



Отладка приложений

Microsoft®

для .NET

Microsoft®

и

WINDOWS®

Джон Роббинс

Microsoft
.net

Wintellect
Raise Your Windows IQ®

Бурные аплодисменты рецензентов

Если вы стали Bugslayer'ом с первой книгой Джона Роббинса, со второй его книгой вы станете управляемым и неуправляемым BugslayerEx'ом.

Кристоф Назаррэ, менеджер разработок Business Objects

Хотя .NET оберегает от многих ошибок, которые мы бы допустили в Win32, отлаживать их все равно приходится. Из книги Джона я узнал много нового о .NET и отладке. Попад в тупик, я прежде всего звоню Джону.

Джеффри Рихтер, соучредитель Wintellect

Это фантастическая книга для Windows- и .NET-разработчиков. Роббинс дает несметное число советов и средств, чтобы сделать процесс более эффективным, не говоря о том, что и более приятным. Он рассматривает отладку с разных сторон: написание кода, который легче отлаживать, инструменты и их скрытые возможности, что происходит внутри отладчика и как расширять Visual Studio.

Брайан Мориарти, специалист по персоналу и чемпион по коду QuickBooks, Intuit

Один из признаков выдающегося разработчика — способность признать, что всегда есть, чему учиться. Новичок вы или гуру, книга Джона все равно чему-нибудь да научит.

Барри Танненбаум, руководитель разработки BoundsChecker, Compuware NuMega Lab

Основное качество, отличающее опытного разработчика от новичка, — способность эффективной отладки. В первом издании этой книги эффективная отладка разложена по полочкам, а в этом описаны все тонкости отладки управляемого кода. Используя арсенал средств, представленных в этой книге и описанные Джоном подходы к отладке, разработчики справятся с самыми трудными ошибками.

Джо Зббот, ведущий проектировщик Microsoft

На этих страницах Джон собрал действительно замечательную коллекцию сведений об отладке. В то время как в других книгах обсуждение отладки ограничивается советами о том, как избежать ошибок и обзором некоторых методик их отслеживания, в книге Джона описываются полезные инструменты и API, которые толком нигде не описаны. Прибавьте к этому массу ценных примеров, и перед вами не книга, а золотая жила для программистов .NET и Win32.

Келли Брок, Electronic Arts

Второе издание книги Джона Роббинса приятно удивило всех его поклонников. Если вы не хотите потратить годы на изучение .NET или Win32, эта книга для вас. Впечатляет, что даже самые сложные темы Джон Роббинс излагает просто и доступно. Мне кажется, что эта книга должна стать эталоном книг для разработчиков. Я программирую для Windows уже 19 лет, и, если мне придется оставить на полке единственную книгу, я оставлю эту.

Озирис Педрозо, Optimizer Consulting

Visual Studio .NET — прекрасное средство разработки, и когда я с ним столкнулся, то решил, что имею все, что нужно. Но Джон Роббинс снова представил книгу, в которой объясняются вещи, о которых я и не знал, что мне их нужно знать! Еще раз спасибо, Джон, за великолепный ресурс для .NET-разработчиков!

Питер Иерарди, Software Evolutions

Это самая увлекательная, глубокая, подробная и жизненная книга о секретах отладки в Windows, написанная опытным ветераном, прошедшим огонь и воду. Прочтите ее и узнаете, как избежать и исправить сложнейшие ошибки. Эта книга — главная надежда человечества на улучшение качества ПО.

Спенсер Лау, разработчик, подразделение SQL Server Microsoft

Если вы хоть раз сорвали сроки проекта из-за ошибок — читайте книгу Джона! Джон не только научит, как искать эти мерзкие ошибки, но и расскажет об инструментах и подходах, которые прежде всего помогут избежать ошибок.

Джеймс Нэфтел, менеджер продукта, XcelleNet

John Robbins

Debugging applications

for Microsoft®

.NET

and Microsoft®

WINDOWS

Microsoft® Press

Джон Роббинс

Отладка приложений
для Microsoft[®]
.NET
и Microsoft[®]
WINDOWS

Москва, 2004

 РУССКАЯ РЕДАКЦИЯ

УДК 004.45
ББК 32.973.26-018.2
P58

Роббинс Джон

P58 Отладка приложений для Microsoft .NET и Microsoft Windows /Пер. с англ. — М.: Издательство «Русская Редакция», 2004. — 736 стр.: ил.

ISBN 978-5-7502-0243-0

В книге описаны тонкости отладки всех видов приложений .NET и Win32: от Web-сервисов XML до служб Windows. Каждая глава снабжена примерами, которые позволят увеличить продуктивность отладки управляемого и неуправляемого кода. На прилагаемом компакт-диске содержится более 6 Мб исходных кодов примеров и полезных отладочных утилит.

Книга состоит из 19 глав, 2 приложений и предметного указателя. Издание снабжено компакт-диском, содержащим исходные тексты примеров, утилиты и инструментальные отладочные средства.

УДК 004.45
ББК 32.973.26-018.2

Подготовлено к изданию по лицензионному договору с Microsoft Corporation, Редмонд, Вашингтон, США.

Macintosh — охраняемый товарный знак компании Apple Computer Inc. ActiveX, BackOffice, JScript, Microsoft, Microsoft Press, MSDN, NetShow, Outlook, PowerPoint, Visual Basic, Visual C++, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, Win32, Windows и Windows NT являются товарными знаками или охраняемыми товарными знаками корпорации Microsoft в США и/или других странах. Все другие товарные знаки являются собственностью соответствующих фирм.

Все названия компаний, организаций и продуктов, а также имена лиц, используемые в примерах, вымышлены и не имеют никакого отношения к реальным компаниям, организациям, продуктам и лицам.

ISBN 0-7356-1536-5 (англ.)
ISBN 978-5-7502-0243-0

- © Оригинальное издание на английском языке, John Robbins, 2003
- © Перевод на русский язык, Microsoft Corporation, 2004
- © Оформление и подготовка к изданию, издательство «Русская Редакция», 2004

Оглавление

Благодарности	XIII
Введение	XIV
Для кого эта книга?	XVI
Как читать эту книгу и что нового во втором издании	XVI
Требования к системе	XVIII
Файлы примеров	XVIII
Обратная связь	XIX
Служба поддержки Microsoft Press	XX
 ЧАСТЬ I	
СУЩНОСТЬ ОТЛАДКИ	1
Глава 1 Ошибки в программах: откуда они берутся и как с ними бороться?	2
Ошибки и отладка	2
Что такое программные ошибки?	3
Обработка ошибок и решения	6
Планирование отладки	14
Необходимые условия отладки	15
Необходимые навыки	15
Выработка мастерства	17
Процесс отладки	18
Шаг 1. Воспроизведи ошибку	19
Шаг 2. Опиши ошибку	20
Шаг 3. Всегда предполагай, что ошибка твоя	20
Шаг 4. Разделяй и властвуй	21
Шаг 5. Мысли творчески	21
Шаг 6. Усишь инструментарий	22
Шаг 7. Начни интенсивную отладку	23
Шаг 8. Проверь, что ошибка устранена	23
Шаг 9. Научись и поделись	25
Последний секрет отладки	25
Глава 2 Приступаем к отладке	26
Следите за изменениями проекта вплоть до его окончания	26
Системы управления версиями	27
Системы отслеживания ошибок	31
Выбор правильных систем	32
Планирование времени построения систем отладки	33
Создавайте все компоновки с использованием символов отладки	34
При работе над управляемым кодом рассматривайте предупреждения как ошибки	38
При работе над неуправляемым кодом рассматривайте предупреждения как ошибки (в большинстве случаев)	41
Разрабатывая неуправляемый код, знайте адреса загрузки DLL	44
Как поступать с базовыми адресами управляемых модулей?	48
Разработайте несложную диагностическую систему для заключительных компоновок	56

Частые сборки программы и дымовые тесты обязательны	57
Частые сборки	58
Дымовые тесты	59
Работу над программой установки следует начинать немедленно	60
Тестирование качества должно проводиться с отладочными компоновками	61
Устанавливайте символы ОС и создайте хранилище символов	62
Исходные тексты и серверы символов	70

Глава 3 Отладка при кодировании 72

Assert, Assert, Assert и еще раз Assert	74
Как и что утверждать	75
Утверждения в .NET Windows Forms или консольных приложениях	83
Утверждения в приложениях ASP.NET и Web-сервисах XML	92
Утверждения в приложениях C++	102
Различные типы утверждений в Visual C++	106
assert, ASSERT и ASSERTE	106
ASSERT_KINDOF и ASSERT_VALID	108
Главное в реализации SUPERASSERT	115
Trace, Trace, Trace и еще раз Trace	130
Трассировка в Windows Forms и консольных приложениях .NET	131
Трассировка в приложениях ASP.NET и Web-сервисах XML	133
Трассировка в приложениях C++	135
Комментировать, комментировать и еще раз комментировать	135
Доверяй, но проверяй (Блочное тестирование)	137

ЧАСТЬ II 141

ПРОИЗВОДИТЕЛЬНАЯ ОТЛАДКА

Глава 4 Поддержка отладки ОС и как работают отладчики Win32 142

Типы отладчиков Windows	143
Отладчики пользовательского режима	143
Отладчики режима ядра	146
Поддержка отлаживаемых программ операционными системами Windows	148
Отладка Just-In-Time (JIT)	148
Автоматический запуск отладчика (опции исполнения загружаемого модуля)	152
MiniDBG — простой отладчик Win32	154
WDBG — настоящий отладчик	173
Чтение памяти и запись в нее	175
Точки прерывания и одиночные шаги	178
Таблицы символов, серверы символов и анализ стека	183
Шаг внутрь, Шаг через и Шаг наружу	191
Итак, вы хотите написать свой собственный отладчик	192
Что после WDBG?	193

Глава 5 Эффективное использование отладчика Visual Studio .NET 195

Расширенные точки прерывания	196
Подсказки к точкам прерывания	197
Быстрое прерывание на функции	199
Модификаторы точек прерывания по месту	205
Несколько точек прерывания на одной строке	208
Окно Watch	209
Вызов методов в окне Watch	210
Команда Set Next Statement	212

Глава 6 Улучшенная отладка приложений .NET в среде Visual Studio .NET	215
Усложненные точки прерывания для программ .NET	216
Условные выражения	216
Окно Watch	220
Автоматическое развертывание собственных типов	221
Советы и хитрости	224
DebuggerStepThroughAttribute и DebuggerHiddenAttribute	224
Отладка в смешанном режиме	225
Удаленная отладка	226
ILDASM и промежуточный язык Microsoft	228
Начинаем работу с ILDASM	229
Основы CLR	234
MSIL, локальные переменные и параметры	235
Важные команды	237
Другие инструменты восстановления алгоритма	242
Глава 7 Усложненные технологии неуправляемого кода в Visual Studio .NET	245
Усложненные точки прерывания для неуправляемого кода	245
Усложненный синтаксис точек прерывания	246
Точки прерывания в системных и экспортируемых функциях	247
Условные выражения	250
Точки прерывания по данным	252
Окно Watch	255
Форматирование данных и вычисление выражений	255
Хронометраж кода в окне Watch	257
Недокументированные псевдорегистры	258
Автоматическое разворачивание собственных типов	258
Удаленная отладка	265
Советы и уловки	268
Отладка внедренного кода	268
Окно Memory и автоматическое обновление	269
Контроль исключений	269
Дополнительные советы по обработке символов	272
Отключение от процессов Windows 2000	272
Обработка дампов-файлов	272
Язык ассемблера x86	274
Основы архитектуры процессоров	275
Кое-какие сведения о встроенном ассемблере Visual C++ .NET	281
Команды, которые нужно знать	282
Частая последовательность команд: вход в функцию и выход из функции	285
Вызов процедур и возврат из них	287
Соглашения вызова	288
Доступ к переменным: глобальные переменные, параметры и локальные переменные	294
Дополнительные команды, которые нужно знать	299
Манипуляции со строками	304
Распространенные ассемблерные конструкции	308
Ссылки на структуры и классы	309
Полный пример	311
Окно Disassembly	313
Исследование стека «вручную»	317
Советы и хитрости	320

Глава 8 Улучшенные приемы для неуправляемого кода с использованием WinDBG	323
Прежде чем начать	324
Основы	326
Что случается при отладке	330
Получение помощи	330
Обеспечение корректной загрузки символов	331
Процессы и потоки	335
Общие вопросы отладки в окне Command	340
Просмотр и вычисление переменных	340
Исполнение, проход по шагам и трассировка	341
Точки прерывания	347
Исключения и события	350
Управление WinDBG	352
Магические расширения	353
Загрузка расширений и управление ими	353
Важные команды расширения	354
Работа с файлами дампа	359
Создание файлов дампа	359
Открытие файлов дампа	360
Отладка дампа	361
Son of Strike (SOS)	362
Использование SOS	363

ЧАСТЬ III

МОЩНЫЕ СРЕДСТВА И МЕТОДЫ ОТЛАДКИ ПРИЛОЖЕНИЙ .NET 371

Глава 9 Расширение возможностей интегрированной среды разработки Visual Studio .NET	372
Расширение IDE при помощи макросов	374
Параметры макросов	375
Проблемы с проектами	376
Элементы кода	377
CommenTater: лекарство от распространенных проблем?	379
Введение в надстройки	387
Исправление кода, сгенерированного мастером Add-In Wizard	389
Решение проблем с кнопками панелей инструментов	391
Создание окон инструментов	393
Создание на управляемом коде страниц свойств окна Options	395
Надстройка SuperSaver	399
Надстройка SettingsMaster	405
Вопросы реализации SettingsMaster	411
Будущие усовершенствования SettingsMaster	412

Глава 10 Мониторинг управляемых исключений	413
Введение в Profiling API	414
Запуск средства профилирования	420
ProfilerLib	422
ExceptionMon	424
Внутрипроцессная отладка и ExceptionMon	425
Использование исключений в .NET	430

Глава 11 Трассировка программы	433
Установка ловушек при помощи Profiling API	433
Запрос уведомлений входа и выхода	434
Реализация функций-ловушек	434
Встраивание	436
Преобразователь идентификаторов функций	436
Использование FlowTrace	437
Некоторые сведения о реализации FlowTrace	439
Что после FlowTrace	441
 ЧАСТЬ IV	
МОЩНЫЕ СРЕДСТВА И МЕТОДЫ ОТЛАДКИ	
НЕУПРАВЛЯЕМОГО КОДА	443
Глава 12 Нахождение файла и строки ошибки по ее адресу	444
Создание и чтение MAP-файла	446
Содержание MAP-файла	447
Получение информации об исходном файле, имени функции и номере строки	450
PDB2MAP: создание MAP-файлов постфактум	452
Использование CrashFinder	454
Некоторые сведения о реализации	457
Что после CrashFinder?	462
Глава 13 Обработчики ошибок	464
Структурная обработка исключений против обработки исключений C++	465
Структурная обработка исключений	465
Обработка исключений C++	468
Избегайте использования обработки исключений C++	470
API-функция SetUnhandledExceptionFilter	475
Использование API CrashHandler	477
Преобразование структур EXCEPTION_POINTERS	502
Минидампы	503
API-функция MiniDumpWriteDump	504
Укрощение MiniDumpWriteDump	505
Глава 14 Отладка служб Windows и DLL, загружаемых в службы	515
Основы служб	515
API	516
Защита	517
Отладка служб	518
Отладка базового кода	518
Отладка службы	519
Глава 15 Блокировка в многопоточных приложениях	527
Советы и уловки, касающиеся многопоточности	527
Не используйте многопоточность	528
Не злоупотребляйте многопоточностью	528
Делайте многопоточными только небольшие изолированные фрагменты программы	528
Выполняйте синхронизацию на как можно более низком уровне	529
Работая с критическими секциями, используйте спин-блокировку	532
Не используйте функции CreateThread/ExitThread	533

Опасайтесь диспетчера памяти по умолчанию	534
Получайте дампы в реальных условиях	535
Уделяйте особое внимание обзору кода	536
Тестируйте многопоточные приложения на многопроцессорных компьютерах	537
Требования к DeadlockDetection	540
Общие вопросы разработки DeadlockDetection	541
Использование DeadlockDetection	542
Реализация DeadlockDetection	545
Перехват импортируемых функций	545
Детали реализации	553
Что после DeadlockDetection?	567

Глава 16 Автоматизированное тестирование 570

Проклятие блочного тестирования: UI	570
Требования к Tester	571
Использование Tester	572
Сценарии Tester	573
Запись сценариев	577
Реализация Tester	580
Уведомления и воспроизведение файлов в TESTER.DLL	580
Реализация TESTREC.EXE	596
Что после Tester?	607

Глава 17 Стандартная отладочная библиотека C и управление памятью 609

Особенности стандартной отладочной библиотеки C	610
Использование стандартной отладочной библиотеки C	611
Ошибка в DCRT	613
Полезные функции DCRT	617
Выбор правильной стандартной отладочной библиотеки C для вашего приложения	618
Использование MemDumperValidator	619
Использование MemDumperValidator в программах C++	626
Использование MemDumperValidator в программах C	627
Глубокая проверка	628
Реализация MemDumperValidator	632
Инициализация и завершение в программах C++	633
И куда же подевались все сообщения об утечках памяти?	634
Использование MemStress	635
Интересные проблемы с MemStress	637
Кучи операционной системы	638
Советы по отслеживанию проблем с памятью	640
Обнаружение записи в неинициализированную память	640
Нахождение записи данных после окончания блока	641
Потрашающие ключи компилятора	647
Ключи проверки ошибок в период выполнения	647
Ключ проверки безопасности буфера	653

Глава 18 FastTrace: высокопроизводительная утилита трассировки серверных приложений 655

Фундаментальная проблема и ее решение	656
Использование FastTrace	657
Объединение журналов трассировки	658
Реализация FastTrace	659

Глава 19 Утилита Smooth Working Set	661
Оптимизация рабочего набора	662
Работа с SWS	666
Настройка компиляторов SWS	666
Выполнение приложений вместе с SWS	668
Генерирование и использование файла порядка	669
Реализация SWS	671
Функция _penter	671
Формат файла .SWS и перечисление символов	675
Период выполнения и оптимизация	680
Что после SWS?	683
 Ч А С Т Ь V	
ПРИЛОЖЕНИЯ	685
Приложение А Чтение журналов Dr. Watson	686
Журналы Dr. Watson	688
Приложение Б Ресурсы для разработчиков приложений .NET и Windows	696
Книги	696
Разработка ПО	697
Отладка и тестирование	698
Технологии .NET	699
Языки C/C++	700
ОС Windows и технологии Windows	700
Процессоры Intel и аппаратные средства ПК	701
Программные средства	702
Web-сайты	703
Предметный указатель	704
Об авторе	710

Моей жене Пэм.

Я тебе еще не говорил сегодня, как я тобой горжусь?

Памяти Хелен Роббинс.

Ты всегда нас объединяла. Нам страшно не хватает тебя.

Благодарности

Если вы читали первое издание этой книги, какие-нибудь статьи в рубрике «Bugs-layer», слушали мои выступления на конференциях или были на моих учебных курсах, я вам очень признателен! Ваш интерес к отладке и написанию правильного кода — это то, что заставило меня много и напряженно поработать над вторым изданием. Благодарю за переписку и дискуссии. Вы подвигли меня на большое дело, спасибо.

Пять экстраординарных людей помогли этой книге выйти в свет, и у меня не хватает слов, чтобы выразить им свою благодарность: Сэлли Стикни (редактор проекта в квадрате!), Роберт Лайон (технический редактор), Джин Росс (технический редактор), Виктория Тулман (редактор рукописи) и Роб Нэнс (художник). Из моих бессвязных записей и уродливых рисунков они сделали книгу, которую вы держите в руках. Они приложили просто громадные усилия, и я не знаю, как их благодарить.

Как и в первом издании, мне помогла замечательная «Команда Рецензентов». Эти душевные ребята получали мои наброски и советовали абсолютно потрясающие отладочные трюки. Они представляют элиту нашего бизнеса, и мне неловко, что я отнял у них столько времени. Вот они, все как на подбор: Джо Эббот (Microsoft), Скотт Байлес (Gas Powered Games), Келли Брок (Electronic Arts), Питер Иерарди (Software Evolutions), Спенсер Лай (Microsoft), Брайан Мориарти (Intuit), Джеймс Нэфтел (XcelleNet), Кристоф Назарпе (Business Objects), Озирис Педрозо (Optimizer Consulting), Энди Пеннел (Microsoft), Джеффри Рихтер (Wintellect) и Барри Танненбаум (Compuware).

Мне также льстит, что я могу считать себя одним из Wintellect'уалов, сделавших огромный вклад в эту книгу: Джим Бэйл, Франческо Балена, Роджер Боссонье, Джейсон Кларк, Пола Дениэлс, Питер ДеБетта, Дино Эспозито, Гэри Эвинсон, Дэн Фергас, Льюис Фрейзер, Джон Лэм, Берни МакКой, Джэф Просиз, Брэнт Ректор, Джеффри Рихтер, Кенн Скрибнер и Крис Шелби.

В заключение, как обычно, огромное спасибо моей жене Пэм. Она пожертвовала многими вечерами и выходными, пока я писал. Даже когда я был в полном отчаянии, она по-прежнему верила в успех, воодушевляла меня, и я довел дело до конца. Дорогая, с этим покончено. Получай своего муженька обратно.

Введение

Ошибки — жуткая гадость. Многоточие... Ошибки являются причиной обреченных на гибель проектов с сорванными сроками, ночными бдениями и опостылевшими коллегами. Ошибки могут превратить вашу жизнь в кошмар, поскольку, если изрядное их число затаится в вашем продукте, пользователи могут прекратить его применение, и вы потеряете работу. Ошибки — серьезный бизнес.

Много раз люди из нашей среды называли ошибки всего лишь досадным недоразумением. Это утверждение далеко от истины, как никакое другое. Любой разработчик расскажет вам о проектах с немыслимым количеством ошибок и даже о компаниях, загнувшихся оттого, что их продукт содержал столько ошибок, что был непригоден. Когда я писал первое издание этой книги, NASA потеряла космический зонд, направленный на Марс, из-за ошибок, допущенных при выработке требований и проектировании ПО. Во время написания данного издания на солдат американского спецназа упала бомба, направленная на другую цель. Причиной была программная ошибка, возникшая при смене источника питания в системе наведения. По мере того как компьютеры управляют все более ответственными системами, медицинскими устройствами и сверхдорогой аппаратурой, программные ошибки вызывают все меньше улыбок и не рассматриваются как нечто самой собой разумеющееся.

Я надеюсь, что эта книга прежде всего поможет вам узнать, как писать программы с минимальным числом ошибок и отлаживать их побыстрее. При правильном подходе вы сэкономите на отладке массу времени. Речь не идет о выработке требований и проектировании, но отлаживать вы наверняка научитесь более грамотно. В этой книге описывается интегральный подход к отладке. Я рассматриваю отладку не как отдельный шаг, а как составную часть общего цикла производства ПО. Я считаю, что ее следует начинать на этапе выработки требований и продолжать вплоть до стадии производства.

Две вещи делают отладку в средах Microsoft .NET и Microsoft Windows сложной и отнимающей много времени. Во-первых, отладка требует опыта — в основном вам потребуется все постигать самим. Даже если у вас специальное образование, бьюсь об заклад, что вы никогда не сталкивались со специальным курсом, посвященным отладке. В отличие от таких эзотерических предметов, как методы автоматической верификации программ на языках программирования, которые ни один дурак не использует, или разработка отладчиков для дико прогрессивных и жутко распараллеленных компьютеров, наука отладки, применяемая в коммерческом ПО, похоже, совсем не популярна в вузовском истеблишменте. Некоторые профессора наставляют: главное — не писать программы с ошибками. Хотя это и выдающаяся мысль и идеал, к которому все мы стремимся, в действительности все слегка по-другому. Изучение систематизированных проверенных методик отлад-

ки не спасет от очередной ошибки, но следование рекомендациям этой книги поможет вам сократить число ошибок, вносимых в код, а те из них, которые все-таки туда прокрались, найти быстрее.

Вторая проблема в том, что, несмотря на обилие прекрасных книг по отдельным технологиям .NET и Windows, ни в одной из них отладка не описана подробно. Для отладки в рамках любой технологии нужно знать гораздо больше, чем отдельные аспекты технологии, описываемой в той или другой книге. Одно дело знать, как встроить элемент управления ASP.NET на страницу, совсем другое — как полностью отладить элемент управления ASP.NET. Для его отладки нужно знать все тонкости .NET и ASP.NET, знать, как различные DLL помещаются в кэш ASP.NET и как ASP.NET находит элементы управления. Многие книги объясняют реализацию таких сложных функций, как соединение с удаленной базой данных с применением современных технологий, но когда в вашей программе не работает «db.Connect (“Foo”)» — а рано или поздно это обязательно случается! — приходится самому разбираться во всей технологической цепочке. Кроме того, хотя есть несколько книг по управлению проектами, в которых обсуждаются вопросы отладки, в них делается упор на управленческие и административные проблемы, а не на задачи разработчиков. Эти книги могут включать прекрасную информацию о планировании отладки, но от этого мало толку, когда вы сталкиваетесь с разрушением базы данных или сбоем при возврате из функции обратного вызова.

Идея этой книги — плод моих проб и ошибок как разработчика и менеджера, старающегося вовремя поставить высококачественный продукт, и как консультанта, пытающегося помочь другим завершить свои разработки в срок. Год за годом я накапливал знания и подходы, применяемые для решения двух описанных проблем, чтобы облегчить разработку Windows-приложений. Для решения первой проблемы (отсутствия формального обучения по вопросам отладки) я написал первую часть этой книги — четкий курс отладки с уклоном в коммерческую разработку. Что касается второй проблемы (потребности в книге по отладке именно в .NET, а также в традиционной Windows-среде), я считаю, что написал книгу, заполняющую пробел между специфическими технологиями и будничными, но жизненно необходимыми практическими методами отладки.

Я считаю, мне просто повезло заниматься почти исключительно вопросами отладки последние восемь лет. Сориентировать свою карьеру на отладку мне помогли несколько событий. Первое: я был одним из первых инженеров, работавших в компании NuMega Technologies (ныне часть Compuware) над такими крупными проектами, как BoundsChecker, TrueTime, TrueCoverage и SoftICE. Тогда же я начал вести рубрику «Bugslayer» в «MSDN Magazine», а затем взялся и за первое издание этой книги. Благодаря фантастической переписке по электронной почте и общению с инженерами, разрабатывающими все мыслимые типы приложений, я получил огромный опыт.

И, наконец, самое важное, что сформировало мое мировоззрение, — участие в создании и работе Wintellect, что позволило мне пойти далеко вперед и помогать в решении весьма серьезных проблем компаниям по всему миру. Представьте, что вы сидите на работе, на часах — полдень, в голове — никаких идей, а клиент может обанкротиться, если вы не найдете ошибку. Сценарий устрашающий, но адреналина хоть отбавляй. Работа с лучшими инженерами в таких компаниях,

как Microsoft, eBay, Intuit и многими другими — лучший из известных мне способов узнать все методы и хитрости для устранения ошибок.

Для кого эта книга?

Я написал эту книгу для разработчиков, которые не хотят допоздна сидеть на работе, отлаживая программы, и хотят улучшить качество своего кода и организации. Я также написал эту книгу для менеджеров и руководителей коллективов, которые хотели бы иметь более эффективные команды разработчиков.

С технической точки зрения, «идеальный читатель» — это некто, имеющий опыт разработки для .NET или Windows от одного до трех лет. Я также рассчитываю, что читатель является членом реальной команды и уже поставил хотя бы один продукт. Хотя я и не сторонник навешивать ярлыки, в программной отрасли разработчики с таким уровнем опыта называются «средними».

Для опытных разработчиков тоже будет польза. Многие из наиболее заинтересованных корреспондентов в переписке по первому изданию этой книги были опытные разработчики, которым, казалось бы, и учиться уже нечему. Я был заинтригован тем, что эта книга помогла им добавить новые инструменты в свой арсенал. Так же, как и в первом издании, группа замечательных друзей под названием «Команда Рецензентов» просматривала и критиковала все главы, прежде чем я отправлял их в Microsoft Press. Эти инженеры, перечисленные в разделе «Благодарности» этой книги, — сливки общества разработчиков, благодаря им каждый читатель этой книги узнает что-нибудь полезное.

Как читать эту книгу и что нового во втором издании

Первое издание было ориентировано на отладку, связанную с Microsoft Visual Studio 6 и Microsoft Win32. Поскольку появилась совершенно новая среда разработки, Microsoft Visual Studio .NET 2003, и совершенно новая парадигма программирования, .NET, есть еще о чем рассказать. На самом деле в первом издании было 512 страниц, а в этой — около 850, так что новой информации хватает. Несколько моих рецензентов сказали: «Непонятно, почему ты называешь это вторым изданием, это же совершенно новая книга!» Чтобы вы правильно понимали, насколько второе издание больше первого, замечу, что в первом издании 2,5 Мб исходных текстов, а в этом — 6,9! Не забывайте: это только исходные тексты и вспомогательные файлы, а не скомпилированные двоичные файлы (скомпилировав все, вы получите более 1 Гб). Что еще интересней, я даже не включил две главы из первого издания во второе. Как видите, это совершенно новая книга.

Я разделил книгу на четыре части. Первые две (главы с 1 по 8) следует читать по порядку, поскольку материал в них изложен в логической последовательности.

В части I «Сущность отладки» (главы с 1 по 3) я даю определение видов ошибок и описываю процесс отладки, которому следуют все порядочные разработчики. По просьбе читателей первого издания я расширил и углубил обсуждение этих тем. Я также рассматриваю инфраструктурные требования, необходимые для правильной коллективной отладки. Настоятельно рекомендую уделить особое внимание вопросу установки сервера символов в главе 2. Наконец, поскольку вы

можете (и должны) уделять огромное внимание отладке на этапе кодирования, я рассказываю про упреждающую отладку при написании кода. Заключительное слово в обсуждении темы первой части — в главе 3, в которой говорится об утверждениях в .NET и Win32.

Часть II «Производительная отладка» (главы с 4 по 8) я начинаю объяснением поддержки отладки со стороны ОС и рассказываю о работе отладчика Win32, так как Win32-отладка имеет больше потаенных мест, чем .NET. Чем лучше вы разберетесь с инструментарием, тем лучше сможете его применять. Я также достаточно глубоко разбираю отладчик Visual Studio .NET, так что вы научитесь выжимать из него по максимуму как в .NET, так и в Win32. Одна вещь, которую я узнал, работая с программистами как опытными, так и очень опытными, — они используют лишь крошечную часть возможностей отладчика Visual Studio .NET. Хотя такие сантименты могут казаться странными в устах автора книги об отладке, я хочу, насколько это возможно, оградить вас от применения отладчика. Читая книгу, вы увидите, что моя цель в первую очередь — научить вас избегать ошибок, а не находить их. Я также хочу научить вас использовать максимум возможностей отладчика, поскольку все-таки настанут времена, когда вы будете его применять.

В части III «Мощные средства и методы отладки приложений .NET» (главы с 9 по 11) я предлагаю несколько утилит для .NET-разработки. В главе 9 описаны потрясающие возможности расширения Visual Studio .NET. Я представляю несколько отличных макросов и надстроек, которые помогут ускорить разработку независимо от того, с чем вы работаете: с .NET или только с Win32. В главах 10 и 11 рассказывается об отличном интерфейсе .NET Profiling API и представляются два инструмента, которые помогут вам отслеживать исключения и ход выполнения ваших .NET-приложений.

В заключительной части «Мощные средства и методы отладки неуправляемого кода» (главы с 12 по 19) предлагаются решения распространенных проблем отладки, с которыми вы столкнетесь при написании Windows-приложений. Я раскрываю темы от поиска исходного файла и номера строки для сбойного адреса, до корректной обработки сбоев приложений. Главы с 15 по 18 были и в первом издании, однако я существенно изменил их текст, а некоторые утилиты (Deadlock-Detection, Tester и MemDumperValidator) полностью переписал. Кроме того, такие утилиты, как Tester, прекрасно работают как с неуправляемым кодом, так и с .NET. И, наконец, я добавил два новых отладочных инструмента для Windows: FastTrace (глава 18) и Smooth Working Set (глава 19).

Приложения (А и Б) содержат дополнительную информацию, которую вы найдете полезной в своих отладочных приключениях. В приложении А я объясняю, как читать и интерпретировать журнал программы Dr. Watson. В приложении Б вы обнаружите аннотированный список ресурсов (книг, инструментов, Web-сайтов), которые помогли мне отточить свое мастерство как разработчика/отладчика.

В первом издании я предложил несколько врезок с фронтовыми очерками об отладке. Реакция была ошеломляющей, и в этом издании я существенно увеличил их число. Надеюсь, поделившись с вами примерами некоторых действительно «хороших» ошибок, я помог обнаружить (или внести!) аналогичные, и вы увидели практическое применение рекомендуемых мной подходов и методик. Мне также хотелось бы помочь вам избежать ошибок, сделанных мной.

У меня был список вопросов, которые мне задали в связи с первым изданием, и на них я ответил во врезках «Стандартный вопрос отладки».

Требования к системе

Чтобы проработать эту книгу, вам потребуются:

- Microsoft Windows 2000 SP3 или более поздняя версия, Microsoft Windows XP Professional или Windows Server 2003;
- Microsoft Visual Studio .NET Professional 2003, Microsoft Visual Studio .NET Enterprise Developer 2003 или Microsoft Visual Studio .NET Enterprise Architect 2003.

Файлы примеров

Я уже сказал, что одних исходных текстов на диске 6,9 Мб. Учитывая, что это больше, чем в ином коммерческом проекте, держу пари, что ни в одной другой книге по .NET или Windows вы столько примеров не найдете. Здесь более 20 утилит или библиотек и более 35 примеров программ, демонстрирующих отдельные конструкции. Между прочим, в это число не входят блочные тесты для утилит и библиотек! Код большинства утилит проверялся в таком огромном количестве коммерческих приложений, что я сбился со счета, когда их число перевалило за 800. Я горжусь, что столько компаний сочли мой код достаточно хорошим для своих продуктов, и надеюсь, что вам он тоже пригодится.

Роберт Лайон, фантастический технический редактор этой книги, собрал DEBUGNET.CHM, который выступает в роли README-файла и содержит информацию о том, как компоновать и использовать код в ваших проектах, а также описывает каждую двоичную компоновку.

С файлами примеров также поставляются следующие стандартные средства от Microsoft:

- Application Compatibility Toolkit (ACT) версия 2.6;
- Debugging Tools for Windows версия 6.1.0017.2.

Я разрабатывал и проверял все проекты в Microsoft Visual Studio .NET Enterprise Edition 2003. Что касается ОС, я тестировал в Windows 2000 Service Pack 3, Windows XP Professional Service Pack 1 и Windows Server 2003 RC2 (прежде называвшуюся Windows .NET Server 2003).

ВНИМАНИЕ! ANSI-код Windows 98/Me

Поскольку Microsoft Windows Me устарела, я не поддерживал ОС, предшествующие Windows 2000. Для Windows 2000 и более поздних я внес соответствующие изменения, в том числе перевел весь свой код в UNICODE. Я использовал макросы из TCHAR.H, и интерфейсы к библиотекам, поддерживающим ANSI-символы, остались. Однако я не компилировал ни одной программы как ANSI/мультитайт, так что здесь могут возникнуть проблемы с компиляцией или ошибки при выполнении.

ВНИМАНИЕ! Сервер символов DBGHELP.DLL

В нескольких утилитах с неуправляемым кодом я использовал сервер символов DBGHELP.DLL, поставляемый с Debugging Tools for Windows версии 6.1.0017.2. Поскольку DBGHELP.DLL теперь можно поставлять со своими приложениями, я включил эту библиотеку в каталоги Release и Output дерева исходных кодов. Поищите более новую версию Debugging Tools for Windows по адресу www.microsoft.com/ddk/debugging и скачать последнюю версию DBGHELP.DLL. Для компиляции DBGHELP.LIB включена в Visual Studio .NET.

Если захотите использовать мои утилиты с неуправляемым кодом, запишите новую версию DBGHELP.DLL в каталог, содержащий утилиту. С Windows 2000 и Windows XP поставляется версия DBGHELP.DLL, предшествующая 6.1.0017.2.

Обратная связь

Мне очень интересно ваше мнение об этой книге. Если у вас есть вопросы или собственные фронтальные очерки об отладке, буду рад их услышать! Идеальное место для ваших вопросов по этой книге и по отладке в целом — форум «Debugging and Tuning» на www.wintellect.com/forum. Прелесть этого форума в том, что здесь вы можете покопаться среди вопросов других читателей и отслеживать возможные исправления и изменения.

Если у вас есть вопросы, которые неудобно публиковать на форуме, отправьте e-mail по адресу john@wintellect.com. Имейте в виду, что я порядочно разъезжаю и получаю очень много электронной почты, так что вы не всегда получите ответ мгновенно. Но я обязательно постараюсь вам ответить.

Спасибо за внимание и счастливой отладки!

*Джон Роббинс
Февраль 2003
Холлис, Нью Гемпшир*

Служба поддержки Microsoft Press

Мы приложили все усилия, чтобы обеспечить точность сведений, изложенных в книге и содержащихся в файлах примеров. Поправки к этой книге предоставляются Microsoft Press через World Wide Web по адресу:

<http://www.microsoft.com/mspress/support/>

Чтобы подключиться к базе знаний Microsoft Press и найти нужную информацию, откройте страницу:

<http://www.microsoft.com/mspress/support/search.asp>

Если у вас есть замечания, вопросы или предложения по поводу этой книги или прилагаемого к ней CD или вопросы, на которые вы не нашли ответа в Knowledge Base, присылайте их в Microsoft Press по электронной почте:

mspinput@microsoft.com

или обычной почтой:

Microsoft Press

Attn: Debugging Applications for Microsoft .NET and Microsoft Windows Editor

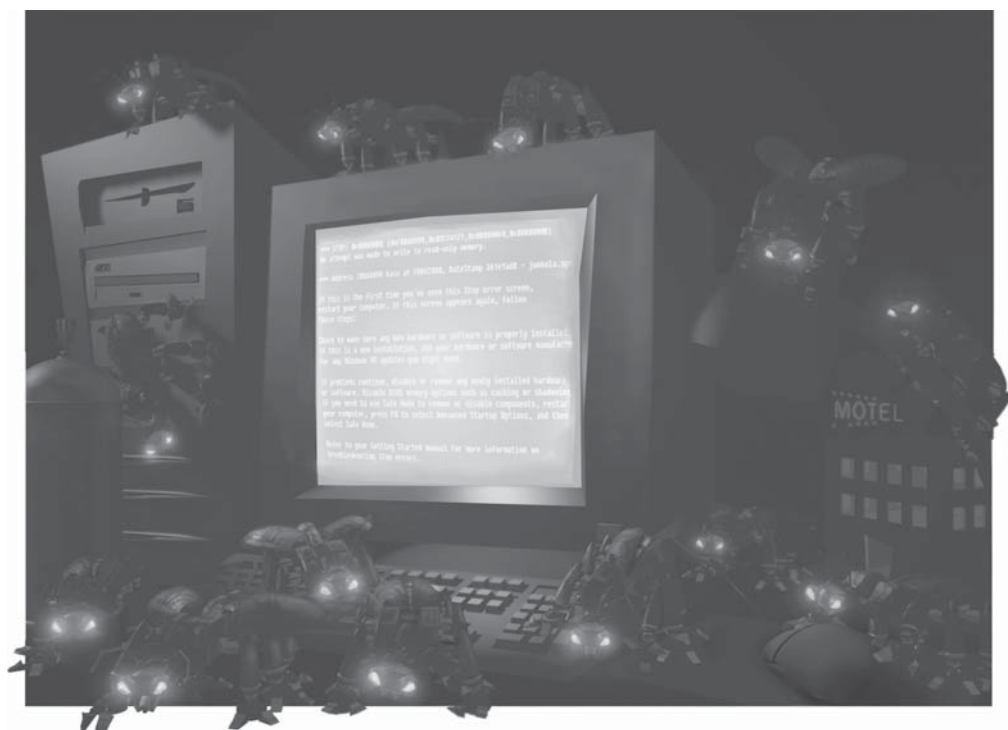
One Microsoft Way

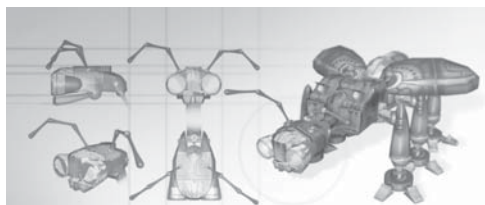
Redmond, WA 98052-6399

Пожалуйста, обратите внимание на то, что по этим адресам не предоставляется техническая поддержка.

ЧАСТЬ I

СУЩНОСТЬ ОТЛАДКИ





Ошибки в программах: откуда они берутся и как с ними бороться?

Отладка — тема очаровательная, какой бы язык программирования или платформу вы ни использовали. Именно на этой стадии разработки ПО инженеры орут на свои компьютеры, пинают их ногами и даже выбрасывают. Людям, обычно немногословным и замкнутым, такая эмоциональность не свойственна. Отладка также известна тем, что заставляет вас проводить над ней ночи напролет. Я не встречал инженера, который бы звонил своей супруге (супругу), чтобы сказать: «Милая (милый), я не могу приехать домой, так как мы получаем огромное удовольствие от разработки UML-диаграмм и хотим задержаться!» Однако я встречал массу инженеров, звонивших домой, причитая: «Милая, я не могу приехать домой, так как мы столкнулись с потрясающей ошибкой в программе».

Ошибки и отладка

Ошибки в программах — это круто! Они помогают вам узнать, как все это работает. Мы все занялись своим делом потому, что нам нравится учиться, а вылавливание ошибок дает нам ни с чем не сравнимый опыт. Не знаю сколько раз, работая над новой книгой, я располагался в своем офисе, выискивая хороший «баг». Как здорово находить и устранять такие ошибки! Конечно же, самые крутые ошибки в программах — это те, что вы найдете до того, как заказчик увидит результат вашей работы. А вот если ошибки в ваших программах находят заказчики, это совсем плохо.

Разработка ПО аномальна по двум причинам. Во-первых, это новая и в чем-то еще незрелая область по сравнению с другими формами инженерного искусства,

такими как конструирование или разработка электрических схем. Во-вторых, пользователи вынуждены принимать ошибки в программах, в частности, в программах для персональных компьютеров. Хотя они и мирятся с ними, но далеко не в восторге, находя их. Но те же самые заказчики никогда не допустят ошибок в конструкции атомного реактора или медицинского оборудования. ПО занимает все большее место в жизни людей, и близок час, когда оно перестанет быть свободным искусством. Я не сомневаюсь, что законы, накладывающие ответственность в других технических областях, в конце концов начнут действовать и для ПО.

Вы должны беспокоиться об ошибках в программах, так как в конечном счете они дорого обходятся вашему бизнесу. Очень скоро заказчики начинают обращаться к вам за помощью, заставляя вас тратить свое время и деньги, поддерживая текущую разработку, в то время как конкуренты уже работают над следующими версиями. Затем невидимая рука экономики нанесет вам удар: заказчики начнут покупать программы конкурентов, а не ваши. ПО в настоящее время востребовано больше, чем капитальные вложения, поэтому борьба за высокое качество программ будет только накаляться. Многие приложения поддерживают расширяемый язык разметки (Extensible Markup Language, XML) для ввода и вывода, и ваши пользователи потенциально могут переключаться с программы одного поставщика на программу другого, переходя с одного Web-сайта на другой. Это благо для пользователей будет означать, что если наши программы будут содержать большое количество ошибок, то ваше и мое трудоустройство окажется под вопросом. Это же будет побуждать к созданию более качественных программ. Позвольте мне сформулировать это иначе: чем больше ошибок в ваших программах, тем вероятней, что вы будете искать новую работу. Нет ничего более ненавистного, чем заниматься поиском работы.

Что такое программные ошибки?

Прежде, чем приступить к отладке, нужно дать определение ошибки. Мое определение таково: нечто, что вызывает головную боль у пользователя. Любая ошибка может быть отнесена к одной из следующих категорий:

- нелогичный пользовательский интерфейс;
- неудовлетворенные ожидания;
- низкая производительность;
- аварийные завершения или разрушение данных.

Нелогичный пользовательский интерфейс

Нелогичный пользовательский интерфейс хоть и не является очень серьезным видом ошибок, очень раздражает. Одна из причин успеха ОС Microsoft Windows — в одинаковом в общих чертах поведении всех разработанных для Windows приложений. Отклоняясь от стандартов Windows, приложение становится «тяжелым» для пользователя. Прекрасный пример такого нестандартного, досаждающего поведения — реализация с помощью клавиатуры функции поиска (Find) в Microsoft Outlook. Во всех других англоязычных приложениях на планете, разработанных для Windows, нажатие Ctrl+F вызывает диалог для поиска текста в текущем окне. А в Microsoft Outlook Ctrl+F переадресует открытое сообщение, что, как я полагаю, является ошибкой. Даже после многих лет работы с Outlook я никак не могу

запомнить, что для поиска текста в открытом сейчас сообщении, надо нажимать клавишу F4.

В клиентских приложениях довольно просто решить все проблемы нелогичности пользовательского интерфейса. Достаточно лишь следовать рекомендациям книги Microsoft Windows User Interface (Microsoft Press, 1999), доступной также в MSDN Online по адресу <http://msdn.microsoft.com/library/en-us/dnuiue/html/welcome.asp>. Если вы чего-то не найдете в этой книге, посмотрите на другие приложения для Windows, делающие что-то похожее на то, что вы пытаетесь реализовать, и следуйте этой модели. Создается впечатление, что Microsoft имеет бесконечные ресурсы и неограниченное время. Если вы задействуете преимущества их всесторонних исследований в процессе решения проблем логичности, то это не будет вам стоить руки или ноги.

Если вы работаете над интерфейсом Web-приложения, ваша жизнь существенно труднее: здесь нет стандартов на пользовательский интерфейс (UI). Как пользователи, мы знаем, что довольно трудно найти хороший UI в браузере. Для разработки хорошего пользовательского интерфейса для Web-клиента я могу порекомендовать две книги. Первая — это образцово-показательная библия Web-дизайна: «Jacob Nielsen, Designing Web Usability: The Practice of Simplicity». Вторая — небольшая, но выдающаяся книга, которую вы должны дать всем доморощенным спецам по эргономике, которые не могут ничего нарисовать, не промочив горло (так некоторые начальники хотят делать UI, а сами никогда не работали на компьютере). Это книга «Steve Krug, Don't Make Me Think! A Common Sense Approach to Web Usability». Разрабатывая что-либо для Web-клиента, помните, что не все пользователи имеют 100-мегабитные каналы. Поэтому сохраняйте UI простым и избегайте загрузки с сервера множества мелочей. Исследуя замечательные клиентские Web-интерфейсы, компания User Interface Engineering (www.uie.com) нашла, что такие простые решения, как *CNN.com*, нравятся всем пользователям. Простой набор понятных ссылок на информационные разделы кажется им выглядящим лучше, чем что-либо еще.

Неудовлетворенные ожидания

Неудовлетворенные ожидания пользователя — одна из самых трудноразрешимых ошибок. Она обычно возникает в самом начале проекта, если компания недостаточно исследует реальные потребности пользователей. При обоих видах проектирования — будь то «коробочные продукты» (разрабатываемые для продажи) или Информационные Технологии (программы собственной разработки для нужд собственного предприятия) — причина этой ошибки восходит к проблемам взаимодействия.

В общем, коллективы разработчиков не общаются напрямую с заказчиками своих программ, поэтому они сами не изучают, что нужно пользователям. В идеале все члены коллектива разработчиков должны навещать заказчиков, чтобы увидеть, что они делают с их программами. У вас откроются глаза, если вы понаблюдаете из-за плеча заказчика, как используется ваша программа. Кроме того, такой опыт позволит вам понять, что, по мнению заказчика, должна делать ваша программа. Вообще-то я бы весьма рекомендовал вам прекратить сейчас чтение и разработать график встреч с заказчиком. Не могу сказать, что этого достаточно, но чем больше вы говорите с заказчиком, тем лучшим разработчиком вы будете.

В дополнение к поездкам к заказчику поможет наличие команды, анализирующей звонки и электронную почту в службу поддержки. Такая обратная связь позволит разработчикам увидеть проблемы, с которыми сталкиваются пользователи.

Порой уровень ожиданий пользователей существенно выше, чем может дать разработка. Такая инфляция ожидания пользователя является классическим результатом очковтирательства, и вы должны сопротивляться представлению в ложном свете возможностей вашей разработки при данной цене. Когда пользователи не знают, чего им ожидать от разработки, они склонны полагать, что разработка содержит больше ошибок, чем на самом деле. Основное правило в такой ситуации — никогда не обещать того, чего вы не можете сделать, и всегда делать то, что обещали.

Низкая производительность

Пользователей очень расстраивают ошибки, приводящие к снижению производительности при обработке реальных данных. Такие ошибки (а причина их — в недостаточном тестировании) порой выявляются только на реальных больших объемах данных. Один из проектов, над которым я работал, BoundsChecker 3.0 компании NuMega, содержал подобную ошибку в первой версии технологии FinalCheck. Эта версия FinalCheck добавляла отладочную и контекстно-зависимую информацию прямо в текст программы, чтобы BoundsChecker подробнее описывал ошибки. Увы, мы недостаточно протестировали FinalCheck на реальных приложениях перед выпуском BoundsChecker 3.0. В итоге гораздо больше пользователей, чем мы предполагали, не смогло задействовать эту возможность. В последующих выпусках мы полностью переписали FinalCheck. Но первая версия имела низкую производительность, и поэтому многие пользователи больше с ней не работали, хотя это была одна из самых мощных и полезных функций. Что интересно, мы выпустили BoundsChecker 3.0 в 1995 году, а семь лет спустя все еще были люди (по крайней мере двое), которые говорили мне, что они не работают с FinalCheck из-за такого негативного опыта!

Бороться с низкой производительностью можно двумя способами. Во-первых, сразу определите требования к производительности. Чтобы узнать, есть ли проблемы производительности, ее нужно с чем-то сравнивать. Важной частью планирования производительности является сохранение ее основных показателей. Если ваше приложение начинает терять 10% этих показателей или больше, остановитесь и определите, почему упала производительность, и предпримите шаги по исправлению положения. Во-вторых, убедитесь, что вы тестируете свои приложения по наиболее близким к реальной жизни сценариям, и начинайте делать это в процессе разработки как можно раньше.

Вот один из наиболее часто задаваемых разработчиками вопросов: «Где взять эти самые реальные данные для тестирования производительности?» Ответ — попросить у заказчиков. Никогда не вредно спросить, можете ли вы получить их данные, чтобы обеспечить тестирование. Если заказчик беспокоится о конфиденциальности своих данных, попробуйте написать программу, которая изменит важную часть информации. Заказчик запустит эту программу и, убедившись, что измененные в результате ее работы данные не являются конфиденциальными, передаст их вам. Чтобы стимулировать заказчика предоставить вам свои данные, бывает полезно передать ему некоторое бесплатное ПО.

Аварийные завершения или разрушение данных

Аварийные завершения и разрушение данных — это то, что ассоциируется с ошибками у большинства программистов и пользователей. Я к этой категории отношу также утечки памяти. Пользователи в принципе могли бы работать с этими ошибками, но аварийные завершения их добивают. Вот почему большая часть этой книги посвящена решению этих проблем. Кроме того, аварийные завершения и разрушение данных — наиболее распространенный тип ошибок. Некоторые из них разрешить легко, другие же почти неразрешимы. Главное, вы никогда не должны поставлять разработку заказчику, зная, что она содержит хотя бы одну такую ошибку.

Обработка ошибок и решения

Хотя поставка ПО без ошибок возможна (при условии, что вы уделяете достаточно внимания деталям), я по опыту знаю, что большинство коллективов разработчиков не достигло такого уровня зрелости разработки ПО. Ошибки — это реальность. Однако вы можете минимизировать количество ошибок в своих приложениях. Это как раз то, что делают коллективы разработчиков, поставляющих высококачественные разработки (и их много). Причины ошибок, в общем, таковы:

- короткие или невозможные для исполнения сроки;
- подход «Сначала кодируй, потом думай»;
- непонимание требований;
- невежество или плохое обучение разработчика;
- недостаточная приверженность к качеству.

Короткие или невозможные для исполнения сроки

Мы все работали в коллективах, в которых «руководство» устанавливало сроки, либо сходя к гадалке, либо, если первое стоило слишком дорого, с помощью магического шара¹. Хоть нам и хотелось бы верить, что в большинстве нереальных графиков работ виновато руководство, чаще бывает, что его не в чем винить. В основу планирования обычно кладется оценка объема работ программистами, а они иногда ошибаются в сроках. Забавные они люди! Они интроверты, но почти всегда оптимисты. Получив задание, программисты верят до глубины души, что могут заставить компьютер пуститься в пляс. Если руководитель приходит и говорит, что в приложение надо добавить преобразование XML, рядовой инженер говорит: «Конечно, босс! Это займет три дня». Конечно же, он может даже не знать, что такое «XML», но он знает, что это займет три дня. Большой проблемой является то, что разработчики и руководители не принимают в расчет время обучения, необходимое для того, чтобы смочь реализовать функцию. В разделе «Планирование времени построения систем отладки» главы 2 я освещу некоторые аспекты, которые надо учитывать при планировании. Кто бы ни был виноват в ошибочной оценке сроков поставки — руководство ли, разработчики или обе сторо-

¹ Детская игрушка «Magic 8-Ball», выпускающаяся в США с 40-х годов, которая «отвечает на любые вопросы». На самом деле содержит 20 стандартных обтекаемых ответов.
— Прим. перев.

ны — главное, что нереальный график ведет к халтуре и снижению качества продукта.

Мне посчастливилось работать в нескольких коллективах, которые поставляли ПО к сроку. Каждый из этих коллективов владел ситуацией, и нам удавалось определять реалистичные сроки поставки. Мы рассчитывали эти сроки на основе набора реализуемых функций. Если компания находила предложенную дату поставки неприемлемой, мы исключали какие-то возможности, чтобы успеть к сроку. Кроме того, план согласовывался с каждым членом коллектива разработчиков прежде, чем мы представляли его руководству. Так поддерживалась вера коллектива в своевременное завершение задания. И что интересно: кроме того, что эти продукты поставлялись в срок, они были самыми качественными из всех, над которыми я работал.

Подход «Сначала кодируй, потом думай»

Выражение «Сначала кодируй, потом думай» придумал мой друг Питер Иерарди. Каждого из нас в той или иной степени можно упрекнуть в таком подходе. Игры с компиляторами, кодирование и отладка — забавное времяпрепровождение. Это то, что нам интересно в нашем деле в первую очередь. Очень немногим из нас нравится сидеть и ваять документы, описывающие, что мы собираемся делать.

Однако, если вы не пишете эти документы, вы столкнетесь с ошибками. Вместо того чтобы в первую очередь подумать, как избежать ошибок, вы начинаете доводить код и разбираться с ошибками. Понятно, что такая тактика усложнит задачу, потому что вы будете добавлять все новые ошибки в уже нестабильный базовый исходный код. Компания, в которой я работаю, помогает в отладке самых трудных задач. Увы, зачастую, будучи приглашенными для оказания помощи в разрешении проблем, мы ничего не могли поделать, потому что проблемы были обусловлены архитектурой программ. Когда мы доводим эти проблемы до руководства заказчика и говорим, что для их решения надо переписать часть кода, мы порой слышим: «Мы вложили в этот код слишком много денег, чтобы его менять». Явный признак того, что у компании проблема «Сначала кодируй, потом думай!» Отчитываясь о работе с клиентом, в качестве причины, по которой мы не смогли помочь, мы просто пишем «СКПД».

К счастью, решить эту проблему просто: планируйте свои проекты. Есть несколько хороших книг о сборе требований и планировании проектов. Я даю ссылки на них в приложении Б и весьма рекомендую вам познакомиться с ними. Хотя это не очень привлекательно и даже немного болезненно, предварительное планирование жизненно важно для исключения ошибок.

В отзывах на первое издание этой книги звучала жалоба на то, что я рекомендовал планировать проекты, но не говорил, как это делать. Недовольство закономерное, и хочу сказать, что я обращаю внимание на эту проблему и здесь, во втором издании. Единственная загвоздка в том, что я на самом деле не знаю как! Вы можете подумать, что я использую неблагоприятный авторский прием — оставлять непонятный вопрос читателю в качестве упражнения. Читайте дальше, и вы узнаете, какие тактики планирования применял я сам. Надеюсь, что они подадут некоторые идеи и вам.

Если вы прочтете мою биографию в конце книги, то заметите, что я не занимался программированием почти до 30 лет, т. е. в действительности это моя вторая профессия. Моей первой профессией было прыгать с самолетов, преследовать врага, так как я был «зеленым беретом». Если это не было подготовкой к программированию, то я не знаю, чем это было! Конечно, если вы встретите меня сейчас, вы увидите лишь толстого коротышку с одутловатым зеленым лицом — результат длительного сидения перед монитором. Но я был настоящим мужиком. Правда!

Проходя службу, я научился планировать. При проведении спецопераций шансы погибнуть достаточно велики, так что вы крайне заинтересованы в наилучшем планировании. Планируя одну из таких операций, командование помещает всю группу в так называемую «изоляция». В форте Брегг (Северная Калифорния), где дислоцируется спецназ, есть места, где действительно изолируют команду для продумывания сценариев операции. Ключевой вопрос при этом был: «Что может привести к гибели?» Что, если, спрыгнув с парашютами, мы минуем точку невозврата, а ВВС не найдут место нашего приземления? А если у нас будут раненые или убитые к моменту прыжка? А что случится, если после приземления мы не найдем командира партизан, с которым предполагалась встреча? А если он приведет с собой больше людей, чем предполагалось? А если засада? Мы всегда придумывали вопросы и искали на них ответы, прежде чем покинуть место изоляции. Идея заключалась в том, чтобы иметь план действий в любой ситуации. Поверьте: если есть шанс гибели при выполнении задания, вы захотите знать и учесть все возможные варианты.

Когда я занялся программированием, я стал использовать этот вид планирования в работе. В первый раз я пришел на совещание и сказал: «Что будет, если Боб помрет до того, как мы минуем стадию выработки требований?» Все заерзали. Поэтому теперь я формулирую вопросы менее живодерски, вроде: «Что будет, если Боб выиграет в лотерею и уволится до того, как мы минуем стадию выработки требований?» Идея та же. Найдите все сомнительные места и путаницу в ваших планах и займитесь ими. Это не просто сделать, слабым инженеров это сводит с ума, но ключевые вопросы всегда всплывут, если вы копнете достаточно глубоко. Скажем, на стадии выработки требований вы будете задавать такие вопросы: «Что, если наши требования не соответствуют пожеланиям пользователей?» Такие вопросы помогут предусмотреть в бюджете время и деньги на выработку согласованных требований. На стадии проектирования вы будете спрашивать: «Что, если производительность не будет достаточно высока?» Такие вопросы напоминают вам о необходимости сесть и определить основные параметры производительности и начать планирование, как вы собираетесь добиваться значений этих параметров при тестировании в реальных условиях. Планировать будет существенно проще, если вы сможете свести все вопросы в таблицу. Просто будьте благодарны, что ваша жизнь не зависит от поставки ПО в срок.

Отладка: фронтовые очерки

Тяжелый случай с СКПД

Боевые действия

Клиент пригласил нас, так как у него возникли серьезные проблемы с быстроедействием, а дата поставки стремительно приближалась. В первую очередь мы попросили сделать 15-минутный обзор, чтобы быстро разобраться в терминологии и получить представление о том, как устроен проект. Клиент дал нам одного из архитекторов системы, и он начал объяснение на доске.

Обычно такие совещания с рисованием кружочков и стрелочек занимают 10–15 минут. Однако архитектор выступал уже 45 минут, а я еще ни в чем толком не разобрался. Наконец я окончательно запутался и снова попросил сделать 10-минутный обзор системы. Мне не нужно было знать все — только основные особенности. Архитектор начал заново, но через 15 минут он осветил только 25% системы!

Исход

Это была большая СОМ-система, и теперь я начал понимать, в чем заключались проблемы быстрогодействия. Было ясно, что кто-то в команде был без ума от СОМ. Он не удовлетворился глотком из стакана с живительной влагой СОМ, а хлебал из 200-литровой бочки. Как я позже понял, системе нужно было 8–10 основных объектов, а эта команда имела 80! Чтобы вы поняли, насколько нелеп такой подход, представьте себе, что практически каждый символ в строке был представлен СОМ-объектом. Классический случай нулевого практического опыта авторов!

Примерно через полдня я ответил руководителю в сторонку и сказал, что в таком виде мы с производительностью ничего не сделаем, потому что ее убивают накладные расходы самой СОМ. Он не очень-то обрадовался, услышав это, и немедленно выдал печально известную фразу: «Мы вложили в этот код слишком много денег, чтобы его менять». Увы, но в этом случае мы практически ничем не смогли помочь.

Полученный опыт

Этот проект страдал из-за нескольких основных проблем с самого начала. Во-первых, члены коллектива отдали проектирование не разработчикам. Во-вторых, они сразу начали кодирование, в то время как надо было начинать с планирования. Не было абсолютно никаких других мыслей, кроме кодирования, и кодирования прямо сейчас. Классический случай проблемы «Сначала кодируй, потом думай», которой предшествовало «Бездумное Проектирование». Я не могу не подчеркнуть это: вам необходимо произвести реалистичную оценку технологии и планировать свою разработку до того, как включите компьютер.

Непонимание требований

Надлежащее планирование также минимизирует основной источник ошибок в разработке — расползания функций. Расползание функций — добавление первоначально не планировавшихся функций — это симптом плохого планирования и неадекватного сбора требований. Добавление функций в последнюю минуту, будь то реакция на давление конкурентов, любимая «штучка» разработчика или нажим руководства, вызывает появление большего числа ошибок в ПО, чем что-либо еще.

Разработка ПО очень зависит от мелочей. Чем больше деталей вы проясните до начала кодирования, тем меньше риск. Есть только один способ достичь должного внимания к мелочам — планировать ключевые события и реализацию своих проектов. Конечно, это не означает, что вам нужно отойти от дел и сочинить тысячи страниц документации, описывающей, что вы собираетесь делать.

Лучший документ такого рода, созданный мной, был просто серией рисунков на бумаге (бумажные прототипы) UI. Основываясь на исследованиях и результатах обучения у Джэйреда Спула и его компании User Interface Engineering, моя команда рисовала UI и прорабатывала сценарии поведения пользователей. Деля это, мы должны были сосредоточиться на требованиях к разработке и точно понять, как пользователи собирались исполнять свои задачи. Если вставал вопрос, какое поведение предполагалось по данному сценарию, мы доставали свои бумажные прототипы и вновь работали над сценарием.

Даже если бы вы могли спланировать все на свете, вы все равно должны понимать требования к своей разработке, чтобы правильно их реализовать. В одной из компаний, где я работал (к счастью, меньше года), требования к разработке казались очень простыми и понятными. На проверку, однако, большинство членов коллектива недостаточно понимало потребности пользователей, чтобы разобраться, что же программа должна делать. Компания допустила классическую ошибку, радикально увеличив «поголовье» разработчиков, не удосужившись обучить новичков. Вследствие этого, хоть и планировалось исключительно все, разработка запоздала на несколько лет, и рынок отверг ее.

В этом проекте были две большие ошибки. Первая: компания не желала тратить время на то, чтобы тщательно объяснить потребности пользователей разработчикам, которые были новичками в предметной области, хотя некоторые из нас просили об обучении. Вторая: многие разработчики, старые и молодые, не проявляли интереса к изучению предметной области. В итоге команда каждый раз меняла направление, когда сотрудники отделов маркетинга и продаж в очередной раз объясняли требования. Код был настолько нестабильным, что понадобились месяцы, чтобы заставить работать безотказно даже простейшие пользовательские сценарии.

Вообще лишь немногие компании проводят обучение своих разработчиков в предметной области. Хоть многие из нас и закончили колледжи, мы многого не знаем о том, как заказчики будут использовать наши разработки. Если компании затрачивают адекватное время, честно помогая своим разработчикам понять предметную область, они могут исключить ошибки, вызванные непониманием требований.

Но проблема не только в компаниях. Разработчики сами обязаны обучаться в предметной области. Некоторые считают, что, создавая средства решения зада-

чи, можно дистанцироваться от предметной области. Нет — разработчик отвечает за решение задачи, а не просто предоставляет возможность решения!

Примером предоставления возможности решения является ситуация, когда вы проектируете пользовательский интерфейс, формально работоспособный, но не соответствующий технологии работы пользователей. Другой пример — построение приложения, позволяющее решать сиюминутные задачи, но не дающее возможности приспособиться к изменяющимся потребностям бизнеса.

При решении пользовательских проблем, а не предоставлении возможности решения разработчик старается узнать предметную область, так что созданное вами ПО становится «расширением» пользователя. Лучший разработчик не тот, кто может манипулировать битами, а тот, кто может решать проблемы пользователя.

Невежество и плохое обучение разработчика

Еще одна существенная причина ошибок исходит от разработчиков, не разбирающихся в ОС, языке программирования или технологиях, используемых в проектах. Увы, программистов, готовых признать такой недостаток и стремящихся к обучению, немного.

Во многих случаях, однако, малограмотность является не столько персональным недостатком, сколько правдой жизни. В наши дни так много пластов и взаимозависимостей вовлечено в разработку ПО, что невозможно найти такого человека, кто знал бы все тонкости каждой ОС, языка программирования и технологии. Не знать не стыдно: это не признак слабости и не делает вас главным недомумом в конторе. В здоровом коллективе признание сильных и слабых сторон каждого его члена работает на успех. Учитывая навыки, имеющиеся или отсутствующие у разработчиков, коллектив может получить максимальную выгоду от вложений в обучение. Устраняя слабые стороны каждого, коллектив сможет лучше приспособливаться к непредвиденным обстоятельствам и, как следствие, наращивать совокупный потенциал всей команды. Коллектив может также точнее планировать разработку, если его члены добровольно признают, что они чего-то не знают. Вы можете предусмотреть время для обучения и создать более реалистичный график работ, если члены команды откровенно признают пробелы в своем образовании.

Лучший способ научиться технологии — создать что-либо с ее помощью. Очень давно, когда NuMega послала меня изучать Microsoft Visual Basic, чтобы мы могли писать программы для разработчиков на Visual Basic, я представил план, чему я собираюсь учиться, и это потрясло моего босса. Идея заключалась в создании приложения, оскорблявшем вас; оно называлось «Обидчик». Версия 1 представляла собой форму с единственной кнопкой, щелчок которой выводил случайное оскорбление из числа закодированных в тексте программы. Вторая версия читала оскорбления из базы данных и позволяла вам добавлять оскорбления, используя форму. Третья версия была подключена к корпоративному серверу Microsoft Exchange и позволяла посылать оскорбления работникам компании. Моему руководителю понравилось то, что я собираюсь делать, чтобы изучить технологию. Все ваши руководители заботятся о том, чтобы всегда была возможность доложить боссу о вашей работе в тот или иной день. Если вы предоставите своему руководителю такую информацию, вы попадете в любимчики. Когда я впервые столкнулся с .NET, я просто снова использовал идею Обидчика, который стал называться Обидчик.NET!

О том, какие навыки и знания критичны для разработчиков, я расскажу в разделе «Необходимые условия отладки».

Стандартный вопрос отладки

Нужно ли пересматривать код?

Безусловно! К сожалению, многие компании подходят к этому совершенно неверно. Одна компания, на которую я работал, требовала формальные пересмотры кода точно так же, как это описано в одном из этих фантастических учебников для программистов, который был у меня в колледже. Все было расписано по ролям: был Архивариус для записи комментариев, Секретарь для ведения протокола, Привратник, открывающий дверь, Руководитель, надувающий щеки, и т. д. На самом деле было 40 человек в комнате, но никто из них не читал код. Это была пустая трата времени.

Вариант пересмотра кода, который мне нравится, как раз неформальный. Вы просто садитесь с распечаткой текста программы и читаете его строка за строкой вместе с разработчиком. При чтении вы отслеживаете входные данные и результаты и можете представить все, что происходит в программе. Подумайте о том, что я только что написал. Если это напоминает вам отладку программы, вы совершенно правы. Сосредоточьтесь на том, что делает программа, — именно в этом назначение обзора кода программы.

Другая хитрость, гарантирующая, что пересмотр кода результативен, — привлечение младших разработчиков для пересмотра кода старших. Это не только дает понять менее опытным, что их вклад значим, но и отличный способ познакомить их с разработкой и показать им любопытные приемы и хитрости.

Недостаточная приверженность к качеству

Последняя причина появления ошибок в проектах, на мой взгляд, самая серьезная. Я не встречал компании или программиста, не говоривших, что они приверженцы качества. Увы, на самом деле это не так. Если вы когда-либо сталкивались с компанией или программистами, работавшими качественно, вы понимаете, о чем речь. Они гордятся своим детищем и готовы корпеть над всеми частями продукта, а не только над теми, что им интересны. Например, вместо того чтобы копаться в деталях алгоритма, они выбирают более простой алгоритм и думают, как лучше его протестировать. В конце концов заказчика интересуют не алгоритмы, а качественный продукт. Компании и отдельные программисты, по настоящему приверженные качеству, демонстрируют одни и те же характерные черты: тщательное планирование, персональную ответственность, основательный контроль качества и прекрасные способности к общению. Многие компании и отдельные программисты проходят через разные этапы разработки больших систем (планирование, кодирование и т. п.), но только тот, кто уделяет внимание деталям, поставляет продукцию в срок и высокого качества.

Хорошим примером приверженности качеству служит мой первый ежемесячный обзор кода в компании NuMega. Во-первых, я был поражен, насколько быст-

ро я получил результаты, хотя обычно приходится умолять руководителей хоть о какой-то обратной связи. Одним из ключевых разделов обзора была запись о количестве зарегистрированных в разработке ошибок. Я был ошеломлен тем, что NuMega будет оценивать эту статистику как часть моего обзора производительности. Однако, хотя отслеживание ошибок — жизненно важная часть сопровождения продукта, никакая другая компания, из числа тех, где я работал, не проводила таких проверок столь очевидным образом. Разработчики знают, где кроются ошибки, но нужно заставлять их включать информацию о них в систему слежения за ошибками. NuMega нашла нужный подход. Когда я увидел раздел обзора, посвященный количеству ошибок, поверьте, я стал регистрировать все, что я нашел, независимо от того, насколько это тривиально. Несмотря на заинтересованность технических писателей, специалистов по качеству, разработчиков и руководителей в здоровой конкуренции регистрировать как можно больше ошибок, несколько сюрпризов все же затаились. Но важнее то, что у нас было реальное представление, в каком состоянии находится проект в каждый момент времени.

Другой отличный пример — первое издание этой книги. На компакт-диске, прилагаемом к книге, было около 2,5 Мб исходных текстов программ (это не компилированные программы — только исходные тексты!). Это очень много, и я рад, что это во много раз больше, чем прилагается к большинству других книг. Многие люди не могут себе даже представить, что я потратил больше половины времени, ушедшего на эту книгу, на тестирование этих программ. Народ балдеет, находя ошибки в кодах Bugslayer², и чего я меньше всего хочу — это получать письма типа «Ага! Ошибочка в Bugslayer!». Без ошибок на том компакт-диске не обошлось, но их было только пять. Моим обязательством перед читателями было дать им только лучшее из того, на что я способен. Моя цель в этом издании — не более пяти ошибок в более чем 6 Мб исходных текстов этого издания.

Руководя разработкой, я следовал правилу, которое, я уверен, стимулировало приверженность к качеству. Каждый член коллектива должен подтвердить готовность продукта при достижении каждой вехи проекта. Если кто-либо не считал, что проект готов, проект не поставлялся. Я бы лучше исправил небольшую ошибку и дал бы дополнительный день на тестирование, чем выпустил бы что-то такое, чем коллектив не мог бы гордиться. Это правило соблюдалось не только для того, чтобы все представляли, что качество обеспечено, это также приводило к пониманию каждым своей доли участия в результате. Я заметил интересный феномен: у членов коллектива никогда не было шанса остановить выпуск из-за чужой ошибки — «хозяин» ошибки всегда опережал остальных.

Приверженность качеству задает тон для всей разработки: она начинается с процесса найма и простирается через контроль качества до кандидата на выпуск. Все компании говорят, что хотят нанимать лучших работников, но лишь немногие предлагают соблазнительную зарплату и пособия. Кроме того, некоторые компании не желают обеспечивать специалистов оборудованием и инструментарием, необходимым для высококачественных разработок. К сожалению, многие компа-

² Название рубрики в журнале «MSDN Magazine». В русском переводе, выпускаемом издательством «Русская Редакция», рубрика называется «Отладка и оптимизация». — *Прим. перев.*

нии не хотят тратить \$500 на инструментарий, позволяющий за несколько минут найти причину ошибки, приводящей к аварийному завершению, но спокойно выбрасывают на ветер тысячи долларов на зарплату разработчиков, неделями барахтающихся в попытках найти эту самую ошибку.

Компания также показывает свою приверженность качеству, когда делает самое сложное — увольняет тех, кто не работает по стандартам, установленным в организации. При формировании команды из высококлассных специалистов вы должны суметь сохранить ее. Все мы видели человека, который, кажется, только кислород переводит, но получает повышения и премии, как вы, хотя вы убиваетесь на работе, пашете ночами, а иногда и в выходные, чтобы завершить продукт. В результате хороший работник быстро осознает, что его усилия того не стоят. Он начинает ослаблять свое рвение или, что хуже, искать другую работу.

Будучи руководителем проекта, я, хоть не без боязни, но уволил одного человека за два дня до Рождества. Я знал, что люди в коллективе чувствовали, что он не работал по стандартам. Если бы после Рождества они вернулись и увидели его на месте, я бы начал терять коллектив, который столько формировал. Я зафиксировал факт низкой производительности этого сотрудника, поэтому у меня были веские причины. Поверьте, в армии мне было стрелять легче, чем «устранить» этого человека. Было бы намного проще не вмешиваться, но мои обязательства перед коллективом и компанией — качественно делать работу, на которую я нанят. Всего за все время моей работы в разных организациях я уволил трех человек. Лучшее было пройти через такое потрясение, чем иметь в коллективе того, кто тормозил работу. Я сильно мучался при каждом увольнении, но я должен был это делать. Быть приверженным качеству очень трудно, и это значит, что вы должны делать то, что будет задерживать вас до ночи, но это необходимо для поставок хорошего ПО и заботы о ваших работниках.

Если вы окажетесь в организации, которая страдает от недостаточной приверженности качеству, то поймете, что нет простых путей переделать ее за одну ночь. Руководитель должен найти подходы к вашим работникам и своему руководству для распространения приверженности качеству во всей организации. Рядовой же разработчик может сделать свой код самым надежным и расширяемым в проекте, что будет примером для остальных.

Планирование отладки

Пришло время подумать о процессе отладки. Многие начинают думать об отладке, только споткнувшись на фазе кодирования, но вы должны думать о ней с самого начала, с фазы выработки требований. Чем больше времени вы уделите процессу планирования, тем меньше времени (и денег) вы потратите на отладку впоследствии.

Как я уже говорил, расползание функций может стать убийцей проекта. Чаще всего незапланированные функции добавляют ошибки и наносят вред проекту. Однако это не означает, что ваши планы должны быть высечены в граните. Иногда нужно изменять или добавлять функции для повышения конкурентоспособности разработки или лучшего удовлетворения потребностей пользователей. Главное, что до того, как вы начнете менять свою программу, надо определить и спланировать,

что конкретно вы будете менять. И помнить, что добавление функции затрагивает не только кодирование, но и тестирование, документацию, а иногда и маркетинговые материалы.

В отличной книге Стива МакКонелла (Steve McConnell) «Code Complete» (Microsoft Press, 1993, стр. 25–26) есть упоминание о стоимости исправления ошибки, которая по мере разработки растет экспоненциально, как и стоимость отладки (во многом по тому же сценарию, как и при добавлении или удалении функций).

Планирование отладки производится совместно с планированием тестирования. Во время планирования вам нужно предусмотреть разные способы ускорения и улучшения обоих процессов. Одна из лучших мер предосторожности — написание утилит для дампа файлов и проверки внутренних структур (а при необходимости и двоичных файлов). Если проект читает и записывает двоичные данные в файлы, вы должны включить в чей-то план написание тестовой программы, выводящей данные в читаемом формате. Программа дампа должна также проверять данные и их взаимозависимости в двоичных файлах. Такой шаг сделает тестирование и отладку проще.

Планируя отладку, вы минимизируете время, проведенное в отладчике, и это ваша цель. Может показаться, что такой совет звучит странно в книге об отладке, но смысл в том, чтобы попытаться избежать ошибок. Если вы встраиваете достаточно отладочного кода в свои приложения, то этот код (а не отладчик) подскажет вам, где сидят ошибки. Я освещу подробнее вопросы, касающиеся отладочного кода, в главе 3.

Необходимые условия отладки

Вы не можете быть хорошим отладчиком, если вы не являетесь хорошим программистом-разработчиком и наоборот.

Необходимые навыки

Хорошие отладчики должны обладать серьезными навыками разрешения проблем, что весьма характерно для ПО. К счастью, вы можете учиться и оттачивать свое мастерство. Великих отладчиков/программистов отличает от хороших отладчиков/программистов то, что кроме умения разрешать проблемы, они понимают, как все части проекта работают в составе проекта в целом.

Вот те области, в которых вы должны быть знатоком, чтобы стать великим или по крайней мере лучшим отладчиком/программистом:

- ваш проект;
- ваш язык программирования;
- используемая технология и инструментарий;
- операционная система/среда;
- центральный процессор.

Знай свой проект

Знание проекта есть первая линия защиты UI, логики работы и проблем производительности. Зная, как и где в исходных текстах реализованы функции, вы сможете быстро понять, кто что делает.

К сожалению, все проекты разные, и единственный путь изучить проект — прочитать проектную документацию, если она есть, и пройти по коду с отладчиком. Современные системы разработки имеют браузеры классов, позволяющие увидеть основы устройства программы. Но вам может понадобиться настоящее средство просмотра, такое как Source Insight от Source Dynamics. Кроме того, вы можете задействовать средства моделирования, такие как Microsoft Visual Studio.NET Enterprise Architect, интегрированную с Microsoft Visio, чтобы увидеть взаимосвязи или диаграммы UML (Unified Modeling Language), описывающие программу. Даже минимальные комментарии в тексте программы лучше, чем ничего, если это предотвратит дизассемблирование.

Знай язык реализации

Знать язык (языки) программирования вашего проекта труднее, чем может показаться. Я говорю не только об умении программировать на этом языке, но и о знании того, как исполняется программа, написанная на нем. Скажем, программисты C++ иногда забывают, что локальные переменные, являющиеся классами или перегруженными операторами, могут создавать временные объекты в стеке. В свою очередь оператор присваивания может выглядеть совершенно невинным и при этом требовать большого объема кода для своего исполнения. Многие ошибки, особенно связанные с производительностью, — результат неправильного применения средств языка программирования. Поэтому полезно изучать индивидуальные особенности используемых языков.

Знай технологию и инструментарий

Владение технологиями — первый большой шаг в борьбе с трудными ошибками. Например, если вы понимаете, что делает COM, чтобы создать COM-объект и вернуть его интерфейс, вы существенно сократите время на поиск причины завершения с ошибкой запроса интерфейса. Это же относится к фильтрам ISAPI. Если у вас проблемы с правильно вызванным фильтром, вам надо знать, где и когда INETINFO.EXE должен был загружать ваш фильтр. Я не говорю, что вы должны знать наизусть файлы и строки исходного текста или книги. Я говорю, что вы должны хотя бы в общем понимать используемые технологии и, что более важно, точно знать, где найти подробное описание того, что вам нужно.

Кроме знания технологии, жизненно важно знать инструментарий. В этой книге значительное место уделяется передовым методам использования отладчика, но многие другие средства (скажем, распространяемые с Platform SDK) остаются за пределами книги. Вы поступите очень мудро, если посвятите один день просмотру и ознакомлению с инструментарием, имеющимся в вашем распоряжении.

Знай свою операционную систему/среду

Знание основ работы ОС/среды позволит просто устранять ошибки, а не ходить вокруг них. Если вы работаете с неуправляемым кодом, вы должны суметь ответить на вопросы типа: что такое динамически подключаемая библиотека (DLL)? Как работает загрузчик образов? Как работает реестр? Для управляемого кода вы должны знать, как ASP.NET находит используемые компоненты, когда вызываются финализаторы, чем отличается домен приложения от сборки и т. д. Многие са-

мые неприятные ошибки появляются из-за неправильного использования средств ОС/среды. Мой друг Мэтт Питрек (Matt Pietrek), научивший меня прелестям отладки, утверждает, что знание ОС/среды и центрального процессора отличает богов отладки от простых смертных.

Знай свой центральный процессор

И последнее, что нужно знать, чтобы стать богом отладки неуправляемого кода, — это центральный процессор. Вы должны хоть что-то знать о центральном процессоре для разрешения наиболее неприятных ошибок. Было бы хорошо, если бы аварийное завершение всегда наступало там, где доступен исходный текст, но обычно при аварийном завершении отладчик показывает окно с дизассемблированным текстом. Я всегда удивляюсь, как много программистов не знает (более того, не хочет знать) язык ассемблера. Он не настолько сложен, три-четыре часа, потраченные на его изучение сэкономят вам бесчисленные часы, затрачиваемые на отладку. Еще раз: я не говорю, что вы должны уметь писать собственные программы на ассемблере, я даже не думаю, что сам умею это делать. Главное, чтобы вы могли прочитать их. Все, что вам нужно знать о языке ассемблера, имеется в главе 7.

Выработка мастерства

Имея дело с технологиями, вы должны непрерывно учиться и идти вперед. Хотя я и не могу помочь вам в работе над вашими конкретными проектами, в приложении Б я перечислил все ресурсы, помогавшие мне (а они помогут и вам) стать лучшим отладчиком.

Кроме чтения книг и журналов по отладке, вам также нужно писать утилиты, причем любые. Лучший способ научиться — это работать, а в нашем случае — программировать и отлаживать. Это не только отточит ваши главные навыки, такие как программирование и отладка, но, если рассматривать эти утилиты как настоящие проекты (т. е. завершать их к сроку и с высоким качеством), то вы разовьете и дополнительные навыки, такие как планирование проектов и оценка графика исполнения.

Кстати, завершенные утилиты — прекрасный материал, который можно показать на собеседовании при приеме на работу. Хотя очень немногие программисты берут свои программы на собеседования, работодатели рассматривают таких кандидатов в первую очередь. То, что вы располагаете рядом работ, выполненных в свободное время дома, — свидетельство того, что вы можете завершать свои работы самостоятельно и что вы увлечены программированием, а это позволит вам практически сразу войти в состав 20% лучших программистов.

Если же мне было нужно больше узнать о языках, технологиях и ОС, очень помогало знакомство с текстом программ других разработчиков. Большое количество текстов программ, с которыми можно познакомиться, витает в Интернете. Запуская разные программы под отладчиком, вы можете увидеть, как другие борются с ошибками. Если что-то мешает вам написать утилиту, вы можете просто добавить функцию к одной из утилит из числа найденных.

Для изучения технологии, ОС и виртуальной машины (процессора) можно порекомендовать и методику восстановления алгоритма (reverse engineering). Это

ускорит ваше изучение языка ассемблера и функций отладчика. Прочитав главы 6 и 7, вы будете достаточно знать о промежуточном языке (Microsoft Intermediate Language, MSIL) и языке ассемблера IA32. Я не советовал бы вам начинать с полного восстановления текста загрузчика ОС — лучше начать с задач поскромнее. Так, весьма поучительно познакомиться с реализацией `CoInitializeEx` для неуправляемого кода и класса `System.Diagnostics.TraceListener` — для управляемого.

Чтение книг и журналов, написание утилит, знакомство с текстами других программистов и восстановление алгоритмов — отличные способы повысить мастерство отладки. Однако самые лучшие ресурсы — это ваши друзья и коллеги. Не бойтесь спрашивать их, как они сделали что-либо или как что-то работает; если их не поджимают сроки, они должны быть рады помочь вам. Я люблю, когда люди задают мне вопросы, так как сам узнаю больше, чем те, кто задает мне вопросы! Я постоянно читаю программистские группы новостей, особенно то, что пишут парни из Microsoft, кого называют MVP (Most Valuable Professionals, наиболее ценные профессионалы).

Процесс отладки

В заключение обсудим процесс отладки. Трудновато было определить процесс, работающий для всех видов ошибок, даже «глюков» (которые, кажется, падают с Луны и никакого объяснения не имеют). Основываясь на своем опыте и беседах с коллегами, я со временем понял подход к отладке, которому интуитивно следуют все великие программисты, а менее опытные (или просто слабые) часто не считают очевидным.

Как вы увидите, чтобы реализовать этот процесс отладки, не нужно быть семи пядей во лбу. Самое трудное — начинать этот процесс каждый раз, приступая к отладке. Вот девять шагов, связанных с рекомендуемым мной подходом к отладке (рис. 1-1).

- Шаг 1. Воспроизведи ошибку.
- Шаг 2. Опиши ошибку.
- Шаг 3. Всегда предполагай, что ошибка твоя.
- Шаг 4. Разделяй и властвуй.
- Шаг 5. Мысли творчески.
- Шаг 6. Усишь инструментарий.
- Шаг 7. Начни интенсивную отладку.
- Шаг 8. Проверь, что ошибка исправлена.
- Шаг 9. Научись и поделись.

В зависимости от ошибки вы можете полностью пропустить некоторые шаги, если проблема и место ее возникновения совершенно очевидны. Вы всегда должны начинать с шага 1 и пройти через шаг 2. Однако где-то между шагами 3 и 7 вы можете найти решение и исправить ошибку. В таких случаях, исправив ошибку, перейдите к шагу 8 для проверки сделанных исправлений.

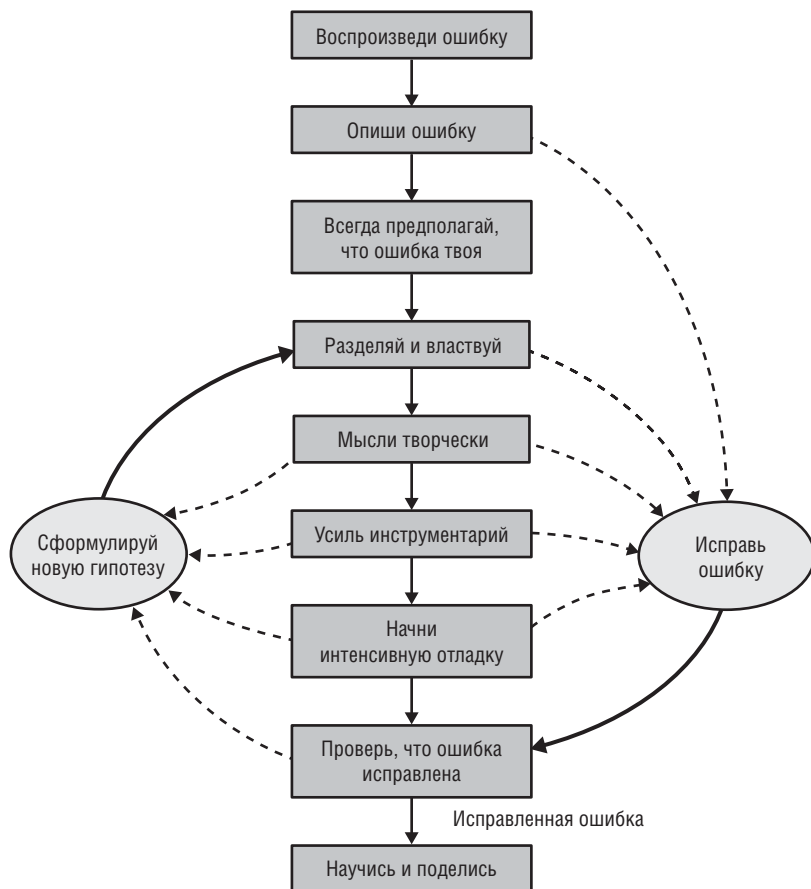


Рис. 1-1. Процесс отладки

Шаг 1. Воспроизведи ошибку

Воспроизведение ошибки — наиболее критичный шаг в процессе отладки. Иногда это трудно или даже невозможно сделать, но, не повторив ошибку, вы, возможно, не устраните ее. Попробуй повторить ошибку, можно дойти до крайности. У меня была ошибка, которую я не мог повторить простым перезапуском программы. Я думал, что какое-то сочетание данных могло быть причиной, но, когда я запускал программу под отладчиком и вводил данные, необходимые для повторения ошибки, прямо в память, все работало. Если вы сталкиваетесь с проблемами синхронизации, возможно, вам придется предпринять некоторые действия по загрузке тех же задач для повторения состояния, при котором возникала ошибка.

Теперь вы, возможно, думаете: «Ну, вот! Конечно же, главное воспроизвести ошибку. Если бы я всегда смог это сделать, мне бы не нужна была ваша книга!» Все зависит от вашего определения слова «воспроизводимость». Мое определение — воспроизведение ошибки на одной машине один раз в течение 24 часов. Этого достаточно для моей компании, чтобы начать работу над ошибкой. Почему? Все

просто. Если вам удастся повторить ошибку на одной машине, то на 30 машинах вам удастся повторить ее 30 раз. Люди сильно заблуждаются, если пытаются повторить ошибку на всех доступных машинах. Если у вас есть 30 человек, чтобы долбить по клавишам, — хорошо. Однако наибольшего эффекта можно добиться, автоматизировав UI, чтобы вывести ошибку на чистую воду. Вы можете воспользоваться программой Tester из главы 17 или коммерческими средствами регрессионного тестирования.

Если вам удалось повторить ошибку в результате каких-то действий, оцените, можете ли вы повторить ошибку, действуя в другом порядке. Какие-то ошибки проявляются только при определенных действиях, другие могут быть воспроизведены различными путями. Идея в том, чтобы посмотреть на поведение программы со всех возможных точек зрения. Повторяя ошибку различными способами, вы гораздо лучше ощущаете данные и граничные условия, вызывающие ее. Кроме того, некоторые ошибки могут маскировать собой другие. Чем больше способов воспроизвести ошибку удастся найти, тем лучше.

Даже если не удастся повторить ошибку, вы все равно должны ее зарегистрировать в протоколе ошибок системы. Если у меня есть ошибка, которую я не могу повторить, я в любом случае регистрирую ее, пометая, что я не смог воспроизвести ее. Позже другой программист, ответственный за эту часть программы, будет понимать, что здесь что-то не так. Регистрируя ошибку, которую вам не удалось повторить, опишите ее как можно подробнее — описание может оказаться достаточно вам или другому специалисту, чтобы решить проблему в другой раз. Хорошее описание особенно важно, так как вы можете установить связь между разными ошибками, которые не удалось воспроизвести, позволяя вам начать рассмотрение различных вариантов поведения.

Шаг 2. Опиши ошибку

Если вы типичный студент технического колледжа, вы скорее всего любите математические и технические дисциплины и не жаловали гуманитарные. В реальной жизни искусство писать почти столь же важно, как и ваше техническое мастерство, так как вам нужно уметь описывать свои ошибки как устно, так и письменно. Сталкиваясь с ошибкой, надо всегда останавливаться после того, как ее удалось воспроизвести, и описывать ее. В идеале вы это делаете в журнале регистрации ошибок системы, и, даже если вы обязаны устранить эту ошибку, описать ее также полезно. Описание ошибки часто помогает устранить ее. Я и не вспомню, сколько раз описание других специалистов помогало мне посмотреть на ошибку с другой стороны.

Шаг 3. Всегда предполагай, что ошибка твоя

За все годы, что я занимаюсь разработкой ПО, лишь несколько раз я сталкивался с ошибкой, причиной которой был компилятор или исполняющая среда. В случае ошибки есть все шансы, что она ваша, и вы всегда должны считать и надеяться, что это так. Если источник ошибки — ваш код, вы по крайней мере можете устранить ее. Если же виноват компилятор или среда, проблема серьезней. Вы должны исключить любую возможность наличия ошибки в вашем коде, прежде чем начнете тратить время на поиск ее где-либо еще.

Шаг 4. Разделяй и властвуй

Если вы воспроизвели ошибку и хорошо описали ее, у вас есть предположения о ее природе и месте, где она прячется. На этом этапе вы начинаете приводить в порядок и проверять свои предположения. Важно помнить фразу: «Обратись к источнику, Люк!»³. Отвлекитесь от компьютера, прочтите исходный текст и подумайте, что происходит при работе программы. Чтение исходного текста заставит вас потратить больше времени на анализ проблемы. Взяв за исходную точку состояние машины на момент аварийного завершения или появления проблемы, проанализируйте различные сценарии, которые могли привести к этой части кода. Если ваше предположение о том, что не так работает, не приводит к успеху, остановитесь и переоцените ситуацию. Вы уже знаете об ошибке немного больше — теперь вы можете переоценить свое предположение и попробовать снова.

Отладка напоминает алгоритм поиска делением пополам. Вы делаете попытки найти ошибку и с каждой итерацией, соответствующей очередному предположению, вы, надо надеяться, исключаете фрагменты программы, где ошибок нет. Продолжая, вы исключаете из программы все больше и больше, пока ошибка не окажется в каком-то одном фрагменте. Так как вы продолжаете развивать гипотезу и узнаете все больше об ошибке, то можете обновить описание ошибки, отразив новые сведения. Делая это, я, как правило, проверяю три-пять основательных гипотез, прежде чем перейду к следующему шагу. Если вы чувствуете, что уже близко, можете проделать немного «легкой» отладки на этом шаге для окончательной проверки предположения. Под «легкой» я понимаю двойную проверку состояний и значений переменных, не просматривая все подряд.

Шаг 5. Мысли творчески

Если ошибка, которую вы пытаетесь исключить, — одна из тех неприятных ошибок, появляющихся только на определенных машинах или которую трудно воспроизвести, посмотрите на нее с разных точек зрения. Это шаг, на котором вы должны начать думать о несоответствии версий, различиях в ОС, проблемах двоичных файлов или их установки и других внешних факторах.

Прием, который иногда, к моему удивлению, работает, состоит в том, чтобы отключиться от проблемы на день-другой. Иногда вы так сосредоточены на проблеме, что за деревьями леса не видите и начинаете пропускать очевидные факты. Отключаясь от ошибки, вы даете шанс поработать над проблемой подсознанию. Я уверен, каждый из читающих эту книгу находил ошибки по дороге с работы домой. Конечно же, трудно отключиться от ошибки, если она задерживает поставку и босс стоит у вас над душой.

В нескольких компаниях, в которых я работал, прерыванием наивысшего приоритета было нечто называемое «разговор об ошибке». Это означает, что вы со-

³ «Use the source, Luke!» — популярный у программистов каламбур, получившийся из фразы героя «Звездных войн» Оби-Ван Кеноби «Use the force, Luke!» («Применяй силу, Люк»). Используется, когда хотят привлечь внимание к исходному тексту, вместо того чтобы искать ответ на вопрос в конференциях или у службы поддержки. В форумах и переписке чаще используют более экспрессивный, хоть и менее легитимный короткий вариант: RTFS (Read The Fucking Source). — *Прим. перев.*

вершенно выбиты из колеи и должны подробно обсудить ошибку с кем-либо. Идея такова: вы идете в чей-то кабинет и представляете свою проблему на доске. Сколько раз я приходил в чужой кабинет, открывал маркер, касался им доски и решал проблему, не проронив ни слова! Именно подготовка разума представить проблему помогает миновать дерево, в которое вы уперлись, и увидеть лес. Человека для разговора об ошибке вы должны выбрать не из числа коллег, с которыми вы тесно работаете над той же частью проекта. Таким образом, вы можете быть уверены, что ваш собеседник не сделает тех же предположений о проблеме, что и вы.

Что интересно, этот «кто-то» даже не обязательно должен быть человеком. Мои кошки, оказывается, — прекрасные отладчики, и они помогли мне найти множество мерзких ошибок. Я собирал их вместе, обрисовывал проблему на доске и давал сработать их сверхъестественным способностям. Конечно же, было трудновато объяснить происходящее почтальону, стоящему на пороге, учитывая, что в такие дни я не принимал душ и ходил в одних трусах.

Есть один человек, с которым следует избегать разговора об ошибках, — это ваша супруга или иная значимая для вас персона. Почему-то тот факт, что вы тесно связаны с этим человеком, означает наличие неразрешимых проблем. Вы, возможно, видели это, пытаясь описать ошибку: глаза собеседника стекленеют, и он или она вот-вот упадет в обморок.

Шаг 6. Усиль инструментарий

Я никогда не понимал, почему некоторые компании позволяют своим программистам разыскивать ошибки неделями, расходуя на это тысячи долларов, хотя соответствующий инструментарий помог бы им найти эту ошибку (и все ошибки, с которыми они встретятся в будущем) за считанные минуты.

Некоторые компании, такие как Compuware и Rational, разрабатывают прекрасные средства как для управляемого, так и для неуправляемого кода. Я всегда пропускаю свои тексты программ через эти средства, прежде чем приступить к трудному этапу отладки. Так как ошибки неуправляемого кода всегда труднее найти, чем ошибки управляемого, эти средства гораздо важнее. Compuware NuMega предлагает BoundsChecker (средство обнаружения ошибок), TrueTime (средство анализа производительности) и TrueCoverage (средство исследования кода программы). Rational предлагает Purify (обнаружение ошибок), Quantify (производительность) и PureCoverage (средство исследования кода). Суть в том, что, если вы не используете средства сторонних производителей для облегчения отладки своих проектов, вы тратите на отладку больше времени, чем необходимо.

Для тех из вас, кто не знаком с этими средствами, объясню, что каждое из них делает. Средство обнаружения ошибок, кроме всего прочего, следит за неправильным обращением к памяти, некорректными параметрами вызовов системных API и СОМ-интерфейсов, утечками памяти и ресурсов. Средство анализа производительности помогает выследить, где ваше приложение работает медленно, — а это всегда совсем не то место, что вы думаете. Информация об исследовании кода программы полезна потому, что, если вы ищете ошибку, вы хотите видеть только реально исполняемые строки программы.

Шаг 7. Начни интенсивную отладку

Я отличаю интенсивную отладку от легкой (упомянутой в шаге 4) по тому, что вы делаете, применяя отладчик. При легкой отладке вы просматриваете только несколько состояний и парочку переменных. При интенсивной же вы проводите много времени, исследуя действия программы. Именно во время интенсивной отладки вы используете расширенные функции отладчика. Ваша цель — как можно больше тяжелой ноши отдать отладчику. В главах с 6 по 8 обсуждаются различные расширенные функции отладчика.

Как и при легкой отладке, в случае интенсивной вам нужно иметь представление о том, где может таиться ошибка, а затем применить отладчик для подтверждения предположения. Никогда не сидите в отладчике из любопытства. Я настоятельно советую записать ваше предположение до запуска отладчика. Это поможет полностью сосредоточиться именно на том, чего вы пытаетесь достичь.

При интенсивной необходимо регулярно просматривать изменения, сделанные для устранения ошибок, обнаруженных с помощью отладчика. Мне нравится работать на этом этапе на двух машинах, установленных рядом. В этом случае я могу устранять ошибку, работая на одной машине, а на другой запускать ту же программу в нормальных условиях. Основная идея заключается в двойной и тройной проверке любых внесенных изменений, чтобы не дестабилизировать нормальную работу программы. Хочу предупредить: начальство терпеть не может, когда вы проверяете программу только на predeterminedных граничных условиях, а не в нормальной среде.

Если вы правильно планируете проект, проводите отладку по шагам, описанным выше, и следуйте рекомендациям главы 2 — будем надеяться, вы не потратите много времени на интенсивную отладку.

Шаг 8. Проверь, что ошибка устранена

Если вы думаете, что ошибка окончательно устранена, следующий шаг в процессе отладки — тестирование, тестирование и еще раз тестирование исправлений. Я уже сказал о необходимости тестировать исправления? Если ошибка — в изолированном модуле в строке программы, вызываемой один раз, тестировать исправление просто. Если же был исправлен центральный модуль, особенно если он управляет структурой данных или чем-то подобным, то надо быть очень внимательным, чтобы исправление не вызвало дополнительных проблем или побочных эффектов в других частях проекта.

При тестировании исправлений, особенно критических частей, надо проверить, что все работает при любом сочетании данных, хороших и плохих. Нет ничего хуже, чем появление двух новых ошибок в результате исправления одной. Если изменяете критический модуль, оповестите остальных членов коллектива о внесенных изменениях. Таким образом, они будут в курсе возможного появления «волновых эффектов».

Отладка: фронтовые очерки

Куда девалась интеграция?

Боевые действия

Один из программистов, с которым я работал в NuMega, думал, что нашел серьезную ошибку в интеграции Visual C++ Integrated Development Environment (VC IDE) компании NuMega, так как она не работала на его машине. Для тех из вас, кто не знаком с VC IDE от NuMega, я немного расскажу о ней. Интеграция программных продуктов от NuMega с VC IDE существует уже много лет. Такая интеграция позволяет появляться окнам, панелям инструментов и меню от NuMega непосредственно в среде VC IDE.

Исход

Этот программист потратил несколько часов, используя отладчик ядра SoftICE для поиска ошибки. Он установил точки останова практически по всей ОС. Наконец он нашел свою «ошибку». Он заметил, что при запуске VC IDE `CreateProcess` вызывается с указанием пути `\\R2D2\VSCommon\MSDev98\Bin\MSDEV.EXE` вместо пути `C:\VSCommon\MSDev98\Bin\MSDEV.EXE`, с которым, как он думал, должен происходить вызов. Иначе говоря, вместо запуска VC IDE с его локальной машины (`C:\VSCommon\MSDev98\Bin\MSDEV.EXE`) он запускал ее со своей старой машины (`\\R2D2\VSCommon\MSDev98\Bin\MSDEV.EXE`). Как такое могло случиться?

Он только что получил новую машину и установил полностью VC IDE компании NuMega для работы. Чтобы упростить себе жизнь, он скопировал ярлыки рабочего стола (файлы LNK) со старой машины, на которой VC IDE была установлена без средств интеграции, на свою новую машину, перетаскивая ярлыки мышью. При перетаскивании ярлыков система изменяет внутренние пути, чтобы отобразить размещение исходных файлов в новых условиях. Поскольку он всегда запускал VC IDE щелчком ярлыка на рабочем столе, который ссылался на старую машину, то и в новых условиях он также запускал VC IDE со старой машины.

Полученный опыт

Этот программист неправильно начинал отладку, сразу же запуская отладчик ядра, вместо попытки повторить проблему разными способами. На шаге 1 процесса отладки («Воспроизведи ошибку») я рекомендовал попытаться повторить ошибку различными способами, чтобы убедиться, что перед вами действительно ошибка, а не несколько ошибок, маскирующих и усложняющих другую. Если бы этот программист следовал шагу 5 («Мысли творчески»), то он был бы освобожден от этой работы, потому что он сначала подумал бы о проблеме вместо немедленного погружения в нее.

Шаг 9. Научись и поделись

Исправляя «хорошую» ошибку (т. е. потребовавшую усилий для того, чтобы ее найти и исправить), вы должны быстро подвести итог пройденному. Мне нравится заносить хорошие ошибки в журнал, чтобы позже можно было посмотреть, что я делал правильно для поиска и решения проблемы. Но еще важнее: я хочу знать, что я делал неправильно, чтобы обходить тупики и отлаживаться и находить ошибки быстрее. Почти все о программировании вы узнаете в процессе отладки, поэтому необходимо использовать все возможности, чтобы извлекать из отладки уроки.

Один из самых важных шагов, который необходимо сделать после исправления хорошей ошибки, — поделиться с коллегами информацией, которую вы получили, исправляя ошибку, особенно если эта ошибка специфична для проекта.

Последний секрет отладки

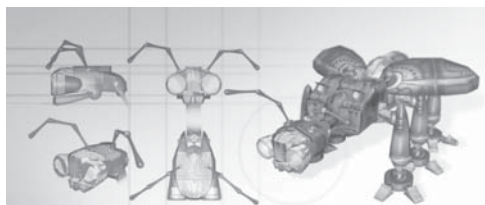
Я бы хотел поделиться с вами последним секретом отладки: отладчик может ответить на все ваши вопросы, только если вы задаете ему корректные вопросы. Еще раз: я советую всегда иметь в голове предположение (что-то такое, что вы хотите доказать или опровергнуть), прежде чем запустить отладчик. Как я рекомендовал на шаге 7, до того как я прикоснусь к отладчику, я записываю свое предположение, чтобы всегда быть уверенным, что у меня есть цель.

Помните: отладчик — это только инструмент вроде отвертки. Он делает только то, что вы ему прикажете. Настоящий отладчик — это мягкое вещество в вашем твердом черепе.

Резюме

В этой главе положено начало определению программных ошибок и описанию решения связанных с ними проблем. Затем обсуждалось, что вы должны знать к моменту начала отладки. Наконец, представлен процесс отладки, которому вы должны следовать в работе над отладкой программы.

Лучший способ отладить программу — исключить ошибки. Если вы хорошо планируете свои проекты, привержены качеству и знаете, как ваша разработка связана с технологиями, ОС и процессором, вы можете минимизировать время, затрачиваемое на отладку.



Приступаем к отладке

В этой главе я опишу важные инфраструктурные инструменты и требования, которые помогут оптимизировать отладку приложений в процессе их создания. Некоторые касаются разработки, а другие представляют собой программные утилиты, однако все они имеют одну общую черту: они позволяют непрерывно следить за развитием проекта. Я убежден, что постоянный контроль — один из важнейших факторов своевременной разработки качественного ПО.

Все идеи, описываемые в этой и следующей главах, основаны на моем опыте работы над реальными программными продуктами, а также на консультировании некоторых компаний. Я не могу представить себе работу без этих инструментов и методов. Многим людям — и мне в том числе — пришлось потратить много сил, чтобы извлечь эти важные уроки, и я с радостью поделюсь ими, чтобы помочь вам сэкономить время и сохранить душевное спокойствие. Возможно, читателям, работающим в группах из двух-трех человек, покажется, что некоторые из моих советов их не касаются, однако это не так. Как бы я ни работал над проектом — в одиночку или в составе большой группы, — я подхожу к нему одинаково. Я принимал участие в самых разных проектах и знаю, что мои рекомендации пригодятся любым группам разработчиков: и маленьким, и самым большим.

Следите за изменениями проекта вплоть до его окончания

Системы управления версиями и отслеживания ошибок — два важнейших инфраструктурных инструмента, поскольку они отражают историю проекта. Некоторые разработчики утверждают, что могут хранить нужные сведения в уме, однако компания все равно должна иметь информацию о ходе проекта на тот случай, если все работавшие над ним сотрудники выигрывают в лотерею и примут решение об увольнении. Обычно документация о требованиях к программе и проектная до-

кументация на всем протяжении проекта ведется плохо, в результате чего единственными реальными документами становятся контрольные журналы систем управления версиями и отслеживания ошибок.

Надеюсь, я вас убедил. Увы, я постоянно сталкиваюсь с группами, которые еще не стали использовать эти инструменты, особенно системы отслеживания ошибок. Как человек, интересующийся историей, я утверждаю: чтобы знать, куда вы идете, вы должны знать, где вы находитесь сейчас и где были раньше. Единственный гарантированный способ достижения этой цели — использование названных мной систем. Наблюдая за частотой обнаружения и решения проблем, применяя систему отслеживания ошибок, можно точнее прогнозировать дату завершения работы над проектом. Система управления версиями дает представление о степени изменения кода, благодаря чему можно определить объем дополнительного тестирования. Кроме того, эти инструменты предоставляют единственный эффективный способ оценки того, насколько действительны изменения, совершаемые в ходе разработки ПО.

Если в вашей группе появится новый разработчик, эти инструменты окупятся за один день. Пусть в самом начале он поработает с системами управления версиями и отслеживания ошибок и проследит путь изменения проекта. Конечно, идеально было бы иметь качественную проектную документацию, но если ее нет, системы управления версиями и отслеживания ошибок по крайней мере предоставят информацию об эволюции кода и укажут на все проблемные области.

Я говорю об этих двух системах одновременно, потому что они неразделимы. Система отслеживания ошибок фиксирует все события, которые могут привести к изменению исходных текстов программы. Система управления версиями регистрирует все сделанные изменения. В идеале следует поддерживать связь между обнаруженными проблемами и изменениями исходных кодов. Это позволяет определить причины и следствия исправления ошибок. Если вы не будете поддерживать такую связь, вы будете часто удивляться некоторым изменениям кода программы. Почти всегда при разработке более поздней версии программы приходится искать программиста, внесшего то или иное изменение, при этом остается только надеяться на то, что он помнит причину своего поступка.

Существуют интегрированные программные продукты, которые автоматически следят за связью изменений исходных текстов программы с ошибками. Если такая возможность в вашей системе отсутствует, поддерживайте связь вручную. Этого можно достигнуть, включая номер ошибки в комментарии, описывающие метод ее исправления. Регистрируя измененный файл в системе управления версиями, указывайте в комментарии к нему номер исправленной ошибки.

Системы управления версиями

Система управления версиями предназначена для контроля не только над исходными кодами проекта. В ней нужно хранить все, что имеет отношение к проекту, включая все планы тестирования, автоматизированные тесты, систему справочной информации и проектную документацию. Некоторые компании включают в нее даже средства сборки приложения (т. е. компилятор, компоновщик, включаемые файлы и библиотеки), позволяющие полностью воссоздать поставленную заказчику версию программы. Если вы сомневаетесь, включать ли какой-нибудь

файл в систему управления версиями, спросите себя, сможет ли эта информация понадобиться в ближайшие пару лет сопровождающим программистам. Если да, ей самое место в системе управления версиями.

Блочные тесты также нужно включать в систему управления версиями

Хотя я только что объяснил важность регистрации в системе управления версиями всего, что только может понадобиться, во многих компаниях этим советом пренебрегают. Одна из крупнейших проблем, с которыми я когда-либо сталкивался в компаниях по разработке ПО, возникла из-за отсутствия в системе управления версиями *блочных тестов* (unit test). Если термин «блочный тест» вам незнаком, я вкратце поясню его. Блочный тест — это фрагмент кода, который управляет выполнением основной программы. (Иногда эти тесты еще называют *тестовыми приложениями* или *средствами тестирования*.) Это тестовый код, создаваемый разработчиком программы для проведения тестирования «прозрачного ящика», или «белого ящика»¹, позволяющего удостовериться в том, что основные операции программы действительно имеют место. Подробное описание блочных тестов см. в главе 25 «Блочное тестирование» книги Стива Макконнелла (Steve McConnell. Code Complete. — Microsoft Press, 1993).

Включение блочных тестов в систему управления версиями позволяет достигнуть двух основных целей. Во-первых, вы облегчите труд разработчиков, которые будут сопровождать программы. Очень часто при модернизации или исправлении кода им — а таким человеком вполне можете оказаться вы сами — приходится изобретать колесо. Это не только требует огромных усилий, но и по-настоящему удручает. Во-вторых, вы упростите сотрудникам отдела контроля качества общее тестирование программы, благодаря чему они смогут сосредоточиться на более важных областях тестирования, таких как производительность и масштабируемость программы, а также ее полнота и соответствие требованиям заказчика. Обязательное включение блочных тестов в систему управления версиями — один из признаков опыта и профессионализма.

Конечно, регистрация блочных тестов в системе управления версиями автоматически означает, что нужно будет поддерживать их соответствие изменениям кода. Да, это возлагает на вас дополнительную ответственность, однако нет ничего хуже, когда сотрудник, осуществляющий поддержку программы, преследует вас и упрекает в разгильдяйстве за то, что блочные тесты больше не работают. Устаревшие блочные тесты в системе управления версиями — большее зло, чем их полное отсутствие.

Просмотрев исходные коды программ, прилагаемых к книге, вы заметите, что все мои блочные тесты входят в их состав. Есть даже отдельный сценарий, позволяющий автоматически создать все блочные тесты для всех моих утилит и примеров. В этой книге я рекомендую только то, что использую сам.

Некоторые читатели, возможно, думают, что поддержка блочных тестов, которую я так отстаиваю, потребует массы дополнительной работы. В действительности это не совсем так, потому что большинство разработчиков (я очень на это наде-

¹ Glass box, white box — программа, поведение которой строго детерминировано. — Прим. перев.

юсь!) уже проводит блочное тестирование. Я только советую регистрировать эти тесты в системе управления версиями, своевременно обновлять их, а также, возможно, написать какой-либо сценарий для их компиляции и компоновки. Следуя правильным методам работы вы сэкономите огромное количество времени. Например, большую часть программ для этой книги я разрабатывал на компьютере с Microsoft Windows 2000 Server. Чтобы сразу приступить к тестированию на компьютере с Microsoft Windows XP, мне нужно было только извлечь код тестов из системы управления версиями и выполнить сценарии их создания. Многие программисты разрабатывают одноразовые блочные тесты, чем усложняют тестирование в среде других ОС из-за невозможности легкого переноса блочных тестов на другую платформу и их компиляции и компоновки. Если все члены группы будут включать блочные тесты в свой код, это позволит им сэкономить много недель работы.

Контроль над изменениями

Отслеживание изменений имеет огромное значение, однако наличие хорошей системы отслеживания ошибок не означает, что разработчикам разрешается вносить крупномасштабные изменения в исходные коды программы, когда захочется. Это сделало бы все отслеживание изменений бесполезным. Идея в том, чтобы контролировать изменения в ходе разработки программы, ограничивая права на совершение определенных типов изменений на определенных этапах проекта; это позволяет постоянно иметь представление о состоянии общих исходных кодов группы. О наилучшей схеме контроля над изменениями, о которой я когда-либо слышал, мне рассказал мой друг Стив Маньян (Steve Munyan); он называет ее «Зеленый, желтый и красный период». В зеленый период любой разработчик может регистрировать любые измененные файлы в общих исходных кодах группы. Начальные стадии проекта обычно полностью выполняются в зеленом периоде, потому что в это время группа разрабатывает новые функции программы.

Желтый период наступает, когда проект входит в стадию исправления ошибок или приближается к прекращению разработки нового кода. В это время изменять код разрешается только для исправления ошибок. Добавлять к программе новые функции и вносить в нее другие изменения нельзя. Чтобы исправить ошибку, разработчик должен получить разрешение у технического лидера группы или руководителя проекта. Исправляя ошибку, он должен описать свои действия и на что они влияют. При этом каждое исправление ошибки превращается по сути в миниобзор кода. Выполняя такой обзор кода, важно помнить об использовании утилиты различия версий из состава системы управления версиями, чтобы гарантировать, что произошли именно те и только те изменения, которые были запланированы. В некоторых группах, в которых я работал, проект находился в стадии желтого периода с самого начала, потому что группе нравилось проводить обзоры кода, требуемые на этом этапе. Мы несколько смягчали требования и позволяли обращаться за утверждением изменений к любому другому члену группы. Интересно, что из-за постоянных обзоров кода разработчики находили много ошибок еще до регистрации файлов в общих исходных кодах группы.

Красный период начинается, когда вы прекращаете разрабатывать новый код или приближаетесь к важной контрольной точке. На этом этапе все изменения

кода требуют утверждения руководителя проекта. Когда я был руководителем проекта (член группы, ответственный за код в целом), я даже шел на изменение прав доступа к системе управления версиями, разрешая членам группы только чтение информации, но не запись. Я делал это главным образом потому, что знал ход мысли разработчиков: «Это всего лишь небольшое изменение; я исправлю ошибку, и это больше ни на что не повлияет». Несмотря на благие намерения, одно небольшое изменение могло означать, что вся группа должна будет начать план тестирования с нуля.

Руководитель проекта должен строго придерживаться правила красного периода. Если выполнение программы приводит к воспроизводимой критической ошибке или искажению данных, решение об изменении принимается автоматически, потому что оно необходимо. Однако обычно принять решение об исправлении конкретной проблемы не так легко. Чтобы решить, насколько важным является исправление ошибки, я всегда задавал себе следующие вопросы, держа в уме интересы компании:

- скольких людей касается эта проблема?
- затронет изменение ядро или второстепенную часть программы?
- если изменение будет сделано, какие компоненты приложения придется тестировать заново?

Позвольте мне дополнить этот список некоторыми конкретными цифрами и описать общие правила для стадий бета-тестирования. Если проблема серьезна, т. е. приводит к аварийному завершению программы или искажению данных и, вероятно, коснется более 15% наших внешних тестировщиков, решение об ее исправлении принимается автоматически. Если ошибка приводит к изменению файла данных, я также принимаю решение об ее исправлении, чтобы позднее нам не пришлось изменять форматы файлов и чтобы бета-тестировщики могли получить более объемные наборы данных для последующих бета-версий программы.

Важность меток

Команда записи метки — одна из наиболее важных команд при работе с системой управления версиями. В Microsoft Visual SourceSafe она называется меткой (label), в MKS Source Integrity — контрольной точкой (checkpoint), а в PVCS Version Manager — меткой версии (version label). Но, как бы она ни называлась, метка указывает на конкретный набор общих для группы исходных текстов программы. Метка позволяет получить нужную версию исходных кодов программы. Если вы создадите ошибочную метку, возможно, вы никогда не получите исходные коды, использованные для создания конкретной версии программы, и не сможете обнаружить причину ее отказа.

Я всегда помечаю:

1. достижение всех контрольных точек работы над программой;
2. все переходы между зеленым, желтым и красным периодами разработки;
3. все компоновки (builds), отсылаемые за пределы группы;
4. все ветви дерева разработки, создаваемые в системе управления версиями;
5. правильное выполнение ежедневной компоновки программы и дымовых тестов (о них см. ниже одноименный раздел этой главы).

Во всех случаях я следую схеме: <Название проекта> <Контрольная точка/Причина> <Дата>, чтобы названия меток были описательными.

Есть и третье правило записи меток, о котором многие забывают. Сотрудники отдела контроля качества обычно работают с компоновками контрольных точек и ежедневными компоновками, поэтому, сообщая об ошибках, они имеют в виду конкретные версии программы. Разработчики изменяют код довольно быстро, поэтому нужно позаботиться, чтобы можно было легко вернуться к версии файлов, нужных для воспроизведения ошибки.

Стандартный вопрос отладки

Что делать, если мы не можем воспроизвести компоновку, посланную за пределы группы?

Отсылая компоновку за пределы группы, обязательно делайте полную копию ее каталога на CD/DVD или ленточном накопителе. Копия должна включать все исходные и промежуточные файлы программы, файлы символов и окончательный результат. Включайте в нее также пакет для установки, отсылаемый заказчику. Следует даже подумать о создании копии средств компоновки программы. CD/DVD и ленточные накопители — очень недорогой способ страховки от будущих проблем.

Даже когда я делал все для сохранения конкретной компоновки в системе управления версиями, повторное создание программы порой приводило к получению двоичных файлов, отличающихся от первоначальной версии. Имея архив полного дерева компоновки, вы сможете отлаживать программы пользователей при помощи тех же двоичных файлов, которые вы им в свое время послали.

Системы отслеживания ошибок

Система отслеживания ошибок не только накапливает сведения об ошибках, но и является прекрасным средством для хранения разных заметок и списка заданий, особенно на этапе разработки исходного кода. Некоторые программисты любят хранить заметки и списки заданий в мобильных ПК, но при этом важная информация часто теряется среди отладочных файлов со случайными шестнадцатеричными данными и рисунков, выполненных для борьбы со сном во время планерок. Сохранив эти заметки в системе отслеживания ошибок и пометив, что они принадлежат вам, вы консолидируете их в одном месте, что облегчит их поиск. Кроме того, хотя вам, возможно, нравится думать, что код, над которым вы работаете, «принадлежит» вам, на самом деле это не так — он принадлежит группе. Если вы будете хранить свой список заданий в системе отслеживания ошибок, другие члены группы, которым понадобится ваш код, смогут проверить ваш список и узнать, что именно вы сделали. И еще одно преимущество хранения списков заданий и заметок в системе отслеживания ошибок: они будут постоянно напоминать вам, что нужно сделать, поэтому вам не придется лихорадочно отлаживать ошибку в последний момент из-за того, что вы про нее забыли или почему-либо еще. Я использую систему отслеживания ошибок постоянно, чтобы важ-

ные заметки и задания можно было внести в нее сразу, как только они придут мне в голову.

Я люблю назначать заметкам и спискам заданий в системе отслеживания ошибок наименьший приоритет ошибок. Это позволяет отделить их от настоящих ошибок, и в то же время ничто не мешает повысить их приоритет, если надо. При этом следует организовать методику сообщений об ошибках так, чтобы они не включали кодов ошибок с наименьшим приоритетом, иначе можно будет запутаться.

Не бойтесь изучать данные системы отслеживания ошибок: они содержат всю правду о проекте. Планируя модернизацию программы, поработайте с системой отслеживания ошибок и найдите модули или функции, в которых было зарегистрировано наибольшее число проблем. Выделите некоторое время на то, чтобы члены группы лучше поработали над соответствующими разделами программы.

При внедрении системы отслеживания ошибок убедитесь, что к ней имеют доступ все, кому это нужно: как минимум, это члены групп разработки и технической поддержки. Если система отслеживания ошибок позволяет назначить разные уровни доступа, возможно, стоит разрешить соответствующий доступ к ней другим людям, скажем, инженерам по сбыту (технические эксперты, работающие в торговых организациях и оказывающие продавцам помощь при продаже сложной продукции) и сотрудникам отдела маркетинга. Например, некоторым членам отделов продаж и маркетинга можно разрешить регистрировать сообщения об ошибках и запросы о реализации функций, но не получать информацию об обнаруженных ошибках. Представители этих двух групп, как правило, больше общаются с клиентами, чем обычные инженеры, и могут предоставить бесценную обратную связь. Естественным, вы должны обучить их составлению отчетов об ошибках. Они с радостью согласятся помочь, но им нужно дать необходимые указания, чтобы они делали это правильно. Если представители этих двух групп будут оставлять свои запросы и сообщения о проблемах в той же системе, что и другие сотрудники, эффективность работы с ней только повысится. Идея системы отслеживания ошибок как раз в том, чтобы все сообщения об ошибках и запросы о реализации функций находились в одном месте. Если эта информация будет храниться в разных местах — в электронном почтовом ящике руководителя проекта, в записных книжках инженеров и, конечно, в системе отслеживания ошибок — уследить за ней будет гораздо сложнее.

Выбор правильных систем

Система управления версиями должна соответствовать вашим потребностям. Очевидно, если вы работаете в компании с требованиями класса «high-end», такими как поддержка нескольких платформ, вам скорее всего придется выбирать более дорогую систему или использовать решение с открытым кодом, скажем, CVS. Если же вы работаете в небольшой группе и разрабатываете программу только для Windows, можно рассмотреть варианты подешевле. Потратьте некоторое время на тщательную оценку системы, которую вы планируете внедрить, уделив особое внимание прогнозированию будущих потребностей. Вам придется работать с системой управления версиями довольно долго, поэтому убедитесь, что она будет развиваться вместе с вашим проектом. Выбор правильной системы управле-

ния версиями очень важен, но еще важнее, чтобы вы вообще ее использовали: хоть какая-то система управления версиями лучше, чем никакая.

Я знаю массу случаев, когда разработчики пытались использовать собственные системы отслеживания ошибок, однако я настоятельно рекомендую потратить средства на коммерческий продукт или использовать решение с открытым кодом. Информация системы отслеживания ошибок слишком важна, чтобы ее хранение можно было доверить приложению, которое трудно поддерживать и которое не сможет развиваться вместе с вашими потребностями в течение длительного срока. Кроме того, это позволяет избежать траты времени на разработку внутренней системы вместо работы над коммерческой программой.

При выборе системы отслеживания ошибок следует руководствоваться теми же критериями, что и при выборе системы управления версиями. Однажды я, как руководитель проекта, выбрал систему отслеживания ошибок, не уделив должного внимания ее наиболее важной части, составлению отчетов об ошибках. Система была достаточно проста в плане внедрения и использования, однако поддержка отчетов в ней была столь ограничена, что сразу же по достижении первой внешней контрольной точки нам пришлось переносить все наши ошибки на другую систему. Мне было очень неловко перед коллегами за эту оплошность.

Как я уже говорил, очень важно, чтобы система отслеживания ошибок обеспечивала интеграцию с системой управления версиями. Большинство систем управления версиями для Windows поддерживают Интерфейс контроля над исходным кодом Microsoft (Microsoft Source Code Control Interface, MSSCCI). Если ваша система отслеживания ошибок также поддерживает MSSCCI, вы сможете согласовать исправления ошибок с конкретными версиями файлов.

Некоторые люди называют код «кровью» группы разработчиков. Если это так, то системы управления версиями и отслеживания ошибок — артерии, от которых зависит правильное кровообращение. Не работайте без них.

Планирование времени построения систем отладки

Составляя план и расписание проекта, выделите время на создание систем отладки. Вам нужно заранее решить, как вы будете реализовывать обработчики критических ошибок (см. главу 13), средства создания дампов файлов и другие инструменты, которые понадобятся для воспроизведения реальных проблем. Мне всегда нравилось рассматривать системы обработки ошибок так, как если бы они являлись одной из функций программы. Это позволяет другим сотрудникам компании узнать, что вы собираетесь делать с ошибками при их появлении.

Планируя системы отладки, разработайте политику предупредительной отладки. Первые и наиболее сложные этапы этого процесса заключаются в определении способа возвращения ошибок. Какое бы решение вы ни приняли, всегда используйте только один способ. Мне известен один давний проект (к счастью я в нем не участвовал), в котором применялись три способа возвращения ошибок: при помощи возвращаемых функциями значений, при помощи исключений `setjmp/longjmp` и при помощи глобальной переменной, аналогичной переменной `errno` стандартной библиотеки C. Эти разработчики провели немало тяжелых минут, пытаясь отследить ошибки, пересекающие границы различных подсистем.

При разработке приложений для платформы .NET выбрать способ обработки ошибок довольно легко. Вы можете или продолжать использовать возвращаемые значения, или применить исключения. Привлекательность .NET в том, что в отличие от неуправляемого кода в ней есть стандартный класс исключений, `System.Exception`, который является базовым для всех прочих исключений. Механизм исключений .NET имеет и один недостаток: вам все же придется вести подробную документацию и проводить инспекцию кода программы, чтобы точно знать, какое исключение генерируется методом. Как вы увидите по моим программам, для сообщений о нормальном завершении блока программы и ожидаемых ошибках я все же предпочитаю использовать возвращаемые значения, так как это немного быстрее, чем выполнение кода `throw` и `catch`. Однако для всех непредвиденных ошибок я всегда использую исключения.

С другой стороны, при написании неуправляемого кода вы по сути вынуждены применять только возвращаемые значения. Проблема в том, что в C++ нет стандартного класса исключений, генерируемых автоматически, а такие технологии, как COM, не позволяют исключениям пересекать границы адресного пространства отделенного потока или процесса. Как я покажу в главе 13, исключения C++ — одна из самых проблемных в смысле производительности и ошибок областей. Окажите себе большую услугу и забудьте об исключениях в языке C++. С теоретической точки зрения они великолепны, но реальность далеко не всегда соответствует теории.

Создавайте все компоновки с использованием символов отладки

Некоторые из моих советов по поводу систем отладки не вызывают никаких сомнений. Так, я годами твержу о том, что все компоновки, в том числе заключительные (release), нужно создавать, применяя полный набор символов отладки — данные, позволяющие отладчику показывать исходные тексты, номера строк, имена переменных и информацию о типах данных вашей программы. Вся эта информация хранится в файлах с расширением .PDB (Program Database, база данных программы), связанных с конкретными модулями. Разумеется, если вы работаете на условиях почасовой оплаты, нет ничего плохого в том, чтобы проводить все рабочее время за отладкой на уровне ассемблера. Увы, большинство из нас не может позволить себе такой роскоши и поэтому нуждается в средствах быстрого обнаружения ошибок.

Конечно, у отладки заключительных компоновок при помощи символов есть свои минусы. Например, оптимизированный код, создаваемый компилятором по требованию (just-in-time compiler, JIT compiler) или компилятором неуправляемого кода, не всегда соответствует потоку исполнения исходного кода, поэтому работать с заключительным кодом сложнее, чем с отладочным. Другая проблема исследования неуправляемых заключительных компоновок в том, что компилятор иногда оптимизирует регистры стека так, что это не позволяет увидеть полный стек вызовов, как при обычной отладочной компоновке. Кроме того, при включении символов отладки в двоичный файл он слегка увеличивается из-за строки раздела отладки, определяющей файл .PDB. Однако эти несколько байтов — нич-

то в сравнении с тем, насколько символы облегчают и ускоряют исправление ошибок.

В проектах, создаваемых при помощи мастеров (wizard), отладочные символы для заключительных компоновок применяются по умолчанию, но при необходимости это можно сделать и вручную. Если вы работаете над проектом C#, откройте диалоговое окно Property Pages (страницы свойств) (рис. 2-1) и выберите папку Configuration Properties (свойства конфигурации). Щелкните в раскрывающемся списке Configuration (конфигурация) пункт All Configurations (все конфигурации) или Release (заключительная конфигурация); выберите в папке Configuration Properties страницу свойств Build (компоновка программы) и задайте в поле Generate Debugging Information (генерировать отладочную информацию) значение True. Это устанавливает флаг `/debug:full` для компилятора CSC.EXE.

По непонятной мне причине диалоговое окно Property Pages проекта, разрабатываемого в среде Microsoft Visual Basic .NET, отличается от аналогичного окна проекта C#, однако ключ компилятора в обоих случаях один и тот же. Подключение полного набора отладочных символов для заключительных компоновок Visual Basic .NET показано на рис. 2-2. Откройте диалоговое окно проекта Property Pages и выберите папку Configuration Properties. Щелкните в раскрывающемся списке Configuration пункт All Configurations или Release; выберите в папке Configuration Properties страницу свойств Build и установите флажок Generate Debugging Information.

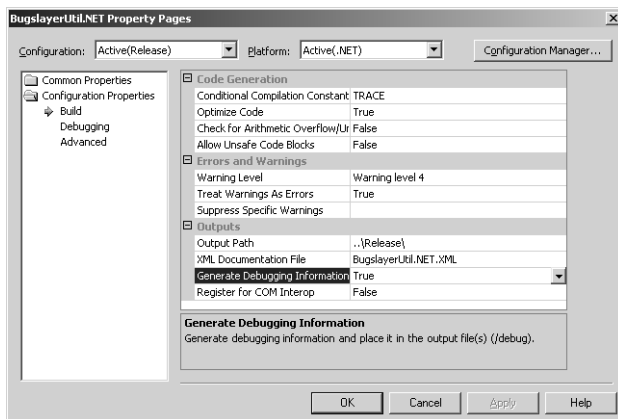


Рис. 2-1. Включение генерирования отладочной информации для проекта C#

Чтобы включить создание PDB-файла для неуправляемой программы C++, нужно задать компилятору ключ `/Zi`. Откройте в окне Property Pages папку C/C++ страницу свойств General (общие свойства) и задайте в поле Debug Information Format (формат отладочной информации) значение Program Database (`/Zi`). Убедитесь, что вы не выбрали пункт Program Database For Edit & Continue (база данных программы для режима «отредактировать и продолжить»), а то заключительная компоновка окажется большой и медленной, так как в нее будет занесена вся дополнительная информация, необходимая для специфического режима отладки, позволяющего внести изменение в программу и продолжить ее выполнение. Правильные параметры компилятора см. на рис. 2-3, где показаны и другие параметры, позволяющие оптимизировать создание компоновок; их я опишу в разделе «Какие допол-

нительные параметры компилятора и компоновщика помогут заранее позаботиться об отладке неуправляемого кода?».

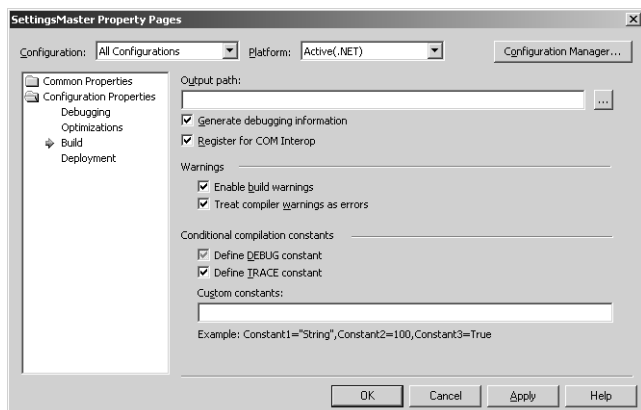


Рис. 2-2. Включение генерирования отладочной информации для проекта Visual Basic .NET

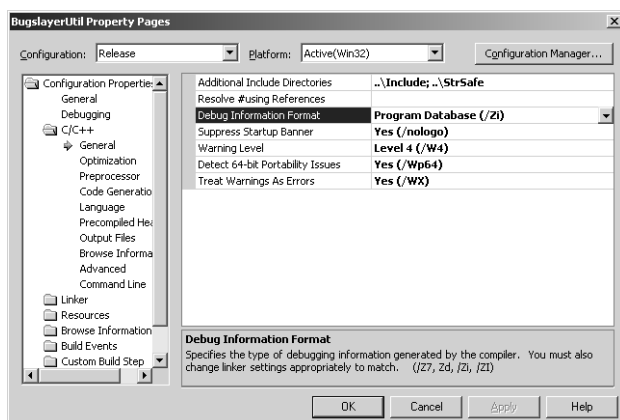


Рис. 2-3. Настройка компилятора C++ для генерирования отладочной информации

После установки ключа компилятора вам понадобится задать соответствующие ключи компоновщика: `/INCREMENTAL:NO`, `/DEBUG` и `/PDB`. Для указания параметров компоновки с приращением нужно открыть окно Property Pages, выбрать папку Linker (компоновщик), страницу свойств General и задать соответствующее значение в поле Enable Incremental Linking (включить компоновку с приращением). Расположение ключа см. на рис. 2-4.

Выберите в окне Property Pages папку Linker, перейдите на страницу Debugging (отладка) и задайте в поле Generate Debug Info (генерировать отладочную информацию) значение Yes (/DEBUG). Чтобы задать ключ `/PDB`, введите в поле Generate Program Database File (генерировать файл базы данных программы), находящееся сразу же под полем Generate Debug Info, значение `$(Каталог_файла)/$(Название_проекта).PDB`. Если вы не заметили, в системе проектов Microsoft Visual Studio

.NET наконец-то решены серьезные проблемы предыдущих версий, связанные с общими ключами компоновки. Значения, начинающиеся с символа \$ и заключенные в скобки, являются макрокомандами, о назначении которых часто можно догадаться по названиям. Об остальных макрокомандах можно узнать, щелкнув на странице свойств почти любое поле ввода и выбрав из списка пункт <Edit...>. Во всплывающем диалоговом окне будут указаны все макрокоманды и во что они преобразуются. Установка ключей /DEBUG и /PDB показана на рис. 2-5. Остальные параметры важны для неуправляемого кода C++. Я опишу их в разделе «Какие дополнительные параметры компилятора и компоновщика помогут заранее позаботиться об отладке неуправляемого кода?».

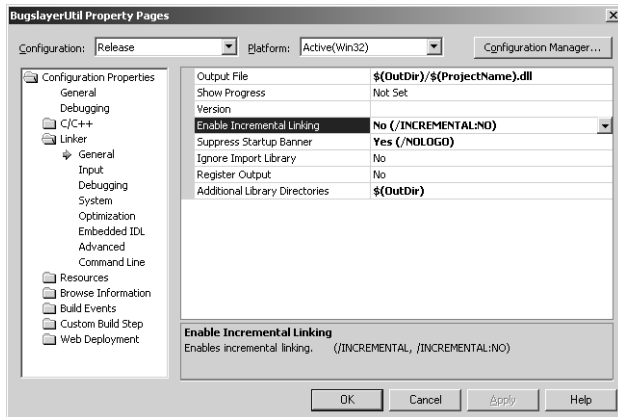


Рис. 2-4. Отключение компоновки с приращением для компоновщика C++

Правильная настройка создания отладочных символов для C++ требует задания еще двух ключей: /OPT:REF и /OPT:ICF. Они находятся в папке Linker на странице Optimization (рис. 2-6). Выберите в разделе References (ссылки) значение Eliminate Unreferenced Data (/OPT:REF) (удалять неиспользуемые данные). В поле Enable COMDAT Folding (удаление избыточных записей COMDAT) выберите Remove Redundant COMDATs (/OPT:ICF) (удалять избыточные записи COMDAT). При установленном ключе /DEBUG компоновщик включает в итоговый файл все функции независимо от того, вызываются они или нет; в случае отладочных компоновок это задано по умолчанию. Ключ /OPT:REF указывает компоновщику включать в итоговый файл только те функции, что вызываются программой. Если вы забудете добавить ключ /OPT:REF, заключительное приложение будет содержать функции, которые никогда не вызываются, что сделает его гораздо более объемным, чем следовало бы. Ключ /OPT:ICF задает комбинирование идентичных записей данных COMDAT, так что для всех ссылок на постоянное значение у вас будет только одна константная переменная.

После создания заключительных компоновок с PDB-файлами, содержащими полную информацию, храните эти файлы в безопасном месте с двоичными файлами, которые вы поставляете заказчику. В случае утраты PDB-файлов вам придется вернуться к отладке на уровне ассемблера. Обращайтесь с ними так же, как с распространяемыми двоичными файлами.

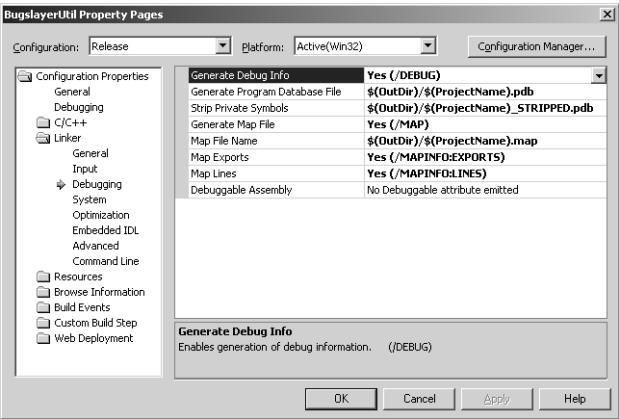


Рис. 2-5. Настройка отладочных параметров компоновщика C++

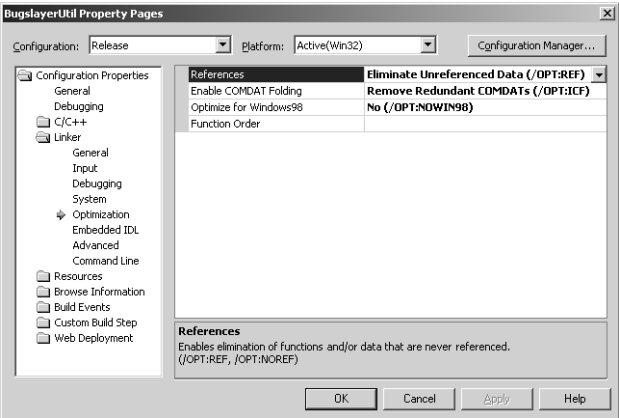


Рис. 2-6. Оптимизация компоновщика C++

Если мысль о ручном изменении параметров проекта с целью его компоновки с символами отладки, а также о правильном указании всех остальных ключей компоновки внушает вам страх, не волнуйтесь, все не так уж и плохо. Для главы 9 я написал очень полезный модуль надстройки SettingsMaster, который возьмет всю работу по изменению параметров проекта на себя. SettingsMaster по умолчанию сконфигурирован так, чтобы задавать все ключи, рекомендуемые в этой главе.

При работе над управляемым кодом рассматривайте предупреждения как ошибки

Если вы писали на управляемом коде что-нибудь более серьезное, чем «Hello World!», вы наверняка заметили, что его компиляторы гораздо строже относятся к ошибкам компилирования. Программисты, привыкшие работать с C++ и не очень хорошо знакомые с .NET, часто удивляются числу дополнительных ограничений этой платформы: например, в C++ вы могли приводить переменные почти к любому типу, и компилятор смотрел на это сквозь пальцы. Компиляторы управляемого кода

не только гарантируют явный тип данных, но и помогут в исправлении ошибок, но для этого их нужно настроить, скажем, сделать инструменты как можно более интеллектуальными.

Если вы откроете документацию к Visual Studio .NET, выберете панель Contents (содержание) и перейдете к разделу Visual Studio .NET\Visual Basic and Visual C#\Reference\Visual C# Language\C# Compiler Options\Compiler Errors CS0001 Through CS9999, вы увидите список всех ошибок компилятора C#. (Ошибки компилятора Visual Basic .NET также включены в документацию, но, к великому удивлению, они не проиндексированы в разделе Contents.) Просматривая список ошибок, вы заметите, что некоторые из них называются предупреждениями (Compiler Warning) и имеют определенный уровень диагностики, например, Compiler Warning (level 4) CS0028. Затем вы обнаружите уровни диагностики от 1 до 4. Генерируя предупреждение, компилятор сообщает, что конкретная конструкция исходного кода правильна с точки зрения синтаксиса, но может быть неверна в данном контексте. В качестве показательного примера можно привести предупреждение CS0183 (The given expression is always of the provided ('type') type [Данное выражение всегда имеет тип ('type')]), проиллюстрированное следующим фрагментом кода:

```
// Генерирует предупреждение CS0183, потому что строка (или, точнее, любой
// тип в .NET) ВСЕГДА имеет базовый тип Object.
public static void Main ( )
{
    String StringOne = " Something pithy. . ." ;
    if ( StringOne is String )    // CS0183
    {
        Console.WriteLine ( StringOne ) ;
    }
}
```

Если компилятор настолько любезен, что сообщает обо всех контекстуальных проблемах подобного рода, разве разумно не обращать на них внимания? Я не люблю называть их предупреждениями, так как на самом деле это ошибки. Если вы когда-нибудь интересовались разработкой компиляторов, особенно синтаксическим анализом, вероятно, вам в голову приходили две мысли: во-первых, что синтаксический анализ очень сложен, и во-вторых, что люди, создающие компиляторы, сделаны из особого теста. (Хорошо это или плохо, решайте сами.) Если разработчики компилятора пошли на то, чтобы включить в него конкретное предупреждение, значит, они хотели сообщить вам нечто очень важное, что, по их мнению, может являться ошибкой. Когда кто-нибудь просит нас помочь найти ошибку, первое, что мы делаем, — проверяем, компилируется ли код без предупреждений. Если это не так, я говорю, что буду рад помочь, но только после того, как будут устранены все предупреждения.

К счастью, Visual Studio .NET по умолчанию создает проекты с подходящим уровнем диагностики, так что вам не понадобится задавать его вручную. Если вы создаете проект C# вручную, присвойте ключу /WARN значение /WARN:4. В создаваемых вручную проектах Visual Basic .NET рассмотрение предупреждений как ошибок по умолчанию отключено, так что включите его.

Уровни диагностики заданы в Visual Studio .NET правильно, однако обращение с предупреждениями как с ошибками по умолчанию отключено. Это неверно. Чистая компиляция кода близка к благочестию, поэтому для компиляторов C# и Visual Basic .NET нужно задать ключ `/WARNASERROR+`. Это не позволит даже начать отладку, пока код не скомпилируется абсолютно чисто. Если вы работаете над проектом C#, откройте окно Property Pages, выберите папку Configuration Properties, страницу Build и задайте в поле Treat Warnings As Errors (считать предупреждения ошибками), расположенном в столбце Errors And Warnings (ошибки и предупреждения), значение True (рис. 2-1). В случае проекта Visual Basic .NET нужно открыть окно Property Pages, папку Configuration Properties, выбрать страницу Build и установить флажок Treat Compiler Warnings As Errors (рис. 2-2).

Если компилятор будет считать предупреждения ошибками, он окажет вам огромную помощь (особенно при работе над проектами C#), прекращая сборку программы при обнаружении таких проблем, как CS0649 [Field 'field' is never assigned to, and will always have its default value 'value' (Полю 'field' никогда не присваивается значение, поэтому оно всегда будет иметь значение 'value', заданное по умолчанию)], которая показывает, что у вас не инициализирован член класса. Однако другие сообщения, такие как CS1573 [Parameter 'parameter' has no matching param tag in XML comment (but other parameters do) (Параметр 'parameter' не имеет соответствующего ему тега в комментарии XML (хотя другие параметры имеют такие теги))], могут быть настолько надоедливыми, что вы захотите отключить обращение с предупреждениями как с ошибками. Не делайте этого!

Сообщение CS1573 выводится, когда вы задаете крайне полезный ключ `/DOC` для создания XML-документации для вашей сборки и не комментируете какой-то использованный параметр. (Я считаю большим преступлением то, что Visual Basic .NET и C++ не поддерживают ключ `/DOC` и документацию XML.) Это самая настоящая ошибка, потому что, если вы создаете документацию XML, кто-нибудь из вашей группы скорее всего будет читать ее, и если вы не опишете все параметры или что-нибудь в этом роде, вы окажете своей группе очень плохую услугу.

Есть одно предупреждение, которое неверно считать ошибкой. Это предупреждение CS1596: [XML documentation not updated during this incremental rebuild; use `/incremental-` to update XML documentation (Во время этой компиляции с приращением документация XML не была обновлена; для ее обновления используйте ключ `/incremental-`)] Документация XML чрезвычайно полезна, но отключение компиляции с приращением очень замедляет создание программы. Отключить эту ошибку невозможно, поэтому эту проблему можно решить, лишь отключив компиляцию с приращением или для отладочных, или для заключительных компонентов. Быстрая компиляция нравится всем, поэтому я отключаю компиляцию с приращением и создаю документацию XML только для заключительных компонентов. Так я обеспечиваю быстроту компиляции и при этом получаю документацию XML, когда она мне нужна.

При работе над неуправляемым кодом рассматривайте предупреждения как ошибки (в большинстве случаев)

По сравнению с управляемым кодом неуправляемый код C++ не только позволяет вам при компиляции выстрелить себе в ногу², но и дает заряженный пистолет со взведенным курком. В C++ предупреждение на самом деле означает, что компилятор делает предположение по поводу намерений программиста. В качестве прекрасного примера можно привести такое предупреждение, как C4244 [‘conversion’ conversion from ‘type1’ to ‘type2’, possible loss of data (Преобразование ‘conversion’ из ‘type1’ в ‘type2’ может привести к потере данных)], которое всегда возникает при преобразовании знакового типа в беззнаковый и наоборот. В данном случае имеется только 50% шансов, что компилятор прочитает ваши мысли и правильно решит, что ему нужно сделать со старшим битом.

Очень часто исправление подобной ошибки тривиально: достаточно, например, выполнить явное приведение типа переменной. Общая идея в том, чтобы сделать код как можно менее неопределенным, чтобы компилятор не был вынужден делать какие-либо предположения. Некоторые из предупреждений просто незаменимы для прояснения кода. Таким является, например, предупреждение C4101 (‘identifier’: unreferenced local variable), сообщающее, что локальная переменная нигде не используется. Исправление этого предупреждения облегчит проведение обзоров кода и сделает программу гораздо понятнее для программистов, которые будут ее сопровождать: никто не будет тратить время на выяснение того, для чего же нужна эта дополнительная переменная и где она используется. Другие предупреждения, такие как C4700 [local variable ‘name’ used without having been initialized (локальная переменная ‘name’ используется, не будучи инициализированной)], указывают на точное место ошибки. Мне известны случаи, когда простое повышение уровня диагностики и исправление появившихся предупреждений приводило к исчезновению ошибок, на поиск которых могли бы уйти недели.

Проекты Visual C++, создаваемые при помощи мастеров, имеют по умолчанию уровень диагностики 3, что соответствует в CL.EXE ключу /W3. Еще выше уровень 4, /W4, а ключ /WX позволяет даже сделать так, чтобы все предупреждения рассматривались компилятором как ошибки. Для задания уровня диагностики откройте окно Property Pages, папку C/C++ и выберите страницу свойств General. В поле Warning Level (уровень диагностики) укажите значение Level 4 (/W4). Двумя строками ниже находится поле Treat Warnings As Errors, в котором следует задать значение Yes (/WX). Правильные значения обоих полей см. на рис. 2-3.

Я с радостью заявил бы, что компиляцию всегда следует выполнять на уровне диагностики 4 и все предупреждения нужно считать ошибками, однако реальность не позволяет мне сделать это. Входящая в состав Visual C++ библиотека стандартных шаблонов (Standard Template Library, STL) имеет много недоработок, не позволяющих работать с ней на уровне диагностики 4. Компилятор также имеет несколько проблем с шаблонами. К счастью, эти проблемы поддаются решению.

² По-английски: «shoot yourself in the foot». Вероятно, автор обыгрывает название известной книги: Allen I. Holub. Enough Rope to Shoot Yourself in the Foot: Rules for C and C++ Programming. — McGraw-Hill, 1995. — *Прим. перев.*

Вы можете подумать, что достаточно задать уровень диагностики 4 и не считать предупреждения ошибками, но такой подход дискредитирует саму суть описанной идеи. Я обнаружил, что разработчики очень быстро перестают обращать внимание на предупреждения в окне Build. Если не исправлять все предупреждения, какими бы безобидными они ни казались, по мере их возникновения, более важные предупреждения начинают теряться в потоке вывода среди других сообщений. Хитрость в том, чтобы более явно указывать, какие предупреждения вы желаете исправлять. Конечно, вы должны избавляться от большинства предупреждений путем улучшения кода программы, однако можно также отключить специфические ошибки, используя директиву `#pragma warning`. Кроме того, она позволяет управлять уровнем диагностики ошибок в конкретных заголовочных файлах.

Хорошим примером уместного понижения уровня диагностики может служить включение заголовочных файлов, которые не компилируются на уровне 4. Понизить уровень диагностики можно через расширенную директиву `#pragma warning`, появившуюся в Visual C++ 6. В следующем фрагменте я понижаю уровень диагностики для включения подозрительного заголовочного файла и сразу же возвращаю ему прежнее значение, чтобы мой код компилировался на уровне 4:

```
#pragma warning ( push , 3 )
#include "IDoNotCompileAtWarning4.h"
#pragma warning ( pop )
```

Директива `#pragma warning` позволяет также запретить отдельные предупреждения. Она полезна, например, когда вы применяете безымянную структуру или объединение и получаете на уровне диагностики 4 ошибку C4201, «nonstandard extension used: nameless struct/union» (использовано нестандартное расширение: структура/объединение не имеет имени). Вот как при помощи директивы `#pragma warning` запретить это предупреждение (заметьте: я закомментировал свои действия и объяснил их). При запрещении отдельных предупреждений ограничивайте диапазон действия `#pragma warning` специфическими разделами программы. Поместив директиву на слишком высоком уровне, вы можете замаскировать другие ошибки своей программы.

```
// Я запрещаю предупреждение "nonstandard extension used: nameless struct/union",
// потому что мне не нужен машино-независимый код
#pragma warning ( disable : 4201 )
struct S
{
    float y;
    struct
    {
        int a ;
        int b ;
        int c ;
    } ;
} *p_s ;
// Снова разрешаю предупреждение.
#pragma warning ( default : 4201 )
```

Существует одно предупреждение, C4100 [«identifier: unreferenced formal parameter» (‘identifier’: неиспользуемый формальный параметр)], исправление которого иногда вызывает недоумение. Если у вас есть параметр, который не применяется, его, пожалуй, следует удалить из определения метода. Однако при написании программы на объектно-ориентированном языке программирования можно выполнить наследование от метода, которому, как потом оказывается, параметр не нужен, но изменять базовый класс нельзя. Вот правильный способ обработки ошибки C4100:

```
// Этот код сгенерирует ошибку C4100:
int ProblemMethod ( int i , int j )
{
    return ( 5 ) ;
}
// Правильный способ избежания ошибки C4100:
int GoodMethod ( int /* i */ , int /* j */ )
{
    return ( 22 ) ;
}
```

Стандартный вопрос отладки

STL, поставляемая с Visual Studio .NET, сложна для понимания и отладки. Что-нибудь может мне помочь?

Я понимаю, что STL из состава Visual Studio .NET писали гораздо более умные люди, чем я, но даже в этом случае ее почти невозможно понять. С одной стороны, концепция STL хороша: эта библиотека широко используется и имеет согласованный интерфейс. С другой стороны, природа STL, поставляемой с Visual Studio .NET, и шаблонов вообще такова, что при возникновении проблемы вам придется приложить гораздо больше усилий для ее понимания, чем для отладки на уровне ассемблера.

Вместо STL из Visual Studio .NET я рекомендую свободно распространяемую STL от компании STLport (www.stlport.org). Библиотека STLport не только бесконечно понятней, но и включает гораздо лучшие средства поддержки многопоточности и отладки. Учитывая эти преимущества и то, что она не налагает никаких ограничений на коммерческое использование, я настоятельно рекомендую использовать именно ее, а не STL из Visual Studio .NET, если, конечно, вам вообще нужна STL.

Если вы не используете STL, этот способ работает прекрасно. Однако при работе с STL он эффективен не всегда. Применяя STL, лучше всего включать в прекомпилированные заголовочные файлы только заголовочные файлы STL. Это значительно облегчает изоляцию директив `#pragma warning (push , 3)` и `#pragma warning (pop)` в заголовочных файлах. Другое важное преимущество заключается в существенном ускорении компиляции. Прекомпилированный заголовочный файл представляет по сути дерево синтаксического анализа, благодаря чему позволяет сэкономить много времени, так как STL — очень объемная библиотека. Наконец, чтобы получить полный контроль над утечками и искажениями памяти при ис-

пользовании стандартной библиотеки C, нужно держать заголовочные файлы STL в одном месте. О стандартной библиотеке C для отладки см. главу 17.

Основной смысл сказанного в том, что с самого начала проекта нужно выполнять компиляцию на уровне диагностики 4 и рассматривать все предупреждения как ошибки. Когда вы впервые повысите уровень диагностики проекта, вы скорее всего будете удивлены числом появившихся предупреждений. Изучите их и исправьте. Возможно, это приведет и к исчезновению нескольких ошибок. Если вы думаете, что заставить программу компилироваться с ключами /W4 и /WX нельзя, я могу доказать обратное: весь неуправляемый код примеров с прилагаемого к этой книге CD компилируется с обоими флагами, заданными для всех конфигураций.

Разрабатывая неуправляемый код, знайте адреса загрузки DLL

Если вы когда-нибудь гуляли по лесу, то знаете: чтобы не заблудиться, очень важно запоминать всякие примечательные объекты. Не имея ориентиров, можно просто ходить по кругу. При аварийном завершении приложения нужно иметь аналогичный ориентир, который поможет найти правильный путь и не блуждать впустую по коду своей программы в окне отладчика.

Первым важным ориентиром при крахе программы являются базовые адреса DLL или элементов управления на базе ActiveX (OCX), указывающие на область их размещения в памяти. Когда клиент предоставит вам адрес аварийного завершения программы, вам нужно быстро определить, к какой DLL он относится, по первым двум или трем цифрам. Я не утверждаю, что вы должны знать адреса всех системных DLL, но нужно помнить хотя бы базовые адреса DLL, используемых в вашем проекте.

Если все ваши DLL будут загружаться по уникальным адресам, вы будете иметь отличные ориентиры, которые помогут искать причину проблемы. Ну, а если все ваши DLL будут иметь одинаковые адреса загрузки? Очевидно, ОС не сможет отобразить их на одну и ту же область памяти. При загрузке DLL, желающей расположиться в уже занятой области памяти, ОС должна будет «переехать» DLL, выделив ей другое место. И как же определить, где какая DLL загружена? Увы, мы не можем узнать, как поступит ОС на разных компьютерах. А значит, при получении адреса аварийного завершения программы вы не будете иметь представления о том, откуда этот адрес взялся. В свою очередь это означает, что ваш начальник будет очень недоволен, так как вы не сможете объяснить ему причину сбоя приложения.

Для проектов, созданных при помощи мастеров, по умолчанию справедливо следующее: библиотеки DLL элементов ActiveX, созданных в среде Visual Basic 6, загружаются по адресу 0x11000000, а DLL, написанные на Visual C++, — по адресу 0x10000000. Готов спорить, что по меньшей мере половина имеющихся на данный момент в мире DLL пытается загрузиться по одному из этих адресов. Изменение адреса загрузки DLL называется модификацией базового адреса (или переездацией) и является простой операцией, позволяющей задать другой адрес загрузки, отличный от используемого по умолчанию.

Прежде чем приступить к обсуждению модификации базового адреса, рассмотрим два простых способа, позволяющих определить наличие конфликтов при

загрузке DLL. Первый подразумевает использование окна Modules (модули) отладчика Visual Studio .NET. Запустите приложение в среде Visual Studio .NET и откройте окно Modules, для чего нужно выбрать меню Debug, подменю Windows или нажать CTRL+ALT+U, если комбинации клавиш настроены по умолчанию. Если базовый адрес модуля был модифицирован, его значок будет отмечен красным кружком с восклицательным знаком. Кроме того, диапазон занимаемых модулем адресов будет отмечен звездочкой. На рис. 2-7 показано окно Modules с переадресованной библиотекой SYMSRV.DLL во время сеанса отладки.

Modules							
Name	Address	Path	Order	Version	Program	Timestamp	
DHPOK32.dll	10000000-10010000	E:\Program Files\Panicware\Pop-Up...	22		[1992] dp...	4/29/2002...	
Sxgb.dll	69F00000-69F14000	E:\WINNT\system32\Sxgb.dll	23	3.00.3.5	[1992] dp...	10/13/2000...	
psapi.dll	690A0000-690A8000	E:\WINNT\system32\psapi.dll	24	5.00.2134.1	[1992] dp...	11/30/1999...	
exgbsys.dll	00800000-00809000*	E:\WINNT\system32\exgbsys.dll	25	3.00.0.5	[1992] dp...	1/19/2000...	
PANICNT.DLL	01080000-0108000*	E:\WINNT\PANICNT.dll	26		[1992] dp...	9/16/2001...	
MSCTF.DLL	60000000-600A5000	E:\WINNT\system32\MSCTF.DLL	27	5.01.2409.7	[1992] dp...	2/20/2001...	
Msh_zwf.dll	61220000-61232000	E:\Program Files\Microsoft Hardware...	28	4.10.851.0	[1992] dp...	4/11/2002...	

Рис. 2-7. Переадресованная DLL в окне Modules отладчика Visual Studio .NET

Второй способ — загрузить бесплатное приложение Process Explorer, написанное моим хорошим другом и когда-то соседом Марком Руссиновичем (Mark Russinovich) из Sysinternals (www.sysinternals.com). Как следует из названия, Process Explorer позволяет узнать разнообразную информацию о процессах, например, загруженные DLL и открытые дескрипторы (handle). Это настолько полезный инструмент, что если у вас его еще нет, немедленно прекратите чтение и загрузите его! Кроме того, вам следует прочитать главу 14, где описаны дополнительные приемы и хитрости, которые могут облегчить отладку при помощи Process Explorer.

Узнать, была ли переадресована DLL, очень легко. Просто выполните описанные ниже действия. На рис. 2-8 показано, как выглядит окно Process Explorer, если DLL процесса была переадресована.

1. Запустите Process Explorer и свой процесс.
 2. Выберите в меню View пункт View DLLs.
 3. Выберите в меню Options пункт Highlight Relocated DLLs (Выделить переадресованные DLL).
 4. Выберите свой процесс в верхней половине основного окна.
- Все переадресованные DLL будут выделены желтым цветом.

The screenshot shows the Windows Task Manager interface. The 'Processes' tab is active, displaying a list of running processes. The 'popUp.exe' process is highlighted in blue. Below the main list, a detailed view of the 'popUp.exe' process is shown, including its properties, command line, and open files.

Process	PID	PPID	CPU	Description	Private	Session	Handles	Window Title
POINT32.Drv	612	0	0	Cursor features dr...	ARISTOTLE.vgho	0	20	
PopUp.exe	694	0	0		ARISTOTLE.vgho	0	01	PopUp.exe
Narvaez.exe	620	0	0	Norton AntiVirus A...	ARISTOTLE.vgho	0	193	
Winlogon.exe	1156	0	0	Windows Task Manag...	ARISTOTLE.vgho	0	71	
NMData.exe	1272	0	0	Compuware NtMLM	ARISTOTLE.vgho	0	01	
4NT.Drv	1096	0	0	4NT	ARISTOTLE.vgho	0	51	4 Command Prompt
process.exe	3080	0	0	Symantec Proce...	ARISTOTLE.vgho	0	58	
cmd.exe	2020	0	0	Powershell	ARISTOTLE.vgho	0	57	

Base	Size	MM	Description	Version	Time	Path
0x310000	0x4000	-		1	45.0000.0000	8/23, E:\WINNT\System32\user32.dll
0x400000	0x58000	-		1	11.0.0.0000.0000	11/3, E:\Program Files\PopUp Killer\PopUpKiller.Drv
0x500000	0x3000	-		1	1.05.2600.1105	8/23, E:\Program Files\PopUp Killer\PopUpKiller.Drv
0x600000	0x40000	-	Microsoft Text Frame Work Service IME	11.1.1	11.1.1	E:\WINNT\System32\IME\IME00000001\IME00000001.dll
0x700000	0x40000	-		11.1.1	E:\Memph\DF5AD01.tmp	
0x800000	0x40000	-		11.1.1	E:\Memph\DF5AD01.tmp	
0x900000	0x7000	-		11.1.1	E:\WINNT\System32\RegAsst\RegAsst00000001\IME00000001.dll	
0xA00000	0x6000	-		1.00.00.0004	9/16, E:\Program Files\PopUp Killer\PopUpKiller.Drv	
0xB00000	0x13000	-		1.00.00.0018	8/23, E:\WINNT\System32\PopUpKiller.Drv	
0xC00000	0x4000	-		1.00.00.0028	8/23, E:\WINNT\System32\PopUpKiller.Drv	
0xD00000	0x30000	-	Windows Common Controls ActiveX Control DLL	6.00.0000.0002	5/22, E:\WINNT\System32\COMMONCTL\COMMONCTL.DLL	
0xE00000	0x4000	-	Microsoft Excel 3.50 for Windows NT[TM] and	3.50.5014.0000	8/23, E:\WINNT\System32\excel2.dll	
0xF00000	0x4000	-	CMUser ActiveX Control DLL	6.00.0000.0018	8/23, E:\WINNT\System32\CMUser.DLL	

Рис. 2-8. Переадресованные DLL в окне программы Process Explorer

Еще один отличный инструмент, показывающий переадресованные DLL не только с модифицированным, но и с исходным адресом, — программа ProcessSpy из прекрасной статьи Кристофа Назарра (Christophe Nasarre) «Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities, Part 2» (Избавление от ада DLL при помощи собственных отладочных инструментов и утилит, часть 2), опубликованной в журнале MSDN Magazine в августе 2002 года. По функциональности программы Process Explorer и ProcessSpy похожи, однако ProcessSpy поставляется с исходным кодом, так что вы можете узнать, как она колдует.

Переадресация DLL ОС не только затрудняет поиск причин краха приложения, но и замедляет его выполнение. При переадресации ОС должна прочитать информацию о модификации адресов, проработать все участки программы, получающие доступ к DLL, и изменить их, потому что DLL будет размещена в памяти не на своем излюбленном месте. Если в приложении будет два конфликта адресов загрузки, время его запуска может увеличиться аж вдвое!

Есть и еще одна крупная проблема: переадресовав модуль, ОС не сможет выгрузить его из памяти полностью, если ей понадобится выделить место для другого кода. Если модуль загружается по предпочитаемому адресу загрузки, ОС может выгрузить его на диск, а затем загрузить обратно. Однако, если базовый адрес модуля был модифицирован, значит, была изменена и область памяти, содержащая код этого модуля. Поэтому ОС должна где-то хранить эту память (возможно, в страничном файле), даже если модуль выгружен из памяти. Легко догадаться, что это может «съедать» большие блоки памяти и замедлять работу компьютера из-за затрат на их перемещение.

Базовый адрес DLL можно модифицировать двумя способами. Первый — с помощью утилиты REBASE.EXE из состава Visual Studio .NET. REBASE.EXE имеет массу опций, но лучше всего вызывать ее из командной строки с ключом /b, указывая после него стартовый базовый адрес и названия DLL. Хочу вас обрадовать: как только вы модифицируете базовый адрес какой-либо DLL, вам почти никогда больше не придется возвращаться к ней. Модифицируйте базовые адреса DLL только до ее регистрации. Если вы модифицируете базовый адрес DLL после ее регистрации, она не загрузится.

В табл. 2-1 приведен фрагмент документации Visual Studio .NET, посвященный модификации базовых адресов DLL. Как видите, рекомендуется использовать алфавитную схему. Я обычно следую ей, потому что она проста. DLL ОС загружаются по адресам от 0x70000000 до 0x78000000, так что следование правилам табл. 2-1 избавит вас от конфликтов с ОС. Конечно, вам всегда следует изучать адресное пространство своих приложений при помощи Process Explorer или ProcessSpy, чтобы узнать, не загружена ли уже какая-нибудь DLL по тому адресу, который вы хотите использовать.

Если в приложение включены четыре DLL — APPLE.DLL, DUMPLING.DLL, GINGER.DLL и GOOSEBERRIES.DLL, для правильной модификации их адресов нужно выполнить REBASE.EXE трижды. Это проиллюстрировано следующими тремя командами:

```
REBASE /b 0x60000000 APPLE.DLL
REBASE /b 0x61000000 DUMPLING.DLL
REBASE /b 0x62000000 GINGER.DLL GOOSEBERRIES.DLL
```

Если в командной строке указать несколько DLL, как я только что поступил с библиотеками GINGER.DLL и GOOSEBERRIES.DLL, утилита REBASE.EXE модифицирует их базовые адреса так, чтобы они загружались друг за другом, начиная с указанного адреса.

Табл. 2-1. Схема модификации базовых адресов DLL

Первая буква названия DLL	Базовый адрес
A–C	0x60000000
D–F	0x61000000
G–I	0x62000000
J–L	0x63000000
M–O	0x64000000
P–R	0x65000000
S–U	0x66000000
V–X	0x67000000
Y–Z	0x68000000

Другой метод модификации базового адреса DLL — указать адрес загрузки при компоновке DLL. В Visual C++ это можно сделать, открыв окно Property Pages, папку Linker и выбрав страницу свойств Advanced (расширенные настройки). Шестнадцатеричный адрес загрузки DLL следует указать в поле Base Address (базовый адрес). Этот адрес будет передан компоновщику LINK.EXE вместе с ключом /BASE (рис. 2-9).

Утилиту REBASE.EXE позволяет автоматически задавать адреса загрузки нескольких DLL одновременно без ограничений, но при задании адресов во время компоновки следует быть внимательнее. Если вы укажете адреса загрузки нескольких DLL слишком близко, то в отладочном окне Module увидите, что их адреса будут модифицированы. Поэтому, чтобы никогда впоследствии не волноваться об адресах загрузки, их нужно задавать с достаточным интервалом.

В примере с REBASE.EXE я задал бы адреса загрузки этих DLL так:

```
APPLE.DLL      0x60000000
DUMPLING.DLL   0x61000000
GINGER.DLL     0x62000000
GOOSEBERRIES.DLL 0x62100000
```

Обратите особое внимание на библиотеки GINGER.DLL и GOOSEBERRIES.DLL, потому что их названия начинаются с одинаковой буквы. В таких случаях я задаю другой адрес загрузки при помощи третьей по старшинству цифры. Если бы я собрался использовать еще одну DLL, название которой также начиналось бы с буквы «G», я бы указал адрес загрузки 0x62200000.

Ознакомиться с проектом, в котором адреса загрузки заданы вручную, можно на примере проекта WDBG из главы 4. Я забыл сказать, что ключ /BASE позволяет указать текстовый файл, содержащий адреса загрузки всех DLL приложения. В проекте WDBG я применил именно такой способ.

Для переадресации DLL и ОСХ можно использовать оба метода: модифицировать базовые адреса DLL при помощи утилиты REBASE.EXE или вручную, однако, пожалуй, лучше всего следовать второму методу и выполнять переадресацию DLL вручную. Все примеры DLL на CD, прилагаемом к этой книге, я переадресовал

вручную. Основное преимущество такого метода в том, что MAP-файл будет содержать специфический заданный адрес. MAP-файл — это текстовый файл, указывающий, по каким адресам компоновщик размещает все символы и строки программы. При заключительных компоновках MAP-файлы следует создавать всегда, так как они являются единственными простыми текстовыми описаниями символов. MAP-файлы окажутся особенно полезными в будущем, когда вам нужно будет найти причину краха программы, а ваш отладчик не сможет работать со старыми символами. Если же переадресацию DLL выполнять посредством REBASE.EXE, создаваемый компоновщиком MAP-файл будет содержать первоначальный базовый адрес, и для его преобразования в модифицированный адрес понадобятся некоторые вычисления (о MAP-файлах см. главу 12).

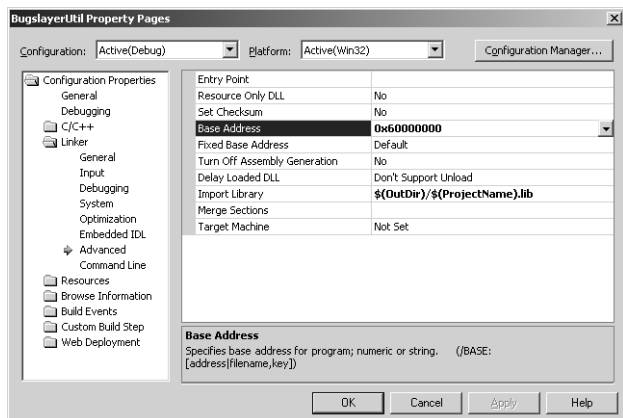


Рис. 2-9. Задание базового адреса DLL

Меня часто спрашивают: «Базовые адреса каких файлов модифицировать?» Следуйте простому правилу: если код написан вами или кем-нибудь из вашей группы, модифицируйте его базовый адрес. В противном случае не трогайте его. Если вы используете компоненты сторонних фирм, вам придется располагать свои двоичные файлы в памяти, учитывая уже занятые этими компонентами области.

Как поступать с базовыми адресами управляемых модулей?

В данный момент вы, возможно, думаете, что, раз уж управляемые компоненты компилируются в DLL, их базовые адреса также следует модифицировать. Более того, если вы изучали ключи компиляторов C# и Visual Basic .NET, то, может быть, видели ключ `/BASEADDRESS` для задания базового адреса. Однако в случае управляемого кода все немного не так. Если вы изучите управляемую DLL с помощью программы DUMPBIN.EXE из состава Visual Studio .NET, служащей для просмотра дампов файлов Portable Executable (PE), или при помощи великолепного инструмента PEDUMP, созданного Мэттом Питреком (Matt Pietrek) (MSDN Magazine, февраль 2002), вы заметите одну импортируемую функцию `_CorDllMain` из библиотеки MSCOREE.DLL и одно значение в таблице переадресации.

Думая, что в управляемых DLL может находиться какой-то исполняемый код, я дизассемблировал несколько DLL, однако в разделе кода модуля все выглядело, как данные. Я еще немного почесал голову и заметил кое-что очень интересное. Точ-

ка входа модуля, т. е. точка, с которой начинается его выполнение, оказалась расположенной по тому же адресу, что и импортируемая функция `_CorDllMain`. Это подтвердило, что в модуле нет неуправляемого исполняемого кода.

В конечном счете модификация базовых адресов управляемых модулей не принесет вам такого же огромного преимущества, как в случае неуправляемого кода. Тем не менее я выполняю ее, так как мне кажется, что загрузчик ОС все-таки не остается в стороне, вследствие чего переадресация управляемой DLL при загрузке будет замедлять запуск программы. Если вы решаете модифицировать базовые адреса управляемых DLL, это нужно делать во время компоновки. Если модифицировать адрес зарегистрированной управляемой DLL при помощи `REBASE.EXE`, система безопасности заметит, что DLL была изменена, и откажется загружать ее.

Стандартный вопрос отладки

Какие дополнительные параметры компилятора C# помогут мне заранее позаботиться об отладке управляемого кода?

Хотя управляемый код устраняет многие ошибки, отравлявшие нашу жизнь при работе с неуправляемым кодом, некоторые ошибки все же могут сказаться на работе вашей программы. К счастью, есть очень полезные ключи командной строки, задав которые можно облегчить обнаружение таких ошибок. Хорошая новость для любителей Visual Basic .NET: эта среда абсолютно правильно настроена по умолчанию, поэтому вам не понадобится задавать дополнительных ключей компилятора. Если вы не желаете настраивать компилятор вручную, модуль надстройки `SettingsMaster` из главы 9 сделает это за вас.

`/checked+`

(проверка целочисленной арифметики)

В областях потенциальных проблем можно использовать ключевое слово `checked`, но это нужно делать при написании кода. Ключ командной строки `/checked+` позволяет включить проверку целочисленного переполнения для всей программы. Если результат окажется вне диапазона допустимых значений типа данных, программа автоматически сгенерирует исключение периода выполнения. Задание этого ключа приводит к небольшому увеличению объема кода, поэтому я предпочитаю оставлять его включенным в отладочных компоновках и использовать ключевое слово `checked` для явной проверки подобных ошибок в заключительных компоновках. Для установки этого ключа нужно открыть окно `Property Pages`, папку `Configuration Properties`, выбрать страницу `Build` и задать в поле `Check For Arithmetic Overflow/ Underflow` (Проверка арифметического переполнения) значение `True`.

`/noconfig`

(игнорировать файл `CSC.RSP`)

Интересно, но задать этот ключ в среде Visual Studio .NET невозможно. Тем не менее, если вы захотите собирать программу из командной строки, знать о его предназначении не помешает. По умолчанию, прежде чем обрабаты-

см. след. стр.

вать командную строку, компилятор C# читает файл CSC.RSP, в котором также указаны ключи командной строки. Чтобы автоматизировать свою работу, вы можете задать в нем любые допустимые ключи. Стандартный файл CSC.RSP из состава Visual Studio .NET содержит огромное число ключей /REFERENCE для распространенных сборок, которые все мы постоянно используем. А вот для таких библиотек, как System.XML.dll, этот ключ не нужен, так как файл CSC.RSP содержит запись /r: System.XML.dll. Файл CSC.RSP находится в каталоге версии .NET Framework: <Название каталога Windows>\Microsoft.NET\Framework\<Номер версии .NET Framework>.

Стандартный вопрос отладки

Какие дополнительные параметры компилятора и компоновщика помогут позаботиться об отладке неуправляемого кода?

Существует много ключей, способных помочь повысить производительность приложения и облегчить его отладку. Кроме того, как я уже говорил, я не совсем согласен со значениями параметров компилятора и компоновщика Visual C++ по умолчанию в проектах, создаваемых при помощи мастеров. Поэтому я всегда изменяю некоторые их параметры. Если вы не желаете делать это вручную, используйте модуль надстройки SettingsMaster из главы 9.

Ключи компилятора CL.EXE

Задать эти ключи вручную можно, открыв окно Property Pages, папку C/C++, страницу Command Line (командная строка) и введя их в поле Additional Options (дополнительные ключи), однако гораздо лучше указывать их в соответствующих им местах. Задание ключей командной строки в поле Additional Options может привести к проблемам, потому что разработчики не привыкли искать их в этом месте.

/EP /P *(препроцессорная обработка с выводом в файл)*

В случае проблем с макрокомандами могут пригодиться ключи /EP и /P. Они приказывают препроцессору обработать исходный файл, преобразовав все макрокоманды в обычную форму и включив все указанные файлы, и сохранить результат в файле с тем же именем, но с расширением .I. Открыв этот файл, вы сможете узнать, во что преобразуются ваши макрокоманды. Убедитесь, что у вас хватает места на диске, потому что файлы .I могут занимать по несколько мегабайт. Чтобы препроцессор сохранил в файле комментарии, нужно также указать ключ /C (не удалять комментарии).

Для задания ключей /EP и /P откройте окно Property Pages, папку C/C++, выберите страницу Preprocessor (препроцессор) и укажите в поле Generate Preprocessed File (генерировать файл, прошедший препроцессорную обработку) значение Without Line Numbers (/EP /P) (без номеров строк). Поле Keep Comments (сохранять комментарии), расположенное на той же странице, позволяет задать компилятору ключ /C. Помните, что эти ключи не

вызывают компиляцию файла .I, поэтому при компоновке программы вы столкнетесь с ошибками. Определив проблему, отключайте их. Знаю на собственном опыте: регистрация проекта в системе управления версиями с заданными ключами /EP и /P не понравится ни вашим товарищам по группе, ни руководителю.

/X (игнорировать стандартный путь включения файлов)

Создание правильной компоновки может оказаться проблематичным, если на компьютере установлены несколько компиляторов и пакетов для разработки ПО (SDK). Если не задан ключ /X, компилятор, вызываемый MAK-файлом, вызовет переменную среды INCLUDE. Ключ /X позволяет контролировать включение заголовочных файлов: он заставляет компилятор игнорировать переменную INCLUDE и искать заголовочные файлы только в местах, указанных явно посредством ключа /I. Задать ключ /X можно, открыв окно Property Pages, папку C/C++, страницу Preprocessor и выбрав соответствующее значение в поле Ignore Standard Include Path (игнорировать стандартный путь включения файлов).

/Zp (выравнивание членов структур)

Этот флаг использовать не следует. Выравнивание членов структур в памяти надо задавать не в командной строке, а в директиве `#pragma pack` в специфических заголовочных файлах. Невыполнение этого условия порой приводит к очень трудноуловимым ошибкам. Начиная проект, разработчики задавали ключ /Zp. Когда они переходили к другой компоновке или если работу над кодом продолжала другая группа, про ключ /Zp забывали, и структуры начинали немного отличаться, так как по умолчанию применялся иной метод выравнивания. На поиск причины тех ошибок пришлось потратить кучу времени. Для установки этого ключа нужно открыть окно Property Pages, папку C/C++, выбрать страницу Code Generation (генерирование кода) и задать нужное значение свойства Struct Member Alignment (выравнивание членов структур).

Используя директиву `#pragma pack`, не забывайте про ее новый вариант `#pragma pack (show)`, выводящий при компиляции значение выравнивания в окно Build. Это поможет вам следить за текущим выравниванием в различных разделах кода.

/Wp64 (определять проблемы совместимости с 64-разрядными платформами)

Этот ключ позволяет сэкономить много времени при работе над совместимостью кода с 64-разрядными системами. Установить его можно, открыв окно Property Pages, папку C/C++, выбрав страницу General и задав в поле Detect 64-bit Portability Issues (определять проблемы совместимости с 64-разрядными платформами) значение Yes (/Wp64). Лучше всего /Wp64 применять с самого начала проекта. Если вы зададите этот ключ, уже проделав значительную работу над программой, то вас поразит количество обнаруженных проблем, так как он предъявляет очень высокие требования. Кро-

ме того, некоторые поставляемые Microsoft макрокоманды, которые, как предполагалось, помогут решить вопросы совместимости с платформами Win64, например `SetWindowLongPtr`, при компиляции с ключом `/Wp64` приводят к выводу сообщений об ошибке.

/RTC (проверка ошибок в период выполнения)

Самые полезные ключи, известные сообществу программистов на C++! Всего их три: `/RTCс` обеспечивает проверку потери данных при их преобразовании в меньший тип, `/RTCu` помогает предотвращать использование неинициализированных переменных, `/RTCf` проверяет кадры стека путем инициализации всех локальных переменных известным значением (`0x00`), предотвращает применение недопустимых индексов локальных переменных и проверяет правильность указателей стека для предотвращения искажения данных. Для установки этих ключей откройте окно Property Pages, папку C/C++, страницу Code Generation и выберите соответствующие значения в полях Smaller Type Check (проверка при преобразовании к меньшему типу) и Basic Runtime Checks (базовые виды проверки периода выполнения). Эти ключи настолько важны, что в главе 17 мы обсудим их особо.

/GS (проверка безопасности буферов)

Один из наиболее распространенных приемов в арсенале создателей вирусов — переполнение буфера, при котором адрес возврата перезаписывается так, чтобы управление получал код злоумышленника. К счастью, ключ `/GS` позволяет включить в программу специальные фрагменты, гарантирующие, что адрес возврата не был перезаписан. Это значительно затрудняет создание вирусов такого типа. Ключ `/GS` задан по умолчанию для заключительных компоновок, и я также советую использовать его в отладочных компоновках. Если когда-нибудь этот ключ сообщит, что кто-то перезаписал только адрес возврата, вы увидите, как много недель ужасно сложной отладки это вам сэкономит. Установите ключ `/GS`, открыв окно Property Pages, папку C/C++, страницу Code Generation и задав в поле Buffer Security Check (проверка безопасности буферов) значение Yes (`/GS`). В главе 17 я объясню, как изменять принятые по умолчанию сообщения об ошибках, обнаруженных ключом `/GS`.

/O1 (минимизировать размер кода)

В проектах C++, создаваемых мастерами, для заключительных компоновок по умолчанию применяется ключ `/O2` (максимизировать скорость). Однако Microsoft создает все свои коммерческие приложения с ключом `/O1`, и вам также следует делать это. Задать этот ключ можно, открыв окно Property Pages, папку C/C++, страницу Optimization и выбрав соответствующее значение свойства Optimization. Программисты Microsoft обнаружили, что после нахождения наилучшего алгоритма и написания компактного кода скорость выполнения приложения можно значительно повысить, уменьшив число ошибок страниц памяти. Как я слышал, они говорят: «Ошибки страниц могут испортить вам весь день!»

Страница представляет собой наименьший блок кода или данных (4 кб для компьютеров с архитектурой x86), с которым диспетчер памяти может работать как с единым целым. Ошибка страницы происходит при обращении к недействительной странице памяти. Это может быть обусловлено самыми разными причинами: например, попыткой получения доступа к странице из списка резервных или измененных страниц или к странице, которая больше не находится в памяти. Для исправления ошибки страницы ОС должна прекратить выполнение программы и загрузить в регистры процессора новый адрес страницы. Если ошибка страницы «мягкая» (т. е. страница уже находится в памяти), накладные расходы не очень велики, тем не менее они все равно лишние. Однако если ошибка «жесткая», ОС вынуждена загрузить в память нужную страницу с диска. Разумеется, это требует выполнения сотен тысяч команд, замедляя работу приложения. Минимизация объема двоичного файла позволяет уменьшить общее число используемых приложением страниц, а значит, и снизить вероятность ошибок страницы. Пусть загрузчик и диспетчер управления кэш-памятью ОС очень хороши, но зачем допускать больше ошибок страниц, если есть возможность уменьшить их число?

Кроме задания ключа `/O1`, рекомендую подумать об утилите Smooth Working Set (SWS) из главы 19, которая помогает вынести наиболее часто вызываемые функции в начало двоичного файла, минимизировав таким образом рабочий набор, т. е. число страниц, находящихся в оперативной памяти. Если часто используемые функции расположены в начале файла, ОС сможет выгрузить ненужные страницы на диск. Это позволит ускорить выполнение приложения.

`/GL` **(оптимизация всей программы)**

Программисты Microsoft много сделали для улучшения генераторов кода, благодаря чему компактность и скорость выполнения программ, создаваемых в среде Visual C++ .NET, заметно улучшились. Одно из крупных изменений состоит в том, что вместо оптимизации отдельных файлов (известных также как компилянды) при компиляции теперь можно выполнять кросс-файловую оптимизацию программы при ее компоновке. Я уверен, что все программисты, впервые компилирующие проект C++ в среде Visual C++ .NET, замечают серьезное уменьшение объема программы. Удивительно, но для заключительных компоновок Visual C++ этот ключ по умолчанию не используется. Установите его: откройте окно Property Pages, папку Configuration Properties, страницу General и задайте в поле Whole Program Optimizations (оптимизация всей программы) значение Yes. Это одновременно установит и соответствующий ключ компоновщика, `/LTCG`.

`/showIncludes` **(выводить список включаемых файлов)**

О назначении этого ключа говорит само название. При компиляции файла он составляет иерархический список всех включаемых файлов, позволяющий узнать, что, куда и откуда включается. Задайте этот ключ, открыв окно

см. след. стр.

Property Pages, папку C/C++, страницу Advanced и указав в поле Show Includes (показывать включаемые файлы) значение Yes (/showIncludes).

Ключи для компоновщика LINK.EXE

Задать эти ключи вручную можно, открыв окно Property Pages, папку Linker, страницу Command Line и введя их в текстовом поле Additional Options, однако гораздо лучше указывать их в соответствующих им местах. Как я уже писал в разделе, посвященном ключам компилятора, программисты не привыкли искать ключи командной строки в текстовом поле Additional Options, так что это может привести к проблемам.

<code>/MAP</code>	<i>(генерировать MAP-файл)</i>
<code>/MAPINFO: LINES</code>	<i>(включать в MAP-файл номера строк)</i>
<code>/MAPINFO: EXPORTS</code>	<i>(включать в MAP-файл информацию об экспортируемых функциях)</i>

Эти ключи обеспечивают создание MAP-файла для компонуемого образа программы (о MAP-файлах см. главу 12). Я советую всегда создавать MAP-файл, так как это единственный способ получения информации о символах в текстовом виде. Используйте все три ключа, чтобы MAP-файл содержал наиболее полную информацию. Задать их можно, открыв окно Property Pages, папку Linker и выбрав нужные значения на странице Debugging.

<code>/NODEFAULTLIB</code>	<i>(игнорировать библиотеки)</i>
----------------------------	---

Многие системные заголовочные файлы включают директивы `#pragma comment (lib#, XXX)`, определяющие, с какой библиотекой компоновать файл, где XXX — название библиотеки. Ключ `/NODEFAULTLIB` указывает компоновщику игнорировать эти директивы. Данный ключ позволяет программисту самому выбирать компонуемые библиотеки и порядок компоновки. Вам придется указывать все нужные библиотеки в командной строке компоновщика, но вы хотя бы будете точно знать, какие библиотеки вы используете и в каком порядке. Управление порядком компоновки может оказаться очень важным, когда один символ встречается в нескольких библиотеках, что может приводить к трудноуловимым ошибкам. Задать этот ключ можно, открыв окно Property Pages, папку Linker, страницу Input (ввод) и указав в поле Ignore All Default Libraries (игнорировать все библиотеки, используемые по умолчанию) значение Yes.

<code>/OPT: NOWIN98</code>

Если от вашей программы не требуется поддержка ОС Windows 9x/Me, этот ключ позволит немного уменьшить размер исполняемых файлов, сняв ограничение, требующее, чтобы их разделы выравнивались по границе 4 кб. Для установки этого ключа нужно открыть окно Property Pages, папку Linker, страницу Optimization и задать нужное значение в поле Optimize For Windows98 (оптимизировать программу для ОС Windows98).

/ORDER**(располагать функции в определенном порядке)**

Если вы собираетесь применять утилиту Smooth Working Set (см. главу 19), ключ **/ORDER** позволит указать файл, описывающий порядок расположения функций. Он отключает компоновку с приращением, поэтому задавайте его только для завершающих компоновок. Этот ключ задается так: откройте в окне Property Pages папку Linker, страницу Optimization и введите значение в поле Function Order (порядок функций).

/VERBOSE**(выводить сообщения о прогрессе компоновки)****/VERBOSE: LIB****(выводить только сообщения, касающиеся поиска библиотек)**

В случае проблем с компоновкой эти сообщения смогут показать вам, какие символы ищет компоновщик и где он их находит. Информация может оказаться очень объемной, но, возможно, она поможет вам найти причину проблемы. Однажды эти два ключа помогли мне при отладке очень странной ошибки, когда на уровне ассемблера вызываемая функция выглядела совсем не так, как я предполагал. Оказалось, что в двух разных библиотеках имелись две различных функции с одинаковыми сигнатурами, и компоновщик использовал неправильный вариант. Задать эти ключи можно, открыв окно Property Pages, папку Linker, страницу General, в поле Show Progress (показывать информацию о прогрессе компоновки).

/LTCG**(генерация кода во время компоновки)**

Используется вместе с ключом компилятора **/GL** для выполнения перекрестной оптимизации компилянтов. Он устанавливается автоматически при задании ключа **/GL**.

/RELEASE**(задание контрольной суммы)**

Если ключ **/DEBUG** указывает компоновщику генерировать отладочный код, то неверно названный ключ **/RELEASE** не делает, как можно было бы предположить, противоположное и не приказывает компоновщику создать оптимизированную заключительную компоновку. Вообще-то этот ключ следовало бы назвать **/CHECKSUM**. Он всего лишь вносит значения контрольной суммы в заголовок файла Portable Executable (PE). Это необходимо для загрузки драйверов устройств, но не нужно приложениям, работающим в пользовательском режиме. Однако установка этого ключа для завершающих компоновок будет совсем не лишней, так как отладчик WinDBG (см. главу 8) всегда выводит соответствующее сообщение, если двоичный файл не содержит значения контрольной суммы. В отладочных компоновках ключ **/RELEASE** использовать не следует, так как он требует отключения компоновки с приращением. Чтобы установить ключ **/RELEASE** для завершающих компоновок, откройте окно Property Pages, папку Linker, страницу Advanced и выберите в поле Set Checksum (использовать контрольную сумму) значение Yes (**/RELEASE**).

см. след. стр.

`/PDBSTRIPPED`**(не включать частные символы в PDB-файл)**

Одной из сложнейших отладочных проблем является получение чистого стека вызовов. Причина, по которой вы не можете получить хорошие стеки вызовов, в том, что код «плавающих стеков» не включает специальных данных о кадре стека с отсутствующим указателем (FPO, Frame pointer omission), которые помогли бы расшифровать имеющийся стек. Так как данные FPO для вашего приложения содержатся в PDB-файлах, вы можете просто предоставить эти файлы клиенту. Конечно, это вполне обоснованно заставит вас и вашего менеджера нервничать, но не забывайте, что до появления Visual C++ .NET у вас было гораздо больше проблем с получением чистых стеков вызовов.

Если вы когда-нибудь устанавливали символы ОС от Microsoft (см. раздел «Установите символы ОС и создайте хранилище символов»), вы, вероятно, заметили, что символы Microsoft предоставляли вам полную информацию о стеках вызовов, не выдавая никаких секретов. Для этого программисты Microsoft делают следующее: они включают в PDB-файлы только открытые функции и крайне важные данные FPO, но не закрытую информацию вроде переменных и данных об исходных кодах и номерах строк.

Ключ `/PDBSTRIPPED` позволяет вам безопасно создавать аналогичный тип символов для своего приложения, не выдавая никаких секретов. Есть новость и получше: сокращенный PDB-файл генерируется одновременно с его полной версией, поэтому я очень рекомендую устанавливать этот ключ для завершающих компоновок. Откройте диалоговое окно проекта Property Pages, папку Linker, страницу Debugging и задайте в поле Strip Private Symbols (не включать закрытые символы) расположение и название файла символов. Я всегда использую строку `$(OutDir)/$(ProjectName)_STRIPPED.PDB`, чтобы было ясно, какой PDB-файл является сокращенной версией, а какой — полной. Если вы отправляете сокращенные PDB-файлы заказчику, удалите из названий часть «_STRIPPED», чтобы их могли загрузить такие программы, как Dr. Watson.

Разработайте несложную диагностическую систему для заключительных компоновок

Больше всего я ненавижу ошибки, которые происходят только на компьютерах одного-двух пользователей. Все остальные работают с программой без проблем, но у этих происходит что-то совсем не то, почти не поддающееся пониманию. Хотя вы всегда можете попросить пользователя прислать непослушный компьютер вам, эта стратегия не всегда удобна. Если клиент живет на одном из островов в Карибском море, вы, конечно, согласились бы слетать туда и отладить проблему на месте. Однако я почему-то не слышал, чтобы многие компании так щепетильно относились к качеству своей продукции. Не встречались мне и разработчики, готовые с радостью отправиться за Северный полярный круг.

Если проблема происходит только на одной-двух машинах, нужно узнать поток выполнения программы на этих компьютерах. Многие разработчики уже

поддерживают слежение за потоком выполнения при помощи регистрационных файлов и журналов событий, но я хочу особо подчеркнуть, насколько важны такие журналы для решения проблем. Протоколирование потока выполнения окажется при решении проблем гораздо полезнее, если вся группа будет подходить к этому организованно.

При протоколировании информации чрезвычайно важно следовать определенному шаблону. Если данные будут иметь согласованный формат, разработчикам будет гораздо легче проанализировать файл и выяснить интересующие их моменты. Если протоколировать информацию правильно, можно получить просто огромный объем полезных данных, а написав сценарий на Perl'e или каком-то другом языке — легко разделить информацию на важную и второстепенную, существенно ускорив ее обработку.

Ответ на вопрос, что протоколировать, зависит главным образом от проекта, однако в любом случае нужно регистрировать хотя бы ошибочные и аномальные ситуации. Кроме того, следует попытаться учесть логический смысл операции программы. Так, если ваша программа работает с файлами, не стоит записывать в журнал такие подробности, как «Переходим в файле к смещению 23»; вместо этого нужно протоколировать открытие и закрытие файла. Тогда, увидев, что последняя запись в журнале гласит «Подготавливаем открытие D:\Foo\BAR.DAT», вы узнаете, что ваш BAR.DAT скорее всего поврежден.

Глубина протоколирования зависит также от вызываемого им снижения производительности. Я обычно протоколирую все, что мне может понадобиться, и наблюдаю за производительностью заключительных компоновок, когда протоколирование не ведется. Современные средства слежения за производительностью позволяют легко узнать, получает ли управление ваш код протоколирования. Если да, вы можете немного снизить объем регистрируемой информации, пока не достигнете приемлемого баланса с производительностью приложения. Определить, что именно протоколировать, сложно. В главе 3 я расскажу, что нужно протоколировать в управляемых приложениях, а в главе 18 покажу, как выполнять высокоскоростную трассировку неуправляемых приложений с минимальными усилиями. Другим полезным средством является очень быстрая, но неправильно названная система Event Tracing, встроенная в Windows 2000 и более поздние версии (см. о ней по адресу: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/perfmon/base/event_tracing.asp).

Частые сборки программы и дымовые тесты обязательны

Два из самых важных элементов инфраструктуры — система сборки программы и комплект дымовых тестов. Система сборки выполняет компиляцию и компоновку программы, а комплект дымовых тестов включает тесты, которые запускают программу и подтверждают, что она работает. Джим Маккарти (Jim McCarthy) в книге «Dynamics of Software Development» (Microsoft Press, 1995) называет ежедневное проведение сборки программы и дымовых тестов сердцебиением проекта. Если эти процессы неэффективны, проект мертв.

Частые сборки

Проект надо собирать каждый день. Порой мне говорят, что некоторые проекты бывают столь огромны, что их невозможно собирать каждый день. Означает ли это, что они включают более 40 миллионов строк кода, лежащих в основе Windows XP или Windows Server 2003? Учитывая, что эти ОС — самые масштабные коммерческие программные проекты в истории и все же собираются каждый день, я так не думаю. Итак, ежедневная сборка программы оправданий не имеет. Вы не только должны собирать проект каждый день, но и автоматизировать этот процесс.

При сборке следует одновременно собирать и заключительную, и отладочную компоновки. Как я покажу ниже, отладочные компоновки очень важны. Неудачная сборка программы — большой грех. Если разработчики зарегистрировали код, который не компилируется, виновного нужно наказать. Публичная порка, вероятно, была бы несколько жесткой формой наказания (хотя и не слишком), но есть и другой метод: заставьте провинившегося публично раскаться в преступлении и покупать пончики для всей группы. По крайней мере в группах, в которых работал я, это всегда давало отличные результаты. Если в вашей группе нет штатного сотрудника, отвечающего за сборку программы, вы можете наказать человека, по вине которого провалилась сборка программы, возложив на него ответственность за сборку до тех пор, пока эта обязанность не перейдет к его товарищу по несчастью.

Одна из лучших практик ежедневной сборки проекта, которую я когда-либо использовал, заключается в оповещении членов группы по электронной почте при окончании сборки. При автоматизированной ночной сборке программы каждый член группы может утром сразу же узнать, увенчалась ли сборка успехом; если нет, группа может предпринять немедленные действия по исправлению ситуации.

Чтобы избежать проблем со сборкой программы, каждый член группы должен иметь одинаковые версии всех инструментов и компонентов сборки. Как я уже упоминал, в некоторых группах это гарантируется путем хранения системы сборки программы в системе управления версиями. Если члены группы работают с разными версиями инструментов, включая разные версии пакетов обновлений (service pack), они создают идеальную почву для ошибок при сборке программы. Если убедительных причин использования кем-нибудь другой версии компилятора нет, никакой разработчик не должен обновлять свои инструменты по собственной воле. Кроме того, все члены группы должны использовать для сборки своих частей программы одни и те же сценарии и компьютеры. Так образуется надежная связь между тем, что создается разработчиками, и тем, что тестируется тестировщиками.

При каждой сборке программы система сборки будет извлекать самую последнюю версию исходных кодов из системы управления версиями. В идеале разработчикам также следует ежедневно использовать файлы системы управления версиями. В случае крупного проекта разработчики должны иметь возможность легкого получения ежедневно компилируемых двоичных файлов, чтобы избежать длительной компиляции программы на своих компьютерах. Нет ничего хуже, чем тратить время на решение сложной проблемы только затем, чтобы обнаружить, что проблема связана с более старой версией файла на машине разработчика. Другое преимущество частого извлечения файлов из системы управления версиями состоит в том, что это помогает навязать правило «никакая сборка программы не

должна заканчиваться неудачей». При частом извлечении файлов из системы управления версиями любая проблема общей сборки программы автоматически становится локальной проблемой каждого разработчика. Если руководителей неудача ежедневной сборки программы раздражает, то разработчики просто лопаются от гнева, если вы нарушаете их локальную сборку. Зная, что неудача общей сборки программы означает неудачу сборки для всех членов группы, разработчики будут более ответственно подходить к регистрации в общих исходных текстах только тщательно проверенного кода.

Стандартный вопрос отладки

Когда прекращать модернизацию компилятора и других инструментов?

Как только вы завершили разработку функциональности приложения, что также известно как стадия бета-1, вам определенно не следует модернизировать никакие инструменты. Схема оптимизации нового компилятора, какой бы хорошей она ни казалась, не оправдывает изменения кода программы. Ко времени достижения стадии бета-1 значительный объем тестирования уже выполнен, и, если вы измените инструменты, начать его придется с нуля.

Дымовые тесты

Так называют тест, проверяющий основные функции приложения. Термин «дымовой тест» берет начало в электронике. На некотором этапе разработки продукции инженеры по электронике подключают устройство в сеть и смотрят, не задымит-ся ли оно (в буквальном смысле). Если устройство не дымит или, что еще хуже, не загорается, значит, группа достигла определенного прогресса. Обычно дымовой тест приложения заключается в проверке его основных функций. Если они работают, можно начинать серьезное тестирование программы. Дымовой тест играет роль базового показателя состояния кода.

Дымовой тест представляет собой просто контрольную таблицу функций, которые может выполнять программа. Начните с малого: установите приложение, запустите его и закройте. По мере цикла разработки дымовые тесты также должны развиваться, чтобы можно было исследовать новые функции программы. Дымовой тест должен включать по крайней мере по одному тесту для каждой функции и каждого крупного компонента программы. Это значит, что, работая в отделе готовой продукции, вы должны тестировать каждую функцию, упомянутую в рекламных проспектах. Если вы сотрудник ИТ-отдела, тестируйте основные функции, которые вы обещали реализовать менеджеру по информатизации и своим клиентам. Помните: дымовой тест вовсе не должен проверять абсолютно все пути выполнения вашей программы, его надо использовать, чтобы узнать, выполняет ли программа основные функции. Как только она прошла дымовой тест, сотрудники отдела технического контроля могут начинать свой тяжелый труд, пытаясь нарушить работу программы новыми изощренными способами.

Чрезвычайно важный компонент дымового теста — та или иная форма теста производительности. Многие забывают про это, в результате чего приходится

расплачиваться на более поздних этапах цикла разработки программы. Если у вас есть сравнительный тест какой-либо операции программы (например, как долго запускалась последняя версия программы), неудачу теста можно определить как замедление выполнения операции на 10% или более. Я всегда удивляюсь тому, сколь часто небольшое изменение в безобидном на вид месте программы может приводить к огромному снижению производительности. Наблюдая за производительностью программы на протяжении всего цикла ее разработки, вы сможете решать проблемы с производительностью до того, как они выйдут из под контроля.

В идеале при проведении дымового теста выполнение программы должно быть автоматизировано, чтобы она могла работать без взаимодействия с пользователем. Инструмент, применяемый для автоматизации ввода информации и выполнения действий с приложением, называется средством регрессивного тестирования. Увы, не всегда можно автоматизировать тестирование каждой функции, особенно при изменении UI. На рынке много хороших средств регрессивного тестирования, поэтому, если вы работаете над крупным сложным приложением и не можете позволить себе, чтобы кто-либо из вашей группы отвечал исключительно за проведение и поддержку дымовых тестов, возможно, следует подумать о покупке такого инструмента. Если уговорить начальника приобрести коммерческий инструмент не получается, можете использовать приложение Tester из главы 16, записывающее ввод мыши и клавиатуры в файл JScript или VBScript, который затем можно воспроизвести.

К неудачному выполнению дымового теста следует относиться так же серьезно, как и к неудачной сборке программы. На создание дымового теста уходит очень много усилий, поэтому никакой разработчик не должен относиться к нему легкомысленно. Именно дымовой тест говорит группе контроля качества о том, что полученная ими версия программы достаточно хороша, чтобы с ней можно было работать, поэтому проведение дымового теста должно быть обязательным. Если у вас есть автоматизированный дымовой тест, возможно, его стоит предоставить и разработчикам, чтобы они также могли автоматизировать свое тестирование. Кроме того, автоматизированный дымовой тест надо проводить с каждой ежедневной сборкой программы, чтобы можно было сразу оценить ее качество. Как и при ежедневной сборке, результаты дымового теста следует сообщать членам группы по электронной почте.

Работу над программой установки следует начинать немедленно

Начинайте работать над программой установки сразу же после начала проекта. Это первая часть вашего приложения, которую видят пользователи. Слишком многие программы оставляют плохое первое впечатление, показывая, что программа установки была создана в последнюю минуту. Если вы начнете работу над программой установки как можно раньше, у вас будет время на ее тестирование и отладку. Разработав ее на ранней стадии проекта, вы сможете включить ее в дымовой тест. Это позволит вам провести ее многократное тестирование, а ваши тесты еще на один шаг приблизятся к имитации того, как пользователи будут работать с программой.

Ранее я рекомендовал собирать и заключительную, и отладочную версии программы. Вам также понадобится программа установки, которая позволит устанавливать обе версии. Хотя управляемые приложения поддерживают хваленый метод установки при помощи команды `XCOPY`, он годится только для простейших программ. Реальные управляемые приложения скорее всего должны будут инициализировать базы данных, помещать сборки в глобальный кэш сборок и выполнять другие операции, которые просто невозможны при обычном копировании. Программисты, разрабатывающие неуправляемые приложения, должны также помнить, что технология COM все еще жива и здорова, а COM требует внесения такого объема информации в реестр, что без программы установки правильная установка приложения становится почти невозможной. Программа установки отладочных компоновок позволяет разработчикам легко установить отладочную версию приложения и быстро приступить к решению проблемы.

Еще одно преимущество как можно более раннего создания программы установки в том, что другие сотрудники компании гораздо раньше смогут начать тестирование приложения. Получив программу установки, сотрудники службы технической поддержки начнут использовать приложение и предоставлять обратную связь достаточно рано, чтобы вы успели придумать оптимальный способ решения обнаруженных ими проблем.

Тестирование качества должно проводиться с отладочными компоновками

Если вы будете следовать моим рекомендациям из главы 3, вы получите несколько прекрасных средств диагностики своего кода. Проблема в том, что диагностика обычно приносит выгоду только разработчикам. Чтобы сотрудники группы контроля качества оказывали более эффективную помощь в отладке ошибок, они также должны использовать отладочные компоновки. Вы будете удивлены тем, как много проблем вы найдете и решите, если группа контроля качества проведет тестирование отладочных компоновок.

Есть одно очень важное условие: запретить вывод информации макросами `ASSERT`, чтобы они не мешали работе автоматизированных тестов отдела контроля качества. В главе 3 я расскажу о применении макросов `ASSERT` для управляемого и неуправляемого кода. И управляемый код, и мой макрос `SUPERASSERT` для неуправляемого кода поддерживают отключение всплывающих информационных окон и вывода других данных, вызывающих неудачу автоматизированных тестов.

На начальных стадиях цикла разработки программы сотрудники группы контроля качества должны тестировать и отладочные, и заключительные компоновки. По мере развития проекта им следует все большее внимание уделять заключительным компоновкам. Пока вы не достигнете точки альфа-версии, когда в программе будет реализовано достаточно функций, чтобы ее можно было показать клиентам, группа контроля качества должна тестировать отладочные компоновки два-три дня в неделю. При приближении к контрольной точке бета-1 время тестирования отладочных компоновок нужно снизить до двух дней в неделю. По достижении точки бета-2, когда все функции программы реализованы и основные ошибки исправлены, это время надо уменьшить до одного дня в неделю.

Миновав контрольную точку предварительной версии (release candidate), следует перейти на тестирование только заключительных компоновок.

Устанавливайте символы ОС и создайте хранилище символов

Как известно любому человеку, который провел более 5 минут над разработкой программ для Windows, секрет эффективной отладки состоит в согласованном использовании корректных символов. Если вы пишете управляемый код, то без символов отладка вообще может оказаться невозможной. Работая без символов над неуправляемым кодом, вы, возможно, не получите чистые стеки вызовов из-за «плавающих стеков» — для этого нужны данные FPO, содержащиеся в PDB-файле.

Если вы думаете, что заставить всех членов группы и сотрудников компании применять корректные символы очень сложно, представьте, насколько хуже обстоит дело в группе разработчиков ОС Microsoft. Они работают над крупнейшим коммерческим приложением в мире, имеющем более 40 миллионов строк кода. Они выполняют сборку каждый день, и в каждый конкретный момент времени во многих странах мира выполняются тысячи различных компоновок ОС. Не правда ли, с этой точки зрения, ваши проблемы с символами — сущая чепуха: даже если вы думаете, что работаете над большим проектом, ваши неудобства ни в какое сравнение не идут с такой огромной символьной болью!

Кроме проблемы с символами перед программистами Microsoft также стояла проблема получения нужных двоичных файлов. Одна из разработанных в Microsoft технологий, призванных помочь отлаживать ошибки, называется минидамп, или аварийный дамп. Минидамп представляет собой файлы, содержащие сведения о состоянии приложения на момент аварийного завершения. Если вы имеете опыт работы с другими ОС, можете называть его дампом ядра. Привлекательность минидампа объясняется тем, что, имея файлы, характеризующие состояние приложения, вы сможете загрузить его в отладчик, и все данные будут такими, как если бы крах приложения произошел на ваших глазах. О создании собственных минидампов, а также о работе с ними в отладчиках я расскажу в следующих главах. Большая проблема минидампов заключается в загрузке правильных двоичных файлов. Даже если вы создадите программу на платформе Windows Server 2003 или более новой, минидамп клиента может быть создан в системе Windows 2000 только с первым пакетом обновления. В этом случае справедливо то же утверждение, что и в ситуации с символами: если вы не можете загрузить точные двоичные файлы, находившиеся в памяти во время создания минидампа, вы полностью заблуждаетесь, если думаете, что он позволит вам легко справиться с проблемой.

Разработчики Microsoft понимали, что им просто необходимо сделать что-то, чтобы облегчить свою жизнь. Мы, программисты, не работающие в Microsoft, также жаловались, что из-за отсутствия символов и двоичных файлов ОС, соответствующих многочисленным обновлениям и исправлениям, установленным на конкретном компьютере, отладка превращается в пытку. Концепция сервера символов проста: хранить все символы и двоичные файлы публичных компоновок в известном месте и наделить отладчики необходимым интеллектом, чтобы они могли использовать корректные символы и двоичные файлы для каждого загружаемого в процесс модуля — независимо от того, загружается ли он вашей программой или

ОС — без взаимодействия с пользователем. Вся прелесть в том, что реальность почти столь же проста! С серверами символов связано несколько проблем, которые я опишу чуть ниже, но, если сервер символов создан и настроен как надо, никто в вашей группе или компании никогда не будет страдать от отсутствия корректных символов или двоичных файлов независимо от того, разрабатывает ли он управляемый, неуправляемый или смешанный код и использует ли отладчик Visual Studio .NET или WinDBG. И еще одна приятная новость: к этой книге я прилагаю несколько файлов, которые возьмут на себя всю работу по получению отличных символов и двоичных файлов для ОС и ваших программ.

В документации к Visual Studio .NET упоминается один метод создания сервера символов для отладки, но он требует выполнения нескольких одинаковых действий для каждой загружаемой программы, что очень неудобно. Кроме того, там не обсуждается самое важное: как заполнить сервер символами и двоичными файлами. Так как именно в этом огромное преимущество применения сервера символов, то для достижения символьной нирваны вам понадобится сделать следующее.

Получить физический сервер, к которому сможет получать доступ любой сотрудник, работающий над вашими проектами, довольно просто. Вы, вероятно, захотите назвать этот сервер `\\SYMBOLS`, чтобы сразу было ясно, какую функцию он выполняет. В оставшейся части я буду использовать именно это имя сервера. Он не обязательно должен быть очень мощным, так как будет выполнять функцию обычного файлового сервера. Однако я очень рекомендую, чтобы сервер имел довольно большой объем дискового пространства. Для начала вполне хватит от 40 до 80 Гб. Установив все серверное ПО, создайте два каталога с общим доступом под названием `OSSYMBOLS` и `PRODUCTSYMBOLS`, разрешив запись и чтение всем разработчикам и сотрудникам отдела контроля качества. Вы, наверное, уже догадались по названиям, что в одном каталоге будут храниться символы и двоичные файлы ОС, а во втором — аналогичные файлы ваших программ. Для простоты администрирования их следует хранить отдельно. Я полагаю, вы сможете получить в свое распоряжение этот сервер. Все сражения за него я оставляю вам в качестве упражнения.

Следующий шаг к достижению символьной нирваны — установка пакета Debugging Tools for Windows. Его можно или загрузить с сайта Microsoft по адресу www.microsoft.com/ddk/debugging, или установить с CD, прилагаемого к книге. Обратите внимание: двоичные файлы для сервера символов созданы группой разработчиков Windows, а не Visual Studio .NET. Проверьте, существует ли обновленная версия Debugging Tools for Windows; похоже, группа разработчиков обновляет этот пакет довольно часто. После установки Debugging Tools for Windows укажите установочный каталог в системной переменной среды `PATH`. Разрешите запись информации в сервер символов и ее чтение для четырех важнейших двоичных файлов: `SYMSRV.DLL`, `DBGHELP.DLL`, `SYMCHK.EXE` и `SYMSTORE.EXE`.

Если вы работаете с прокси-сервером, требующим регистрации при каждом подключении к Интернету, я вам сочувствую. К счастью, группа разработчиков Windows не осталась безучастной к вашей боли. В пакет Debugging Tools for Windows версии 6.1.0017 входит новая версия библиотеки `SYMSRV.DLL`, удовлетворяющая требованиям компаний, следящих за каждым Интернет-пакетом. Изучите в доку-

ментации к Debugging Tools for Windows раздел «Using Symbol Servers and Symbol Stores» (Использование серверов и хранилищ символов), в котором обсуждается работа с прокси-серверами и межсетевыми экранами. Там сказано, как задать переменную среды `_NT_SYMBOL_PROXY`, чтобы избежать ввода имени пользователя и пароля при каждом запросе на загрузку символов. Следите за появлением новых версий Debugging Tools for Windows на сайте www.microsoft.com/ddk/debugging. Группа разработчиков Windows постоянно работает над улучшением серверов символов, поэтому я рекомендую следить за появлением новых версий этого пакета.

Как только вы установите Debugging Tools for Windows, вам останется только создать системную среду для Visual Studio и отладчика WinDBG. Лучше всего задать переменную среды в системных параметрах (т. е. параметрах для всего компьютера). Для получения доступа к этой области в Windows XP/Server 2003 нужно щелкнуть правой кнопкой значок My Computer (Мой компьютер) и выбрать в контекстном меню пункт Properties (Свойства). Выберите вкладку Advance (Дополнительно) и нажмите кнопку Environment Variables (Переменные среды) в нижней части страницы. Диалоговое окно Environment Variables показано на рис. 2-10. Если переменной среды `_NT_SYMBOL_PATH` нет, создайте ее и присвойте ей следующее значение (обратите внимание, что указанное выражение должно быть введено в одной строке):

```
SRV*\\Symbols\OSSymbols*http://msdl.microsoft.com/download/symbols;  
SRV*\\Symbols\ProductSymbols
```

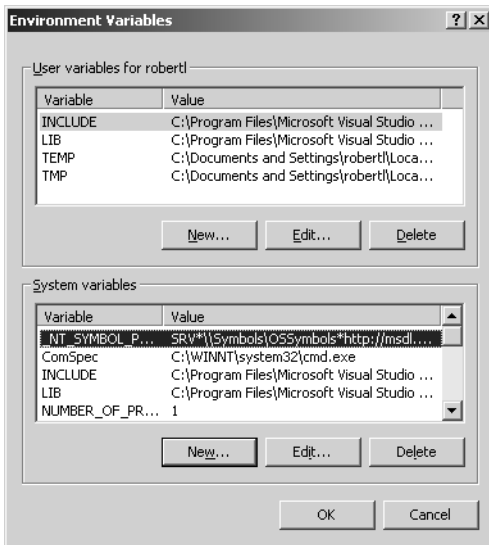


Рис. 2-10. Диалоговое окно *Environment Variables*

Переменная `_NT_SYMBOL_PATH` будет указывать Visual Studio .NET и WinDBG, где искать ваши серверы символов. В указанной строке заданы два отдельных сервера символов, отделенные точкой с запятой: один для символов ОС, а другой для символов ваших программ. Буквы SRV в начале обеих частей строки приказывают отладчикам загрузить библиотеку SYMSRV.DLL и передать ей значения, распо-

ложенные после SRV. В случае первого сервера символов вы сообщаете SYMSRV.DLL, что символы ОС будут храниться в каталоге `\\Symbols\OSSymbols`; вторая звездочка является HTTP-адресом, который SYMSRV.DLL будет использовать для загрузки любых символов (но не двоичных файлов), отсутствующих в сервере символов. Этот раздел переменной `_NT_SYMBOL_PATH` обеспечит обновление символов ОС. Вторая часть переменной `_NT_SYMBOL_PATH` говорит библиотеке SYMSRV.DLL о том, что специфические символы ваших программ следует искать только в общем каталоге `\\Symbols\ProductSymbols`. Если вы хотите задать другие пути поиска, можете добавить их к строке переменной `_NT_SYMBOL_PATH`, разделив их точками с запятой. Так, в следующей строке указано, чтобы поиск символов ваших программ осуществлялся и в корневом системном каталоге `System32`, потому что именно в этот каталог Visual Studio .NET помещает PDB-файлы стандартной библиотеки C и MFC при установке:

```
SRV*\\Symbols\OSSymbols*http://msdl.microsoft.com/download/symbols;  
SRV*\\Symbols\ProductSymbols;c:\windows\system32
```

В полной степени достоинства сервера символов обнаруживаются при его заполнении символами ОС, загруженными с сайта Microsoft. Если вы опытный «охотник на насекомых», то, вероятно, уже установили символы ОС. Однако это всегда немного разочаровывает, так как почти на всех компьютерах установлены те или иные пакеты исправлений, а определенные символы ОС никогда не включают символы этих пакетов. К счастью, серверы символов гарантируют, что вы всегда сможете получить абсолютно правильные символы ОС без всякого труда! Это огромное благо, которое здорово облегчит вашу жизнь. Оно стало возможным благодаря тому, что Microsoft открыла доступ к символам для всех ОС от Microsoft Windows NT 4 до последних версий Windows XP/.NET Server 2003, включая все пакеты обновлений и исправления.

В начале следующего сеанса отладки отладчик автоматически увидит, что переменная `_NT_SYMBOL_PATH` задана и, если нужного ему файла символов не найдется, начнет загрузку символов ОС с Web-сайта Microsoft и поместит их в ваше хранилище символов. Внесем ясность: сервер символов загрузит с сайта только нужные ему символы, а не все символы ОС. Размещение хранилища символов в общем каталоге сэкономит вам много времени: если один из членов группы уже загрузил нужный вам символ, вам не понадобится загружать его повторно.

В самом по себе хранилище символов нет ничего удивительного. Это обычная база данных, которая для нахождения файлов использует файловую систему. На рис. 2-11 показано, как выглядит часть дерева моего сервера символов в окне Windows Explorer. Корневой каталог называется `OSSymbols`, и все файлы символов, такие как `ADVAPI32.PDB`, находятся на первом уровне. Под именем каждого файла символов находится каталог, название которого соответствует дате/времени, сигнатуре и прочей информации, необходимой для полного определения конкретной версии файла символов. Помните: при наличии нескольких вариантов файла (например, `ADVAPI32.PDB`) для различных версий ОС, у вас будет и несколько каталогов, соответствующих каждому варианту. В каталоге сигнатур скорее всего будет находиться конкретный файл символов для данного варианта. Есть меры предосторожности, которые нужно соблюдать, создавая при помощи специаль-

ных текстовых файлов указатели на другие файлы в хранилище символов, но если вы будете делать все так, как я рекомендую, все будет в порядке.

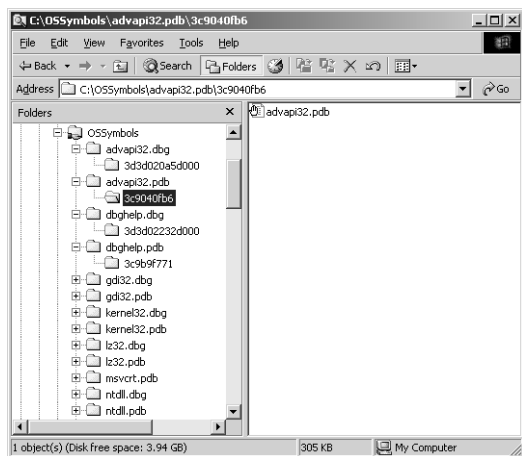


Рис. 2-11. Пример базы данных сервера символов

Загрузка символов во время отладки очень полезна, однако она не способствует получению двоичных файлов ОС. Кроме того, лучше было бы не возлагать ответственность за получение символов на разработчиков, а изначально наполнить серверы символов всеми двоичными файлами и символами всех поддерживаемых вами ОС. Это позволило бы вам работать с любыми минидампами клиентов и любыми отладочными проблемами, с которыми вы столкнетесь в своем отделе.

Пакет Debugging Tools for Windows (в состав которого входит WinDBG) включает два очень полезных инструмента: Symbol Checker (SYMCHK.EXE), предназначенный для загрузки в ваш символьный сервер символов Microsoft, и Symbol Store (SYMSTORE.EXE), который заботится о загрузке в хранилище символов двоичных файлов. Я понимал, что для наполнения своего сервера символами и двоичными файлами для всех версий ОС, которые я хочу поддерживать, мне придется работать с обоими инструментами, поэтому я решил автоматизировать этот процесс. Я хотел, чтобы создание сервера символов ОС было простым и легким, чтобы он постоянно был заполнен последними двоичными файлами и символами и чтобы это практически не требовало работы.

Создавая первый сервер символов ОС, установите первую версию ОС без всяких пакетов обновлений и исправлений. Установите пакет Debugging Tools for Windows и укажите его установочный каталог в переменной PATH. Для получения двоичных файлов и символов ОС запустите мой файл OSSYMS.JS, про который я расскажу чуть ниже. Когда OSSYMS.JS завершит свою работу, установите первый пакет обновлений и выполните OSSYMS.JS повторно. Установив все пакеты обновлений и скопировав все их двоичные файлы и символы, установите все обновления, рекомендованные функцией Windows Update ОС Windows 2000/XP/.NET Server 2003, и запустите OSSYMS.JS в последний раз. Повторите этот процесс для всех ОС, которые вам нужно поддерживать. Теперь, чтобы ваш сервер символов постоянно находился в отличном состоянии, нужно будет только запускать OSSYMS.JS каждый раз, когда вы установите исправление или новый пакет обновлений. Ради

целей планирования я подсчитал, что это требует чуть менее 1 Гб для каждой версии ОС и примерно такого же объема для каждого пакета обновлений.

Возможно, вы думаете, что OSSYMS.JS (и вспомогательный файл WRITENOT-FIXES.VBS, который нужно скопировать в тот же каталог, что и OSSYMS.JS) представляет собой простую оболочку для вызова программ SYMCHK.EXE и SYMSTORE.EXE, но это не так. На самом деле это очень полезная оболочка. Если вы изучите ключи командной строки обеих программ, вам непременно захочется автоматизировать их работу, потому что в ключах очень легко запутаться. Запустив программу OSSYMS.JS без параметров командной строки, вы увидите текст, описывающий все ее функции:

```
OSsyms - Version 1.0 - Copyright 2002-2003 by John Robbins
Debugging Applications for Microsoft .NET and Microsoft Windows
```

```
Fills your symbol server with the OS binaries and symbols.
Run this each time you apply a service pack/hot fix to get the perfect
symbols while debugging and for mini dumps.
SYMSTORE.EXE and SYMCHK.EXE must be in the path.
```

```
Usage: OSsyms <symbol server> [-e|-v|-b|-s|-d]
```

```
<symbol server> - The symbol server in \\server\share.
-e               - Do EXEs as well as DLLs.
-v               - Do verbose output.
-d               - Debug the script. (Shows what would execute.)
-b               - Don't add the binaries to the symbol store.
-s               - Don't add the symbols to the symbol store.
                  (Not recommended)
```

Единственный необходимый параметр — путь к серверу символов в формате \\сервер\общий_каталог. Когда вы запускаете программу OSSYMS.JS, она сначала определяет версию ОС и уровень установленного пакета обновлений и находит все исправления. Это позволяет приложению SYMSTORE.EXE правильно заполнить информацию о программе, ее версии и поле комментария, чтобы вы могли точно определить, какие символы и двоичные файлы хранятся в сервере символов. Про специфические ключи командной строки SYMSTORE.EXE и то, как узнать, что находится в вашей базе данных, я расскажу ниже. Огромная важность информации об установленных пакетах обновлений и исправлений объясняется тем, что при получении минидампа она позволяет быстро определить, есть ли в сервере символов двоичные файлы и символы для этого конкретного случая.

После сбора нужной системной информации программа OSSYMS.JS выполняет рекурсивный поиск всех двоичных файлов DLL в каталоге ОС (%SYSTEMROOT%) и копирует их в сервер символов. Выполнив копирование, OSSYMS.JS вызывает программу SYMCHK.EXE для автоматической загрузки из Интернета всех имеющихся символов для этих DLL. Если вы хотите сохранить в сервере символов все EXE-файлы и их символы, укажите в командной строке OSSYMS.JS после пути к серверу символов ключ-е.

Чтобы узнать, какие двоичные файлы и символы были сохранены в сервере символов, а какие были проигнорированы (с указанием причин), прочитайте

информацию, содержащуюся в текстовых файлах DllBinLog.TXT и DllSymLog.TXT, в которых описаны результаты добавления в сервер двоичных файлов и символов DLL соответственно. В случае EXE-файлов соответствующие файлы называются ExeBinLog.TXT и ExeSymLog.TXT.

Выполнение OSSYMS.JS может потребовать времени. Копирование двоичных файлов в сервер символов выполняется быстро, однако загрузка символов из сети Интернет может затянуться. При загрузке символов ОС для DLL и EXE-файлов нужно будет загрузить скорее всего около 400 Мб данных. Следует избегать добавления двоичных файлов в сервер символов несколькими компьютерами одновременно. Это объясняется тем, что SYMSTORE.EXE использует в качестве базы данных файловую систему и текстовый файл, поэтому она не поддерживает транзакций. Программа SYMCHK.EXE не использует текстовую базу данных SYMSTORE.EXE, поэтому сохранение символов несколькими разработчиками одновременно вполне допустимо.

Microsoft постоянно размещает на своем сайте все большее число символов для своей продукции. Программа OSSYMS.JS достаточно гибка, чтобы можно было легко указывать серверу символов дополнительные каталоги хранения двоичных файлов и соответствующих символов. Чтобы добавить в сервер символов новые двоичные файлы, найдите глобальную переменную `g_AdditionalWork`, расположенную в начале файла OSSYMS.JS. Этой переменной присвоено значение `null`, поэтому в функции `main` она не обрабатывается. Чтобы сохранить в сервере символов новый набор файлов, создайте `Array` и добавьте в него в качестве элемента класс `SymbolsToProcess`. Ниже показано, как включить сохранение в сервере символов всех DLL, которые находятся в каталоге Program Files. Заметьте: первый элемент не обязан быть переменной среды — он может быть названием конкретного каталога, скажем, «`c:\ Program Files`». Однако использование общей системной переменной среды позволяет избежать жесткого задания названий дисков.

```
var g_AdditionalWork = new Array
(
    new SymbolsToProcess ( "%ProgramFiles%" , // Начальный каталог.
                          "*.dll" , // Ищем все DLL.
                          "PFD11BinLog.TXT" , // Журнал для двоичных файлов.
                          "PFD11SymLog.TXT" ) // Журнал для символов.
);
```

Я объяснил, как сохранить в сервере символов двоичные файлы и символы ОС. Давайте теперь рассмотрим, как с помощью программы SYMSTORE.EXE сделать то же самое для ваших программ. SYMSTORE.EXE имеет много ключей командной строки (табл. 2-2).

Табл. 2-2. Важные ключи командной строки программы SYMSTORE

Ключ	Описание
add	Добавляет файлы в хранилище символов.
del	Удаляет из хранилища символов конкретный набор файлов.
/f File	Добавляет в хранилище символов конкретный файл или каталог.
/r	Рекурсивно добавляет в хранилище символов файлы или каталоги.
/s Store	Корневой каталог хранилища символов.

Табл. 2-2. Важные ключи командной строки ... (продолжение)

Ключ	Описание
/t Product	Название программы.
/v Version	Версия программы.
/c	Дополнительные комментарии.
/o	Подробный вывод, полезный для отладки.
/i ID	Идентификатор транзакции из файла history.txt, используемый при удалении файлов.
/?	Справка.

Наилучший способ использования SYMSTORE.EXE состоит в автоматическом сохранении EXE-, DLL- и PDB-файлов дерева проекта после его ежедневной сборки (если дымовой тест покажет, что программа работает), после каждой контрольной точки и при передаче компоновки за пределы группы. Если вы не обладаете дисковым пространством огромного объема, то разработчикам не следует сохранять в сервере символов свои локальные компоновки. Например, следующая команда сохраняет в хранилище символов все PDB- и двоичные файлы, которые будут обнаружены во всех каталогах, дочерних по отношению к каталогу D:\BUILD (включая и его).

```
symstore add /r /f d:\build\*. * /s \\Symbols\ProductSymbols
/t "MyApp" /v "Build 632" /c "01/22/03 Daily Build"
```

При добавлении файлов ключ /t (название программы) требуется всегда, но для ключей /v (версия) и /c (комментарии) это, увы, не так. Советую всегда использовать ключи /v и /c, потому что информация о том, какие файлы хранятся в сервере символов вашей программы, никогда не может оказаться лишней. По мере заполнения сервера символов вашей программы это приобретает особую важность. Символы, хранящиеся в сервере символов ОС, имеют меньший объем из-за того, что они не включают всех частных символов и типов, однако символы вашей программы могут достигать огромных размеров, что может приводить к заметному уменьшению дискового пространства при работе над полугодовым проектом.

Непрерывно сохраняйте в сервере символов все компоновки, соответствующие достижению контрольных точек, и компоновки, отсылаемые за пределы группы. Однако мне нравится держать в хранилище символов двоичные файлы и символы ежедневных компоновок не более чем за последние четыре недели. Как видно из табл. 2-2, SYMSTORE.EXE поддерживает и удаление файлов.

Для гарантии того, что вы удаляете те файлы, которые действительно собирались удалить, нужно посмотреть специальный каталог 000admin, находящийся в общем каталоге сервера символов. В этом каталоге есть файл HISTORY.TXT, содержащий историю всех транзакций сервера символов и, если вы добавляли файлы в сервер символов, набор пронумерованных файлов, включающих списки файлов, которые на самом деле были добавлены в сервер символов в результате транзакций.

HISTORY.TXT является файлом со значениями, разделенными запятыми (Comma separated value, CSV), поля которого приведены в табл. 2-3 (для добавления файлов) и в табл. 2-4 (для удаления файлов).

Табл. 2-3. Поля CSV файла HISTORY.TXT для добавления файлов

Поле	Описание
ID	Номер транзакции. Это число имеет 10 разрядов, поэтому в общей сложности сервер символов может выполнить 9,999,999,999 транзакций.
Add	При добавлении файлов это поле всегда имеет значение add.
File или Ptr	Показывает, что было добавлено: файл (file) или указатель (ptr) на файл, находящийся в другом месте.
Date	Дата транзакции.
Time	Время начала транзакции.
Product	Название программы, указанное после ключа /t.
Version	Версия программы, указанная после ключа /v (необязательный параметр) .
Comment	Текст комментария, указанный после ключа /c (необязательный параметр) .
Unused	Неиспользуемое поле, зарезервированное на будущее.

Табл. 2-4. Поля CSV файла HISTORY.TXT для удаления файлов

Поле	Описание
ID	Номер транзакции.
Del	При удалении файлов это поле всегда имеет значение del.
Deleted Transaction	10-разрядный номер удаленной транзакции.

Как только вы определили номер транзакции, которую желаете удалить, сделать это при помощи SYMSTORE.EXE очень просто:

```
symstore del /i 0000000009 /s \\Symbols\ProductSymbols
```

При удалении файлов из сервера символов я заметил одну странную вещь: не выводится абсолютно никакой информации, подтверждающей, что удаление увенчалось успехом. Если вы забудете указать какой-то важный ключ командной строки, например, само название сервера символов, вы не получите никаких предупреждений и, возможно, будете ошибочно думать, что файлы были удалены. Поэтому после удаления я всегда проверяю файл HISTORY.TXT, чтобы убедиться, что удаление действительно имело место.

Исходные тексты и серверы символов

После упорядочения символов и двоичных файлов следующий элемент головоломки — упорядочение исходных файлов. Правильные стеки вызовов — прекрасное достижение, но пошаговое изучение комментариев к исходному коду не нравится никому. К сожалению, пока Microsoft не интегрирует компиляторы с системой управления версиями, чтобы по мере создания компоновок компиляторы могли извлекать и помечать исходные тексты программы, вам придется кое-что делать вручную.

Возможно, вы не заметили, но все компиляторы из состава Visual Studio .NET уже включают в PDB-файлы полный путь к исходным файлам программы. В пре-

дыдущих версиях компиляторов это не поддерживалось, что чрезвычайно осложняло получение нужных исходных текстов. Полный путь повышает ваши шансы на получение необходимых исходных файлов программы при отладке ее предыдущих версий или изучении минидампа.

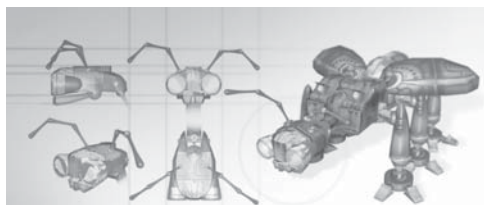
На компьютере для сборки программы следует при помощи команды `SUBST` отобразить корень дерева проекта на диск S:. В результате этого при сборке программы диск S: будет корневым каталогом информации об исходных текстах, включаемой во все PDB-файлы, которые вы будете добавлять в хранилище символов. Если разработчику нужно будет отладить предыдущую версию исходного кода, он сможет извлечь ее из системы управления версиями и отобразить ее при помощи команды `SUBST` на диск S:. Благодаря этому отладчик, показывая исходный код программы, сможет загрузить правильную версию файлов символов с минимальным количеством проблем.

Хотя я вкратце описал серверы символов, вам непременно следует полностью прочитать раздел «Symbols» в документации к пакету Debugging Tools for Windows. Технология серверов символов настолько важна для успешной отладки, что в ваших интересах знать о ней как можно больше. Надеюсь, я смог доказать важность серверов символов и описать способы их лучшего применения. Если вы еще не создали свой сервер символов, я приказываю вам прекратить чтение и сделать это.

Резюме

В этой главе я описал чрезвычайно важные инфраструктурные требования по минимизации времени отладки. Они варьируются от систем управления версиями и отслеживания ошибок, параметров компилятора и компоновщика до преимуществ ежедневных сборок и дымовых тестов и важности использования символов.

Возможно, ваша уникальная среда разработки предъявляет какие-нибудь дополнительные инфраструктурные требования, однако предложенные в этой главе рекомендации справедливы для всех сред. В их важности я убедился, работая над реальными проектами. Если вы еще не используете в своей компании какие-либо из этих инфраструктурных инструментов и методов, я настоятельно советую немедленно реализовать их. Они позволят вам сэкономить на отладке многие сотни часов.



Отладка при кодировании

В главе 2 я заложил основу общепроектной инфраструктуры, обеспечивающей более эффективную работу. В этой главе мы определим, как облегчить отладку, когда вы погрязли в кодовых баталиях. Большинство называет этот процесс защитным программированием (defensive programming), но я предпочитаю думать о нем несколько шире и глубже — как о профилактическом программировании (proactive programming) или отладке при кодировании. По моему определению, защитное программирование — это код обработки ошибок, сообщающий вам, что возникла ошибка. Профилактическое программирование позволяет узнать, почему возникла ошибка.

Создание защищенного кода — лишь часть борьбы за исправление ошибок. Обычно специалисты пытаются провести очевидные защитные маневры — скажем, проверить, что указатель на строку в C++ не равен `NULL`, — но они часто не принимают дополнительных мер: не проверяют тот же параметр, чтобы удостовериться в наличии достаточного объема памяти для хранения строки максимально допустимого размера. Профилактическое программирование подразумевает выполнение всех возможных действий, чтобы избежать необходимости применения отладчика и вместо этого заставить код самостоятельно сообщать о проблемных участках. Отладчик — одна из самых больших в мире «черных дыр» для времени, и, чтобы ее избежать, нужны точные сообщения кода о любых отклонениях от идеала. При вводе любой строки кода остановитесь и подумайте, что вы предполагаете в хорошем развитии ситуации и как проверить, что именно такое состояние будет при каждом исполнении этой строки кода.

Все просто: ошибки не появляются в коде по волшебству. «Секрет» в том, что вы и я вносим их при написании кода и эти досадные ошибки могут появляться из тысяч источников. Они могут стать следствием таких критических проблем, как недостатки дизайна приложения, или таких простых, как опечатки. Хотя не-

которые ошибки легко устранить, есть и такие, которых не исправить без серьезных изменений в коде. Хорошо бы взвалить вину за ошибки в вашем коде на гремлинов, но следует признать, что именно вы и ваши коллеги вносите их туда. (Если вы читаете эту книгу, значит, в основном в ошибках виноваты ваши коллеги.)

Поскольку вы и другие разработчики отвечаете за ошибки в коде, возникает проблема поиска путей создания системы проверок и отчетов, позволяющей находить ошибки в процессе работы. Я всегда называл такой подход «доверяй, но проверяй» по знаменитой фразе Рональда Рейгана о том, как Соединенные Штаты собираются приводить в жизнь один из договоров об ограничении ядерных вооружений с бывшим Советским Союзом. Я верю, что мы с моими коллегами будем использовать код правильно. Однако для предотвращения ошибок я проверяю все: данные, передаваемые другими в мой код, внутренние операции в коде, любые допущения, сделанные в моем коде, данные, передаваемые моим кодом наружу, данные, возвращаемые от вызовов, сделанных в моем коде. Можно хоть что-то проверить — я проверяю. В столь навязчивой проверке нет ничего личного по отношению к коллегам, и у меня нет (серьезных) психических проблем. Я просто знаю, откуда появляются ошибки, и знаю, что если вы хотите обнаруживать ошибки как можно раньше, то ничего нельзя оставлять без проверки.

Прежде чем продолжить, подчеркну один закон моей философии разработки: ответственность за качество кода целиком лежит на инженерах-разработчиках, а не на тестировщиках, техническом персонале или менеджерах. Именно мы с вами пишем, реализуем и исправляем код, так что только мы можем принять значимые меры, чтобы сделать создаваемый нами код настолько безошибочным, насколько это возможно.

Одно из самых удивительных мнений, с которыми мне, как консультанту, доводилось сталкиваться, заключается в том, что разработчики должны только разрабатывать, а тестировщики — только тестировать. Основная проблема такого подхода в том, что разработчики пишут большие порции кода и отправляют их тестировщикам, весьма поверхностно убедившись в правильности работы. Не говоря уже о том, что ситуации, когда разработчики не отвечают за тестирование кода, приводят к несоблюдению сроков и низкому качеству продукта.

По-моему, разработчик — это тестировщик и разработчик: если разработчик не тратит хотя бы 40–50% времени разработки на тестирование своего кода, он не разрабатывает. Обязанность тестировщика — сосредоточиться на таких проблемах, как подгонка, тестирование на устойчивость и производительность. Тестировщик крайне редко должен сталкиваться с поиском причин краха. Крах кода напрямую относится к компетенции инженера-разработчика. Ключ тестирования, выполняемого разработчиком, — в блочном тестировании (unit test). Ваша задача — запустить максимально большой фрагмент кода, чтобы убедиться, что он не приводит к краху и соответствует установленным спецификациям и требованиям. Вооруженные результатами блочного тестирования модулей тестировщики могут сосредоточиться на проблемах интеграции и общесистемном тестировании. Мы подробно поговорим о тестировании модулей в разделе «Доверяй, но проверяй (Блочное тестирование)».

Assert, Assert, Assert и еще раз Assert

Надеюсь, большинство из вас уже знает, что такое утверждение (assertion), так как это самый важный инструмент профилактического программирования в арсенале отладочных средств. Для тех, кто не знаком с этим термином, дам краткое определение: утверждение объявляет, что в определенной точке программы должно выполняться некое условие. Если условие не выполняется, говорят, что утверждение нарушено. Утверждения используются в дополнение к обычной проверке на ошибки. Традиционно утверждения — это функции или макросы, выполняемые только в отладочных компоновках и отображающие окно с сообщением о том, что условие не выполнено. Я расширил определение утверждений, включив туда компилируемый по условию код, проверяющий условия и предположения, которые слишком сложно обработать в функции или макросе обычного утверждения. Утверждения — ключевой компонент профилактического программирования, потому что они помогают разработчикам и тестировщикам не только определить наличие, но и причины возникновения ошибки.

Даже если вы слышали об утверждениях и порой вставляете их в свой код, вы можете знать их недостаточно, чтобы применять эффективно. Разработчики не могут быть слишком жирными или слишком худыми — они не могут использовать слишком много утверждений. Метод, которому я всегда следовал, чтобы определить достаточное количество утверждений, прост: утверждений достаточно, если мои подчиненные жалуются на появление множества информационных окон о нарушении утверждений, как только они пытаются вызвать мой код, используя неверную информацию или предположения.

Достаточное количество утверждений даст вам основную информацию для выявления проблем на ранних стадиях. Без утверждений вы потратите массу времени на отладчик, продвигаясь в обратном направлении от сбоя в поисках того места, откуда все стало не так. Хорошее утверждение сообщит, где и почему нарушены условия. Хорошее утверждение при нарушении условия позволит вам перейти в отладчик, чтобы вы смогли увидеть полное состояние программы в точке сбоя. Плохое утверждение скажет, что что-то не так, но не объяснит что, где и почему.

Побочное преимущество от утверждений в том, что они служат прекрасной дополнительной документацией к вашему коду. Утверждения отражают ваши намерения. Я уверен, что вы прилагаете массу усилий, чтобы сохранять документацию соответствующей текущему положению, но я уверен и в том, что документация нескольких проектов испарилась. Хорошие утверждения позволяют сопровождающему разработчику вместо общих условий сбоя точно увидеть, какой диапазон значений параметра вы ожидаете или что, по вашим предположениям, может пойти не так в ходе нормального исполнения. Утверждения никогда не заменяют точных комментариев, но, используя их для прояснения загадочного «вот что я имел в виду, а совсем не то, что написано в документации», вы сэкономите кучу времени при работе над проектом.

Как и что утверждать

Мой стандартный ответ на вопрос «что утверждать?» — утверждайте все. Я бы с удовольствием заявил, что утверждение следует создать для каждой строки кода, но это нереальная, хоть и прекрасная цель. Следует утверждать каждое условие, поскольку именно оно может в будущем оказаться решением мерзкой ошибки. Не переживайте, что внесение слишком большого числа утверждений снизит производительность программы, — как правило, утверждения активны только в отладочных сборках, а созданные возможности по обнаружению ошибок с лихвой перевесят небольшую потерю производительности.

В утверждениях не следует менять переменные или состояния программы. Воспринимайте все данные, которые вы проверяете в утверждениях, как доступные только для чтения. Поскольку утверждения активны только в отладочных сборках, если вы изменяете данные, применяя утверждения, отладочные и финальные сборки будут работать по-разному, и отследить различия будет очень трудно.

В этом разделе я хочу сосредоточиться на том, как использовать утверждения и что утверждать. Я покажу это на примерах кодов. Замечу, что в этих примерах `Debug.Assert` — это утверждение .NET из пространства имен `System.Diagnostics`, а `ASSERT` — встроенный метод C++, который я представлю ниже.

Отладка: фронтовые очерки

Удар по карьере

Боевые действия

Давным-давно я работал в компании, у программного продукта которой были серьезные проблемы с надежностью. Как старший Windows-инженер этого чудовищного проекта, я обнаружил, что многие проблемы возникали от недостаточного понимания причин сбоев в обращениях к другим модулям. Я написал служебную записку, в которой советовал то же, что и в этой главе, рассказав участникам проекта, почему и когда им следовало использовать утверждения. Я обладал некоторыми полномочиями и внес это в критерии оценки кода, чтобы следить за правильным использованием утверждений.

Отправив записку, я ответил на несколько вопросов, возникших у людей по поводу утверждений, и думал, что все пришло в порядок. Три дня спустя мой начальник ворвался в мой кабинет и начал вопить, что я всех подвел, приказав отозвать служебную записку об утверждениях. Я был ошеломлен, и у нас начался весьма жаркий спор по поводу данных мною рекомендаций. Я не вполне понимал, что пытается сказать мой босс, но это было как-то связано с тем, что стабильность продукта упала еще сильнее. Пять минут мы кричали друг на друга, и я вызвался доказать начальнику что люди использовали утверждения неверно. Он вручил мне распечатку кода, выглядевшую примерно так:

```
BOOL DoSomeWork ( HMODULE * pModArray , int iCount , LPCTSTR szBuff )
{
    ASSERT ( if ( ( pModArray == NULL ) &&
```

см. след. стр.


```
        ( IsBadWritePtr ( pModArray ,  
                        ( sizeof ( HMODULE ) * iCount ) ) &&  
        ( iCount != 0 ) &&  
        ( szBuff != NULL ) ) )  
        {  
            return ( FALSE ) ;  
        }  
    ) ;  
    for ( int i = 0 ; i < iCount ; i++ )  
    {  
        pModArray[ i ] = m_pDataMods[ i ] ;  
    }  
    :  
}
```

Исход

Стоит отметить, что мы с боссом не очень-то ладили. Он считал меня зеленым юнцом, не стоящим и не знающим абсолютно ничего, а я его — невежественным тупицей, который без бутылки ни в чем не разберется. По мере чтения кода мои глаза все больше вылезали из орбит! Человек, писавший его, абсолютно не понимал предназначения утверждений и просто проходил код, заключая все обычные процедуры обработки ошибок в утверждения. Поскольку в финальных сборках утверждения отключаются, человек, писавший код, полностью удалял проверку на ошибки из финальных сборок!

К этому моменту я уже побагровел и орал во весь голос: «Того, кто это написал, нужно уволить! Не могу поверить, что у нас работает такой невероятный и полный @#!&*\$ идиот!» Мой начальник притих, выхватил распечатку из моих рук и тихо сказал: «Это мой код». Ударом по карьере стал мой истерический смех, понесшийся вдогонку ретирующемуся боссу.

Полученный опыт

Подчеркну: используйте утверждения как дополнение к обычным средствам обработки ошибок, а не вместо них. Если у вас есть утверждение, то рядом в коде должна быть какая-то процедура обработки ошибок. Что до моего босса, то когда несколько недель спустя я пришел к нему в кабинет увольняться, поскольку получил работу в компании получше, он был готов танцевать на столе и петь о том, что это был лучший день в его жизни.

Как утверждать

Первое правило: каждый элемент нужно проверять отдельно. Если вы проверяете несколько условий в одном утверждении, то не сможете узнать, какое именно вызвало сбой. В следующем примере я демонстрирую одну и ту же функцию с разными утверждениями. Хотя утверждение в первой функции обнаружит неверный параметр, оно не сможет сообщить, какое условие нарушено или даже какой из трех параметров неверен.

// Ошибочный способ написания утверждений. Какой параметр неверен?

```
BOOL GetPathItem ( int i , LPTSTR szItem , int iLen )
{
    ASSERT ( ( i > 0
                ( NULL != szItem
                ( ( iLen > 0 ) && ( iLen < MAX_PATH )
                ( FALSE == IsBadStringPtr ( szItem , iLen ) ) ) );
    :
}
```

// Правильный способ. Каждый параметр проверяется отдельно,
// так что вы сможете узнать, какой из них неверный.

```
BOOL GetPathItem ( int i , LPTSTR szItem , int iLen )
{
    ASSERT ( i > 0 );
    ASSERT ( NULL != szItem );
    ASSERT ( ( iLen > 0 ) && ( iLen < MAX_PATH ) );
    ASSERT ( FALSE == IsBadStringPtr ( szItem , iLen ) );
    :
}
```

Утверждая условие, старайтесь проверять его полностью. Например, если в .NET ваш метод принимает в виде параметра строку и вы ожидаете наличия в ней неких данных, то проверка на null опишет ошибочную ситуацию лишь частично.

// Пример частичной проверки ошибочной ситуации.

```
bool LookupCustomerName ( string CustomerName )
{
    Debug.Assert ( null != CustomerName , "null != CustomerName" );
    :
}
```

Ее можно описать полностью, добавив проверку на пустую строку.

// Пример полной проверки ошибочной ситуации.

```
bool LookupCustomerName ( string CustomerName )
{
    Debug.Assert ( null != CustomerName , "null != CustomerName" );
    Debug.Assert ( 0 != CustomerName.Length , "\"\" != CustomerName.Length" );
    :
}
```

Еще одна мера, которую я всегда принимаю, — проверка на особые значения. В следующем примере сначала приводится неверная проверка на положительные значения, а затем показано, как это сделать правильно:

// Пример плохо написанного утверждения: nCount должен быть положительным,
// но утверждение не срабатывает, если nCount отрицательный.

```
void UpdateListEntries ( int nCount )
{
    ASSERT ( nCount );
    :
}
```

```
// Правильное утверждение, проверяющее необходимое значение в явном виде.  
void UpdateListEntries ( int nCount )  
{  
    ASSERT ( nCount > 0 );  
    :  
}
```

Неверный пример проверяет только то, что `nCount` не равен 0, что составляет лишь половину нужной информации. Утверждения, в которых допустимые значения проверяются явно, сами себе служат документацией и, кроме того, гарантируют обнаружение неверных данных.

Что утверждать

Теперь мы можем перейти к вопросу о том, что утверждать. Если вы еще не догадались по приведенным до сих пор примерам, позвольте прояснить, что в первую очередь следует утверждать передающиеся в метод параметры. Утверждение параметров особенно важно для интерфейсов модулей и методов классов, вызываемых другими участниками вашей команды. Поскольку эти шлюзовые функции являются точками входа в ваш код, стоит убедиться в корректности всех параметров и предположений. В истории «Удар по карьере» я уже обращал ваше внимание на то, что утверждения ни в коем случае не должны вытеснять обычную обработку ошибок.

По мере продвижения в глубь модуля, параметры его закрытых методов будут требовать все меньше проверки в зависимости от места их происхождения. Во многом решение о том, допустимость каких параметров проверять, сводится к здравому смыслу. Не вредно проверять каждый параметр каждого метода, однако, если параметр передается в модуль извне и однажды уже полностью проверялся, делать это снова не обязательно. Но, утверждая каждый параметр в каждой функции, вы можете обнаружить внутренние ошибки модуля.

Я нахожусь строго между двумя крайностями. Определение подходящего для вас количества утверждений параметров потребует некоторого опыта. Получив представление о том, где в вашем коде обычно возникают проблемы, вы поймете, где и когда проверять внутренние параметры модуля. Я научился одной предосторожности: добавлять утверждения параметров при каждом нарушении работы моего кода из-за плохого параметра. Тогда ошибка не будет повторяться, так как ее обнаружит утверждение.

Еще одна обязательная для утверждения область — возвращаемые методами значения, поскольку они сообщают, была ли работа метода успешной. Одна из самых больших проблем, с которыми я сталкивался, отлаживая код других разработчиков, в том, что они просто вызывают методы, не проверяя возвращаемое значение. Как часто приходилось искать ошибку лишь затем, чтобы выяснить, что ранее в коде произошел сбой в каком-то методе, но никто не позаботился проверить возвращаемое им значение! Конечно, к тому времени, как вы обнаружите нарушителя, ошибка уже проявится, так что через какие-нибудь 20 минут программа обрушится или повредит данные. Правильно утверждая возвращаемые значения, вы по крайней мере узнаете о проблеме при ее появлении.

Напомню: я не выступаю за применение утверждений для каждого возможного сбоя. Некоторые сбои являются ожидаемыми, и вам следует соответствующим образом их обрабатывать. Инициация утверждения при каждом неудачном поиске в базе данных скорее всего заставит всех отключить утверждения в проекте. Учтите это и утверждайте возвращаемые значения там, где это важно. Обработка в программе корректных данных никогда не должна приводить к срабатыванию утверждения.

И, наконец, я рекомендую использовать утверждения, когда вам нужно проверить предположение. Так, если спецификации класса требуют 3 Мб дискового пространства, надо проверить это предположение утверждением условной компиляции внутри данного класса, чтобы убедиться, что вызывающие выполняют свою часть обязательств. Еще пример: если ваш код должен обращаться к базе данных, надо проверять, существуют ли в ней необходимые таблицы. Тогда вы сразу узнаете, в чем проблема, и не будете недоумевать, почему другие методы класса возвращают такие странные значения.

В обоих предыдущих примерах, как и в большинстве случаев утверждения предположений, нельзя проверять предположения в общем методе или макросе утверждения. В таких случаях поможет технология условной компиляции, которую я упомянул в предыдущем абзаце. Поскольку код, выполняемый в условной компиляции, работает с «живыми» данными, следует соблюдать особую осторожность, чтобы не изменить состояние программы. Чтобы избежать серьезных проблем, которые могут появиться от введения кода с побочными эффектами, я предпочитаю, если возможно, реализовывать такие типы утверждений отдельными методами. Таким образом вы избежите изменения локальных переменных внутри исходного метода. Кроме того, компилируемые по условию методы утверждений могут пригодиться в окне Watch, что вы увидите в главе 5, когда мы будем говорить об отладчике Microsoft Visual Studio .NET. Листинг 3-1 демонстрирует компилируемый по условию метод, который проверяет существование таблицы до начала интенсивной работы с данными. Заметьте: этот метод предполагает, что вы уже передали строку подключения и имеете полный доступ к базе данных. `AssertTableExists` подтверждает существование таблицы, чтобы вы могли опираться на это предположение, не получая странных сообщений о сбоях из глубин вашего кода.

Листинг 3-1. `AssertTableExists` проверяет существование таблицы

```
[Conditional("DEBUG")]
static public void AssertTableExists ( string ConnStr ,
                                     string TableName )
{
    SqlConnection Conn = new SqlConnection ( ConnStr );

    StringBuilder sBuildCmd = new StringBuilder ( );

    sBuildCmd.Append ( "select * from dbo.sysobjects where " );
    sBuildCmd.Append ( "id = object_id(' " );
    sBuildCmd.Append ( TableName );
    sBuildCmd.Append ( "')" );
```

см. след. стр.

```
// Выполняем команду.
SqlCommand Cmd = new SqlCommand ( sBuildCmd.ToString ( ) , Conn );

try
{
    // Открываем базу данных.
    Conn.Open ( ) ;

    // Создаем набор данных для заполнения.
    DataSet TableSet = new DataSet ( ) ;

    // Создаем адаптер данных.
    SqlDataAdapter TableDataAdapter = new SqlDataAdapter ( ) ;

    // Устанавливаем команду для выборки.
    TableDataAdapter.SelectCommand = Cmd ;

    // Заполняем набор данных из адаптера.
    TableDataAdapter.Fill ( TableSet ) ;

    // Если что-нибудь появилось, таблица существует.
    if ( 0 == TableSet.Tables[0].Rows.Count )
    {
        String sMsg = "Table : '" + TableName +
                      "' does not exist!\r\n" ;
        Debug.Assert ( false , sMsg ) ;
    }
}
catch ( Exception e )
{
    Debug.Assert ( false , e.Message ) ;
}
finally
{
    Conn.Close ( ) ;
}
}
```

Прежде чем описать специфические проблемы различных утверждений для .NET и машинного кода, хочу показать пример того, как я обрабатываю утверждения. В листинге 3-2 показана функция `StartDebugging` отладчика машинного кода из главы 4. Этот код — точка перехода из одного модуля в другой, так что он демонстрирует все утверждения, о которых говорилось в этом разделе. Я выбрал метод C++, потому что в «родном» C++ всплывает гораздо больше проблем и поэтому надо утверждать больше условий. Я рассмотрю некоторые проблемы этого примера ниже в разделе «Утверждения в приложениях C++».

Листинг 3-2. Пример исчерпывающего утверждения

```

HANDLE DEBUGINTERFACE_DLLINTERFACE __stdcall
StartDebugging ( LPCTSTR          szDebuggee          ,
                  LPCTSTR          szCmdLine           ,
                  LPDWORD          lpPID               ,
                  CDebugBaseUser * pUserClass          ,
                  LPHANDLE         lpDebugSyncEvents   )
{
    // Утверждаем параметры.
    ASSERT ( FALSE == IsBadStringPtr ( szDebuggee , MAX_PATH ) );
    ASSERT ( FALSE == IsBadStringPtr ( szCmdLine , MAX_PATH ) );
    ASSERT ( FALSE == IsBadWritePtr ( lpPID , sizeof ( DWORD ) ) );
    ASSERT ( FALSE == IsBadReadPtr ( pUserClass ,
                                     sizeof ( CDebugBaseUser * ) ) );
    ASSERT ( FALSE == IsBadWritePtr ( lpDebugSyncEvents ,
                                     sizeof ( HANDLE ) *
                                     NUM_DEBUGEVENTS ) );

    // Проверяем их существование.
    if ( ( TRUE == IsBadStringPtr ( szDebuggee , MAX_PATH ) ) ||
        ( TRUE == IsBadStringPtr ( szCmdLine , MAX_PATH ) ) ||
        ( TRUE == IsBadWritePtr ( lpPID , sizeof ( DWORD ) ) ) ||
        ( TRUE == IsBadReadPtr ( pUserClass ,
                                 sizeof ( CDebugBaseUser * ) ) ) ||
        ( TRUE == IsBadWritePtr ( lpDebugSyncEvents ,
                                 sizeof ( HANDLE ) *
                                 NUM_DEBUGEVENTS ) ) )
    {
        SetLastError ( ERROR_INVALID_PARAMETER );
        return ( INVALID_HANDLE_VALUE );
    }

    // Строка для события стартового подтверждения.
    TCHAR szStartAck [ MAX_PATH ] = _T ( "\\0" );

    // Загружаем строку для стартового подтверждения.
    if ( 0 == LoadString ( GetDllHandle ( ) ,
                          IDS_DBGEVENTINIT ,
                          szStartAck ,
                          MAX_PATH ) )
    {
        ASSERT ( !"LoadString IDS_DBGEVENTINIT failed!" );
        return ( INVALID_HANDLE_VALUE );
    }

    // Описатель стартового подтверждения, которого будет ждать
    // эта функция, пока не запустится отладочный поток.
    HANDLE hStartAck = NULL ;

    // Создаем событие стартового подтверждения.

```

см. след. стр.

```

hStartAck = CreateEvent ( NULL , // Безопасность по умолчанию.
                        TRUE , // Событие с ручным сбросом.
                        FALSE , // Начальное состояние=Not signaled.
                        szStartAck ) ; // Имя события.

ASSERT ( NULL != hStartAck ) ;
if ( NULL == hStartAck )
{
    return ( INVALID_HANDLE_VALUE ) ;
}

// Связываем параметры.
THREADPARAMS stParams ;
stParams.lpPID = lpPID ;
stParams.pUserClass = pUserClass ;
stParams.szDebuggee = szDebuggee ;
stParams.szCmdLine = szCmdLine ;

// Описатель для отладочного потока.
HANDLE hDbgThread = INVALID_HANDLE_VALUE ;

// Пробуем создать поток.
UINT dwTID = 0 ;
hDbgThread = (HANDLE)_beginthreadex ( NULL ,
                                     0 ,
                                     DebugThread ,
                                     &stParams ,
                                     0 ,
                                     &dwTID ) ;

ASSERT ( INVALID_HANDLE_VALUE != hDbgThread ) ;
if ( INVALID_HANDLE_VALUE == hDbgThread )
{
    VERIFY ( CloseHandle ( hStartAck ) ) ;
    return ( INVALID_HANDLE_VALUE ) ;
}

// Ждем, пока отладочный поток не придет в норму и продолжаем.
DWORD dwRet = ::WaitForSingleObject ( hStartAck , INFINITE ) ;
ASSERT ( WAIT_OBJECT_0 == dwRet ) ;
if ( WAIT_OBJECT_0 != dwRet )
{
    VERIFY ( CloseHandle ( hStartAck ) ) ;
    VERIFY ( CloseHandle ( hDbgThread ) ) ;
    return ( INVALID_HANDLE_VALUE ) ;
}

// Избавляемся от описателя подтверждения.
VERIFY ( CloseHandle ( hStartAck ) ) ;

// Проверяем, что отладочный поток еще выполняется. Если это не так,
// отлаживаемое приложение, вероятно, не может запуститься.

```



```

DWORD dwExitCode = ~STILL_ACTIVE ;
if ( FALSE == GetExitCodeThread ( hDbgThread , &dwExitCode ) )
{
    ASSERT ( !"GetExitCodeThread failed!" ) ;
    VERIFY ( CloseHandle ( hDbgThread ) ) ;
    return ( INVALID_HANDLE_VALUE ) ;
}
ASSERT ( STILL_ACTIVE == dwExitCode ) ;
if ( STILL_ACTIVE != dwExitCode )
{
    VERIFY ( CloseHandle ( hDbgThread ) ) ;
    return ( INVALID_HANDLE_VALUE ) ;
}

// Создаем события синхронизации, чтобы главный поток
// мог сообщить отладочному циклу, что делать.
BOOL bCreateDbgSyncEvts =
    CreateDebugSyncEvents ( lpDebugSyncEvents , *lpPID ) ;
ASSERT ( TRUE == bCreateDbgSyncEvts ) ;
if ( FALSE == bCreateDbgSyncEvts )
{
    // Это серьезная проблема. Отладочный поток выполняется, но
    // я не смог создать события синхронизации, необходимые потоку
    // пользовательского интерфейса для управления отладочным потоком.
    // Мое единственное мнение – выходить. Я закрою отладочный поток
    // и просто выйду. Больше я ничего не могу сделать.
    TRACE ( "StartDebugging : CreateDebugSyncEvents failed\n" ) ;
    VERIFY ( TerminateThread ( hDbgThread , (DWORD)-1 ) ) ;
    VERIFY ( CloseHandle ( hDbgThread ) ) ;
    return ( INVALID_HANDLE_VALUE ) ;
}

// Просто на случай, если кто-то изменит функцию
// и не сможет правильно указать возвращаемое значение.
ASSERT ( INVALID_HANDLE_VALUE != hDbgThread ) ;

// Жизнь прекрасна!
return ( hDbgThread ) ;
}

```

Утверждения в .NET Windows Forms или консольных приложениях

Перед тем как перейти к мелким подробностям утверждений .NET, хочу отметить одну ключевую ошибку, которую я встречал практически во всех кодах .NET, особенно во многих примерах, из которых разработчики берут код для создания своих приложений. Все забывают, что можно передать в объектном параметре значение null. Даже когда разработчики используют утверждения, код выглядит примерно так:

```
void DoSomeWork ( string TheName )  
{  
    Debug.Assert ( TheName.Length > 0 ) ;  
:  
}
```

Если `TheName` имеет значение `null`, то вместо срабатывания утверждения вызов свойства `Length` приводит к исключению `System.NullReferenceException`, тут же обрушивая ваше приложение. Это тот ужасный случай, когда утверждение вызывает нежелательный побочный эффект, нарушая основное правило утверждений. И, разумеется, отсюда следует, что если разработчики не проверяют наличие пустых объектов в утверждениях, то не делают этого и при обычной проверке параметров. Окажите себе огромную услугу: начните проверять объекты на `null`.

То, что приложения .NET не должны заботиться об указателях и блоках памяти означает, что по крайней мере 60% утверждений, использовавшихся нами в дни C++, ушли в прошлое. В сфере утверждений команда .NET добавила в пространство имен `System.Diagnostics` два объекта — `Debug` и `Trace`, активных, только если в компиляции приложения вы определили `DEBUG` или `TRACE` соответственно. Оба эти определения могут быть указаны в диалоговом окне `Property Pages` проекта. Как вы видели, метод `Assert` обрабатывает утверждения в .NET. Довольно интересно, что и `Debug` и `Trace` обладают похожими методами, включая `Assert`. Мне кажется, что наличие двух возможных утверждений, компилирующихся по разным условиям, может сбить с толку. Следовательно, поскольку утверждения должны быть активны только в отладочных сборках, для утверждений я использую только `Debug.Assert`. Это позволяет избежать сюрпризов от конечных пользователей, звонящих мне с вопросами о странных диалоговых окнах или сообщениях о том, что что-то пошло не так. Я настоятельно рекомендую вам делать то же самое, внося свой вклад в целостность мира утверждений.

Есть три перегруженных метода `Assert`. Все они принимают значение булевского типа в качестве первого или единственного параметра, и, если оно равно `false`, иницируется утверждение. Как видно из предыдущих примеров, где я использовал `Debug.Assert`, один из методов принимает второй параметр типа `string`, который отображается в выдаваемом сообщении. Последний перегруженный метод `Assert` принимает третий параметр типа `string`, предоставляющий еще больше данных при срабатывании утверждения. По моему опыту случай с двумя параметрами — самый простой для использования, так как я просто копирую условие, проверяемое в первом параметре, и вставляю его как строку. Конечно, теперь, когда нужное в утверждении условное выражение находится в кавычках, проверяя правильность кода, следует контролировать, чтобы строковое значение всегда совпадало с реальным условием. Следующий код демонстрирует все три метода `Assert` в действии.

```
Debug.Assert ( i > 3 )  
Debug.Assert ( i > 3 , "i > 3" )  
Debug.Assert ( i > 3 , "i > 3" , "This means I got a bad parameter")
```

Объект `Debug` в .NET интересен тем, что позволяет представлять результат разными способами. Исходящая информация от объекта `Debug` (и соответственно объекта `Trace`) проходит через другой объект — `TraceListener`. Классы-потомки

`TraceListener` добавляются в свойство объекта `Debug` — набор `Listener`. Прелесть такого подхода в том, что при каждом нарушении утверждения объект `Debug` перебирает набор `Listener` и по очереди вызывает каждый объект `TraceListener`. Благодаря этой удобной функциональности даже при появлении новых усовершенствованных способов уведомления для утверждений вам не придется вносить серьезных изменений в код, чтобы задействовать их преимущества. Более того, в следующем разделе я покажу, как добавить новые объекты `TraceListener`, вообще не изменяя код, что обеспечивает превосходную расширяемость!

Используемый по умолчанию объект `TraceListener` называется `DefaultTraceListener`. Он направляет исходящую информацию в два разных места, самым заметным из которых является диалоговое окно утверждения (рис. 3-1). Как видите, большая его часть занята информацией из стека и типами параметров. Также указаны источник и строка для каждого элемента. В верхних строках окна выводятся строковые значения, переданные вами в `Debug.Assert`. На рис. 3-1 я в качестве второго параметра передал в `Debug.Assert` строку «`Debug.Assert assertion`».

Результат нажатия каждой кнопки описан в строке заголовка информационного окна. Единственная интересная клавиша — `Retry`. Если вы исполняете код в отладчике, вы просто переходите в отладчик на строку, следующую за утверждением. Если вы не в отладчике, щелчок `Retry` инициирует специальное исключение и запускает селектор отладчика по требованию, позволяющий выбрать зарегистрированный отладчик для отладки утверждения.

В дополнение к выводу в информационном окне `Debug.Assert` также направляет всю исходящую информацию через `OutputDebugString`, поэтому ее получает подключенный отладчик. Эта информация предоставляется в схожем формате, который показан в следующем коде. Поскольку `DefaultTraceListener` выполняет вывод через `OutputDebugString`, вы можете воспользоваться прекрасной программой Марка Руссиновича (Mark Russinovich) `DebugView` (www.sysinternals.com), чтобы просмотреть его, не находясь в отладчике. Ниже я расскажу об этом подробнее.

```
-- DEBUG ASSERTION FAILED --  
-- Assert Short Message --  
Debug.Assert assertion  
-- Assert Long Message --
```

```
at HappyAppy.Fum() d:\asserterexample\asserter.cs(15)  
at HappyAppy.Fo(StringBuilder sb) d:\asserterexample\asserter.cs(20)  
at HappyAppy.Fi(IntPtr p) d:\asserterexample\asserter.cs(24)  
at HappyAppy.Fee(String Blah) d:\asserterexample\asserter.cs(29)  
at HappyAppy.Baz(Double d) d:\asserterexample\asserter.cs(34)  
at HappyAppy.Bar(Object o) d:\asserterexample\asserter.cs(39)  
at HappyAppy.Foo(Int32 i) d:\asserterexample\asserter.cs(46)  
at HappyAppy.Main() d:\\asserterexample\asserter.cs(76)
```

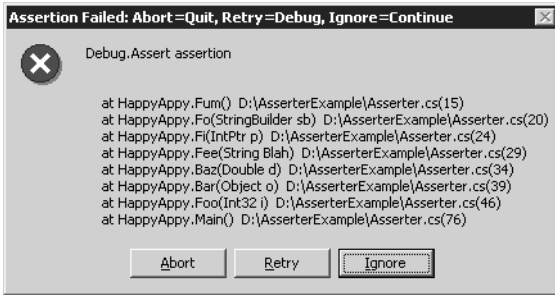


Рис. 3-1. Информационное окно `DefaultTraceListener`

Обладая информацией, предоставляемой `Debug.Assert`, вы никогда больше не будете раздумывать, почему сработало утверждение! .NET Framework также предоставляет два других объекта `TraceListener`. Для записи исходящей информации в текстовый файл используйте класс `TextWriterTraceListener`, а для записи ее в журнал событий — класс `EventLogTraceListener`. К сожалению, классы `TextWriterTraceListener` и `EventLogTraceListener` практически бесполезны, потому что записывают только поля сообщений ваших утверждений и не включают информацию о стеке. Хорошая новость в том, что реализовать собственные объекты `TraceListener` несложно, поэтому в рамках `BugslayerUtil.NET.DLL` я пошел дальше и написал для вас исправленные версии `TextWriterTraceListener` и `EventLogTraceListener`: `BugslayerTextWriterTraceListener` и `BugslayerEventLogTraceListener` соответственно.

И `BugslayerTextWriterTraceListener`, и `BugslayerEventLogTraceListener` — вполне заурядные классы. `BugslayerTextWriterTraceListener` наследует напрямую от `TextWriterTraceListener`, и все, что он делает, — переопределяет метод `Fail`, который `Debug.Assert` вызывает для вывода информации. Помните, что при использовании `BugslayerTextWriterTraceListener` или `TextWriterTraceListener` соответствующий текстовый файл с исходящей информацией не сбрасывается на диск, если не задать `true` атрибуту `autoflush` элемента `trace` в конфигурационном файле приложения, не вызвать явно `Close` для потока или файла или не задать `Debug.AutoFlush` значение `true`, чтобы каждая запись автоматически вызывала сброс на диск. По каким-то причинам класс `EventLogTraceListener` является закрытым, поэтому я не мог наследовать от него напрямую и создал потомок прямо от абстрактного класса `TraceListener`. Однако я все-таки получил информацию о стеке весьма интересным способом. Как показано ниже, стандартный класс `StackTrace`, предоставляемый .NET, позволяет в любой момент легко получить информацию о стеке.

```
StackTrace StkTrc = new StackTrace ( ) ;
```

В сравнении с действиями, которые надо было выполнять в машинном коде, чтобы получить такую информацию, способ, предоставляемый .NET, служит прекрасным примером того, как .NET облегчает вашу жизнь. `StackTrace` возвращает набор объектов `StackFrame`, представляющих стек. Просмотрев документацию на `StackFrame`, вы увидите, что в нем есть все виды интересных методов для получения строки и номера источника. Объект `StackTrace` содержит метод `ToString`, и я был абсолютно уверен, что через него как-то можно добавлять источник и строку в итоговую информацию о стеке. Увы, я ошибался. Поэтому мне пришлось 30

минут писать и тестировать класс `BugslayerStackTrace`, наследующий от `StackTrace` и переопределяющий `ToString`, чтобы иметь возможность добавить информацию об источнике и строке к каждому методу. В листинге 3-3 показаны два метода из `BugslayerStackTrace`, выполняющие эти действия.

Листинг 3-3. `BugslayerStackTrace`, собирающий полную информацию о стеке, в том числе сведения об источнике и строке

```

/// <summary>
/// Создает читаемое представление информации о стеке.
/// </summary>
/// <returns>
/// Читаемое представление информации о стеке.
/// </returns>
public override string ToString ( )
{
    // Обновляем StringBuilder для хранения всего необходимого.
    StringBuilder StrBld = new StringBuilder ( ) ;

    // Первое, что надо внести, – перевод строки.
    StrBld.Append ( DefaultLineEnd ) ;

    // Зациклить и сделать! Здесь нельзя использовать foreach,
    // так как StackTrace не наследует от IEnumerable.
    for ( int i = 0 ; i < FrameCount ; i++ )
    {
        StackFrame StkFrame = GetFrame ( i ) ;
        if ( null != StkFrame )
        {
            BuildFrameInfo ( StrBld , StkFrame ) ;
        }
    }
    return ( StrBld.ToString ( ) ) ;
}

/*//////////////////////////////////////
// Закрытые методы
//////////////////////////////////////*/

/// <summary>
/// Выполняет мелкую работу по преобразованию фрейма
/// в строку и внесению его в StringBuilder.
/// </summary>
/// <param name="StrBld">
/// StringBuilder для внесения результатов.
/// </param>
/// <param name="StkFrame">
/// Фрейм стека для преобразования.
/// </param>
private void BuildFrameInfo ( StringBuilder StrBld ,

```

см. след. стр.

```
StackFrame StkFrame )
{
    // Получаем метод через механизм отражения.
    MethodBase Meth = StkFrame.GetMethod ( ) ;

    // Если ничего не получили, выходим отсюда.
    if ( null == Meth )
    {
        return ;
    }

    // Присваиваем метод.
    String StrMethName = Meth.ReflectedType.Name ;

    // Вносим отступ функции (function indent), если он есть.
    if ( null != FunctionIndent )
    {
        StrBld.Append ( FunctionIndent ) ;
    }

    // Получаем тип и имя класса.
    StrBld.Append ( StrMethName ) ;
    StrBld.Append ( "." ) ;
    StrBld.Append ( Meth.Name ) ;
    StrBld.Append ( "(" ) ;

    // Вносим параметры, включая все их имена.
    ParameterInfo[] Params = Meth.GetParameters ( ) ;
    for ( int i = 0 ; i < Params.Length ; i++ )
    {
        ParameterInfo CurrParam = Params[ i ] ;
        StrBld.Append ( CurrParam.ParameterType.Name ) ;
        StrBld.Append ( " " ) ;
        StrBld.Append ( CurrParam.Name ) ;
        if ( i != ( Params.Length - 1 ) )
        {
            StrBld.Append ( ", " ) ;
        }
    }

    // Закрываем список параметров.
    StrBld.Append ( ")" ) ;

    // Получаем источник и строку, только если они есть.
    if ( null != StkFrame.GetFileName ( ) )
    {
        // Мне надо определять источник? Если да, то нужно
        // вставить в конце разрыв строки и отступ.
        if ( null != SourceIndentString )
        {

```

```

        StrBld.Append ( LineEnd ) ;
        StrBld.Append ( SourceIndentString ) ;
    }
    else
    {
        // Просто добавляем пробел.
        StrBld.Append ( ' ' ) ;
    }

    // Здесь получаем имя файла и строку с проблемой.
    StrBld.Append ( StkFrame.GetFileName ( ) ) ;
    StrBld.Append ( "(" ) ;
    StrBld.Append ( StkFrame.GetFileLineNumber().ToString());
    StrBld.Append ( ")" ) ;
}
// Всегда добавляйте перевод строки.
StrBld.Append ( LineEnd ) ;
}

```

Теперь, когда у вас есть другие классы `TraceListener`, которые стоит добавить в набор `Listeners`, мы в коде можем добавлять и удалять объекты `TraceListener`. Как и в любом наборе .NET, чтобы добавить объект в набор, вызовите метод `Add`, а чтобы избавиться от объекта — метод `Remove`. Стандартный `TraceListener` называется «Default». Вот как добавить `BugslayerTextWriterTraceListener` и удалить `DefaultTraceListener`:

```

Stream AssertFile = File.Create ( "BSUNBTWTLTest.txt" ) ;

BugslayerTextWriterTraceListener tListener =
    new BugslayerTextWriterTraceListener ( AssertFile ) ;

Debug.Listeners.Add ( tListener ) ;

Debug.Listeners.Remove ( "Default" ) ;

```

Управление объектом `TraceListener` через файлы конфигурации

Если вы разрабатываете консольные приложения и приложения Windows Forms, то по большей части `DefaultTraceListener` должен удовлетворить все ваши потребности. Однако появляющееся время от времени информационное окно может нарушить работу любых автоматизированных тестов. Или, может быть, вы используете компонент сторонних производителей в службе Win32, и его отладочная сборка правильно использует `Debug.Assert`. В обоих случаях вам потребуется отключить информационное окно, вызываемое `DefaultTraceListener`. Можно добавить код для удаления объекта `DefaultTraceListener`, но его можно удалить и не прикасаясь к коду.

Любому двоичному коду .NET может быть сопоставлен внешний конфигурационный файл XML. Этот файл располагается в том же каталоге, что и двоичный файл, и имеет такое же имя с добавленным в конце словом `.CONFIG`. Например, конфигурационный файл для `FOO.EXE` называется `FOO.EXE.CONFIG`. Можно лег-

ко добавить конфигурационный файл к проекту, добавив новый XML-файл с именем APP.CONFIG. Этот файл будет автоматически скопирован в каталог конечных файлов и назван в соответствии с именем двоичного файла.

Элемент `assert`, расположенный внутри `system.diagnostics` в конфигурационном файле XML, имеет два атрибута. Если задать `false` первому атрибуту — `assertuienabled`, .NET не будет отображать информационные окна, но исходящая информация по-прежнему будет направляться через `OutputDebugString`. Второй атрибут — `logfile` — позволяет указать файл, в который следует записывать любой вывод утверждений. Интересно что при указании файла в атрибуте `logfile`, в этом файле также появятся все операторы трассировки, о которых я расскажу ниже. В следующем отрывке показан минимальный конфигурационный файл. Он демонстрирует, как просто отключить информационные окна утверждений. Не забудьте: главный конфигурационный файл MACHINE.CONFIG включает такие же параметры, что и обычные конфигурационные файлы, так что с их помощью вы вправе отключить информационные окна на всей машине.

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <system.diagnostics>
    <assert assertuienabled="false"
           logfile="tracelog.txt" />
  </system.diagnostics>
</configuration>
```

Как я уже отмечал, можно добавлять и удалять приемники информации (`listeners`), не затрагивая код, и, как вы, вероятно, догадались, это как-то связано с конфигурационным файлом. В документации он выглядит вполне очевидным, но на момент написания этой книги документация содержала ошибки. Экспериментально я выявил все нужные приемы для корректного управления приемниками без изменений кода.

Все действия выполняются над элементом `trace` конфигурационного файла. Этот элемент содержит один очень важный необязательный атрибут, которому всегда следует задавать `true`, — `autoFlush`. Сделав так, вы предписываете сбрасывать исходящий буфер на диск при каждой операции записи. В противном случае вам придется добавлять в код вызовы для сброса информации.

Внутри `trace` содержится элемент `listener`, через который добавляются и удаляются объекты `TraceListener`. Удалить объект `TraceListener` очень просто. Укажите элемент `remove` и задайте его атрибуту `name` строковое имя нужного объекта `TraceListener`. Ниже приведен полный конфигурационный файл, удаляющий `DefaultTraceListener`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <system.diagnostics>
    <trace autoFlush="true" indentSize="0">
      <listeners>
        <remove name="Default" />
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

Элемент `add` содержит два необходимых атрибута: `name` представляет строку, определяющую имя объекта `TraceListener` в том виде, в котором оно помещается в свойство `TraceListener.Name`, а `type` вызывает замешательство, и я объясню почему. В документации показано только добавление типа, находящегося в глобальном кэше сборок (GAC), и сказано, что добавление собственного приемника гораздо сложнее, чем нужно. Один необязательный атрибут — `initializeData` — представляет строку, передаваемую конструктору объекта `TraceListener`.

Чтобы добавить объект `TraceListener` из GAC, в элементе `type` надо только полностью указать класс объекта `TraceListener`. Согласно документации для добавления объекта `TraceListener`, не находящегося в GAC, вам придется иметь дело со всей атрибутикой вроде региональных параметров (`culture`) и маркеров открытых ключей (`public key tokens`). К счастью, все, что нужно сделать, — это просто указать полностью класс, добавить запятую и имя сборки. Во избежание инициации исключения `System.Configuration.ConfigurationException` не добавляйте запятую и имя класса. Вот как правильно добавить глобальный класс `TextWriterTraceListener`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <system.diagnostics>
    <trace autoflush="true" indentsize="0">
      <listeners>
        <add name="CorrectWay"
              type="System.Diagnostics.TextWriterTraceListener"
              initializeData="TextLog.log"/>
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

Чтобы добавить объекты `TraceListener`, не находящиеся в GAC, надо разместить сборку, содержащую потомки класса `TraceListener`, в одном каталоге с двоичным файлом. Испробовав все комбинации путей и параметров конфигурации, я выяснил, что включить сборку из другого каталога через конфигурационный файл нельзя. Добавляя потомок класса `TraceListener`, поставьте запятую и имя сборки. Вот как добавить `BugslayerTextWriterTraceListener` из `BugslayerUtil.NET.DLL`:

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <system.diagnostics>
    <trace autoflush="true" indentsize="0">
      <listeners>
        <add name="AGoodListener"
              type=
"Wintellect.BugslayerTextWriterTraceListener,BugslayerUtil.NET"
              initializeData="BSUTWTL.log"/>
      </listeners>
    </trace>
  </system.diagnostics>
</configuration>
```

Утверждения в приложениях ASP.NET и Web-сервисах XML

Я действительно рад видеть платформу для разработки, в которую изначально заложены идеи по обработке утверждений. Пространство имен `System.Diagnostics` содержит все эти полезные классы, квинтэссенция которых — `Debug`. Как и большинство из вас, я начал изучать .NET с создания консольных приложений и приложений Windows Forms, поскольку в то время они проще всего уживались в моей голове. Когда я перешел к ASP.NET, я уже использовал `Debug.Assert` и подумал, что Microsoft правильно поступила, избавившись от информационных окон. Безусловно, они поняли, что при работе в ASP.NET мне потребуется возможность при срабатывании утверждения перейти в отладчик. Представьте мое удивление, когда я инициировал утверждение и ничего не прекратилось! Я увидел обычный вывод утверждения в окне Output отладчика, но не увидел вызовов `OutputDebugString` с информацией об утверждении. Поскольку Web-сервисы XML в .NET по существу являются приложениями ASP.NET без пользовательского интерфейса, я проделал то же самое с Web-сервисом и получил те же результаты. (Далее в этом разделе в термине ASP.NET я буду совмещать ASP.NET и Web-сервисы XML.) Поразительно! Это означало, что в ASP.NET нет настоящих утверждений! А без них можно и не программировать! Единственная хорошая новость в том, что в приложениях ASP.NET `DefaultTraceListener` не отображает обычное информационное окно.

Без утверждений я чувствовал себя голым и знал, что с этим надо что-то делать. Подумав, не создать ли новый объект для утверждений, я решил, что правильнее всего будет держаться `Debug.Assert` как единственного способа обработки утверждений. Это позволяло мне решить сразу несколько ключевых проблем. Первая заключалась в наличии единого способа работы с утверждениями для всей платформы .NET — я совсем не хотел беспокоиться о том, будет ли код запущен в Windows Forms или ASP.NET, и применять неверные утверждения. Вторая проблема касалась библиотек сторонних производителей, в которых имеется `Debug.Assert`: как их использовать, чтобы их утверждения появлялись в том же месте, где и все другие.

Третья проблема состояла в том, чтобы сделать обращение к библиотеке утверждений максимально безболезненным. Написав массу утилит, я понял важность легкой интеграции библиотеки утверждений в приложение. Последняя проблема, которую я хотел решить, заключалась в наличии серверного элемента управления, позволяющего легко видеть утверждения на странице. Весь код находится в `BugslayerUtil.NET.DLL`, так что вы можете открыть этот проект с тестовой программой `BSUNAssertTest`, расположенной в подкаталоге `Test` каталога `BugslayerUtil.NET`. Прежде чем открыть проект, не забудьте создать виртуальный каталог в Microsoft Internet Information Services (IIS), ссылающийся на каталог `BSUNAssertTest`.

Проблемы, которые я хотел решить, указывали на создание специального класса, наследуемого от `TraceListener`. Через секунду я расскажу об этом коде, но независимо от того, насколько классным получился бы `TraceListener`, мне нужен был способ подключить свой объект `TraceListener` и удалить `DefaultTraceListener`. Как бы там ни было, это требовало изменений в коде с вашей стороны, потому что мне нужно выполнить некоторый код. Чтобы упростить применение утверждений и обеспечить максимально ранний вызов библиотеки утверждений, я использовал класс, наследуемый от `System.Web.HttpApplication`, так как его конструктор и метод `Init`

вызываются в приложении ASP.NET в первую очередь. Первым шагом на пути к нирване утверждений является наследование от вашего класса `Global` из `Global.ASAX.cs` (или `Global.ASAX.vb`) с использованием моего класса `AssertHttpApplication`. Это позволит правильно подключить мой `ASPTraceListener` и поместить в ссылку на него в отделе состояния приложения в разделе «`ASPTraceListener`», так что вы сможете в ходе работы изменять параметры вывода. Если все, что вам нужно в приложении, — это возможность остановить его при срабатывании утверждения, то больше от вас ничего не потребуется.

Для вывода утверждений на страницу я написал очень простой элемент управления, который вполне логично называется `AssertControl`. Чтобы добавить его на панель инструментов, щелкните правой кнопкой вкладку `Web Forms` и выберите из контекстного меню команду `Add/Remove Items`. В диалоговом окне `Customize Toolbox` перейдите на вкладку `.NET`, щелкните кнопку `Browse` и в окне `File Open` перейдите к `BugslayerUtil.NET.DLL`. Теперь вы можете просто перетаскивать `AssertControl` на любую страницу, в которой вам потребуются утверждения. Вам не придется прописывать элемент управления в вашем коде, потому что класс `ASPTraceListener` обнаружит его на странице и создаст соответствующий вывод. `AssertControl` будет найден, даже если он вложен в другой элемент управления. Если при обработке страницы на сервере ни одно утверждение не инициировалось, `AssertControl` не выводит ничего. Иначе он отображает те же сообщения утверждений и информацию о стеке, что выводятся в `Windows-` или консольных приложениях. Поскольку на странице могут инициироваться несколько утверждений, `AssertControl` отображает их все. На рис. 3-2 показана страница `BSUNAssertTest` после инициации утверждения. Текст в нижней части страницы — это вывод `AssertControl`.

Вся работа выполняется в классе `ASPTraceListener`, большая часть которого представлена в листинге 3-4. Чтобы объединить в себе все необходимое, `ASPTraceListener` включает несколько свойств, позволяющих перенаправлять и изменять вывод в процессе работы (табл. 3-1).

Табл. 3-1. Свойства вывода и управления `ASPTraceListener`

Свойство	Значение по умолчанию	Описание
<code>ShowDebugLog</code>	<code>true</code>	Показывает вывод в подключенном отладчике.
<code>ShowOutputDebugString</code>	<code>false</code>	Показывает вывод через <code>OutputDebugString</code> .
<code>EventSource</code>	<code>null/Nothing</code>	Имя источника события для записи вывода в журнал событий. Внутри <code>BugslayerUtil.NET.DLL</code> не получают разрешения и не выполняются проверки безопасности для доступа к журналу событий. Перед установкой <code>EventSource</code> вам придется запросить разрешения.
<code>Writer</code>	<code>null/Nothing</code>	Объект <code>TextWriter</code> для записи вывода в файл.
<code>LaunchDebuggerOnAssert</code>	<code>true</code>	Если подключен отладчик, он сразу останавливает выполнение при инициации утверждения.

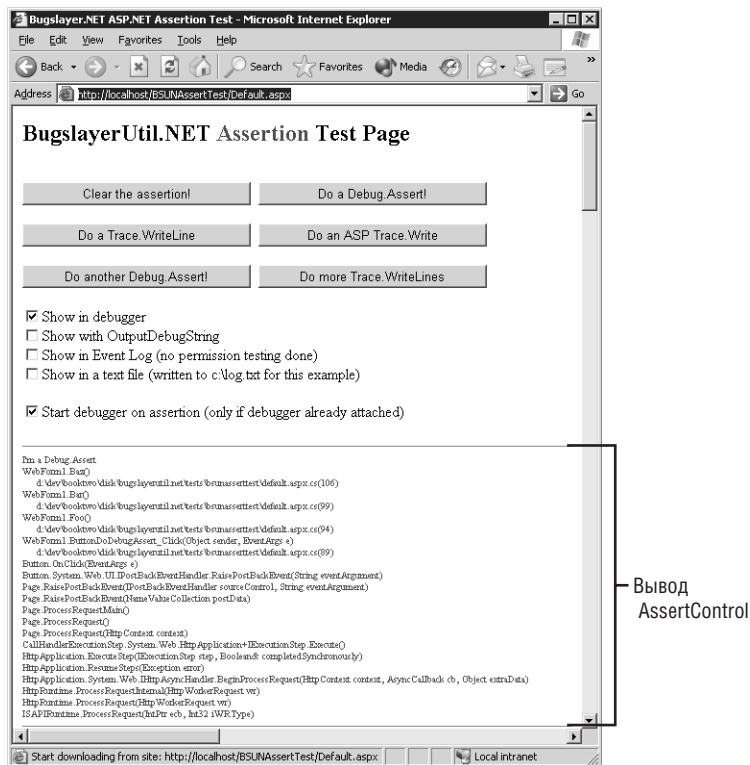


Рис. 3-2. Приложение ASP.NET, отображающее утверждение через AssertControl

Всю работу по выводу информации утверждения, которая включает поиск элементов управления утверждений на странице, выполняет метод `ASPTraceListener.HandleOutput`, показанный в листинге 3-4. Моя первая попытка создания метода `HandleOutput` была гораздо запутаннее. Я мог получить текущий `IHttpHandler` для текущего HTTP-запроса из статического свойства `HttpContext.Current.Handler`, но не нашел способа определить, являлся ли обработчик реальной `System.Web.UI.Page`. Если бы я смог выяснить, что это страница, я мог бы легко идти дальше и найти элементы управления утверждений на странице. Моя первая попытка заключалась в написании кода с использованием интерфейсов отражения, чтобы я смог сам просматривать цепи наследования. Когда я заканчивал примерно пятисотую строку кода, Джефф Просиз (Jeff Prosise) невинно поинтересовался, не слышал ли я про оператор `is`, который определяет совместимость типа объекта, существующего в период выполнения, с заданным типом. Создание функциональности моего собственного оператора `is` стало интересным упражнением, но мне надо было совсем другое.

Получив объект `Page`, я начал искать на странице `AssertControl`. Я знал, что он мог заключаться в другом элементе управления, поэтому задействовал небольшую рекурсию для полного просмотра. Разумеется, при этом надо было убедиться в наличии вырождающегося цикла, иначе я легко мог закончить зацикливанием.

В `ASPTTraceListener.FindAssertControl` я решил задействовать преимущество ключевого слова `out`, которое позволяет передавать параметр метода ссылкой, но не требует его инициализации. Логичнее рассматривать ненайденный элемент управления как `null`, и ключевое слово `out` позволяло это сделать.

Последнее, что я делаю с утверждением в методе `ASPTTraceListener.HandleOutput`, — определяю, переходить ли при инициации утверждения в отладчик. Прекрасный объект `System.Diagnostics.Debugger` позволяет общаться с отладчиком из вашего кода. Если в последнем идет отладка кода, свойство `Debugger.IsAttached` будет иметь значение `true`, и, просто вызвав `Debugger.Break`, вы можете имитировать точку прерывания в отладчике. Конечно, такое решение предполагает, что вы отлаживаете этот конкретный Web-сайт. Мне еще нужно предусмотреть случай вызова отладчика, когда вы работаете не из него.

В классе `Debugger` содержится замечательный метод `Launch`, позволяющий запустить отладчик и подключить его к вашему процессу. Однако, если учетная запись пользователя, под которой выполняется процесс, не находится в группе `Debugger Users`, `Debugger.Launch` не сработает. Если нужно подключать отладчик из кода утверждения, когда отладчик не запущен, придется получить учетную запись для работы ASP.NET, находящуюся в группе `Debugger Users`. Прежде чем продолжить, должен сказать, что, разрешая ASP.NET вызывать отладчик, вы потенциально создаете угрозу безопасности, поэтому делайте это только на отладочных машинах, не подключенных к Интернету.

ASP.NET в Windows 2000 и XP работает под учетной записью ASPNET, так что именно ее надо добавить в группу `Debugger Users`. Добавив учетную запись, перезапустите IIS, чтобы `Debugger.Launch` отобразил диалог Just-In-Time (JIT) Debugging. В Windows Server 2003 ASP.NET работает под учетной записью NETWORK SERVICE. Добавив NETWORK SERVICE в группу `Debugger Users`, перезагрузите машину.

Обеспечив работу `Debugger.Launch` настройкой параметров безопасности, я должен был убедиться, что `Debugger.Launch` будет вызываться только при подходящих условиях. Вызов `Debugger.Launch`, когда в систему сервера никто не вошел, привел бы к большим проблемам, потому что отладчик по требованию мог бы ждать нажатия клавиши в окне, до которого никто не смог бы добраться! В классе `ASPTTraceListener` мне следовало убедиться, что HTTP-запрос производится с локальной машины, потому что это указывает на то, что кто-то вошел в систему и отлаживает утверждение. Метод `ASPTTraceListener.IsRequestFromLocalMachine` проверяет, не является ли 127.0.0.1 адресом хоста или не равна ли серверная переменная `LOCAL_ADDR` адресу хоста пользователя.

Последнее замечание по поводу вызова отладчика касается Terminal Services. Если у вас открыто окно Remote Desktop Connection с подключением к серверу, Web-адрес для любых запросов к серверу, как и следует ожидать, будет представляться в виде IP-адреса сервера. По умолчанию мой код утверждения при совпадении адреса запроса с адресом сервера вызывает `Debugger.Launch`. Тестируя приложение ASP.NET и запустив с помощью Remote Desktop браузер на сервере, я получил сильный шок при срабатывании утверждения. (Помните, что я не отлаживал процесс ни на одной машине.)

Я ожидал увидеть информационное окно с предупреждением о нарушении правил безопасности или диалоговое окно JIT Debugger, но увидел лишь завис-

ший браузер. Я был здорово растерян, пока не подошел к серверу и не подвигал мышь. Там на фоне экрана регистрации находилось мое информационное окно! Мне стало ясно, что, хотя это выглядело как ошибка, все было объяснимо. Поскольку информационное окно или диалог JIT Debugger вызываются из-под учетной записи ASPNET/NETWORK SERVICE, ASP.NET не знает, что подключение осуществлялось через сеанс Terminal Services. Эти учетные записи не могут отслеживать, из какого сеанса был вызван `Debugger.Launch`. Соответственно вывод направлялся только на реальный экран компьютера.

Хорошая новость в том, что если вы подключили отладчик, то независимо от того, сделали вы это в окне Remote Desktop Connection или на другой машине, вызов `Debugger.Launch` работает точно так, как должен, и прерывает выполнение, переходя в отладчик. Кроме того, если вы направили вызов серверу из браузера на другой машине, то вызов `Debugger.Launch` не остановит выполнение. Мораль: если для подключения к серверу вы собираетесь использовать Remote Desktop Connection и запустить браузер внутри этого окна (скажем, на сервере), вам следует подключить отладчик к процессу ASP.NET на этом сервере.

То, что Microsoft не предусмотрела утверждения в ASP.NET, непростительно, но, вооружившись хотя бы `AssertControl`, вы можете начать программировать. Если вы ищете элемент управления, чтобы научиться писать к ним расширения, `AssertControl` может послужить экспериментальным скелетом. Интересным расширением `AssertControl` могло бы стать использование в коде JavaScript для создания улучшенного UI вроде диалогового окна `Web`, чтобы сообщать пользователям о возникших проблемах.

Листинг 3-4. Важные методы `ASPTraceListener`

```
public class ASPTraceListener : TraceListener
{
    /* КОД УДАЛЕН ДЛЯ КРАТКОСТИ */

    // Метод, вызываемый при нарушении утверждения.
    public override void Fail ( String Message
                               , String DetailMessage )
    {
        // По независимым от меня причинам практически невозможно
        // всегда знать число элементов в стеке для Debug.Assert.
        // Иногда их 4, иногда – 5. Увы, единственный способ, которым
        // я могу решить эту проблему, – выяснить вручную. Лентяй.
        StackTrace StkSheez = new StackTrace ( ) ;
        int i = 0 ;
        for ( ; i < StkSheez.FrameCount ; i++ )
        {
            MethodBase Meth = StkSheez.GetFrame(i).GetMethod ( ) ;

            // Если ничего не получили, выходим отсюда.
            if ( null != Meth )
            {
                if ( "Debug" == Meth.ReflectedType.Name )
```



```

        {
            i++ ;
            break ;
        }
    }
}

BugslayerStackTrace Stk = new BugslayerStackTrace ( i ) ;
HandleOutput ( Message , DetailMessage , Stk ) ;
}

/* КОД УДАЛЕН ДЛЯ КРАТКОСТИ * /

/// <summary>
/// Закрытый заголовок сообщения об утверждении.
/// </summary>
private const String AssertionMsg = "ASSERTION FAILURE!\r\n" ;
/// <summary>
/// Закрытая строка с переводом каретки и возвратом строки.
/// </summary>
private const String CrLf = "\r\n" ;
/// <summary>
/// Закрытая строка с разделителем.
/// </summary>
private const String Border =
    "-----\r\n" ;

/// <summary>
/// Выводит утверждение или сообщение трассировки.
/// </summary>
/// <remarks>
/// Обрабатывает весь вывод утверждения или трассировки.
/// </remarks>
/// <param name="Message">
/// Отображаемое сообщение.
/// </param>
/// <param name="DetailMessage">
/// Отображаемый подробный комментарий.
/// </param>
/// <param name="Stk">
/// Значение, содержащее информацию о стеке для утверждения.
/// Если не равно null, эта функция вызвана из утверждения.
/// Вывод трассировки устанавливает этот параметр в null.
/// </param>
protected void HandleOutput ( String Message ,
                             String DetailMessage ,
                             BugslayerStackTrace Stk )
{
    // Создаем StringBuilder для помощи в создании
    // текстовой строки для вывода.

```

см. след. стр.

```
StringBuilder StrOut = new StringBuilder ( ) ;

// Если StackArray не null, это утверждение.
if ( null != Stk )
{
    StrOut.Append ( Border ) ;
    StrOut.Append ( AssertionMsg ) ;
    StrOut.Append ( Border ) ;
}

// Присоединяем сообщение.
StrOut.Append ( Message ) ;
StrOut.Append ( CrLf ) ;

// Присоединяем подробное сообщение, если оно есть.
if ( null != DetailMessage )
{
    StrOut.Append ( DetailMessage ) ;
    StrOut.Append ( CrLf ) ;
}

// Если это утверждение, показываем стек под разделителем.
if ( null != Stk )
{
    StrOut.Append ( Border ) ;
}

// Просматриваем и присоединяем
// всю имеющуюся информацию о стеке.
if ( null != Stk )
{
    Stk.SourceIndentString = "          " ;
    Stk.FunctionIndent = "      " ;
    StrOut.Append ( Stk.ToString ( ) ) ;
}

// Поскольку в нескольких местах
// мне понадобится строка, создаем ее.
String FinalString = StrOut.ToString ( ) ;

if ( ( true == m_ShowDebugLog          ) &&
      ( true == Debugger.IsLogging ( ) ) )
{
    Debugger.Log ( 0 , null , FinalString ) ;
}
if ( true == m_ShowOutputDebugString )
{
    OutputDebugStringA ( FinalString ) ;
}
if ( null != m_EvtLog )
```

```
{
    m_EvtLog.WriteEntry ( FinalString ,
        System.Diagnostics.EventLogEntryType.Error );
}
if ( null != m_Writer )
{
    m_Writer.WriteLine ( FinalString );
    // Добавляем CRLF, просто на всякий случай.
    m_Writer.WriteLine ( "" );
    m_Writer.Flush ( );
}

// Всегда выполняйте вывод на страницу!
if ( null != Stk )
{
    // Выполняем вывод предупреждения в текущий TraceContext.
    HttpContext.Current.Trace.Warn ( FinalString );

    // Ищем на странице AssertionControl.

    // Сначала убедимся, что описатель представляет страницу!
    if ( HttpContext.Current.Handler is System.Web.UI.Page )
    {
        System.Web.UI.Page CurrPage =
            (System.Web.UI.Page)HttpContext.Current.Handler ;

        // Обходим сложности, если на странице нет
        // элементов управления (в чем я сомневаюсь!)
        if ( true == CurrPage.HasControls ( ) )
        {
            // Ищем элемент управления.
            AssertControl AssertCtl = null ;
            FindAssertControl ( CurrPage.Controls ,
                               out AssertCtl      ) ;

            // Если он есть, добавляем утверждение.
            if ( null != AssertCtl )
            {
                AssertCtl.AddAssertion ( Message
                                         ,
                                         DetailMessage ,
                                         Stk
                                         ) ;
            }
        }
    }

    // Наконец, если нужно, запускаем отладчик.
    if ( true == m_LaunchDebuggerOnAssert )
    {
        // Если отладчик уже подключен, я могу просто применить
        // Debugger.Break. Не важно, где именно запущен отладчик,
```

см. след. стр.

```

        // если он работает в этом процессе.
        if ( true == Debugger.IsAttached )
        {
            Debugger.Break ( ) ;
        }
        else
        {
            // С изменениями в модели безопасности версии
            // .NET RTM, учетная запись ASPNET, которую использует
            // ASPNET_WP.EXE, перенесена из System в User.
            // Для работы Debugger.Launch надо добавить
            // ASPNET в группы Debugger Users. Хотя в отладочных
            // системах это безопасно, в рабочих системах
            // следует соблюдать осторожность.
            bool bRet = IsRequestFromLocalMachine ( ) ;
            if ( true == bRet )
            {
                Debugger.Launch ( ) ;
            }
        }
    }
}
else
{
    // TraceContext доступен прямо из HttpContext.
    HttpContext.Current.Trace.Write ( FinalString ) ;
}
}

/// <summary>
/// Определяет, пришел ли запрос от локальной машины.
/// </summary>
/// <remarks>
/// Проверяет, равен ли IP-адрес адресу 127.0.0.1
/// или серверной переменной LOCAL_ADDR.
/// </remarks>
/// <returns>
/// Возвращает true, если запрос пришел от локальной машины,
/// в противном случае – false.
/// </returns>
private bool IsRequestFromLocalMachine ( )
{
    // Получаем объект для запроса.
    HttpRequest Req = HttpContext.Current.Request ;

    // Замкнут ли клиент на себя?
    bool bRet = Req.UserHostAddress.Equals ( "127.0.0.1" ) ;
    if ( false == bRet )
    {
        // Получаем локальный IP-адрес из серверных переменных.

```

```

        String LocalStr =
            Req.ServerVariables.Get ( "LOCAL_ADDR" ) ;
        // Сравниваем локальный IP-адрес с IP-адресом запроса.
        bRet = Req.UserHostAddress.Equals ( LocalStr ) ;
    }
    return ( bRet ) ;
}

/// <summary>
/// Ищет на странице элементы управления утверждений.
/// </summary>
/// <remarks>
/// Все элементы управления утверждений носят имя "AssertControl",
/// так что этот метод просто просматривает набор элементов
/// управления на странице и ищет это имя. Кроме того,
/// он рекурсивно просматривает вложенные элементы.
/// </remarks>
/// <param name="CtlCol">
/// Набор элементов для просмотра.
/// </param>
/// <param name="AssertCtrl">
/// Исходящий параметр, который содержит найденный элемент управления.
/// </param>
private void FindAssertControl ( ControlCollection CtlCol ,
                                out AssertControl AssertCtrl )

{
    // Просматриваем все элементы управления из массива.
    foreach ( Control Ctl in CtlCol )
    {
        // Это тот элемент?
        if ( "AssertControl" == Ctl.GetType().Name )
        {
            // Да! Выходим.
            AssertCtrl = (AssertControl)Ctl ;
            return ;
        }
        else
        {
            // Если этот элемент имеет вложенные, просматриваем их тоже.
            if ( true == Ctl.HasControls ( ) )
            {
                FindAssertControl ( Ctl.Controls ,
                                    out AssertCtrl ) ;
                // Если один из вложенных элементов
                // содержал искомый, то можно выходить.
                if ( null != AssertCtrl )
                {
                    return ;
                }
            }
        }
    }
}

```

см. след. стр.

```
        }
    }
}
// В этом наборе его не нашли.
AssertCtrl = null ;
return ;
}
```

Утверждения в приложениях C++

Многие годы в старой компьютерной шутке, сравнивающей языки программирования с машинами, C++ всегда сравнивают с болидом Формулы 1: быстрый, но опасный для вождения. В другой шутке говорится, что C++ дает вам пистолет, чтобы прострелить себе ногу, и, когда вы проходите «Hello World!», курок уже почти спущен. Я думаю, можно сказать, что C++ — это болид Формулы 1 с двумя ружьями, чтобы вы могли прострелить себе ногу во время аварии. Тогда как даже малейшая ошибка способна обрушить ваше приложение, интенсивное использование утверждений в C++ — единственный способ получить шанс на отладку таких приложений.

С и C++ также включают все виды функций, которые помогут максимально подробно описать условия утверждений (табл. 3-2).

Табл. 3-2. Вспомогательные функции для описательных утверждений С и C++

Функция	Описание
GetObjectType	Функция подсистемы интерфейса графических устройств (GDI), возвращающая тип описателя GDI.
IsBadCodePtr	Проверяет, что указатель на область памяти может быть запущен.
IsBadReadPtr	Проверяет, что по указателю на область памяти можно считать указанное количество байт.
IsBadStringPtr	Проверяет, что по указателю на строку можно читать данные до ограничителя строки NULL или до указанного максимального числа символов.
IsBadWritePtr	Проверяет, что по указателю на область памяти можно записать указанное количество байт.
IsWindow	Проверяет, является ли параметр <code>HWND</code> допустимым окном.

Функции `IsBad*` небезопасны в многопоточной среде. В то время как один поток вызывает `IsBadWritePtr`, чтобы проверить права доступа к участку памяти, другой поток может менять содержимое памяти на которую указывает указатель. Эти функции дают вам лишь описание ситуации на отдельный момент времени. Некоторые читатели первого издания этой книги утверждали, что, поскольку функции `IsBad*` небезопасны в многопоточной среде, их вообще лучше не трогать, раз они могут вызвать ложное ощущение безопасности. Категорически не согласен. Гарантировать полностью безопасную проверку памяти в многопоточной среде практически нельзя, если только вы не выполняете доступ к каждому байту в рамках

структурной обработки исключений. Такое возможно, но код станет настолько медленным, что вы не сможете работать на компьютере. Еще одна проблема, которую порой сильно преувеличивают, в том, что функции `IsBad*` в очень редких случаях могут проглатывать исключения `EXCEPTION_GUARD_PAGE`. За все годы, которые я занимаюсь разработкой под Windows, я никогда не сталкивался с этой проблемой. Я, безусловно, согласен мириться с такими недостатками функций `IsBad*` за те преимущества, которые получаю от информированности о плохом указателе.

Следующий код демонстрирует одну из ошибок, которые я совершал в утверждениях C++:

```
// Неверное использование утверждения.
BOOL CheckDriveFreeSpace ( LPCTSTR szDrive )
{
    ULARGE_INTEGER ulgAvail ;
    ULARGE_INTEGER ulgNumBytes ;
    ULARGE_INTEGER ulgFree ;
    if ( FALSE == GetDiskFreeSpaceEx ( szDrive      ,
                                       &ulgAvail    ,
                                       &ulgNumBytes ,
                                       &ulgFree      ) )
    {
        ASSERT ( FALSE ) ;
        return ( FALSE ) ;
    }
    :
}
```

Хотя я использовал правильное утверждение, я не отображал невыполненное условие. Информационное окно утверждения показывало лишь выражение «FALSE», что не очень-то помогало. Используя утверждения, старайтесь сообщать в информационном окне максимально подробную информацию о сбое утверждения.

Мой друг Дейв Энжел (Dave Angel) обратил мое внимание на то, что в C и C++ можно просто применить логический оператор `NOT (!)`, используя строку в качестве операнда. Такая комбинация дает гораздо лучшее выражение в информационном окне утверждения, так что вы хотя бы имеете представление о том, что случилось, не заглядывая в исходный код. Вот правильный способ утверждения условия сбоя:

```
// Правильное использование утверждения
BOOL CheckDriveFreeSpace ( LPCTSTR szDrive )
{
    ULARGE_INTEGER ulgAvail ;
    ULARGE_INTEGER ulgNumBytes ;
    ULARGE_INTEGER ulgFree ;
    if ( FALSE == GetDiskFreeSpaceEx ( szDrive      ,
                                       &ulgAvail    ,
                                       &ulgNumBytes ,
                                       &ulgFree      ) )
    {
        ASSERT ( !"GetDiskFreeSpaceEx failed!" ) ;
    }
```



```

    return ( FALSE ) ;
}
:
}

```

Фокус Дейва можно усовершенствовать, применив логический условный оператор AND (&&) так, чтобы выполнять нормальное утверждение и выводить текст сообщения. Вот как это сделать (заметьте: при использовании логического AND в начале строки не ставится «!»):

```

BOOL AddToDataTree ( PTREENODE pNode )
{
    ASSERT ( ( FALSE == IsBadReadPtr ( pNode , sizeof ( TREENODE) ) ) &&
            "Invalid parameter!" ) ;
    :
}

```

Макрос VERIFY

Прежде чем перейти к макросам и функциям утверждений, с которыми вы столкнетесь при разработке под Windows, а также к связанным с ними проблемам, я хочу поговорить о макросе VERIFY, широко используемом в разработках на основе библиотеки классов Microsoft Foundation Class (MFC). В отладочной сборке макрос VERIFY ведет себя, как обычное утверждение, поскольку он определен как ASSERT. Если условие равно 0, макрос VERIFY инициирует обычное информационное окно утверждения, предупреждая о проблемах. В финальной сборке макрос VERIFY не выводит информационного окна, однако его параметр остается в исходном коде и вычисляется в ходе нормальной работы.

По сути макрос VERIFY позволяет создавать обычные утверждения с побочными эффектами, и эти побочные эффекты остаются в финальных сборках. В идеале ни в каких типах утверждений не следует использовать условия, вызывающие побочные эффекты. Однако макрос VERIFY может пригодиться: когда функция возвращает код ошибки, который вы все равно не стали бы проверять иначе. Например, если при вызове ResetEvent для очистки освободившегося описателя события происходит сбой, то не остается ничего другого, кроме как завершить работу приложения, поэтому большинство разработчиков вызывает ResetEvent, не проверяя возвращаемое значение ни в отладочных, ни в финальных сборках. Если выполнять вызов через макрос VERIFY, то по крайней мере в отладочных сборках вы будете получать уведомления о том, что нечто пошло не так. Конечно, тот же результат можно получить и благодаря ASSERT, но VERIFY позволяет избежать создания новой переменной только для хранения и проверки возвращаемого значения из вызова ResetEvent — переменной, которая скорее всего будет использована только в отладочных сборках.

Думаю, большинство программистов MFC использует макрос VERIFY потому, что им так удобнее, но попробуйте отказаться от этой привычки. В большинстве случаев вместо применения VERIFY следовало бы проверять возвращаемые значения. Хороший пример частого использования VERIFY — функция-член CString::LoadString, загружающая строки ресурсов. Здесь макрос VERIFY сгодится для отладочных сбо-

рок, так как при сбое `LoadString` он предупреждает вас об этом. Однако в финальных сборках сбой `LoadString` приведет к вызову неинициализированной переменной. Если повезет, вы получите пустую строку, но чаще всего это ведет к краху финальной сборки. Мораль: проверяйте возвращаемые значения. Если хотите задействовать макрос `VERIFY`, подумайте, не послужит ли игнорирование возвращаемого значения причиной проблем в финальных сборках.

Отладка: фронтовые очерки

Исчезающие файлы и потоки

Боевые действия

В работе над версией `BoundsChecker` в `NuMega` мы испытывали невероятные трудности со случайными сбоями, которые было почти невозможно повторить. Единственной зацепкой было то, что описатели файлов и потоков внезапно становились недействительными. Это означало, что файлы случайно закрывались и иногда нарушалась синхронизация потоков. Разработчики пользовательского интерфейса также сталкивались с периодическими сбоями, но только при работе в отладчике. Наконец эти проблемы привели к тому, что все члены команды прекратили свою работу и стали пытаться исправить эти ошибки.

Исход

Команда чуть было не облила меня смолой и не вываляла в перьях, потому что, как выяснилось, виноват в этой проблеме был я. Я отвечал за отладочные циклы в `BoundsChecker`. В отладочном цикле используется отладочный API `Windows` для запуска и управления другими процессами и отлаживаемой программой, а также для реакции на события отладки, генерируемые отладчиком. Как добросовестный программист, я видел, что функция `WaitForDebugEvent` возвращала описатели для некоторых уведомлений о событиях отладки. Например, при запуске процесса в отладчике последний мог получать структуру, содержащую описатель процесса и начальный поток для него.

Я был очень осторожен и знал, что, если API предоставил описатель какого-то объекта, который вам больше не нужен, следует вызвать `CloseHandle`, чтобы освободить память, занимаемую этим объектом. Поэтому, когда отладочный API предоставлял описатель, я закрывал его, как только он мне становился не нужен. Это выглядело вполне оправданно.

Однако, к моему великому огорчению, я не читал написанное мелким шрифтом в документации отладочного API, где говорилось, что отладочный API сам закрывает каждый процесс и создаваемые им описатели потоков. Получалось так, что я удерживал некоторые описатели, возвращаемые отладочным API, до тех пор пока они были мне нужны, но закрывал их после использования — после того как их уже закрыл отладочный API.

Чтобы понять, как это привело к нашей проблеме, надо знать, что, когда вы закрываете описатель, ОС помечает его значение как свободное. Micro-

soft Windows NT 4, которую мы тогда использовали, весьма агрессивна в отношении повторного применения значений описателей. (Microsoft Windows 2000/XP демонстрируют такую же агрессивность по отношению к значениям описателей.) Элементы нашего UI, интенсивно применявшие многопоточность и открывавшие много файлов, постоянно создавали и использовали новые описатели. Поскольку отладочный API закрывал мои описатели, и ОС обращалась к ним повторно, иногда элементы UI получали один из описателей, с которыми работал я. Закрывая позже свои копии описателей, я на самом деле закрывал потоки и описатели файлов UI!

Я едва избежал смолы и перьев, показав что эта же ошибка присутствовала в отладочных циклах предыдущих версий BoundsChecker. До сих пор нам просто везло. Разница в том, что та версия, над которой мы работали, имела новый улучшенный UI, гораздо интенсивнее работавший с файлами и потоками, что создало условия для выявления моей ошибки.

Полученный опыт

Если б я читал написанное мелким шрифтом в документации отладочного API, то избежал бы этих проблем. Кроме того — и это главный урок! — я понял, что нужно всегда проверять возвращаемые значения `CloseHandle`. Хотя, закрывая неверный поток, вы не сможете ничего предпринять, ОС сообщает вам, что что-то не так, и к этому надо относиться со вниманием.

Замечу также: если, работая в отладчике, вы пытаетесь дважды закрыть описатель или передать неверное значение в `CloseHandle`, ОС Windows инициирует исключение «Invalid Handle» (0xC0000008). Увидев такое значение исключения, можете прерваться и выяснить, почему это произошло.

А еще я понял, что очень полезно бегать быстрее своих коллег, когда они гонятся за тобой с котлами смолы и мешками перьев.

Различные типы утверждений в Visual C++

Хотя в C++ я описываю все свои макросы и функции утверждений с помощью простого `ASSERT`, о котором расскажу через секунду, сначала все же хочу коротко остановиться на других типах утверждений, доступных в Visual C++, и немного рассказать об их использовании. Тогда, встретив какое-нибудь из них в чужом коде, вы сможете его узнать. Кроме того, хочу предупредить вас о проблемах, возникающих с некоторыми реализациями.

`assert`, `_ASSERT` и `_ASSERTE`

Первый тип утверждения из библиотеки исполняющей системы C — макрос `assert` из стандарта ANSI C. Эта версия переносима на все компиляторы и платформы C и определяется включением `ASSERT.H`. В мире Windows, если в работе с консольным приложением инициируется утверждение, `assert` направит вывод в `stderr`. Если ваше Windows-приложение содержит графический пользовательский интерфейс, `assert` отобразит сведения о сбое в информационном окне.

Второй тип утверждения в библиотеке исполняющей системы С ориентирован только на Windows. В него входят утверждения `_ASSERT` и `_ASSERTE`, определенные в `CRTDBG.H`. Единственная разница между ними в том, что вариант `_ASSERTE` также выводит выражение, переданное в виде параметра. Поскольку это выражение так важно, особенно при тестировании инженерами отладки, всегда выбирайте `_ASSERTE`, применяя библиотеку исполняющей среды С. Оба макроса являются частью исключительно полезного отладочного кода библиотеки исполняющей среды, и утверждения — лишь одна из многих его функций.

Хотя `assert`, `_ASSERT` и `_ASSERTE` удобны и бесплатны, у них есть недостатки. Макрос `assert` содержит две проблемы, способные несколько вас огорчить. Первая заключается в том, что отображаемое имя файла усекается до 60 символов, так что иногда вы не сможете понять, какой файл инициировал исключение. Вторая проблема `assert` проявляется в работе с проектами, не содержащими UI, такими как службы Windows или внепроцессные COM-серверы. Поскольку `assert` направляет свой вывод в `stderr` или в информационное окно, вы можете его пропустить. В случае информационного окна ваше приложение зависнет, так как вы не можете закрыть информационное окно, если не отображаете UI.

С другой стороны, макросы исполняющей среды С решают проблему с отображением по умолчанию информационного окна, позволяя через вызов функции `_CrtSetReportMode` перенаправить утверждение в файл или в функцию `API OutputDebugString`. Однако все поставляемые Microsoft утверждения страдают одним пороком: они изменяют состояние системы, а это нарушение главного закона для утверждений. Влияние побочных эффектов на вызовы утверждений едва ли не хуже, чем полный отказ от их использования. Следующий пример демонстрирует, как поставляемые утверждения могут вносить различия между отладочными и финальными сборками. Сможете ли вы обнаружить проблему?

```
// Направляем сообщение в окно. Если время ожидания истекло, значит, другой
// поток завис, так что его нужно закрыть. Напомню, единственный способ
// проверить сбой SendMessageTimeout – вызвать GetLastError.
// Если функция возвращает 0 и код последней ошибки 0,
// время ожидания SendMessageTimeout истекло.
_ASSERTE ( NULL != pDataPacket );
if ( NULL == pDataPacket )
{
    return ( ERR_INVALID_DATA );
}
LRESULT lRes = SendMessageTimeout ( hUIWnd
                                   , WM_USER_NEEDNEXTPACKET
                                   , 0
                                   , (LPARAM)pDataPacket
                                   , SMT_BLOCK
                                   , 10000
                                   , &pdwRes
                                   ) ;

_ASSERTE ( FALSE != lRes );
if ( FALSE == lRes )
{
    // Получаем код последней ошибки.
    DWORD dwLastError = GetLastError ( ) ;
```

```

if ( 0 == dwLastError )
{
    // UI завис или обрабатывает данные слишком медленно.
    return ( ERR_UI_IS_HUNG ) ;
}
// Если ошибка другая, значит, проблема в данных,
// передаваемых через параметры.
return ( ERR_INVALID_DATA ) ;
}
return ( ERR_SUCCESS ) ;
:

```

Проблема здесь в том, что поставляемые утверждения уничтожают код последней ошибки. До проверки исполняется «`_ASSERT (FALSE != lRes)`», отображается информационное окно, и код последней ошибки меняется на 0. Так что в отладочных сборках всегда будет казаться, что завис UI, а в финальных сборках проявятся случаи, когда переданные `SendMessageTimeout` параметры были неверны.

То, что предоставляемые системой утверждения уничтожают код последней ошибки, может никак не отразиться на вашем коде, но я видел и другое: две ошибки, на обнаружение которых ушло немало времени, были вызваны именно этой проблемой. Но, к счастью, если вы будете использовать утверждение, представленное ниже в этой главе в разделе «`SUPERASSERT`», я позабочусь об этой проблеме за вас и расскажу кое-что, о чем не сообщают системные версии утверждений.

ASSERT_KINDOF и ASSERT_VALID

Если вы программируете, применяя MFC, в вашем распоряжении есть два дополнительных макроса утверждений, специфичных для MFC и являющих собой фантастические примеры профилактической отладки. Если вы объявляли классы с помощью `DECLARE_DYNAMIC` или `DECLARE_SERIAL`, то, используя макрос `ASSERT_KINDOF`, можете проверить, является ли указатель на потомок `CObject` определенным классом или потомком определенного класса. Утверждение `ASSERT_KINDOF` — всего лишь оболочка метода `CObject::IsKindOf`. Следующий фрагмент сначала проверяет параметр в утверждении `ASSERT_KINDOF`, а затем выполняет действительную проверку ошибок в параметрах.

```

BOOL DoSomeMFCStuffToAFrame ( CWnd * pWnd )
{
    ASSERT ( NULL != pWnd ) ;
    ASSERT_KINDOF ( CFrameWnd , pWnd ) ;
    if ( ( NULL == pWnd ) ||
        ( FALSE == pWnd->IsKindOf ( RUNTIME_CLASS ( CFrameWnd ) ) ) )
    {
        return ( FALSE ) ;
    }
    :
    // Выполняем прочие действия MFC; pWnd гарантированно
    // является CFrameWnd или его потомком.
    :
}

```

Второй специфичный для MFC макрос утверждений — `ASSERT_VALID`. Это утверждение интерпретирует `AfxAssertValidObject`, который полностью проверяет корректность указателя на класс-потомок `CObject`. После проверки правильности указателя `ASSERT_VALID` вызывает метод `AssertValid` объекта. `AssertValid` — это метод, который может быть переопределен в потомках для проверки всех внутренних структур данных в классе. Этот метод предоставляет прекрасный способ глубокой проверки ваших классов. Переопределяйте `AssertValid` во всех ключевых классах.

SUPERASSERT

Рассказав вам о проблемах с поставляемыми утверждениями, я хочу продемонстрировать, как я исправил и расширил утверждения так, чтобы они действительно сообщали, как и почему возникли проблемы, и делали еще больше. На рис. 3-3 и 3-4 показаны примеры диалоговых окон `SUPERASSERT`, сообщающих об ошибках. В первом издании этой книги вывод `SUPERASSERT` представлял собой информационное окно, в котором показывалось расположение невыполненного утверждения, код последней ошибки, преобразованный в текст, и стек вызова. Как видно из рисунков, `SUPERASSERT` определенно подрос! (Однако я не стал называть его `SUPERPUPERASSERT`!)

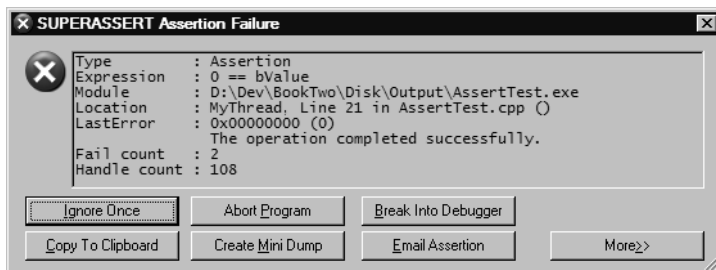


Рис. 3-3. Пример свернутого диалогового окна `SUPERASSERT`

Самое изумительное в труде писателя — потрясающие дискуссии с читателями, в которых я участвовал по электронной почте и лично. Мне повезло учиться у таких поразительно умных ребят! Вскоре после выхода первого издания между Скоттом Байласом (Scott Bilas) и мной состоялся интересный обмен письмами по электронной почте, в которых мы обсудили его мысли о том, что должны делать сообщения утверждений и как их использовать. Изначально я применял информационное окно, так как хотел оставить утверждение максимально легковесным. Однако, обменявшись массой интересных соображений со Скоттом, я убедился, что сообщения утверждений должны предлагать больше функций, таких как подавление утверждений (assertion suppression). Скотт даже предложил код для свертывания диалоговых окон (dialog box folding), свой макрос `ASSERT` для отслеживания числа пропусков (ignore) и т. п. Вдохновленный идеями Скотта, я создал новую версию `SUPERASSERT`. Я сделал это сразу после выхода первой версии и с тех пор использовал новый код во всех своих разработках, так что он прошел серьезную обкатку.

На рис. 3-3 показаны части диалогового окна, которые видны постоянно. Поле ввода Failure содержит причину сбоя (Assertion или Verify), невыполненное выражение, место сбоя, расшифрованный код последней ошибки и число сбоев дан-

ного конкретного утверждения. Если утверждение работает под Windows XP, Server 2003 и выше, оно также отображает общее число описателей ядра (kernel handle) в процессе. В SUPERASSERT я преобразую коды последней ошибки в их текстовое описание. Получение сообщений об ошибках в текстовом виде исключительно полезно при сбоях функций API: вы видите, почему произошел сбой, и можете быстрее запустить отладчик. Так, если в `GetModuleFileName` происходит сбой по причине малого объема буфера ввода, SUPERASSERT установит код последней ошибки равным 122, что соответствует `ERROR_INSUFFICIENT_BUFFER` из `WINERROR.H`. Сразу увидите текст «The data area passed to a system call is too small» («Область данных, переданная системному вызову, слишком мала»), вы поймете, в чем именно проблема и как ее устранить. На рис. 3-3 — стандартное сообщение Windows об ошибке, но вы вправе добавить свои ресурсы сообщений к преобразованию сообщений о последней ошибке в SUPERASSERT. Подробнее о собственных ресурсах сообщений см. раздел MSDN «Message Compiler». Дополнительный стимул к использованию ресурсов сообщений в том, что они здорово облегчают локализацию ваших приложений.

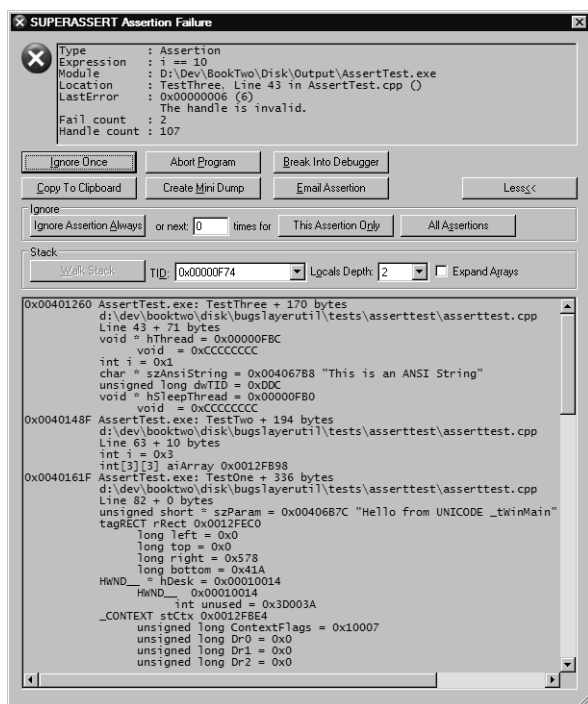


Рис. 3-4. Пример развернутого диалогового окна SUPERASSERT

Кнопка `Ignore Once`, расположенная под полем ввода `Failure`, просто продолжает выполнение. Она выделена по умолчанию, так что, нажав `Enter` или пробел, вы можете сразу продолжить работу, изучив причину сбоя. `Abort Program` вызывает `ExitProcess`, чтобы попытаться корректно завершить приложение. Кнопка `Break Into Debugger` инициирует вызов `DebugBreak`, так что вы можете начать отладку сбоя, перейдя в отладчик или запустив отладчик по требованию. Кнопка `Copy To Clipboard`

из второго ряда копирует в буфер обмена весь текст из поля ввода Failure, а также информацию из всех потоков для которых есть данные из стека. Последняя кнопка — More>> или Less<< — разворачивает и сворачивает диалоговое окно.

Кнопки Create Mini Dump и Email Assertion требуют некоторого пояснения. Если загруженная в пространство процесса версия DBGHELP.DLL содержит экспортируемые функции Minidump, то кнопка Create Mini Dump доступна. Если функции Minidump недоступны, кнопка отключена. Чтобы лучше сохранить состояние приложения, SUPERASSERT приостанавливает все прочие потоки приложения. Это значит, что SUPERASSERT не может использовать стандартный диалог для работы с файлами, так как этот диалог инициирует некоторые фоновые потоки, остающиеся после его закрытия. Когда SUPERASSERT приостанавливает все потоки, код стандартного файлового диалога зависает, поскольку его работа зависит от приостановленного потока. Следовательно, я не могу задействовать стандартный диалог. Поэтому диалог, появляющийся после щелчка Create Mini Dump, — это простое диалоговое окно с полем ввода, предлагающее вам ввести полный путь и имя для минидампа.

Кнопка Email Assertion активна, только если вы внесли в свой исходный файл специальное определение, указывающее адрес электронной почты, на который следует отправлять информацию утверждения. Эта потрясающая функция служит тестерам для отправки соответствующего утверждения нужному разработчику. Письмо содержит всю информацию, включая все данные о стеке, поэтому разработчики смогут точно определить причины инициации утверждения. Чтобы автоматически получить возможность отправки сообщения по электронной почте, в начале каждого исходного файла включите показанный ниже код. Для определения SUPERASSERT_EMAIL не обязательно использовать SUPERASSERT, но я вам это настоятельно советую.

```
#ifdef SUPERASSERT_EMAIL
#undef SUPERASSERT_EMAIL
// Пожалуйста, укажите собственный адрес электронной почты!
#define SUPERASSERT_EMAIL "john@wintellect.com"
#endif
```

Развернутое диалоговое окно SUPERASSERT (рис. 3-4) включает все виды полезных компонентов. Группа Ignore позволяет управлять пропуском утверждений. Первая кнопка — Ignore Assertion Always — отмечает данное утверждение как пропускаемое всегда. Вы также можете указать число пропусков, набрав его в поле ввода. Чтобы установить число пропусков нужному утверждению, щелкните This Assertion Only. Для пропуска всех последующих утверждений, где бы они ни возникли, щелкните All Assertions. Сначала я не думал, что утверждения вообще следует пропускать, но, получив возможность пропускать утверждение, инициировавшееся на каждой итерации цикла, я не представляю, как жил без этого.

Последняя часть диалогового окна посвящена стеку вызова. Стек вызова имелся и в первой версии SUPERASSERT, но, приглядевшись к стеку в поле ввода, вы увидите все локальные переменные и их текущие значения для каждой функции! Библиотека SymbolEngine из BugslayerUtil.DLL способна расшифровывать все основные типы, структуры, классы и массивы. Она также достаточно умна для расшифровки важных значений, таких как символьные указатели и символьные массивы. Как

видно на рис. 3-4, отображаются как строки ANSI, так и Unicode, а также расшифрованные структуры PEET. Правильная расшифровка символов стала одним из сложнейших фрагментов кода, которые я писал! Если вам интересны грязные подробности, то в главе 4 вы сможете больше узнать о библиотеке SymbolEngine. Хорошая новость в том, что я уже сделал все самое трудное, так что, если не хотите, можете даже не думать об этом!

Первая кнопка группы Stack — Walk Stack — позволяет просматривать стек. На рис. 3-4 Walk Stack неактивна, так как стек уже просмотрен. Раскрывающийся список Thread ID позволяет выбрать поток, который вы хотите просмотреть. Если приложение содержит единственный поток, список Thread ID в диалоговом окне не показывается. Раскрывающийся список Locals Depth позволяет выбрать, насколько глубоко будут раскрываться локальные переменные. Этот список сродни стрелкам с плюсами в окне Watch отладчика рядом с раскрывающимися элементами. Чем больше число, тем больше информации о соответствующих локальных переменных будет отображаться. Так, если у вас есть локальная переменная типа `int**`, то, чтобы увидеть целочисленное значение, на которое она указывает, следует установить глубину раскрытия равной 3. Флажок Expand Arrays предписывает SUPERASSERT раскрывать все встречающиеся типы массивов. Вычисление указателей или типов с глубокой вложенностью, а также крупных массивов — операция ресурсоемкая, поэтому вам вряд ли захочется раскрывать массивы без необходимости. Конечно, SUPERASSERT поступает правильно, на лету переоценивая все локальные переменные при изменении глубины раскрытия или по запросу, так что вы можете получить нужную информацию, когда она нужна.

SUPERASSERT включает несколько глобальных параметров, которые также можно менять по ходу работы. Диалоговое окно Global SUPERASSERT Options (рис. 3-5) открывается при выборе Options из системного меню.



Рис. 3-5. Диалоговое окно Global SUPERASSERT Options

Группа Stack Walking определяет объем информации о стеке, получаемой при появлении диалогового окна SUPERASSERT. По умолчанию просматривается только поток, в котором содержится утверждение, но, если нужна максимальная скорость появления окна, вы можете установить просмотр стека только вручную. Группа Additional Mini Dump Information определяет объем информации, записываемой во все минидампы. (Подробнее см. документацию к перечислению MINIDUMP_TYPE.)

Установка Play Sounds On Assertions приводит к проигрыванию стандартного звука сообщения при появлении диалогового окна SUPERASSERT. Если установлен Force Assertion To Top, диалоговое окно SUPERASSERT появляется поверх остальных окон. Если ведется отладка процесса, появление поверх других окон не устанавливается, так как SUPERASSERT может заблокировать отладчик. Все глобальные параметры SUPERASSERT хранятся в разделе реестра HKCU\Software\Bugslayer\SUPERASSERT. Кроме того, в реестре сохраняются координаты последнего отображения диалогового окна и его состояние (свернутое/развернутое), так что SUPERASSERT каждый раз появляется там, где вы ожидаете его увидеть.

Хочу отметить еще некоторые детали SUPERASSERT. Во-первых, как видно на рис. 3-3 и 3-4, SUPERASSERT содержит область захвата в нижнем правом углу для изменения размеров диалогового окна. Кроме того, SUPERASSERT поддерживает работу с несколькими мониторами, поэтому в системном меню есть команда, которая центрирует диалоговое окно на текущем мониторе, чтобы вы могли вернуть SUPERASSERT в нужную позицию. Но из рисунков не видно, что SUPERASSERT можно не отображать вовсе. Может показаться, что такой вариант непродуктивен, но уверяю вас, это не так! Если вы последовали моим рекомендациям из главы 2 и начали тестировать свои отладочные сборки с помощью инструмента регрессионного тестирования, то знаете, что обработка случайных сообщений от утверждений практически невозможна. Из-за проблем с обработкой сообщений от утверждений ваши тестировщики с меньшей вероятностью согласятся применять отладочные сборки. В моем коде утверждений вы вправе указать, что вывод следует направить в `OutputDebugString`, описатель файла, журнал событий или любую комбинацию этих вариантов. Такая гибкость позволяет запускать код и получать все данные от утверждений, не теряя возможности автоматизировать отладочные сборки. Наконец, код SUPERASSERT чрезвычайно умен в отношении выбора условий появления. Он всегда проверяет, зарегистрирован ли на рабочей станции интерактивный пользователь. Если на этой рабочей станции не зарегистрирован ни один интерактивный пользователь, SUPERASSERT не станет появляться и останавливать ваше приложение.

Благодаря информации, предоставляемой SUPERASSERT, я пользуюсь отладчиком реже, чем когда-либо, что является огромной победой в борьбе за скорость отладки. При срабатывании утверждения я размещаю диалоговое окно SUPERASSERT на втором мониторе. Я просматриваю информацию о локальных переменных и начинаю читать исходный код на главном мониторе. Я обнаружил, что способен исправить примерно на 20% больше ошибок, не запуская отладчик. Первая редакция была весьма полезна, вторая — потрясает!

Использование SUPERASSERT

Интегрировать SUPERASSERT в приложения довольно легко. Для этого надо просто включить BUGSLAYERUTIL.H, который, вероятно, лучше всего включается в прекомпилируемый заголовок, и подключиться к BUGSLAYERUTIL.LIB, чтобы внести BUGSLAYERUTIL.DLL в адресное пространство. Это предоставляет вам макрос ASSERT и автоматически перенаправляет все существующие вызовы ASSERT и assert моим функциям. Мой код не перехватывает макросы _ASSERT и _ASSERTE, так как, используя отладочную библиотеку исполняющей среды, вы можете выполнять дополни-

тельные действия или особый вывод, и я не хочу нарушать существующие решения. Мой код также не затрагивает `ASSERT_KINDOF` и `ASSERT_VALID`.

Перенаправить вывод, скажем, в журнал событий или текстовый файл, позволяет макрос `SETDIAGASSERTOPTIONS`, который принимает несколько очевидных макросов битовых полей, определяющих место вывода. Все эти макросы битовых полей определены в `DIAGASSERT.H`.

Два слова об игнорировании утверждений

Всегда плохо, когда другой разработчик или тестер тащат вас к своей машине, чтобы обвинить ваш код в аварии. Еще хуже, когда вы начинаете поиск проблемы с вопроса, не нажимал ли он кнопку `Ignore` в появившемся окне утверждения. Он клянется вам, что не делал этого, но вы знаете, что эта авария не могла произойти без инициации определенного утверждения. Когда вы, наконец, заставляете его сознаться в том, что он все-таки нажимал кнопку `Ignore`, вы готовы оторвать ему голову. Если бы он сообщил об этом утверждении, вы легко устранили бы проблему!

Кнопка `Ignore`, если вы еще не догадались, может представлять большую опасность: людей так и тянет ее нажать! Хотя это было бы слегка сурово, я серьезно раздумывал над тем, чтобы не создавать кнопку `Ignore` в `SUPERASSERT` и заставить вас разбираться с утверждениями и вызвавшими их причинами. В некоторых компаниях разработчики добавляют простой способ проверки, не игнорировались ли утверждения в текущем прогоне. Это позволяет им проверить, не нажималась ли кнопка `Ignore`, прежде чем тратить время на разбор аварии.

Если вы используете `SUPERASSERT` и хотите увидеть, сколько всего было инициировано утверждений, проверьте глобальную переменную `g_iTotalAssertions` из `SUPERASSERT.CPP`. Конечно, для этого надо подключить к аварийной программе отладчик или иметь дампы памяти аварийной программы. Чтобы получить общее число утверждений программно, вызовите `GetSuperAssertionCount`, экспортируемый из `BUGSLAYERUTIL.DLL`.

Возможно, вы захотите подумать о том, чтобы добавить в `SUPERASSERT` ведение журнала для записи всех инициировавшихся утверждений. Тогда вы будете автоматически получать текущее число нажатий пользователями кнопки `Ignore` и сможете утверждать, что к аварии привели действия пользователей. Некоторые компании автоматически записывают утверждения в центральную базу данных, чтобы отслеживать частоту их появления и обнаруживать неоправданное использование кнопки `Ignore` разработчиками и тестерами.

Поскольку я заговорил о вашей защите от рефлекторного стремления пользователей нажать кнопку `Ignore`, будет справедливо упомянуть о том, что, возможно, так поступаете и вы сами. Утверждения никогда не должны появляться при нормальной работе — только если что-то неверно. Вот прекрасный пример неправильного применения утверждения, с которым я столкнулся, помогая отлаживать приложение. Когда я выбрал элемент из часто используемого меню, которому не был сопоставлен объект воздействия (`target item`), утверждение инициировалось перед обычной обработкой ошибок. В этом случае обычной обработки ошибок было более чем достаточно. Если вы получаете жалобы о слишком частой инициации утверждений, тщательно проверьте, нужны ли они на своих местах.

Главное в реализации SUPERASSERT

Код SUPERASSERT может показаться вам слегка «закрученным», так как состоит из двух отдельных наборов кода: SUPERASSERT.CPP и DIAGASSERT.CPP. На самом деле DIAGASSERT.CPP — это первая версия SUPERASSERT. Я оставил его потому, что в создании нового SUPERASSERT было довольно много UI-кода, а поскольку я не мог применять SUPERASSERT на нем самом, мне потребовался второй набор утверждений, чтобы сделать разработку SUPERASSERT чище и устойчивей. Наконец, поскольку для SUPERASSERT требуется больше вспомогательного кода, в одном случае вам понадобится старый код (там, где нельзя использовать SUPERASSERT) для создания утверждений в RawDllMain, до того как инициализируется что-либо в вашем модуле.

Первая интересная часть SUPERASSERT — это макрос, интерпретирующий вызов функции SuperAssertion (листинг 3-5). Так как SUPERASSERT должен отслеживать количество пропущенных локальных утверждений, макрос при каждом использовании создает новую область видимости. В пределах этой области он объявляет две статических целочисленных переменных для отслеживания числа нарушений данного утверждения и количества раз, когда пользователь захотел проигнорировать данное утверждение. После проверки результата выражения оставшаяся часть макроса получает текущий указатель стека и фрейма, и вызывает настоящую функцию SuperAssertion.

Листинг 3-5. Главный макрос ASSERT

```
#ifdef _M_IX86
#define NEWASSERT_REALMACRO( exp , type )
{
    /* Локальный экземпляр количества пропусков и общего числа
       срабатываний */
    static int sIgnoreCount = 0 ;
    static int sFailCount   = 0 ;
    /* Локальный стек и кадр в месте утверждения */
    DWORD dwStack ;
    DWORD dwStackFrame ;
    /* Проверяем выражение */
    if ( ! ( exp ) )
    {
        /* Хьюстон, у нас проблемы */
        _asm { MOV dwStack , ESP }
        _asm { MOV dwStackFrame , EBP }
        if ( TRUE == SuperAssertion ( TEXT ( type )
                                     , TEXT ( #exp )
                                     , TEXT ( __FUNCTION__ ) ,
                                     , TEXT ( __FILE__ )
                                     , __LINE__
                                     , SUPERASSERT_EMAIL
                                     , (DWORD64)dwStack
                                     , (DWORD64)dwStackFrame
                                     , &sFailCount
```

см. след. стр.


```

// Синоним типа GetProcessHandleCount.
typedef BOOL (__stdcall *GETPROCESSHANDLECOUNT)(HANDLE , PDWORD) ;
/*//////////////////////////////////////
// Прототипы масштаба файла.
////////////////////////////////////*/
// Выполняет действия по отображению диалога утверждения.
static INT_PTR PopTheFancyAssertion ( TCHAR * szBuffer      ,
                                      LPCSTR  szEmail       ,
                                      DWORD64 dwStack       ,
                                      DWORD64 dwStackFrame  ,
                                      DWORD64 dwIP          ,
                                      int *   piIgnoreCount  ) ;

// Пытается получить модуль, вызвавший утверждение.
static SIZE_T GetModuleWithAssert ( DWORD64 dwIP   ,
                                    TCHAR *  szMod  ,
                                    DWORD   dwSize ) ;

// Да, это встраиваемая функция, но, чтобы ее
// задействовать, надо создать для нее прототип.
extern "C" void * _ReturnAddress ( void ) ;
#pragma intrinsic ( _ReturnAddress )

// Функция, скрывающая махинации по получению
// открытых потоков в процессе.
static BOOL SafelyGetProcessHandleCount ( PDWORD pdwHandleCount ) ;

/*//////////////////////////////////////
// Глобальные объявления масштаба файла.
////////////////////////////////////*/
// Глобальное число игнорируемых утверждений.
int g_iGlobalIgnoreCount = 0 ;
// Общее число утверждений.
static int g_iTotalAssertions = 0 ;
// Критическая секция, защищающая все.
static CCriticalSection g_cCS ;
// Указатель на функцию GetProcessHandleCount.
static GETPROCESSHANDLECOUNT g_pfnGPH = NULL ;

/*//////////////////////////////////////
// Реализация!
////////////////////////////////////*/
// Отключаем ошибку "unreachable code" для этой функции,
// вызывающей ExitProcess.
#pragma warning ( disable : 4702 )
BOOL RealSuperAssertion ( LPCWSTR szType      ,
                          LPCWSTR szExpression ,
                          LPCWSTR szFunction  ,

```

см. след. стр.


```

        LPCWSTR  szFile      ,
        int      iLine      ,
        LPCSTR   szEmail    ,
        DWORD64  dwStack     ,
        DWORD64  dwStackFrame ,
        DWORD64  dwIP        ,
        int *     piFailCount ,
        int *     piIgnoreCount )
{
    // В начале всегда увеличиваем общее число утверждений,
    // иницированных на данный момент.
    g_iTotalAssertions++ ;

    // Увеличиваем количество сбоев данного конкретного экземпляра.
    if ( NULL != piFailCount )
    {
        *piFailCount = *piFailCount + 1 ;
    }

    // Смотрим, нельзя ли быстро завершить работу с диалогом.
    // Значение "-1" значит "игнорировать все".
    if ( ( g_iGlobalIgnoreCount < 0 ) ||
        ( ( NULL != piIgnoreCount ) && *piIgnoreCount < 0 ) )
    {
        return ( FALSE ) ;
    }

    // Если число игнорирований всех утверждений
    // еще не исчерпано, можно выходить.
    if ( g_iGlobalIgnoreCount > 0 )
    {
        g_iGlobalIgnoreCount- ;
        return ( FALSE ) ;
    }

    // Надо ли пропустить это локальное утверждение?
    if ( ( NULL != piIgnoreCount ) && ( *piIgnoreCount > 0 ) )
    {
        *piIgnoreCount = *piIgnoreCount - 1 ;
        return ( FALSE ) ;
    }

    // Содержит возвращаемое значение функций обработки строк (STRSAFE).
    HRESULT hr = S_OK ;

    // Сохраняем код последней ошибки, чтобы не сбить
    // его при работе с диалогом утверждения.
    DWORD dwLastError = GetLastError ( ) ;

```

```

TCHAR szFmtMsg[ MAX_PATH ] ;
DWORD dwMsgRes = ConvertErrorToMessage ( dwLastError ,
                                         szFmtMsg ,
                                         sizeof ( szFmtMsg ) /
                                         sizeof ( TCHAR ) ) ;

if ( 0 == dwMsgRes )
{
    hr = StringCchCopy ( szFmtMsg
                        ,
                        sizeof ( szFmtMsg ) / sizeof ( TCHAR ) ,
                        _T ( "Last error message text not available\r\n" ) ) ;
    ASSERT ( SUCCEEDED ( hr ) ) ;
}

// Получаем информацию о модуле.
TCHAR szModuleName[ MAX_PATH ] ;
if ( 0 == GetModuleWithAssert ( dwIP , szModuleName , MAX_PATH ) )
{
    hr = StringCchCopy ( szModuleName
                        ,
                        sizeof ( szModuleName ) / sizeof (TCHAR) ,
                        _T ( "<unknown application>" )
                        ) ;
    ASSERT ( SUCCEEDED ( hr ) ) ;
}

// Захватываем синхронизирующий объект,
// чтобы не дать другим потокам достигнуть этой точки.
EnterCriticalSection ( &g_cCS.m_CritSec ) ;

// Буфер для хранения сообщения с выражением.
TCHAR szBuffer[ 2048 ] ;
#define BUFF_CHAR_SIZE ( sizeof ( szBuffer ) / sizeof ( TCHAR ) )

if ( ( NULL != szFile ) && ( NULL != szFunction ) )
{
    // Выделяем базовое имя из полного имени файла.
    TCHAR szTempName[ MAX_PATH ] ;
    LPTSTR szFileName ;
    LPTSTR szDir = szTempName ;

    hr = StringCchCopy ( szDir
                        ,
                        sizeof ( szTempName ) / sizeof ( TCHAR ) ,
                        szFile
                        ) ;
    ASSERT ( SUCCEEDED ( hr ) ) ;
    szFileName = _tcsrchr ( szDir , _T ( '\\' ) ) ;
    if ( NULL == szFileName )
    {
        szFileName = szTempName ;
        szDir = _T ( "" ) ;
    }
    else
    {

```

см. след. стр.

```

    *szFileName = _T ( '\\0' );
    szFileName++ ;
}
DWORD dwHandleCount = 0 ;
if ( TRUE == SafelyGetProcessHandleCount ( &dwHandleCount ) )
{
    // Используем новые функции STRSAFE,
    // чтобы не выйти за пределы буфера.
    hr = StringCchPrintf (
        szBuffer
        ,
        BUFF_CHAR_SIZE
        ,
        _T ( "Type          : %s\r\n" ) \
        _T ( "Expression   : %s\r\n" ) \
        _T ( "Module       : %s\r\n" ) \
        _T ( "Location      : %s, Line %d in %s (%s)\r\n" ) \
        _T ( "LastError      : 0x%08X (%d)\r\n" ) \
        _T ( "              : %s" ) \
        _T ( "Fail count    : %d\r\n" ) \
        _T ( "Handle count : %d" ) ,
        szType
        ,
        szExpression
        ,
        szModuleName
        ,
        szFunction
        ,
        inline
        ,
        szFileName
        ,
        szDir
        ,
        dwLastError
        ,
        dwLastError
        ,
        szFmtMsg
        ,
        *piFailCount
        ,
        dwHandleCount
        );
    ASSERT ( SUCCEEDED ( hr ) );
}
else
{
    hr = StringCchPrintf (
        szBuffer
        ,
        BUFF_CHAR_SIZE
        ,
        _T ( "Type          : %s\r\n" ) \
        _T ( "Expression   : %s\r\n" ) \
        _T ( "Module       : %s\r\n" ) \
        _T ( "Location      : %s, Line %d in %s (%s)\r\n" ) \
        _T ( "LastError      : 0x%08X (%d)\r\n" ) \
        _T ( "              : %s" ) \
        _T ( "Fail count    : %d\r\n" ) \
        ,
        szType
        ,
        szExpression
        ,
        szModuleName
        ,
        szFunction
        ,
        inline
        ,

```

```

        szFileName
        szDir
        dwLastError
        dwLastError
        szFmtMsg
        *piFailCount
    ASSERT ( SUCCEEDED ( hr ) );
}
}
else
{
    if ( NULL == szFunction )
    {
        szFunction = _T ( "Unknown function" );
    }
    hr = StringCchPrintf ( szBuffer
        ,
        BUFF_CHAR_SIZE
        ,
        _T ( "Type      : %s\r\n" ) \
        _T ( "Expression : %s\r\n" ) \
        _T ( "Function  : %s\r\n" ) \
        _T ( "Module    : %s\r\n" ) \
        _T ( "LastError : 0x%08X (%d)\r\n" ) \
        _T ( "      %s" ) ,
        szType
        ,
        szExpression
        ,
        szFunction
        ,
        szModuleName
        ,
        dwLastError
        ,
        dwLastError
        ,
        szFmtMsg
    );
    ASSERT ( SUCCEEDED ( hr ) );
}

if ( DA_SHOWODS == ( DA_SHOWODS & GetDiagAssertOptions ( ) ) )
{
    OutputDebugString ( szBuffer );
    OutputDebugString ( _T ( "\n" ) );
}

if ( DA_SHOWEVENTLOG ==
    ( DA_SHOWEVENTLOG & GetDiagAssertOptions ( ) ) )
{
    // Делаем запись в журнал событий,
    // только если все действительно кошерно.
    static BOOL bEventSuccessful = TRUE ;
    if ( TRUE == bEventSuccessful )
    {
        bEventSuccessful = OutputToEventLog ( szBuffer );
    }
}
}

```

```

if ( INVALID_HANDLE_VALUE != GetDiagAssertFile ( ) )
{
    static BOOL bWriteSuccessful = TRUE ;

    if ( TRUE == bWriteSuccessful )
    {
        DWORD dwWritten ;
        int    iLen = lstrlen ( szBuffer ) ;
        char * pToWrite = NULL ;

#ifdef UNICODE
        pToWrite = (char*)_alloca ( iLen + 1 ) ;

        BSUWide2Ansi ( szBuffer , pToWrite , iLen + 1 ) ;
#else
        pToWrite = szBuffer ;
#endif

        bWriteSuccessful = WriteFile ( GetDiagAssertFile ( ) ,
                                       pToWrite
                                       ,
                                       iLen
                                       ,
                                       &dwWritten
                                       ,
                                       NULL
                                       ) ;

        if ( FALSE == bWriteSuccessful )
        {
            OutputDebugString (
                _T ( "\n\nWriting assertion to file failed.\n\n" ) ) ;
        }
    }
}

// По умолчанию воспринимаем возвращаемое значение как IGNORE.
// Это особенно уместно, если пользователю не нужно окно MessageBox.
INT_PTR iRet = IDIGNORE ;

// Отображаем диалог, только если он нужен пользователю
// и если процесс выполняется интерактивно.
if ( ( DA_SHOWMSGBOX == ( DA_SHOWMSGBOX & GetDiagAssertOptions() ) ) &&
      ( TRUE == BSUIsInteractiveUser ( ) ) )
{
    iRet = PopTheFancyAssertion ( szBuffer
                                ,
                                szEmail
                                ,
                                dwStack
                                ,
                                dwStackFrame
                                ,
                                dwIP
                                ,
                                piIgnoreCount ) ;
}

// Я закончил критическую секцию!
LeaveCriticalSection ( &g_cCS.m_CritSec ) ;

```

```

SetLastError ( dwLastError );

// Хочет ли пользователь перейти в отладчик?
if ( IDRETRY == iRet )
{
    return ( TRUE );
}

// Хочет ли пользователь прервать программу?
if ( IDABORT == iRet )
{
    ExitProcess ( (UINT)-1 );
    return ( TRUE );
}

// Единственный оставшийся вариант – игнорировать утверждение.
return ( FALSE );
}

// Занимается отображением диалогового окна утверждения.
static INT_PTR PopTheFancyAssertion ( TCHAR * szBuffer
                                     ,
                                     LPCSTR szEmail
                                     ,
                                     DWORD64 dwStack
                                     ,
                                     DWORD64 dwStackFrame
                                     ,
                                     DWORD64 dwIP
                                     ,
                                     int * piIgnoreCount )
{
    // В этой подпрограмме я не выделяю память, потому что это может вызвать
    // фатальные проблемы. Я собираюсь сильно повысить приоритет этих потоков,
    // чтобы забрать ресурсы от других потоков и приостановить их.
    // Если на этом этапе я попытаюсь выделить память, то могу попасть
    // в ситуацию, когда потоки с малым приоритетом будут владеть CRT
    // или синхронизирующим объектом кучи и он понадобится этому потоку.
    // Следовательно, мы получим большое веселое зависание.
    // (Да, я так уже делал, вот почему я это знаю!)
    THREADINFO aThreadInfo [ k_MAXTHREADS ];
    DWORD aThreadIds [ k_MAXTHREADS ];

    // Первый поток в массиве информации о потоках – это ВСЕГДА текущий
    // поток. Это массив с нулевой базой, так что код диалога может
    // рассматривать все потоки как равные. Однако в этой функции массив
    // рассматривается как массив с единичной базой, поэтому текущий поток
    // не приостанавливается вместе с остальными.
    UINT uiThreadHandleCount = 1 ;

    aThreadInfo[ 0 ].dwTID = GetCurrentThreadId ( ) ;
    aThreadInfo[ 0 ].hThread = GetCurrentThread ( ) ;
    aThreadInfo[ 0 ].szStackWalk = NULL ;

```

см. след. стр.

```

// Сначала надо сразу повысить приоритет текущего потока. Я не хочу,
// чтобы создавались новые потоки, пока я готовлюсь их приостановить.
int iOldPriority = GetThreadPriority ( GetCurrentThread ( ) );
VERIFY ( SetThreadPriority ( GetCurrentThread ( )
                           THREAD_PRIORITY_TIME_CRITICAL ) );

DWORD dwPID = GetCurrentProcessId ( );

DWORD dwIDCount = 0 ;
if ( TRUE == GetProcessThreadIds ( dwPID
                                   ,
                                   k_MAXTHREADS
                                   ,
                                   (LPDWORD)&aThreadIds
                                   ,
                                   &dwIDCount
                                   ) )
{
    // Должен быть хоть один поток!!
    ASSERT ( 0 != dwIDCount ) ;
    ASSERT ( dwIDCount < k_MAXTHREADS ) ;

    // Вычисляем количество описателей.
    uiThreadHandleCount = dwIDCount ;
    // Если количество описателей равно 1, это однопоточное
    // приложение, и мне ничего не нужно делать!
    if ( ( uiThreadHandleCount > 1
          ( uiThreadHandleCount < k_MAXTHREADS ) ) &&
        {
            // Открываем каждый описатель, приостанавливаем его
            // и сохраняем описатель, чтобы запустить его позже.
            int iCurrHandle = 1 ;
            for ( DWORD i = 0 ; i < dwIDCount ; i++ )
            {
                // Конечно, не останавливать этот поток!!
                if ( GetCurrentThreadId ( ) != aThreadIds[ i ] )
                {
                    HANDLE hThread =
                        OpenThread ( THREAD_ALL_ACCESS ,
                                    FALSE
                                    ,
                                    aThreadIds [ i ] ) ;
                    if ( ( NULL != hThread
                          ( INVALID_HANDLE_VALUE != hThread ) ) &&
                        {
                            // Если SuspendThread возвращает -1,
                            // хранить значение этого потока незначем.
                            if ( (DWORD)-1 != SuspendThread ( hThread ) )
                            {
                                aThreadInfo[iCurrHandle].hThread = hThread ;
                                aThreadInfo[iCurrHandle].dwTID =
                                    aThreadIds[ i ] ;
                                aThreadInfo[iCurrHandle].szStackWalk = NULL;
                                iCurrHandle++ ;
                            }
                        }
                }
            }
        }
    }
}

```



```

        else
        {
            VERIFY ( CloseHandle ( hThread ) );
            uiThreadHandleCount- ;
        }
    }
    else
    {
        // Или для этого потока установлена какая-то защита,
        // или он закрылся сразу после того, как я собрал
        // информацию о потоках. Значит, надо уменьшить
        // общее число описателей потоков, или их будет
        // на один больше.
        TRACE( "Can't open thread: %08X\n" ,
                aThreadIds [ i ]
            );
        uiThreadHandleCount- ;
    }
}

}

}

}

// Возвращаем прежнее значение приоритета потока!
SetThreadPriority ( GetCurrentThread ( ) , iOldPriority );

// Убеждаемся, что ресурсы приложения установлены.
JfxGetApp()->m_hInstResources = GetBSUIInstanceHandle ( );

// Сам диалог утверждения.
JAssertionDlg cAssertDlg ( szBuffer
                            ,
                            szEmail
                            ,
                            dwStack
                            ,
                            dwStackFrame
                            ,
                            dwIP
                            ,
                            piIgnoreCount
                            ,
                            (LPTHREADINFO)&aThreadInfo
                            ,
                            uiThreadHandleCount
                            );

INT_PTR iRet = cAssertDlg.DoModal ( );

if ( ( 1 != uiThreadHandleCount ) &&
      ( uiThreadHandleCount < k_MAXTHREADS ) )
{
    // Снова повышаем приоритет потока!
    int iOldPriority = GetThreadPriority ( GetCurrentThread ( ) );
    VERIFY ( SetThreadPriority ( GetCurrentThread ( )
                                ,
                                THREAD_PRIORITY_TIME_CRITICAL
                            ) );

    // Если в ходе работы я приостановил другие потоки, надо
    // запустить их, закрыть описатели и удалить массив.

```

```

    for ( UINT i = 1 ; i < uiThreadHandleCount ; i++ )
    {
        VERIFY ( (DWORD)-1 !=
            ResumeThread ( aThreadInfo[ i ].hThread ) ) ;
        VERIFY ( CloseHandle ( aThreadInfo[ i ].hThread ) ) ;
    }
    // Возвращаем прежнее значение приоритета потока.
    VERIFY ( SetThreadPriority ( GetCurrentThread ( ) ,
        iOldPriority ) ) ;
}
return ( iRet ) ;
}

```

BOOL BUGSUTIL_DLLINTERFACE

```

SuperAssertionA ( LPCSTR  szType      ,
                  LPCSTR  szExpression ,
                  LPCSTR  szFunction  ,
                  LPCSTR  szFile      ,
                  int      iLine      ,
                  LPCSTR  szEmail     ,
                  DWORD64 dwStack     ,
                  DWORD64 dwStackFrame ,
                  int *    piFailCount ,
                  int *    piIgnoreCount )
{
    int iLenType = lstrlenA ( szType ) ;
    int iLenExp  = lstrlenA ( szExpression ) ;
    int iLenFile = lstrlenA ( szFile ) ;
    int iLenFunc = lstrlenA ( szFunction ) ;

    wchar_t * pWideType = (wchar_t*)
        HeapAlloc ( GetProcessHeap ( ) ,
                    HEAP_GENERATE_EXCEPTIONS ,
                    ( iLenType + 1 ) *
                    sizeof ( wchar_t ) ) ;

    wchar_t * pWideExp = (wchar_t*)
        HeapAlloc ( GetProcessHeap ( ) ,
                    HEAP_GENERATE_EXCEPTIONS ,
                    ( iLenExp + 1 ) *
                    sizeof ( wchar_t ) ) ;

    wchar_t * pWideFile = (wchar_t*)
        HeapAlloc ( GetProcessHeap ( ) ,
                    HEAP_GENERATE_EXCEPTIONS ,
                    ( iLenFile + 1 ) *
                    sizeof ( wchar_t ) ) ;

    wchar_t * pWideFunc = (wchar_t*)
        HeapAlloc ( GetProcessHeap ( ) ,
                    HEAP_GENERATE_EXCEPTIONS ,
                    ( iLenFunc + 1 ) *
                    sizeof ( wchar_t ) ) ;
}

```

```

BSUAnsi2Wide ( szType , pWideType , iLenType + 1 ) ;
BSUAnsi2Wide ( szExpression , pWideExp , iLenExp + 1 ) ;
BSUAnsi2Wide ( szFile , pWideFile , iLenFile + 1 ) ;
BSUAnsi2Wide ( szFunction , pWideFunc , iLenFunc + 1 ) ;

BOOL bRet ;
bRet = RealSuperAssertion ( pWideType
                           ,
                           pWideExp
                           ,
                           pWideFunc
                           ,
                           pWideFile
                           ,
                           iLine
                           ,
                           szEmail
                           ,
                           dwStack
                           ,
                           dwStackFrame
                           ,
                           (DWORD64)_ReturnAddress ( )
                           ,
                           piFailCount
                           ,
                           piIgnoreCount
                           ) ;

VERIFY ( HeapFree ( GetProcessHeap ( ) , 0 , pWideType ) ) ;
VERIFY ( HeapFree ( GetProcessHeap ( ) , 0 , pWideExp ) ) ;
VERIFY ( HeapFree ( GetProcessHeap ( ) , 0 , pWideFile ) ) ;

return ( bRet ) ;
}

BOOL BUGSUTIL_DLLINTERFACE
SuperAssertionW ( LPCWSTR szType
                 ,
                 LPCWSTR szExpression
                 ,
                 LPCWSTR szFunction
                 ,
                 LPCWSTR szFile
                 ,
                 int iLine
                 ,
                 LPCSTR szEmail
                 ,
                 DWORD64 dwStack
                 ,
                 DWORD64 dwStackFrame
                 ,
                 int * piFailCount
                 ,
                 int * piIgnoreCount
                 )
{
    return ( RealSuperAssertion ( szType
                                 ,
                                 szExpression
                                 ,
                                 szFunction
                                 ,
                                 szFile
                                 ,
                                 iLine
                                 ,
                                 szEmail
                                 ,
                                 dwStack
                                 ,
                                 dwStackFrame
                                 ,
                                 (DWORD64)_ReturnAddress ( )
                                 ,
                                 piFailCount
                                 ,
                                 piIgnoreCount
                                 ) ) ;
}

```

см. след. стр.

```

// Возвращает количество инициаций утверждения в приложении.
// В этом количестве учитываются все пропуски утверждения.
int BUGSUTIL_DLLINTERFACE GetSuperAssertionCount ( void )
{
    return ( g_iTotalAssertions );
}

static BOOL SafelyGetProcessHandleCount ( PDWORD pdwHandleCount )
{
    static BOOL bAlreadyLooked = FALSE ;
    if ( FALSE == bAlreadyLooked )
    {
        HMODULE hKernel32 = ::LoadLibrary ( _T ( "kernel32.dll" ) );
        g_pfnGPH = (GETPROCESSHANDLECOUNT)
            ::GetProcAddress ( hKernel32
                               ,
                               "GetProcessHandleCount" );
        FreeLibrary ( hKernel32 );
        bAlreadyLooked = TRUE ;
    }
    if ( NULL != g_pfnGPH )
    {
        return ( g_pfnGPH ( GetCurrentProcess ( ) , pdwHandleCount ) );
    }
    else
    {
        return ( FALSE );
    }
}

static SIZE_T GetModuleWithAssert ( DWORD64 dwIP ,
                                     TCHAR * szMod ,
                                     DWORD dwSize )
{
    // Пытаемся получить базовый адрес памяти для значения из стека.
    // По базовому адресу я попытаюсь получить модуль.
    MEMORY_BASIC_INFORMATION stMBI ;
    ZeroMemory ( &stMBI , sizeof ( MEMORY_BASIC_INFORMATION ) );
    SIZE_T dwRet = VirtualQuery ( (LPCVOID)dwIP
                                   ,
                                   &stMBI
                                   ,
                                   sizeof ( MEMORY_BASIC_INFORMATION ) );

    if ( 0 != dwRet )
    {
        dwRet = GetModuleFileName ( (HMODULE)stMBI.AllocationBase ,
                                     szMod
                                     ,
                                     dwSize
                                     );

        if ( 0 == dwRet )
        {
            // Сдаемся и просто возвращаем EXE.
            dwRet = GetModuleFileName ( NULL , szMod , dwSize );
        }
    }
}

```

```
    }  
    return ( dwRet );  
}
```

Сам код диалога в ASSERTDLG.CPP довольно скромнен, так что его не стоило приводить в книге. Когда мы со Скоттом Байласом обсуждали, на чем должно быть написано диалоговое окно, мы решили, что это должен быть простой язык, не требующий дополнительных двоичных файлов, кроме DLL, содержащей диалоговое окно, — все указывало на MFC. Когда я писал диалоговое окно, библиотека шаблонов Windows Template Library (WTL) еще не вышла. Но скорее всего я и не стал бы ее использовать, так как отношусь к шаблонам с опаской. Лишь немногие разработчики на самом деле понимают все переплетения в шаблонах, и большинство ошибок, с которыми приходилось бороться моей компании, были прямым следствием применения шаблонов. Несколько лет назад мы с Джеффри Рихтером (Jeffrey Richter) участвовали в проекте, для которого требовался исключительно легковесный UI, и разработали простую библиотеку классов UI под именем JFX. Джеффри будет утверждать, что JFX означает «Jeffrey's Framework», но на самом деле это «John's Framework», что бы он ни говорил. Как бы то ни было, для создания UI я использовал JFX. Полный исходный код содержится среди файлов примеров к этой книге. В каталоге JFX есть пара тестовых программ, показывающих, как использовать JFX, и код диалога SUPERASSERT. Хорошая новость: JFX исключительно мал и компактен — финальная версия BugslayerUtil.DLL, включающая гораздо больше, чем просто SUPERASSERT, занимает менее 70 Кб.

Стандартный вопрос отладки

Почему в условных операторах ты всегда размещаешь константы слева?

Я всегда использую операторы вроде «if (INVALID_HANDLE_VALUE == hFile)» вместо «if (hFile == INVALID_HANDLE_VALUE)». Я делаю это во избежание ошибок. Вы можете пропустить один знак равенства, и тогда первая версия приведет к ошибке компиляции. Вторая версия может не вызвать предупреждения (в зависимости от уровня диагностики компилятора), и вы измените значение переменной. Компиляторы при попытке присвоить значение константе выдают ошибку компиляции. Если вам приходилось искать ошибки, связанные со случайным присвоением, вы знаете, как трудно их обнаружить.

Присмотревшись к моему коду, вы увидите, что я размещаю константные переменные в левой части равенств. Как и в случае константных значений, компилятор сообщит об ошибке при попытке присвоить значение константной переменной. Выяснилось, что гораздо проще исправлять ошибки компиляции, чем искать ошибки в отладчике.

Некоторые разработчики жаловались (иногда очень громко), что мой способ написания условных операторов ухудшает читаемость кода. Не согласен. На чтение и перевод моих условных операторов требуется на одну секунду больше времени. Я готов пожертвовать этой секундой, чтобы не тратить огромное количество времени позже.

Trace, Trace, Trace и еще раз Trace

Утверждения — возможно лучший прием профилактического программирования из всех, что вы узнали, а операторы Trace при правильном использовании вместе с утверждениями действительно позволят отлаживать приложения без отладчика. Для некоторых опытных программистов среди вас операторы Trace — суть отладка в стиле `printf`. Мощность отладки в стиле `printf` нельзя недооценивать, поскольку так отлаживалось большинство приложений до изобретения интерактивных отладчиков. Трассировка в .NET интригует, так как, когда Microsoft впервые публично упомянула про .NET, ключевые преимущества были ориентированы не на разработчиков, а на администраторов сетей и ИТ-персонал, ответственных за развертывание написанных разработчиками приложений. Одним из важнейших новых преимуществ Microsoft называла возможность для ИТ-персонала легко включать трассировку, чтобы находить проблемы в приложениях! Читая это, я был ошеломлен, поскольку это говорило о том, что Microsoft откликнулась на страдания наших конечных пользователей, сталкивающихся с ошибками в программах.

Тонкость трассировки — в определении объема нужной информации для решения проблем на машинах, на которых не установлена среда разработки. Записать слишком много — получатся большие файлы, работа с которыми станет мучой, слишком мало — вы не сможете решить проблему. Действия по балансировке требуют наличия ровно такого объема записанной информации, чтобы избежать экстренного перелета за 8 000 километров к пользователю, у которого только что появилась та мерзкая ошибка, — перелета, в котором вам придется сидеть на среднем сиденье рядом с плачущим ребенком и больным пассажиром. В общем, это значит, что вам понадобятся два уровня трассировки: один, отражающий основную работу в программе, чтобы видеть, что и когда вызывалось, и второй — для добавления в файл ключевых данных, чтобы вы могли отыскивать проблемы, связанные с потоками данных.

К сожалению, все приложения разные, так что я не могу назвать вам точное число операторов трассировки или другие признаки данных, которых будет достаточно для журнала трассировки. Один из лучших подходов, которые я видел, заключался в том, чтобы дать нескольким новым членам команды пример журнала и спросить, даст ли он им достаточно информации для начала поиска проблемы. Если через пару часов они с отвращением отказываются, вероятно, информации мало. Если же через час-два у них в общих чертах появится представление о том, где находилось приложение на момент повреждения или краха, — это признак того, что ваш журнал содержит нужный объем информации.

Как я отмечал в главе 2, следует иметь общекомандную систему ведения журналов. Частью разработки этой системы должно стать определение формата трассировки, особенно для облегчения работы с отладочной трассировкой. Без такого формата эффективность трассировки быстро исчезает, так как никто не захочет пробираться сквозь тонны текста без особых на то причин. Хорошая новость для приложений .NET в том, что Microsoft проделала большую работу, чтобы облегчить управление выводом. В машинных приложениях вам придется создавать собственные системы, но ниже я дам вам кое-какие рекомендации в разделе «Трассировка в приложениях C++».

Перед тем как ринуться в разбор особенностей для разных платформ, хочу упомянуть об одном исключительном инструменте, который всегда должен быть на машинах для разработки: DebugView. Мой бывший сосед Марк Руссинович (Mark Russinovich) написал DebugView и массу других потрясающих инструментов, которые можно скачать с сайта Sysinternals (www.sysinternals.com). У них отличная цена (бесплатно!), многие инструменты доступны с исходным кодом и решают некоторые очень сложные проблемы, а потому вам стоит посещать Sysinternals хоть раз в месяц. DebugView отслеживает все вызовы к `OutputDebugString` пользовательского режима или к `DbgPrint` режима ядра, так что вы сможете видеть всю отладочную информацию, не работая в отладчике. Что делает DebugView еще более полезным, так это его способность работать с другими машинами, и вы сможете следить за всеми машинами распределенной системы с одного компьютера.

Трассировка в Windows Forms и консольных приложениях .NET

Как я сказал, Microsoft наделала маркетингового шума вокруг трассировки в приложениях .NET. В общем, они неплохо потрудились при создании хорошей архитектуры, которая лучше управляет трассировкой в реальных разработках. Говоря об утверждениях, я уже упоминал объект `Trace`, поскольку он необходим для трассировки. Как и `Debug`, для обработки вывода объект `Trace` использует концепцию применения `TraceListener`. Поэтому мой код утверждений в ASP.NET менял приемники для обоих объектов: так весь вывод направляется в одно место. В коде утверждений из ваших разработок вам лучше поступать так же. Вызовы методов объекта `Trace` активны, только если определен параметр `TRACE`. По умолчанию он определен в проектах и отладочных, и финальных сборок, создаваемых Visual Studio .NET, поэтому скорее всего методы уже активны.

Объект `Trace` содержит четыре метода для вывода информации трассировки: `Write`, `WriteIf`, `WriteLine` и `WriteLineIf`. Вероятно, вы догадались о разнице между `Write` и `WriteLine`, но понять методы `*If` сложнее: они позволяют осуществлять условную трассировку. Если первый параметр метода `*If` принимает значение `true`, выполняется трассировка, `false` — нет. Это довольно удобно, но при неосторожном обращении может привести к серьезным проблемам с производительностью. Так, написав код, вроде показанного в первой части следующего отрывка, вы будете испытывать издержки от конкатенации строк при каждом выполнении этой строки кода, так как необходимость трассировки определяется внутри вызова `Trace.WriteLineIf`. Гораздо лучше следовать второму примеру из фрагмента, где оператор `if` для вызова `Trace.WriteLine` используется только при необходимости, минимизируя издержки от конкатенации строк.

```
// Испытываем издержки каждый раз.  
Trace.WriteLineIf ( bShowTrace , "Parameters: x=" + x + " y=" + y );
```

```
// Выполняем конкатенацию, только когда это необходимо.  
if ( true == bShowTrace )  
{  
    Trace.WriteLine ("Parameters: x=" + x + " y=" + y );  
}
```

Думаю, разработчики .NET оказали нам всем большую услугу, добавив класс `TraceSwitch`. При наличии методов `*If` в объекте `Trace`, позволяющих выполнять трассировку по условию, остается лишь шаг до определения класса, предоставляющего несколько уровней трассировки и единый способ их установки. Важнейшая часть `TraceSwitch` — это имя, присваиваемое ему в первом параметре конструктора. (Второй параметр — это описательное имя.) Имя позволяет управлять объектом снаружи приложения, о чем я расскажу через секунду. В объектах `TraceSwitch` заключены уровни трассировки (табл. 3-3). Для проверки соответствия `TraceSwitch` определенному уровню служит набор свойств, таких как `TraceError`, возвращающих `true`, если объект соответствует данному уровню. В сочетании с методами `*If` использование объектов `TraceSwitch` вполне очевидно.

```
public static void Main ( )
{
    TraceSwitch TheSwitch = new TraceSwitch ( "SwitchyTheSwitch",
                                              "Example Switch" );

    TheSwitch.Level = TraceLevel.Info ;
    Trace.WriteLineIf ( TheSwitch.TraceError ,
                       "Error tracing is on!" ) ;
    Trace.WriteLineIf ( TheSwitch.TraceWarning ,
                       "Warning tracing is on!" ) ;
    Trace.WriteLineIf ( TheSwitch.TraceInfo ,
                       "Info tracing is on!" ) ;
    Trace.WriteLineIf ( TheSwitch.TraceVerbose ,
                       "VerboseSwitching is on!" ) ;
}
```

Табл. 3-3. Уровни `TraceSwitch`

Уровень трассировки	Значение
Off — Выкл.	0
Error — Ошибки	1
Warnings (and errors) — Предупреждения (и ошибки)	2
Info (warnings and errors) — Информация (предупреждения и ошибки)	3
Verbose (everything) — Полная информация	4

Чудо объектов `TraceSwitch` в том, что ими легко управлять снаружи приложения из вездесущего файла `CONFIG`. В элементе `switches`, вложенном в элемент `system.diagnostics`, указываются элементы `add`, с помощью которых добавляются и устанавливаются имена и уровни. В листинге 3-7 показан полный конфигурационный файл для приложения. В идеале для каждой сборки в приложении надо иметь отдельный объект `TraceSwitch`. Помните, что параметры `TraceSwitch` также можно применять к глобальному файлу `MACHINE.CONFIG`.

Листинг 3-7. Установка флагов `TraceSwitch` в конфигурационном файле

```
<?xml version="1.0" encoding="UTF-8" ?>
<configuration>
  <system.diagnostics>
```



```
<switches>
  <add name="Wintellect.ScheduleJob" value="4" />
  <add name="Wintellect.DataAccess" value="0" />
</switches>
</system.diagnostics>
</configuration>
```

Трассировка в приложениях ASP.NET и Web-сервисах XML

Несмотря на наличие прекрасно продуманных объектов `Trace` и `TraceSwitch`, ASP.NET и — как расширение — Web-сервисы XML содержат совершенно иную систему трассировки. Исходя из размещения вывода трассировки ASP.NET, я могу понять причину этих различий, но все равно считаю, что они сбивают с толку. Класс `System.Web.UI.Page` содержит собственный объект `Trace`, наследуемый от `System.Web.TraceContext`. Чтобы не путать эти два разных варианта трассировки, я буду ссылаться на вариант ASP.NET как на `TraceContext.Trace`. Два ключевых метода `TraceContext.Trace` — это `Write` и `Warn`. Оба они обрабатывают вывод трассировки, но `Warn` записывает вывод красным цветом. Каждый метод имеет три перегруженных версии, и оба принимают одинаковые параметры: обычное сообщение и категорию с вариантами сообщений, но есть версия, принимающая категорию, сообщение и `System.Exception`. Эта последняя версия записывает строку исключения, а также источник и строку где было сгенерировано исключение. Чтобы избежать лишних издержек в обработке, когда трассировка отключена, проверяйте, имеет ли свойство `IsEnabled` значение `true`.

Самый простой способ включить трассировку — задать атрибуту `Trace` директивы `@Page`, располагающейся в начале ваших ASPX-файлов, значение `true`.

```
<%@ Page Trace="true" %>
```

Волшебная маленькая директива включает тонны информации трассировки, которая появляется в нижней части страницы, что довольно удобно, но так ее видите и вы, и пользователи. Честно говоря, информации трассировки так много, что я очень хотел бы, чтобы она была поделена на несколько уровней. Иметь информацию о файлах cookie (Cookies), наборах заголовков (Headers Collections) и серверных переменных (Server Variables) приятно, но чаще всего она не нужна. Все разделы вполне очевидны, но я хочу выделить раздел `Trace Information`, так как здесь появляются все вызовы к `TraceContext.Trace`. Даже если вы не вызывали `TraceContext.Trace.Warn/Write`, вы все равно увидите информацию в разделе `Trace Information`, потому что ASP.NET сообщает о вызове нескольких своих методов. В этом разделе и появляется красный текст при вызове `TraceContext.Trace.Warn`.

Устанавливать атрибут `Trace` в начале каждой страницы приложения скучно, поэтому разработчики ASP.NET ввели в `WEB.CONFIG` раздел, позволяющий управлять трассировкой. Этот раздел, вполне логично названный `trace`, показан ниже:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.web>
```

```
<trace
  enabled="false"
  requestLimit="10"
  pageOutput="false"
  traceMode="SortByTime"
  localOnly="true"
/>
</system.web>
</configuration>
```

Атрибут `enabled` управляет включением трассировки для данного приложения. Атрибут `requestLimit` указывает, сколько запросов трассировки кэшировать в памяти для каждого приложения. (Через секунду мы обсудим, как просмотреть эти кэшированные запросы.) Элемент `pageOutput` сообщает ASP.NET, показывать ли вывод трассировки. Если `pageOutput` задано `true`, вывод появляется на странице, как если бы вы установили атрибут `Trace` в директиве `Page`. Вероятно, вам не захочется менять элемент `traceMode` поскольку так информация в разделе трассировки `Trace Information` отсортирована по времени. Если вы хотите увидеть сортировку по категориям, задайте `traceMode` значение `SortByCategory`. Последний атрибут — `localOnly` — сообщает ASP.NET, должен ли вывод быть видим только на локальной машине или он должен быть виден для всех клиентских приложений.

Чтобы увидеть кэшированные запросы трассировки, когда `pageOutput` задано `false`, добавьте к каталогу приложения HTTP-обработчик `trace.axd`, который отобразит страницу, позволяющую выбрать сохраненную информацию трассировки, которую вы хотите увидеть. Скажем, если имя вашего каталога — `http://www.wintellect.com/schedules`, то, чтобы увидеть сохраненную информацию трассировки, используйте путь `http://www.wintellect.com/schedules/trace.axd`. Достигнув предела `requestLimit`, ASP.NET прекращает записывать информацию трассировки. Запись можно перезапустить, просмотрев страницу `trace.axd` и щелкнув ссылку `Clear Current Trace` в верхней части страницы.

Как видите, если не соблюдать осторожность в трассировке, ее увидят конечные пользователи, а это всегда пугает, так как разработчики печально известны операторами трассировки, способными повредить карьере, если вывод попадет в плохие руки. К счастью, установив `localOnly` в `true`, вы сможете просматривать трассировку только на локальном сервере, даже при доступе к журналу трассировки через HTTP-обработчик `trace.axd`. Чтобы просмотреть журналы трассировки вашего приложения, вам просто придется применить величайший программный продукт, известный человечеству, — `Terminal Services`, и вы получите доступ к серверу прямо из своего офиса, даже не вставая из-за стола. Стоит также изменить раздел `customErrors` файла `WEB.CONFIG` для использования страницы `default-Redirect`, чтобы при попытке доступа к `trace.axd` с удаленной машины конечные пользователи не увидели ошибку ASP.NET «Server Error in 'Имя_приложения' Application». Кроме того, тех, кто пытается получить доступ к `trace.axd`, стоит заносить в журнал, особенно потому, что неудавшаяся попытка доступа, вероятно, указывает на хакера.

Сейчас кто-то из вас, возможно, думает об одной проблеме с трассировкой в ASP.NET: ASP.NET содержит `TraceContext.Trace`, управляющий свой вывод в одно

место, а `DefaultTraceListener` для объекта `System.Diagnostics.Trace` отправляет свой вывод куда-то еще. В обычном ASP.NET это огромная проблема, но если вы применяете код утверждений из `BugslayerUtil.NET`, описанный выше, то `ASPTraceListener` также используется как единый `TraceListener` для объекта `System.Diagnostics.Trace`, так что я перенаправляю всю информацию трассировки в `TraceContext.Trace`, чтобы вся она появлялась в одном месте.

Трассировка в приложениях C++

Почти всю трассировку в таких приложениях выполняет макрос C++, обычно носящий имя `TRACE` и активный только в отладочных сборках. В конечном счете функция, вызываемая им, вызовет `OutputDebugString` из Windows API, так что информацию трассировки можно видеть в отладчике или в `DebugView`. Помните: вызов `OutputDebugString` приводит к переходу в режим ядра. Это не очень важно для отладочных сборок, но может отрицательно сказаться на производительности финальных сборок, так что учтите все вызовы, которые могут остаться в финальных сборках. Вообще в поисках способов повысить производительность Windows в целом, команда Windows удалила массу трассировок, на которые мы все привыкли полагаться, таких как сообщение о конфликте загрузки DLL, появлявшееся при загрузке DLL, и это привело к очень хорошему росту производительности.

Если у вас нет макроса `TRACE`, можете использовать мой — из состава `BugslayerUtil.DLL`. Всю работу выполняют функции `DiagOutputA/W` из `DIAGASSERT.CPP`. Преимущество моего кода в том, что вы можете вызвать `SetDiagOutputFile`, передав ему как параметр описатель файла, и записывать всю трассировку в файл.

В дополнение к макросу `TRACE` в главе 18 описывается мой инструмент `FastTrace` для серверных приложений C++. Последнее, что хочется делать в приложениях, интенсивно использующих многопоточность, — это принуждать все потоки блокироваться на синхронизирующий объект при включении трассировки. Инструмент `FastTrace` дает максимально возможную производительность трассировки без потерь важных потоков информации.

Комментировать, комментировать и еще раз комментировать

Однажды мой друг Франсуа Полин (François Poulin), который весь день занимается сопровождением кода, написанного другими, пришел со значком, на котором было написано: «Кодируй так, как будто тот, кто сопровождает твой код, — буйнопомешанный, который знает, где ты живешь». Франсуа, несомненно, псих, но в его словах есть огромный смысл. Хотя вам может казаться, что ваш код является собой образец ясности и совершенно очевиден, без подробных комментариев для сопровождающих разработчиков он так же плох, как сырой ассемблер. Ирония в том, что сопровождающим разработчиком вашего кода легко можете стать вы сами! Незадолго до начала работы над вторым изданием этой книги я получил по электронной почте письмо из компании, в которой работал лет 10 назад, с просьбой обновить проект, который я для них писал. Взглянуть на код, который я писал так давно, было потрясюще! Потрясало и то, насколько плохие я делал комментарии. Вводя каждую строку кода, вспоминайте значок Франсуа.

Наша задача двойственна: разработать решение для пользователя и сделать его пригодным к сопровождению в будущем. Единственный способ сделать код сопровождаемым — комментировать его. Под словами «комментировать его» я подразумеваю не просто создание комментариев, повторяющих то, что делает код; я подразумеваю документирование ваших предположений, подходов к решению задачи и причин, по которым выбран именно такой подход. Также следует соотносить свои комментарии с кодом. Обычные кроткие программисты сопровождения могут впасть в сомнамбулическое состояние, пытаясь обновить код, делающий не то, что он должен делать согласно комментариям.

Создавая комментарии, я руководствуюсь следующими правилами.

- Каждая функция или метод требуют одного-двух предложений, проясняющих:
 - что делает подпрограмма;
 - какие в ней приняты допущения;
 - что должно содержаться в каждом из входных параметров;
 - что должно содержаться в каждом из выходных параметров в случае успеха и неудачи;
 - каждое из возможных возвращаемых значений;
 - каждое исключение, самостоятельно генерируемое функцией.
- Каждая часть функции, не являющаяся совершенно понятной из кода, требует одного-двух предложений, объясняющих что она делает.
- Любой интересный алгоритм заслуживает полного описания.
- Любые нетривиальные ошибки, исправленные в коде, должны быть прокомментированы с указанием номера ошибки и описания исправлений.
- Удачно размещенные операторы трассировки и утверждения, а также хорошие схемы именования тоже могут служить хорошими комментариями и давать прекрасный контекст для кода.
- Комментируйте так, словно вам самому придется сопровождать этот код через пять лет.
- Старайтесь не оставлять в модулях закоментированный «мертвый» код. Другие разработчики никогда не понимают, следовало ли удалить закоментированный код насовсем или это было сделано лишь временно для тестирования. Вернуться к участкам кода, которых больше нет в текущей версии, вам поможет система контроля версий.
- Если вам хочется сказать: «Я настоящий хакер» или «Это было действительно сложно», — то, вероятно, лучше не комментировать функцию, а переписать ее.

Корректное и полное документирование в коде отличает профессионала от того, кто просто играет в него. Дональд Кнут (Donald Knuth) как-то заметил, что хорошо написанная программа должна читаться как хорошо написанная книга. Хотя я не представляю себя захваченным сюжетом исходного кода TeX, я абсолютно согласен с мнением д-ра Кнута.

Я рекомендую вам изучить главу 19 или сногшибательную книгу Стива МакКоннелла (Steve McConnell) «Совершенный Код» (Code Complete. — Microsoft Press, 1993). В этой главе рассказано как я учился писать комментарии. С правильными

комментариями, даже если ваш программист сопровождения окажется психом, вам ничто не угрожает.

Раз уж мы обсуждаем комментарии, хочу заметить, как сильно я люблю комментарии XML-документации, введенные в C#, и как преступно то, что они не поддерживаются остальными языками от Microsoft. Надеюсь, в будущем все языки получат первоклассные комментарии XML-документации. Имея ясный формат комментариев, который может быть извлечен при компоновке, вы можете начать создание целостной документации для вашего проекта. По правде говоря, я так люблю комментарии XML-документации, что создал не очень сложный макрос `CommenTater` (см. главу 9), который добавляет и обновляет ваши комментарии XML-документации и следит, чтобы вы не забывали добавлять их.

Доверяй, но проверяй (Блочное тестирование)

Я всегда считал, что Энди Гроув (Andy Grove) — бывший председатель совета директоров Intel — был прав, назвав свою книгу «Выживают только одержимые» («Only the Paranoid Survive»). Это особенно верно для программистов. У меня много хороших друзей — прекрасных программистов, но когда дело касается взаимодействия их кода с моим, я проверяю их данные до последнего бита. Вообще-то у меня даже есть здоровый скепсис в отношении себя самого. С помощью утверждений, трассировки и комментариев я проверяю разработчиков своей команды, вызывающих мой код. С помощью блочного тестирования я проверяю себя. Блочные тесты — это строительные леса, которые вы возводите, чтобы вызвать ваш код из-за пределов программы как целого и убедиться, что код работает в соответствии с ожиданиями.

Первое, что я делаю для самопроверки, — начинаю писать блочные тесты одновременно с кодом, разрабатывая их параллельно. Определив интерфейс модуля, я пишу для него функции-заглушки (stub functions) и сразу создаю тестовую программу (или «обвязку» — harness) для вызова этих интерфейсов. Добавляя фрагменты функциональности, я добавляю новые варианты тестов в тестовую программу. С таким подходом я могу протестировать каждое следующее изменение в отдельности и распределить создание тестовой программы по циклу разработки. Если всю обычную работу вы делаете после реализации главного кода, то, как правило, у вас маловато времени для качественной работы над тестовой программой и реализации эффективного теста.

Второй способ проверить себя — подумать о том, как тестировать код, прежде чем его писать. Старайтесь не попасть в ловушку, думая, что, до того как вы сможете тестировать код, приложение должно быть написано полностью. Если вы обнаружили, что стали жертвой такого заблуждения, сделайте шаг назад и пересмотрите тестирование. Я понимаю, что иногда компиляция вашего кода зависит от важной функциональности другого разработчика. В таких случаях ваш тест должен состоять из заглушек для интерфейсов, с которыми возможна компиляция. Как минимум, запрограммируйте интерфейсы вручную, чтобы они возвращали нужные данные и вы смогли откомпилировать и запустить свой код.

Побочное преимущество от обеспечения тестируемости ваших разработок в том, что вы быстро находите проблемы, которые можете устранить, чтобы сде-

лать ваш код более расширяемым и пригодным для многократного использования. Поскольку многократное использование — это Святой Грааль программистов, то все, что бы вы ни сделали для повышения используемости вашего кода, будет не напрасно. Хороший пример такой удачи — `BugslayerStackTrace` из `BugslayerUtil.NET.DLL`. Когда я впервые реализовывал код трассировки в ASP.NET, я встроил код для просмотра стека в класс `ASPTraceListener`. При тестировании я быстро понял, что информация о стеке может понадобиться мне и в других местах. Я извлек код просмотра стека из `ASPTraceListener` и поместил в отдельный класс — `BugslayerStackTrace`. Когда мне потребовалось написать классы `BugslayerTextWriterTraceListener` и `BugslayerEventLogTraceListener`, у меня уже был базовый код, заранее созданный и полностью протестированный.

При кодировании следует выполнять блочные тесты постоянно. Кажется, я мыслю отдельными функциональными модулями примерно по 50 строчек кода. Каждый раз, добавляя или изменяя что-либо, я перезапускаю блочный тест, чтобы проверить, не нарушил ли я чего-нибудь. Я не люблю сюрпризов и поэтому стараюсь свести их к минимуму. Настоятельно рекомендую вам выполнять блочные тесты перед внесением своего кода в главные исходные файлы. В некоторых компаниях существуют специальные тесты внесения (*check-in tests*), которые должны выполняться до внесения кода. Я видел, как эти тесты внесения радикально снижали число неудавшихся компоновок и дымовых тестов.

Ключ к наиболее эффективному блочному тестированию заключается в двух словах: покрытие кода (*code coverage*). Если из этой главы вы не вынесете ничего, кроме этих двух слов, я буду считать, что она удалась. Покрытие кода — это просто процент строк, запущенных в вашем модуле. Если в вашем модуле 100 строк и вы запустили 85, то покрытие кода составляет 85%. Простая истина в том, что незапущенная строка — это строка, ждущая своей аварии.

Как консультанта, меня постоянно спрашивают, есть ли единый рецепт отличного кода. Сейчас я в том месте, откуда я впадаю в «религиозный экстаз», — настолько сильна моя вера в покрытие кода. Если бы вы сейчас стояли передо мной, то я бы прыгал вверх-вниз, восхваляя достоинства покрытия кода с евангелистским рвением. Многие разработчики говорили мне, что следование моему совету и попытки получить хорошее покрытие кода привели к резкому повышению качества кода. Это действует, и в этом весь секрет.

Получить статистику покрытия кода можно двумя способами. Первый способ сложный и включает использование отладчика и установку точек прерывания в каждой строке вашего модуля. По мере выполнения строк удаляйте точки прерывания. Продолжайте выполнять код, пока не удалите все точки прерывания, и вы получите стопроцентное покрытие. Легкий путь заключается в применении инструмента для покрытия от сторонних производителей, такого как `TrueCoverage` от Compuware NuMega, `Visual PureCoverage` от Rational или `C-Cover` от Bullseye. Лично я не вношу код в главные исходные файлы, пока не запущу минимум 85–90% строк моего кода. Знаю, некоторые из вас сейчас застонали. Да, получение хорошего покрытия кода может занять много времени. Иногда приходится выполнять гораздо больше тестов, чем вы когда-либо думали, и это может требовать времени. Получение хорошего покрытия подразумевает запуск вашего приложения в отладчике и изменение переменных с данными для запуска участков кода, до кото-

рых трудно добраться иначе. Однако ваша работа в том, чтобы писать целостный код, и, по моему мнению, покрытие кода, пожалуй, — единственный способ добиться этого на этапе блочного тестирования.

Нет ничего хуже бездействующего QA-персонала, застрявшего на аварийных сборках. Если в ходе блочного тестирования вы получите 90%-ое покрытие кода, ваши люди из отдела анализа качества могут использовать свое время для тестирования приложения на разных платформах и проверки работоспособности интерфейсов между подсистемами. Работа QA-отдела в том, чтобы тестировать продукт как единое целое и сосредоточиться на качестве в целом, а ваша — в том, чтобы протестировать модуль и сосредоточиться на качестве этого модуля. Когда обе стороны делают свою работу, результатом становится высококачественный продукт.

Ладно, я не жду, что разработчики будут проводить тесты на всех ОС Microsoft семейства Win32, которые могут применяться пользователями. Однако, если они смогут получить 90%-ое покрытие хотя бы для одной ОС, команда выиграет две трети борьбы за качество. Если вы не используете один из инструментов покрытия от сторонних производителей, вы обманываете себя с качеством.

Помимо покрытия кода, в своих проектах блочного тестирования я часто запускаю инструменты определения ошибок и проверки производительности от сторонних фирм (см. главу 1). Эти инструменты помогают мне гораздо раньше отлавливать ошибки в цикле разработки, поэтому я трачу меньше времени на общую отладку. Однако из всех инструментов определения ошибок и контроля производительности, что у меня есть, я использую продукты покрытия кода на несколько порядков чаще, чем что-либо еще. К тому времени как я получаю достаточную величину покрытия кода, я решаю почти все ошибки и проблемы с производительностью в коде.

Если вы будете следовать рекомендациям этого раздела, то к концу разработки получите вполне эффективные блочные тесты, но на этом работа не заканчивается. Если вы посмотрите на коды, прилагаемые к этой книге, то в главном каталоге с исходным кодом для каждого инструмента увидите каталог Tests. В этом каталоге хранятся мои блочные тесты для данного инструмента. Я сохраняю блочные тесты как часть кодовой базы, чтобы их легко могли найти. Кроме того, когда я вношу изменения в исходный код, то легко могу провести тест и проверить, не нарушил ли я чего-нибудь. Настоятельно рекомендую вам зарегистрировать ваши тесты в своей системе контроля версий. И наконец, хотя большинство блочных тестов вполне очевидно, не забудьте задокументировать все важные допущения, чтобы другие разработчики не тратили время на борьбу с вашими тестами.

Резюме

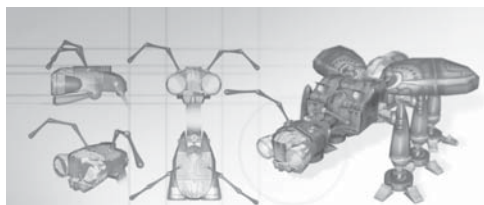
В этой главе были представлены лучшие технологии профилактического программирования, используемые для отладки при кодировании. Лучшая методика заключается в повсеместном применении утверждений, чтобы получить контроль над ошибочными ситуациями. Представленные коды утверждений .NET в `Bugslayer-Util.NET.DLL` и код `SUPERASSERT` устраняют все проблемы с утверждениями, предоставляемыми компиляторами Microsoft. В дополнение к утверждениям правильная трассировка и комментарии могут облегчить вам и другим людям сопровождение и отладку кода. Наконец, самый важный критерий оценки качества для программистов — блочное тестирование. Если вы сможете правильно протестировать свой код перед внесением его в главные исходные файлы, то избежите массы ошибок и проблем для обслуживающих инженеров в будущем.

Единственный способ правильно протестировать модуль — запустить при выполнении тестов инструмент для учета покрытия кода. До внесения кода в главные исходные файлы надо стараться получить покрытие минимум в 85–90%. Чем больше времени вы потратите на отладку при разработке, тем меньше его потребуется для отладки позднее.

ЧАСТЬ II

ПРОИЗВОДИТЕЛЬНАЯ ОТЛАДКА





Поддержка отладки ОС и как работают отладчики Win32

Изучение работы инструментария — ключевая часть нашей работы. Зная возможности инструментов, вы можете максимизировать их отдачу и меньше времени тратить на отладку. В основном отладчики очень помогают, но иногда они способны быть источником коварных проблем. Особенно интересна отладка неуправляемого кода, поскольку здесь вмешивается ОС и меняет поведение процессов, так как они работают под отладчиком. Кроме того, имеется весьма интересная поддержка внутри самой ОС, помогающая в некоторых сложных ситуациях при отладке. В этой главе я объясню, что такое отладчик, покажу, как работают отладчики в ОС Microsoft Win32, а также мы обсудим хитрые приемы, необходимые для эффективного использования средств отладки Win32.

После краткого обзора Win32-отладчиков я перейду к особенностям специальных функций, доступных при запуске процесса под отладчиком. Чтобы показать, как работают отладчики, я представлю пару, исходные коды которых находятся в прилагаемых к этой книге файлах примеров: MinDBG выполняет тот минимум функций, который позволяет ему называться отладчиком, а WDBG является примером настоящего отладчика Win32 и делает все, что положено, включая манипуляции с таблицами символов для просмотра локальных переменных и структур, управление точками прерывания, генерацию дизассемблированного кода, а также координацию с графическим интерфейсом пользователя (GUI). При обсуждении WDBG я также освещу такие темы, как работа точек прерывания, и расскажу о типах файлов символов. В завершение я расскажу о написанной мной очень крутой оболочке для сервера символов, которая упрощает работу с локальными

переменными и аргументами. Этот сервер был самым трудным кодом, написанным мной для этой книги, и я уверен, что вы найдете его весьма полезным!

Почему нет главы, посвященной отладчикам .NET?

Вы, возможно, удивляетесь, почему в этой книге нет главы, посвященной работе отладчиков Microsoft .NET. Сначала я предполагал написать такую главу, но в результате исследования отладочного API .NET (.NET Debugging API) я понял, что в отличие от практически недокументированных отладчиков Win32 команда разработчиков исполняющей среды .NET проделала огромную работу по описанию отладочного интерфейса .NET. Кроме того, приведенный здесь пример отладчика показывает, как сделать все, что требуется от отладчика .NET. Этот пример почти на 98% — консольный отладчик CORDBG. В нем нет только команд дизассемблирования неуправляемого кода. Работа над отладчиком .NET заняла у меня пару недель, и я быстро понял, что здесь делать нечего (разве что изложить своими словами прекрасную документацию по .NET) и мне не удастся показать что-либо новое, кроме того, что видно из примера CORDBG. Файлы Debug.doc и DebugRef.doc, описывающие отладочный API .NET, уже установлены на ваш компьютер в процессе установки Visual Studio .NET и находятся в каталоге <Каталог установки Visual Studio .NET>\SDK\v1.1\Tool Developers Guide\Docs.

И последнее. Прежде чем погрузиться в эту главу, я хочу определить два термина, которые буду использовать на протяжении всей книги: отладчик (debugger) и отлаживаемая программа (debuggee). Отладчик — это просто процесс, способный управлять другим процессом для его отладки, а отлаживаемая программа — это процесс, запускаемый под отладчиком. В некоторых ОС отладчик называют родительским процессом, а отлаживаемую программу — дочерним.

Типы отладчиков Windows

Если вы программировали для Win32, то, возможно, слышали о нескольких типах отладчиков. В мире Microsoft Windows доступны два типа: отладчики пользовательского режима и отладчики режима ядра.

Большинство программистов в основном знакомо с отладчиками пользовательского режима. Не будет сюрпризом узнать, что отладчики первого типа предназначены для отладки приложений пользовательского режима. Отладчики второго типа, как следует из их названия, позволяют отлаживать ядро ОС. Такие отладчики применяют главным образом разработчики драйверов.

Отладчики пользовательского режима

Отладчики пользовательского режима служат для отладки любых приложений, работающих в пользовательском режиме. Сюда входят любые программы с GUI, а также, что для вас будет неожиданностью, такие приложения, как службы Windows. Обычно отладчики пользовательского режима используют графические интерфейсы. Главный признак отладчиков пользовательского режима — это то, что они применяют отладочный API Win32. Так как ОС помечает отлаживаемую программу как

работающую в специальном режиме, вы можете вызвать функцию API `IsDebuggerPresent` для определения, работает ли ваш процесс под отладчиком. Проверка, работаете ли вы под отладчиком, может пригодиться, если вам требуется больше диагностической информации, только когда к вашему процессу подключен отладчик.

В Microsoft Windows 2000 и более ранних ОС проблема отладочного API Win32 заключается в том, что если процесс был однажды запущен под отладчиком и отладчик завершается, то отлаживаемая программа тоже завершается. Иначе говоря, отлаживаемая программа была постоянно отлаживаемой. Это ограничение было прекрасно, когда все работали над клиентскими приложениями, но оно было бедствием при отладке серверных приложений, особенно когда программисты пытались отлаживать рабочие серверы. В Microsoft Windows XP/Server 2003 и более поздних версиях вы можете подсоединять к работающим процессам и отсоединять от них все, что вам понадобится, без каких-либо условий. В Visual Studio .NET вы можете отсоединиться от процесса, выбрав `Detach` (отсоединить) в диалоговом окне `Processes` (процессы).

Интересно, что Visual Studio .NET теперь предлагает службу Visual Studio Debugger Proxy (`DbgProxy`) под Windows 2000, позволяющую отлаживать процесс, а затем от него отсоединиться. `DbgProxy` работает как отладчик, т. е. ваше приложение работает под отладчиком. Теперь вы и под Windows 2000 можете отсоединить, а затем повторно присоединить к процессу все, что надо. Но я все еще наблюдаю одну проблему программистов: независимо от используемой ими ОС (Windows XP/Server 2003 или `DbgProxy` под Windows 2000), они продолжают «вечную отладку», забывая задействовать преимущества новой возможности отсоединения.

Для интерпретирующих языков и исполняющих сред, применяющих принцип виртуальной машины, сами виртуальные машины предлагают полный комплект отладки и не используют отладочный API Win32. Вот некоторые примеры сред такого типа: виртуальные машины Java от Microsoft или Sun, механизм сценариев Microsoft для Web-приложений и, конечно, общезыковая исполняющая среда Microsoft .NET (common language runtime, CLR).

Как я уже говорил, отладка приложений .NET освещена в документах (каталог `Tool Developers Guide`). Я также не буду касаться отладочных интерфейсов Java и языков сценариев, которые выходят за рамки данной книги. О том, как писать отладчик сценариев, см. в MSDN тему «Microsoft Windows Script Interfaces-Introduction». Как и при отладке в CLR .NET, объекты отладчика сценариев предоставляют богатые интерфейсы для доступа к сценариям, в том числе встроенным в документы.

Отладочным API Win32 пользуется неожиданно большое количество программ. Сюда входят отладчик Visual Studio .NET при отладке неуправляемого кода, который я освещаю в деталях в главах 5 и 7, отладчик Windows (Windows Debugger, WinDBG), обсуждаемый в главе 8, `BoundsChecker` от Compuware NuMega, программа `Platform SDK Depends` (которая может быть установлена в составе Visual Studio .NET), отладчики Borland Delphi и C++ Builder, а также символьный отладчик NT (NT Symbolic Debugger, NTSD). Я уверен, имеется и много других.

Стандартный вопрос отладки

Как мне защитить Win32-программу от вмешательства отладчика?

Программисты, работающие на вертикальном рынке приложений с собственными алгоритмами чаще всего меня спрашивают о том, как защитить свои приложения и не дать конкурентам вмешаться в них с помощью отладчика. Вы, конечно, можете вызвать `IsDebuggerPresent`, который скажет, работает ли отладчик пользовательского режима, но если у человека есть хоть чуточку мозгов, то первое, что он сделает при восстановлении алгоритма, — заставит `IsDebuggerPresent` возвращать 0 и, таким образом, будет казаться, что отладчика нет.

Совершенного способа защититься от настырного хакера, имеющего физический доступ к вашим исполняемым кодам, нет, но вы хотя бы можете немного усложнить ему жизнь во время исполнения программы. Весьма интересно, что до сих пор во всех ОС Microsoft `IsDebuggerPresent` работает одинаково. Нет никакой гарантии, что они не изменят этого, но есть хорошие шансы, что все останется так же и в будущем.

Следующая функция, которую вы можете добавить к своему коду, делает то же, что и `IsDebuggerPresent`. Конечно же, добавление только этой функции не исключит возможности вмешиваться в ваш процесс с помощью отладчика. Чтобы затруднить отладку, между основными командами разбросаны другие безобидные команды, так что хакеры не смогут искать `IsDebuggerPresent` по последовательности байтов. Об антихакерских технологиях можно написать целую книгу. Однако, если вы можете провести «двухчасовой тест», означающий, что если среднему программисту требуется более двух часов на взлом вашего приложения, то ваше приложение, вероятно, защищено от всех хакеров, кроме самых настырных и талантливых.

`BOOL AntiHackIsDebuggerPresent (void)`

```
{
    BOOL bRet = TRUE ;
    __asm
    {
        // Получить блок информации потока (Thread Information block, TIB).
        MOV     EAX , FS:[00000018H]
        // Байты со смещением 0x30 в TIB – это поле указателя, который
        // указывает на структуру, имеющую отношение к отладчику.
        MOV     EAX , DWORD PTR [EAX+030H]
        // Второй DWORD в этой отладочной структуре указывает,
        // что процесс отлаживается.
        MOVZX   EAX , BYTE PTR [EAX+002H]
        // Возвращаем результат.
        MOV     bRet , EAX
    }
    return ( bRet ) ;
}
```

Отладчики режима ядра

Отладчики режима ядра располагаются между центральным процессором и ОС. Это значит, что, когда вы останавливаетесь в отладчике режима ядра, ОС тоже останавливается. Как можно себе представить, внезапная остановка ОС полезна, если вы работаете над проблемами согласования по времени и синхронизации.

Существуют три отладчика режима ядра: отладчик ядра KD, WinDBG и SoftICE.

Отладчик ядра KD

Windows 2000/XP/Server 2003 интересны тем, что на самом деле часть отладчика режима ядра является частью NTOSKRNL.EXE — главного файла ядра ОС. Этот отладчик доступен как в рабочей, так и в отладочной версии ОС. Для переключения в режим отладки ядра для систем на базе процессоров x86 установите параметр загрузки /DEBUG в файле BOOT.INI и дополнительно /DEBUGPORT при необходимости установить порт связи для отладчика режима ядра на порт, отличный от порта по умолчанию (COM1). KD работает на отдельной машине, называемой хостом, и взаимодействует с целевой машиной через нуль-модемный кабель или, скажем, через кабель интерфейса 1394 (FireWire) при работе с Windows XP/Server 2003.

Отладчик режима ядра NTOSKRNL.EXE делает достаточно для управления центральным процессором, позволяя отлаживать ОС. Основная работа по отладке — управление символами, обработка расширенных точек прерывания и дизассемблирование — происходит на стороне KD. Когда-то в Microsoft Windows NT 4 Device Driver Kit (DDK) был описан протокол связи через нуль-модемный кабель. Однако Microsoft больше не приводит описание этого протокола.

KD входит в Debugging Tools for Windows (отладочные средства Windows), которые можно загрузить с <http://www.microsoft.com/ddk/debugging> (текущая версия на момент написания этой книги также доступна на прилагаемом к книге компакт-диске). Вся сила KD становится очевидной при ознакомлении с командами, предлагаемыми им для доступа к внутренним состояниям ОС. Если вам когда-либо хотелось увидеть, что происходит в ОС, эти команды покажут вам это. Знание работы драйверов устройств Windows поможет разобраться с выводом этих команд. Интересно, что при всей своей мощи KD почти никогда не применялся за пределами Microsoft, так как это консольное приложение и им весьма утомительно пользоваться при отладке на уровне исходного кода. Однако для команд разработчиков ОС Microsoft этот отладчик ядра — единственный выбор.

WinDBG

WinDBG входит в состав Debugging Tools for Windows. Этот гибридный отладчик можно задействовать и как отладчик режима ядра, и как отладчик пользовательского режима, а при небольшой доработке WinDBG позволяет одновременно отлаживать программы режима ядра и пользовательского режима. При отладке в режиме ядра WinDBG предлагает все возможности KD, так как он обращается к тому же отладочному ядру, что и KD. Однако WinDBG предоставляет графический интерфейс, который вовсе не так легко задействовать, как отладчик Visual Studio .NET, хоть и проще, чем KD. WinDBG позволяет отлаживать драйверы устройств почти так же просто, как будто вы работаете с приложениями пользовательского режима.

Как отладчик пользовательского режима WinDBG весьма хорош, и я настоятельно рекомендую, чтобы вы установили его. WinDBG предлагает гораздо больше возможностей, чем отладчик Visual Studio .NET, так как предоставляет вам куда больше сведений о вашем процессе. Однако за это надо платить: WinDBG сложнее в использовании, чем отладчик Visual Studio .NET. И все же я бы посоветовал вам потратить некоторое время и силы на изучение WinDBG, а я вам покажу ключевые возможности и приемы работы с ним в главе 8. Эти затраты окупятся за счет того, что он поможет вам найти ошибку значительно быстрее, чем используя отладчик Visual Studio .NET. Я провожу около 95% времени в отладчике Visual Studio .NET, а остальное время — в WinDBG.

SoftICE

Этот отладчик режима ядра компании Compuware NuMega, как мне известно, — единственный коммерческий отладчик режима ядра на рынке. Это также единственный отладчик режима ядра, работающий на одной машине. В отличие от других отладчиков режима ядра SoftICE прекрасно отлаживает программы пользовательского режима. Как я уже говорил, отладчики режима ядра располагаются между центральным процессором и ОС. SoftICE также располагается между центральным процессором и ОС при отладке программ пользовательского режима, останавливая всю ОС.

Вас может не вдохновить то, что SoftICE может остановить ОС. Но давайте рассмотрим такой случай. Что, если вам нужно отлаживать чувствительный к временным задержкам код? При использовании такой функции API, как `SendMessageTimeout`, вы легко выйдете за пределы этого времени, пока вы проходите по шагам в другом потоке с помощью обычного отладчика с графическим интерфейсом. Используя SoftICE, вы можете ходить от оператора к оператору сколь угодно долго, так как таймер, от которого зависит исполнение `SendMessageTimeout`, не будет работать, пока вы работаете под SoftICE. SoftICE — единственный отладчик, позволяющий эффективно отлаживать многопоточные приложения. То, что SoftICE останавливает всю ОС, когда он активен, означает, что разрешение проблем согласования времени производится гораздо проще.

То, что SoftICE располагается между центральным процессором и ОС, упрощает и отладку межпроцессного взаимодействия. Если вы занимаетесь COM-программированием с множеством внешних серверов, вы можете просто устанавливать точки прерывания во всех процессах и ходить по шагам между ними. Наконец, в SoftICE вы запросто пройдете по шагам из пользовательского режима в режим ядра и обратно.

Другое важное преимущество SoftICE над другими отладчиками в том, что в нем собрана феноменальная коллекция информационных команд, которые позволяют увидеть практически все, что происходит в ОС. Хотя KD и WinDBG тоже имеют солидный набор таких команд, в SoftICE их гораздо больше. В SoftICE вы можете просмотреть практически все: от состояния всех событий синхронизации до полной информации о `HWND` и расширенной информации о любом потоке системы. SoftICE может рассказать вам все, что происходит в вашей системе.

Как можно ожидать, вся эта замечательная грубая сила имеет свою цену. SoftICE, как и любой отладчик режима ядра, имеет весьма крутую кривую обучения, так

как по существу он сам является ОС. Однако ваши затраты на обучение окупятся с лихвой от предоставляемых им преимуществ.

Поддержка отлаживаемых программ операционными системами Windows

В дополнение к определению API, который отладчик должен вызывать, чтобы считаться отладчиком, Windows предоставляет несколько возможностей, позволяющих найти проблемы в ваших приложениях. Некоторые из них не настолько известны и могут сбить вас с толку при первой встрече с ними.

Отладка Just-In-Time (JIT)

Из некоторых маркетинговых материалов по Visual Studio .NET может показаться, что в Visual Studio за JIT-отладкой скрывается чудо, однако чудеса происходят в самой ОС. При отказе приложения Windows анализирует состояние раздела реестра `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug`, чтобы определить, какой отладчик ей вызвать для отладки приложения. Если этот раздел пуст, Windows XP выводит стандартное диалоговое окно аварийного завершения, а Windows 2000 — информационное окно с адресом аварийного завершения. Если в этом разделе реестра задано значение и заполнены остальные значения, под Windows XP в левом нижнем углу становится активной кнопка `Debug` (отладка), и вы получаете возможность отлаживать приложение. Под Windows 2000 доступна кнопка `Cancel`, позволяющая запустить отладчик.

JIT-отладка использует три следующих важных значения в разделе `AeDebug` реестра:

- `Auto`;
- `UserDebuggerHotKey`;
- `Debugger`.

Если `Auto` содержит значение 0 (ноль), ОС генерирует стандартное диалоговое окно аварийного завершения и делает доступной кнопку `Cancel`, позволяя присоединить отладчик. При значении 1 (единица) отладчик запускается автоматически. Если вы хотите свести с ума кого-нибудь из своих коллег, установите незаметно на их системах значение `Auto`, равное 1, — они не будут понимать, почему при каждом аварийном завершении приложения у них запускается отладчик. Значение `UserDebuggerHotKey` идентифицирует «горячую» клавишу перехода к отладке (мы очень скоро обсудим ее использование). Последнее и самое важное значение `Debugger` указывает отладчик, который должна запускать ОС при аварийном завершении приложения. Есть только одно требование к отладчику: он должен поддерживать присоединение к процессу. После обсуждения значения `UserDebuggerHotKey` я объясню подробнее значение `Debugger` и его формат.

«Быстрые» клавиши прерывания и значение `UserDebuggerHotKey`

Иногда нужно переключиться на отладчик побыстрее. Если вы отлаживаете консольное приложение, нажатие `Ctrl+C` или `Ctrl+Break` вызовет специальное исключение `DBG_CONTROL_C`, которое переключит вас прямо в отладчик и позволит начать отладку.

У ОС Windows есть милая возможность: в приложениях с графическим интерфейсом вы можете переключиться на отладчик в любой момент времени. При работе под отладчиком по умолчанию нажатие клавиши F12 заставляет вызвать `DebugBreak` почти в тот момент, когда была нажата кнопка. Кстати, даже если вы используете клавишу F12 как акселератор или иначе обрабатываете ввод с клавиатуры сообщений для клавиши F12, вы все равно попадете в отладчик.

Клавиша прерывания по умолчанию — F12, но, если надо, можно указать и другую. Значение `UserDebuggerHotKey` есть цифровое значение `VK_*`, соответствующее клавише, которую вы желаете применять как «горячую» клавишу отладчика. Так, если вы хотите для переключения в отладчик задействовать `Scroll Lock`, установите значение `UserDebuggerHotKey` в `0x91` и для вступления нового значения в силу перезагрузите компьютер. Замечательной шуткой для ваших коллег может оказаться замена значения `UserDebuggerHotKey` на `0x45` (латинская буква E) — каждый раз, когда они нажмут клавишу E, программа переключится на отладчик. Однако я не несу никакой ответственности, если ваши коллеги ополчатся на вас и сделают вашу жизнь несчастной.

Значение Debugger

В разделе реестра `AeDebug` есть значение `Debugger`, которое и определяет основные действия. Сразу после установки ОС значение `Debugger` выглядит похожим на строку, передаваемую функции `API wsprintf: drwtsn32 -p %ld -e %ld -g`. Так оно и есть: `-p` является идентификатором аварийно завершающегося процесса, а `-e` — описатель события, нужный отладчику, чтобы сигнализировать, что в его цикле произошел выход из первого потока. Сигнал об этом событии сообщает ОС, что отладчик успешно присоединился к процессу. `-g` говорит программе Dr. Watson, что надо продолжить выполнение программы после присоединения.

Вы всегда можете изменить значение `Debugger`, чтобы вызывать другой отладчик. Чтобы сделать отладчик Visual Studio .NET «родным» отладчиком, откройте Visual Studio .NET и выберите `Options` из меню `Tool`. В диалоговом окне `Options` выберите папку `Debugging`, затем — страницу свойств `Just-In-Time` и убедитесь, что установлен флажок рядом с пунктом `Native`. Вы можете настроить WinDBG или Dr. Watson своим предпочтительным отладчиком путем запуска из командной строки `WinDBG -I` (заметьте: ключ чувствителен к регистру ввода) или `DRWTSN32 -I`. Изменив значение `Debugger`, обязательно завершите Task Manager (Диспетчер задач), если он выполнялся. Диспетчер задач кэширует раздел реестра `AeDebug` во время своей работы, поэтому, если вы попытаетесь отладить процесс из списка на страничке `Processes` (Процессы) Диспетчера задач, отладчик может не заработать, если предыдущим отладчиком был Visual Studio .NET.

Выбор отладчика, запускающегося при аварийном завершении

Хорошо иметь возможность оперативной отладки, когда отладчик вызывается при аварийном завершении приложения, но здесь есть существенное ограничение: вы можете иметь одновременно только один такой отладчик, задаваемый значением `Debugger`. Как мы увидим в следующих главах, отладчики имеют сильные и слабые стороны в зависимости от конкретной ситуации. Ничего не может быть хуже, если «выскочит» не тот отладчик, о котором вы знаете, что он позволит запросто найти ошибку, которую вы несколько недель пытались воспроизвести.

Это серьезная проблема, и я решил приложить руку к ее решению. Однако, поскольку все, похожее на ошибку, запускает JIT-отладку под Visual Studio .NET, я сделал много проб и ошибок, чтобы воплотить свою идею. Прежде всего расскажу, как работает программа Debugger Chooser или, для краткости, DBGCHOOSEr.

Идея, заложенная в DBGCHOOSEr, состоит в том, что она работает как программа-прокладка, вызываемая при аварийном завершении отлаживаемой программы и передающая настоящему отладчику информацию, нужную для отладки приложения. Для настройки DBGCHOOSEr сначала скопируйте ее в каталог своего компьютера, где она не может быть случайно удалена. ОС пытается запустить отладчик, заданный значением `Debugger` раздела реестра `AeDebug`, и, если отладчик недоступен, у вас не будет шансов отладить приложение в случае его аварийного завершения. Для инициализации DBGCHOOSEr просто запустите его (рис. 4-1). Первый запуск DBGCHOOSEr устанавливает умолчания, характерные для большинства машин программистов. Если какие-то ваши отладчики не указаны здесь, укажите их пути. Уделите особое внимание отладчику Visual Studio .NET, так как оболочка оперативного отладчика, используемая Visual Studio .NET, отсутствует в пути по умолчанию. По щелчку кнопки ОК в диалоговом окне настройки DBGCHOOSEr записывает параметры отладчика в INI-файл, хранящийся в каталоге Windows, и настраивает себя отладчиком по умолчанию в разделе реестра `AeDebug`.

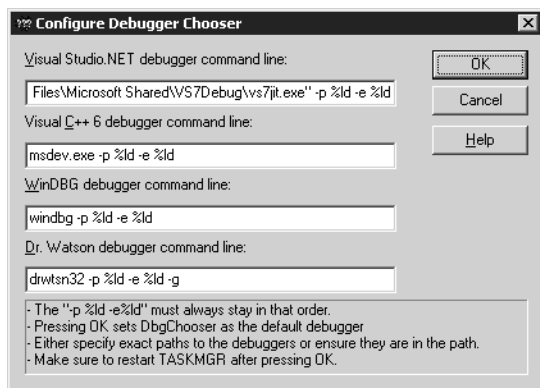


Рис. 4-1. Диалоговое окно настройки DBGCHOOSEr

Как только случится одно из редких (я надеюсь) аварийных завершений, после щелчка кнопки `Debug` диалогового окна аварийного завершения вы увидите диалоговое окно выбора отладчика (рис. 4-2). Просто выберите нужный отладчик и начните отладку.

В реализации DBGCHOOSEr нет ничего особенного. Первое, что может заинтересовать, это то, что, когда вызывается `CreateProcess` для выбранного пользователем отладчика, нужно обеспечить установку флага наследования описателей в `TRUE`. Чтобы с описателями все было классно, я заставил DBGCHOOSEr ждать завершения порожденного отладчика. Таким образом, я знаю, что все наследуемые описатели сохранены для отладчика. Хотя прийти к этой идее было труднее, чем ее реализовать, чтобы заставить Visual Studio .NET правильно работать, пришлось немного потрудиться. Все классно работало с WinDBG, Microsoft Visual C++ 6 и

Dr. Watson, но, когда я подошел к Visual Studio .NET (на самом деле к VS7JIT.EXE, который в свою очередь вызывает отладчик Visual Studio .NET), стало выскакивать сообщение, что JIT-отладка заблокирована и отладку запустить невозможно.

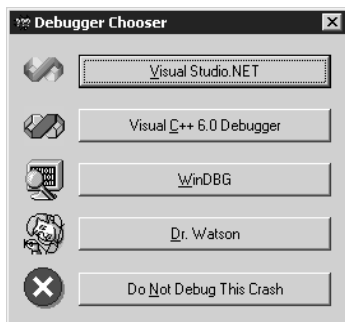


Рис. 4-2. Диалоговое окно выбора отладчика программы DBGCHOOSE

Во-первых, я был в некотором замешательстве от того, что происходит, но с помощью прекрасной программы мониторинга реестра Regmon от Марка Руссиновича и Брайса Когсвелла с www.sysinternals.com я увидел, что VS7JIT.EXE проверял значение `Debugger` раздела реестра `AeDebug`, установлен ли он как оперативный отладчик. Если нет, выскакивало сообщение о том, что оперативная отладка заблокирована. У меня была возможность проверить, что это так, остановив DBGCHOOSE в отладчике, когда он был активизирован благодаря аварийному завершению, и изменив значение раздела реестра `Debugger` так, чтобы он указывал на VS7JIT.EXE. Я не понимал, почему VS7JIT.EXE считает это столь важным, что он не может заниматься отладкой, если он не является оперативным отладчиком. Я быстренько написал в DBGCHOOSE, как обмануть VS7JIT.EXE путем подмены значения `Debugger` на VS7JIT.EXE перед его порождением, и все в этом мире стало прекрасно. Чтобы сделать DBGCHOOSE.EXE вновь оперативным отладчиком, я создал поток, который ждет 5 секунд и восстанавливает значение `Debugger`.

Как я упоминал, когда завел речь о DBGCHOOSE, мое решение несовершенно из-за проблем в оперативном отладчике Visual Studio .NET. В Windows XP я проверял различные варианты запуска и работы Visual Studio .NET, но нашел, что VS7JIT.EXE прекращает свою работу. Поиграв с ним немного, я понял, что в действительности исполняются два экземпляра VS7JIT.EXE, в то время как Visual Studio .NET запускается как оперативный отладчик. Один экземпляр порождает Visual Studio .NET IDE (среду интерактивной разработки), а другой работает под DCOM-сервером RPCSS. В редких случаях, только при тестировании готовой реализации, я приводил систему в состояние, когда попытка породить VS7JIT.EXE была безуспешной, так как не мог запуститься экземпляр DCOM. В основном я сталкивался с этой проблемой, работая над кодом восстановления значения `Debugger` раздела реестра `AeDebug`. Идя по такому пути реализации DBGCHOOSE, я столкнулся с этой проблемой пару раз, и только когда тестировал различные случаи одновременного аварийного завершения нескольких процессов. Я не смог вычислить точную причину и никогда не видел этого при нормальной работе.

Автоматический запуск отладчика (опции исполнения загружаемого модуля)

Трудней всего отлаживать приложения, запускаемые другими процессами. В эту категорию попадают службы Windows и внепроцессные СОМ-серверы. Зачастую можно вызвать API-функцию `DebugBreak`, чтобы заставить отладчик присоединиться к вашему процессу. Однако `DebugBreak` не работает с двумя экземплярами. Во-первых, иногда она не работает со службами Windows. Если вам надо отлаживать процедуру запуска службы, то вызов `DebugBreak` позволит отладчику присоединиться, но время, затраченное отладчиком на свой запуск, может превысить тайм-аут запуска службы, и Windows ее остановит. Во-вторых, `DebugBreak` не работает, если вам нужно отлаживать внепроцессный СОМ-сервер. При вызове `DebugBreak` обработчик ошибок СОМ обнаружит исключение, возникающее в точке прерывания и прекратит выполнение внешнего СОМ-сервера. К счастью, Windows позволяет указать, что приложение должно запускаться в отладчике. Это позволяет запустить отладчик прямо с первого оператора. Прежде чем разрешить эту возможность, убедитесь, что при конфигурировании своей службы вы разрешили ей взаимодействовать с рабочим столом.

Для настройки автоматической отладки лучше всего указать эту опцию в редакторе реестра. В разделе `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options` создайте свой раздел, имя которого совпадает с именем файла вашего приложения. Так, если ваше приложение называется `FOO.EXE`, создайте раздел реестра `FOO.EXE`. В разделе реестра вашего приложения создайте новое строковое значение `Debugger` и введите в нем полный путь и имя файла выбранного вами отладчика.

Теперь, когда вы запускаете свое приложение, отладчик запускается автоматически при загрузке приложения. Если надо указать отладчику какие-либо параметры командной строки, укажите их также в значении `Debugger`. Например, если вы хотите задействовать WinDBG и автоматически инициировать отладку сразу после запуска WinDBG, заполните `Debugger` строкой `d:\windbg\windbg.exe -g`.

Для использования Visual Studio .NET в качестве предпочтительного отладчика придется сделать немного больше. Первая проблема в том, что Visual Studio .NET не может отлаживать исполняемый модуль без файла решения. Если вы разрабатываете исполняемый модуль (иначе говоря, вы располагаете решением и исходным кодом), можете применить это решение. Однако последняя открытая компоновка и будет запускаться. Значит, если вам надо отлаживать поставляемую компоновку (release build) или двоичный образ, для которого у вас нет исходного кода, откройте проект, настройте активное решение как Release и закройте решение. Если вы не располагаете файлом решения для исполняемого файла, в меню File выберите пункт Open Solution и откройте исполняемый образ как решение. Запустите отладку и, когда появится запрос сохранения файла решения, сохраните его.

Имея решение, вы сможете им пользоваться, а командная строка, указываемая в параметре `Debugger`, будет выглядеть следующим образом. Если вы не добавили вручную каталог Visual Studio .NET <Каталог установки VS.NET>\Common7\IDE к системной переменной среды `PATH`, укажите полный путь и каталог для `DEVENV.EXE`.

Ключ `/run` командной строки `DEVENV.EXE` заставляет его начать отладку решения, указанного в командной строке.

```
g:\vsnet\common7\ide\devenv /run d:\disk\output\wdbg.sln
```

Вторая проблема, с которой вы встретитесь, в том, что строковый параметр `Debugger` может быть не более 65 символов в длину. Если вы установили Visual Studio .NET по умолчанию, то почти наверняка путь будет очень длинным. Все, что вам нужно сделать, — это поработать с командой `SUBST` и назначить пути к `DEVENV.EXE` и вашему решению буквам устройств.

Ветераны могут помнить, что параметр `Debugger` легко установить с помощью `GFLAGS.EXE` — небольшой утилиты, поставляемой вместе с WinDBG. Увы, `GFLAGS.EXE` работает неправильно и принимает командную строку длиной только до 25 символов для параметра `Debugger`. В итоге проще всего создать раздел реестра для процесса и параметр `Debugger` вручную.

Стандартный вопрос отладки

Мой шеф посылает мне так много почты, что я не могу ничего делать. Есть ли какой-нибудь способ замедлить эту жуткую почту от ШСИ?

Хотя многие начальники «стараются сделать как лучше», их непрекращающиеся сообщения по электронной почте могут отвлекать вас и не давать вам работать. К счастью, есть простое решение, которое очень хорошо работает и даст вам около недели замечательного спокойствия, в результате вы сможете работать, укладываясь в сроки. Чем менее технически опытен шеф и администраторы сети, тем больше времени вы получите.

В предыдущем разделе я говорил о разделе реестра `Image File Execution Options` и о том, что, когда вы настроите параметр `Debugger` вашего процесса, процесс будет автоматически запускаться под отладчиком. Вот как избавиться от почты ШСИ (шефа, сидящего как на иголках).

1. Зайдите в кабинет шефа.
2. Откройте `REGEDIT.EXE`. Если шеф в кабинете, объясните ему, что вам надо запустить на его машине утилиту, которая позволит ему получить доступ к Web-сервисам XML, над которыми вы работаете (на самом деле не важно, создадите вы Web-сервисы XML или нет — одни только эти модные словечки заставят босса охотно предоставить вам возможность поковыряться в его машине).
3. В разделе `Image File Execution Options` создайте раздел `OUTLOOK.EXE` (замените его на имя другой почтовой программы, если используется не Microsoft Outlook). Скажите боссу, что вы делаете это для того, чтобы предоставить ему почтовый доступ к Web-сервисам XML.
4. Создайте параметр `Debugger` и введите значение `SOLE.EXE`. Скажите шефу, что SOL нужен для того, чтобы ваши Web-сервисы XML получили доступ к машинам Sun Solaris.
5. Закройте `REGEDIT.EXE`.
6. Скажите шефу, что у него все настроено и он может пользоваться Web-сервисами XML. Теперь главное — удалиться из кабинета с серьезным

см. след. стр.

лицом. (Не дать себе рассмеяться во время этого эксперимента значительно труднее, чем кажется, поэтому сначала попрактикуйтесь на своих коллегах!)

В этой ситуации вы всего-навсего сделали так, что при каждом запуске шефом Outlook в действительности будет запускаться Косынка (Solitaire). (Так как большинство руководителей все равно проводит свое рабочее время, играя в Косынку, ваш шеф отвлечется на пару игр прежде, чем до него дойдет, что он хотел запустить Outlook.) Возможно, он так и будет щелкать ярлык Outlook, пока не откроет столько копий Косынки, что ему не хватит виртуальной памяти и понадобится перезагрузить машину. После парочки таких дней многократных циклов щелчков ярлыка и перезагрузки машины ваш шеф вызовет к себе администратора сети посмотреть его машину.

Администратор возбудится, потому что теперь он имеет задачу поинтереснее, чем сбрасывать пароли барышням из бухгалтерии. Он будет забавляться в кабинете шефа с его машиной по меньшей мере день, удерживая таким образом шефа в стороне от машины. Если кто-то спросит ваше мнение, вот готовый ответ: «Я слышал о странностях взаимодействия EJB и NTFS через основы архитектуры DCOM, необходимой для доступа к MFT с использованием алгоритма сортировки методом наименьших квадратов». Администратор заберет у шефа его машину и несколько дней будет развлекаться с ней на своем рабочем месте. В конце концов он заменит жесткий диск и переустановит все заново, на что уйдет еще день-два. К тому времени, когда шеф получит свою машину обратно, у него скопится почта за четыре дня, на разбор которой у него уйдет еще минимум один день, а вы можете спокойно игнорировать сообщения еще день или два. Если же почта ШСИ опять начинает учащаться, просто повторите вышеперечисленные шаги еще раз.

Важное замечание: вы используете этот метод на свой страх и риск.

MiniDBG — простой отладчик Win32

На первый взгляд, отладчик Win32 — простая программа, к которой предъявляется всего парочка требований. Первое: отладчик должен устанавливать специальный флаг `DEBUG_ONLY_THIS_PROCESS` в параметре `dwCreationFlags` функции `CreateProcess`. Этот флаг сообщает ОС, что вызывающий поток должен войти в цикл отладки для управления запущенным процессом. Если отладчик может управлять несколькими процессами, порожденными изначальной отлаживаемой программой, он должен указывать флаг `DEBUG_PROCESS` при создании процесса.

Поскольку используется вызов `CreateProcess`, отладчик и отлаживаемая программа исполняются в разных процессах, благодаря чему устойчивость Win32-систем в процессе отладки весьма высока. Даже если отлаживаемая программа производит беспорядочную запись в память, она все равно не сможет привести к сбою отладчика. (Отладчики 16-разрядных версий Windows и ОС Macintosh до OS X весьма чувствительны к повреждению отлаживаемых программ, поскольку исполняются в одном процессе с ними.)

Второе требование: после запуска отлаживаемой программы отладчик должен войти в свой цикл путем вызова функции API `WaitForDebugEvent` для приема отладочных уведомлений. Завершив обработку некоторого события отладки, он вызывает `ContinueDebugEvent`. Имейте в виду, что функции отладочного API могут быть вызваны только тем потоком, что установил специальные флаги отладки при создании процесса путем вызова `CreateProcess`. Вот какой небольшой по объему код нужен для создания отладчика Win32:

```
void main ( void )
{
    CreateProcess ( ..., DEBUG_ONLY_THIS_PROCESS, ... );

    while ( 1 == WaitForDebugEvent ( ... ) )
    {
        if ( EXIT_PROCESS )
        {
            break ;
        }
        ContinueDebugEvent ( ... ) ;
    }
}
```

Как видите, минимальный отладчик Win32 не требует многопоточности, пользовательского интерфейса или чего-либо еще. И все же, как и в большинстве Windows-приложений, разница между минимальным и приемлемым значительна. В действительности отладочный API Win32 почти требует, чтобы цикл отладчика работал в отдельном потоке. Как следует из имени, `WaitForDebugEvent` (ждать события отладки) блокирует внутренние события ОС, пока отлаживаемая программа не выполнит действия, заставляющие ОС остановить исполнение отлаживаемой программы, после чего ОС может сообщить отладчику об этом событии. Если отладчик имеет единственный поток, то пользовательский интерфейс полностью заморожен, пока в отлаживаемой программе не возникнет событие отладки.

Все время в режиме ожидания отладчик принимает уведомления о событиях в отлаживаемой программе. Следующая структура `DEBUG_EVENT`, заполняемая функцией `WaitForDebugEvent`, содержит всю информацию о событии отладки (табл. 4-1):

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
    };
};
```



```
        RIP_INFO RipInfo;
    } u;
} DEBUG_EVENT
```

Табл. 4-1. События отладки

Событие отладки	Описание
<i>CREATE_PROCESS_DEBUG_EVENT</i>	<p>Генерируется, когда в рамках отлаживаемого процесса создается новый процесс или когда отладчик начинает отладку уже активного процесса. Ядро системы генерирует это событие отладки до начала выполнения процесса в пользовательском режиме и до того, как ядро генерирует другие события отладки для нового процесса.</p> <p>Структура <i>DEBUG_EVENT</i> содержит структуру <i>CREATE_PROCESS_DEBUG_INFO</i>, содержащую описатель нового процесса, описатель файла образа исполняемого процесса, описатель начального потока процесса и другую информацию, описывающую процесс.</p> <p>Описатель процесса имеет права доступа <i>PROCESS_VM_READ</i> и <i>PROCESS_VM_WRITE</i>. Если отладчик имеет те же права доступа к описателю процесса, он может читать память процесса и производить запись в нее через функции <i>ReadProcessMemory</i> и <i>WriteProcessMemory</i>.</p> <p>Описатель исполняемого файла процесса имеет права доступа <i>GENERIC_READ</i> и открыт для совместного чтения.</p> <p>Описатель начального потока процесса имеет права доступа к потоку <i>THREAD_GET_CONTEXT</i>, <i>THREAD_SET_CONTEXT</i> и <i>THREAD_SUSPEND_RESUME</i>. Если отладчик имеет эти типы доступа к потоку, он читает регистры потока и записывает в них с помощью функций <i>GetThreadContext</i> и <i>SetThreadContext</i>, а также может приостанавливать поток и возобновлять его исполнение с помощью функций <i>SuspendThread</i> и <i>ResumeThread</i>.</p>
<i>CREATE_THREAD_DEBUG_EVENT</i>	<p>Генерируется, когда в отлаживаемом процессе создается новый поток или когда начинается отладка уже активного процесса. Это событие отладки генерируется до того, как новый поток начнет свое исполнение в пользовательском режиме.</p> <p>Структура <i>DEBUG_EVENT</i> содержит структуру <i>CREATE_THREAD_DEBUG_INFO</i>. Последняя содержит описатель нового потока и его адрес запуска. Описатель имеет права доступа к потоку <i>THREAD_GET_CONTEXT</i>, <i>THREAD_SET_CONTEXT</i> и <i>THREAD_SUSPEND_RESUME</i>. Если отладчик имеет эти же права, он может читать регистры потока и записывать в них с помощью функций <i>GetThreadContext</i> и <i>SetThreadContext</i>, а также приостанавливать исполнение потока и возобновлять его с помощью функций <i>SuspendThread</i> и <i>ResumeThread</i>.</p>

Табл. 4-1. События отладки (продолжение)

Событие отладки	Описание
<i>EXCEPTION_DEBUG_EVENT</i>	<p>Генерируется, когда в отлаживаемом процессе возникает исключение. Возможные исключения включают попытку обращения к недоступной памяти, исполнение оператора, на котором установлена точка прерывания, попытку деления на 0 и любые другие исключения, перечисленные в разделе документации MSDN «Structured Exception Handling» (структурная обработка исключений).</p> <p>Структура <i>DEBUG_EVENT</i> содержит структуру <i>EXCEPTION_DEBUG_INFO</i>. Последняя описывает исключение, вызвавшее событие отладки.</p> <p>Кроме стандартных условий возникновения исключений, может происходить дополнительное исключение в процессе отладки консольного приложения. При вводе с консоли Ctrl+C ядро генерирует исключение <i>DBG_CONTROL_C</i> для процессов, обрабатывающих в процессе отладки сигнал Ctrl+C. Этот код исключения не предназначен для обработки в приложениях. Приложение никогда не должно иметь обработчик этого исключения. Оно нужно только отладчику и применяется, только когда отладчик присоединен к консольному процессу.</p> <p>Если процесс не находится в состоянии отладки или если отладчик оставляет исключение <i>DBG_CONTROL_C</i> необработанным, производится поиск списка функций-обработчиков исключений приложения. (О функциях-обработчиках исключений консольного процесса см. документацию MSDN по функции <i>SetConsoleCtrlHandler</i>.)</p>
<i>EXIT_PROCESS_DEBUG_EVENT</i>	<p>Возникает, когда завершается последний поток процесса или вызывается функция <i>ExitProcess</i>. Оно возникает сразу после того, как ядро выгружает все DLL процесса и обновляет код завершения процесса.</p> <p>Структура <i>DEBUG_EVENT</i> содержит структуру <i>EXIT_PROCESS_DEBUG_INFO</i>, описывающую код завершения процесса.</p> <p>При возникновении этого события отладчик освобождает все внутренние структуры, ассоциированные с процессом. Описатель, указывающий в отладчике на завершающийся процесс и описатели всех потоков этого процесса, закрываются ядром. Отладчик не должен закрывать эти описатели.</p>
<i>EXIT_THREAD_DEBUG_EVENT</i>	<p>Возникает, когда завершается поток, являющийся частью отлаживаемого процесса. Ядро генерирует это событие сразу после обновления кода завершения потока.</p> <p>Структура <i>DEBUG_EVENT</i> содержит структуру <i>EXIT_THREAD_DEBUG_INFO</i>, описывающую код завершения потока.</p> <p>При возникновении этого события отладчик освобождает все внутренние структуры, ассоциированные с потоком. Описатель, указывающий в отладчике на завершающийся процесс, закрывается системой. Отладчик не должен закрывать этот описатель.</p>

см. след. стр.

Табл. 4-1. События отладки (продолжение)

Событие отладки	Описание
	Событие отладки не возникает, если завершающийся поток является последним потоком процесса. В этом случае вместо него возникает событие отладки <i>EXIT_PROCESS_DEBUG_EVENT</i> .
<i>LOAD_DLL_DEBUG_EVENT</i>	<p>Возникает при загрузке DLL отлаживаемым процессом. Это событие возникает, когда системный загрузчик разрешает ссылки на DLL или когда отлаживаемый процесс вызывает функцию <i>LoadLibrary</i>, а также при каждой загрузке DLL в адресное пространство процесса. Если счетчик ссылок на DLL уменьшается до 0, DLL выгружается. При следующей загрузке DLL снова возникает это событие.</p> <p>Структура <i>DEBUG_EVENT</i> содержит структуру <i>LOAD_DLL_DEBUG_INFO</i>, которая включает описатель файла вновь загруженной DLL, ее базовый адрес и другие данные, описывающие DLL.</p> <p>Обычно при обработке этого события отладчик загружает таблицу символов, ассоциированную с DLL.</p>
<i>OUTPUT_DEBUG_STRING_EVENT</i>	<p>Возникает, когда отлаживаемый процесс обращается к функции <i>OutputDebugString</i>.</p> <p>Структура <i>DEBUG_EVENT</i> содержит структуру <i>OUTPUT_DEBUG_STRING_INFO</i>, которая описывает адрес, размер и формат отладочной строки.</p>
<i>UNLOAD_DLL_DEBUG_EVENT</i>	<p>Возникает, когда отлаживаемый процесс выгружает DLL с помощью функции <i>FreeLibrary</i>. Это событие возникает только при последней выгрузке DLL из адресного пространства процесса (т. е. когда счетчик ссылок на DLL станет равным 0).</p> <p>Структура <i>DEBUG_EVENT</i> содержит структуру <i>UNLOAD_DLL_DEBUG_INFO</i>, которая описывает базовый адрес DLL в адресном пространстве процесса, выгружающего DLL.</p> <p>Обычно при получении этого события отладчик выгружает таблицу символов, ассоциированную с DLL.</p> <p>При завершении процесса ядро автоматически выгружает все DLL процесса, но не генерирует событие отладки <i>UNLOAD_DLL_DEBUG_EVENT</i>.</p>

При обработке событий отладки, возвращаемых функцией `WaitForDebugEvent`, отладчик полностью управляет отлаживаемой программой, так как ОС останавливает все потоки отлаживаемой программы и не управляет ими, пока не вызвана функция `ContinueDebugEvent`. Если отладчику нужно читать из адресного пространства отлаживаемой программы или записывать в него, он может вызвать функции `ReadProcessMemory` и `WriteProcessMemory`. Если память имеет атрибут «только для чтения», можно использовать функцию `VirtualProtect`, чтобы изменить уровень защиты при необходимости произвести запись в эту часть памяти. Если отладчик редактирует код отлаживаемой программы, используя вызовы функции `WriteProcessMemory`, надо вызывать функцию `FlushInstructionCache` для очистки кэша ко-

манд для этой части памяти. Если вы забыли вызвать `FlushInstructionCache`, ваши изменения смогут работать, только если эта память не кэшируется центральным процессором. Если память уже кэширована центральным процессором, изменения не вступят в силу до повторного считывания в кэш центрального процессора. Вызов `FlushInstructionCache` особенно важен в многопроцессорных машинах. Если отладчику нужно получить или установить текущий контекст отлаживаемой программы или регистров центрального процессора, он может вызвать `GetThreadContext` или `SetThreadContext`.

Единственным событием отладки Win32, которому требуется особая обработка, является точка прерывания загрузчика, или начальная точка прерывания. После того как ОС посылает первые уведомления `CREATE_PROCESS_DEBUG_EVENT` и `LOAD_DLL_DEBUG_EVENT` для неявно загруженных модулей, отладчик принимает `EXCEPTION_DEBUG_EVENT`. Это событие отладки и является точкой прерывания загрузчика. Отлаживаемая программа исполняет эту точку прерывания, так как `CREATE_PROCESS_DEBUG_EVENT` указывает только, что процесс загружен, а не что он исполняется. Точка прерывания загрузчика, которую ОС заставляет сработать при каждой загрузке отлаживаемой программы, является тем первым событием, благодаря которому отладчик узнает, что отлаживаемая программа уже исполняется. В настоящих отладчиках инициализация основных структур данных, таких как таблицы символов, происходит при создании процесса, и отладчик начинает показывать дизассемблированный код или редактировать код отлаживаемой программы в точке прерывания загрузчика.

При возникновении точки прерывания загрузчика отладчик должен зарегистрировать, что он «видел» точку прерывания и может обрабатывать все остальные точки прерывания. Вся остальная обработка первой точки прерывания (а в общем, и остальных точек) зависит от типа центрального процессора. Для семейства Intel Pentium отладчик должен продолжить исполнение путем вызова функции `ContinueDebugEvent` с указанием флага `DBG_CONTINUE`, что позволит продолжить исполнение отлаживаемой программы.

Листинг 4-1 демонстрирует MinDBG — минимальный отладчик, доступный в наборе файлов к этой книге. MinDBG обрабатывает все события отладки и правильно исполняет отлаживаемый процесс. Кроме того, он показывает, как присоединиться к существующему процессу и отсоединиться от отлаживавшегося процесса. Для запуска процесса под MinDBG передайте имя процесса в командной строке с нужными отлаживаемой программе параметрами. Для присоединения к существующему процессу и его отладки, укажите в командной строке десятичный идентификатор процесса, предвадив его символом «минус» («-»). Так, если идентификатор процесса равен 3245, вам надо передать в командной строке -3245, чтобы заставить отладчик присоединиться к этому процессу. Если вы работаете под Windows XP/Server 2003 и более поздними системами, можете отсоединиться от процесса простым нажатием Ctrl+Break. Имейте в виду, что при работе с MinDBG на самом деле обработчики событий отладки не делают ничего, кроме как показывают некоторую базовую информацию. Превращение минимального отладчика в настоящий потребует значительных усилий.

Листинг 4-1. MINDBG.CPP

```

/*-----
Отладка приложений для Microsoft .NET и Microsoft Windows
Copyright (c) 1997-2003 John Robbins - All rights reserved.

Самый простой в мире отладчик программ Win32
-----*/

/*////////////////////////////////////
// Обычные включаемые файлы.
////////////////////////////////////*/
#include "stdafx.h"

/*////////////////////////////////////
// Прототипы и типы.
////////////////////////////////////*/
// Показывает минимальную справку.
void ShowHelp ( void ) ;

// Обработчик нажатия Break.
BOOL WINAPI CtrlBreakHandler ( DWORD dwCtrlType ) ;

// Функции отображения.
void DisplayCreateProcessEvent ( CREATE_PROCESS_DEBUG_INFO & stCPDI ) ;
void DisplayCreateThreadEvent ( DWORD dwTID ,
                                CREATE_THREAD_DEBUG_INFO & stCTDI ) ;
void DisplayExitThreadEvent ( DWORD dwTID ,
                              EXIT_THREAD_DEBUG_INFO & stETDI ) ;
void DisplayExitProcessEvent ( EXIT_PROCESS_DEBUG_INFO & stEPDI ) ;
void DisplayDllLoadEvent ( HANDLE hProcess ,
                          LOAD_DLL_DEBUG_INFO & stLDDI ) ;
void DisplayDllUnloadEvent ( UNLOAD_DLL_DEBUG_INFO & stULDDI ) ;
void DisplayODSEvent ( HANDLE hProcess ,
                      OUTPUT_DEBUG_STRING_INFO & stODSI ) ;
void DisplayExceptionEvent ( EXCEPTION_DEBUG_INFO & stEDI ) ;

// Определение типа для DebugActiveProcessStop.
typedef BOOL (WINAPI *PFNDEBUGACTIVEPROCESSSTOP)(DWORD) ;

/*////////////////////////////////////
// Глобальные переменные области видимости файла.
////////////////////////////////////*/
// Флаг, показывающий необходимость отсоединения.
static BOOL g_bDoTheDetach = FALSE ;

/*////////////////////////////////////
// Точка входа.
////////////////////////////////////*/
void _tmain ( int argc , TCHAR * argv[ ] )
{

```

```
// Проверка наличия аргументов в командной строке.
if ( 1 == argc )
{
    ShowHelp ( ) ;
    return ;
}

// Необходим достаточно большой буфер для команды
// или параметров командной строки.
TCHAR szCmdLine[ MAX_PATH + MAX_PATH ] ;
// Идентификатор процесса, если производится присоединение к нему.
DWORD dwPID = 0 ;

szCmdLine[ 0 ] = _T ( '\\0' ) ;

// Проверка, начинается ли командная строка со знака "-", так как это
// означает идентификатор процесса, к которому мы присоединяемся.
if ( _T ( '-' ) == argv[1][0] )
{
    // Попытка вычлнить идентификатор процесса из командной строки.

    // Передвинуться за символ '-' в строке.
    TCHAR * pPID = argv[1] + 1 ;
    dwPID = _tstol ( pPID ) ;
    if ( 0 == dwPID )
    {
        _tprintf ( _T ( "Invalid PID value : %s\\n" ) , pPID ) ;
        return ;
    }
}
else
{
    dwPID = 0 ;

    // Я собираюсь запустить процесс.
    for ( int i = 1 ; i < argc ; i++ )
    {
        _tcscat ( szCmdLine , argv[ i ] ) ;
        if ( i < argc )
        {
            _tcscat ( szCmdLine , _T ( " " ) ) ;
        }
    }
}

// Место для возвращаемого значения.
BOOL bRet = FALSE ;

// Установить обработчик CTRL+BREAK.
bRet = SetConsoleCtrlHandler ( CtrlBreakHandler , TRUE ) ;
```

см. след. стр.

```

if ( FALSE == bRet )
{
    _tprintf ( _T ( "Unable to set CTRL+BREAK handler!\n" ) );
    return ;
}

// Если идентификатор процесса равен 0, я запускаю процесс.
if ( 0 == dwPID )
{
    // Попытаемся запустить отлаживаемый процесс. Этот вызов функции
    // выглядит, как обычный вызов CreateProcess, кроме специального
    // необязательного флага DEBUG_ONLY_THIS_PROCESS.
    STARTUPINFO      stStartInfo      ;
    PROCESS_INFORMATION stProcessInfo  ;

    memset ( &stStartInfo , NULL , sizeof ( STARTUPINFO      ) );
    memset ( &stProcessInfo , NULL , sizeof ( PROCESS_INFORMATION));

    stStartInfo.cb = sizeof ( STARTUPINFO ) ;

    bRet = CreateProcess ( NULL
                           ,
                           szCmdLine
                           ,
                           NULL
                           ,
                           NULL
                           ,
                           FALSE
                           ,
                           CREATE_NEW_CONSOLE |
                           DEBUG_ONLY_THIS_PROCESS
                           ,
                           NULL
                           ,
                           NULL
                           ,
                           &stStartInfo
                           ,
                           &stProcessInfo
                           ) ;

    // Не забудьте закрыть описатели процесса и потока,
    // возвращаемые CreateProcess.
    VERIFY ( CloseHandle ( stProcessInfo.hProcess ) ) ;
    VERIFY ( CloseHandle ( stProcessInfo.hThread ) ) ;

    // Посмотрим, запустился ли процесс отлаживаемой программы.
    if ( FALSE == bRet )
    {
        _tprintf ( _T ( "Unable to start %s\n" ) , szCmdLine ) ;
        return ;
    }

    // Сохранить идентификатор процесса на случай
    // необходимости отсоединения.
    dwPID = stProcessInfo.dwProcessId ;
}
else
{

```

```

bRet = DebugActiveProcess ( dwPID );
if ( FALSE == bRet )
{
    _tprintf ( _T ( "Unable to attach to %u\n" ), dwPID );
    return ;
}
}

// Отлаживаемая программ запущена, поэтому запускаем цикл отладчика.
DEBUG_EVENT stDE
;
BOOL      bSeenInitialBP  = FALSE ;
BOOL      bContinue       = TRUE  ;
HANDLE     hProcess        = INVALID_HANDLE_VALUE ;
DWORD      dwContinueStatus ;

// Цикл до тех пор, пока не потребуется остановиться.
while ( TRUE == bContinue )
{
    // Пауза до возникновения события отладки.
    BOOL bProcessDbgEvent = WaitForDebugEvent ( &stDE , 100 );

    if ( TRUE == bProcessDbgEvent )
    {
        // Обработка конкретных событий отладки.
        // Так как MinDBG – это только минимальный отладчик,
        // он обрабатывает только несколько событий.
        switch ( stDE.dwDebugEventCode )
        {
            case CREATE_PROCESS_DEBUG_EVENT :
            {
                DisplayCreateProcessEvent(stDE.u.CreateProcessInfo);
                // Сохраним описатель, который понадобится позже.
                // Заметьте: вы не можете закрыть этот описатель.
                // Если вы это сделаете, CloseHandle завершится с ошибкой.
                hProcess = stDE.u.CreateProcessInfo.hProcess ;

                // Описатель файла можно закрыть безболезненно.
                // Если вы закроете поток, CloseHandle провалится
                // глубоко в ContinueDebugEvent, когда вы будете
                // завершать приложение.
                VERIFY(CloseHandle(stDE.u.CreateProcessInfo.hFile));

                dwContinueStatus = DBG_CONTINUE ;
            }
            break ;
            case EXIT_PROCESS_DEBUG_EVENT :
            {
                DisplayExitProcessEvent ( stDE.u.ExitProcess );
                bContinue = FALSE ;
                dwContinueStatus = DBG_CONTINUE ;
            }
        }
    }
}

```

```
    }
    break ;

    case LOAD_DLL_DEBUG_EVENT :
    {
        DisplayDllLoadEvent ( hProcess , stDE.u.LoadDll ) ;

        // Не забудьте закрыть описатель соответствующего файла.
        VERIFY ( CloseHandle( stDE.u.LoadDll.hFile ) ) ;

        dwContinueStatus = DBG_CONTINUE ;
    }
    break ;
    case UNLOAD_DLL_DEBUG_EVENT :
    {
        DisplayDllUnLoadEvent ( stDE.u.UnloadDll ) ;
        dwContinueStatus = DBG_CONTINUE ;
    }
    break ;

    case CREATE_THREAD_DEBUG_EVENT :
    {
        DisplayCreateThreadEvent ( stDE.dwThreadId ,
                                   stDE.u.CreateThread ) ;
        // Заметьте, что вы не можете закрыть описатель потока.
        // Если вы это сделаете, CloseHandle провалится глубоко
        // в ContinueDebugEvent.

        dwContinueStatus = DBG_CONTINUE ;
    }
    break ;
    case EXIT_THREAD_DEBUG_EVENT :
    {
        DisplayExitThreadEvent ( stDE.dwThreadId ,
                                 stDE.u.ExitThread ) ;
        dwContinueStatus = DBG_CONTINUE ;
    }
    break ;

    case OUTPUT_DEBUG_STRING_EVENT :
    {
        DisplayODSEvent ( hProcess , stDE.u.DebugString ) ;
        dwContinueStatus = DBG_CONTINUE ;
    }
    break ;

    case EXCEPTION_DEBUG_EVENT :
    {
        DisplayExceptionEvent ( stDE.u.Exception ) ;
```



```
// Единственное исключение, требующее специальной
// обработки, – это точка прерывания загрузчика.
switch(stDE.u.Exception.ExceptionRecord.ExceptionCode)
{
    case EXCEPTION_BREAKPOINT :
    {
        // Если возникает исключение по точке прерывания
        // и оно первое, я продолжаю свое веселье, иначе
        // я передаю исключение отлаживаемой программе.
        if ( FALSE == bSeenInitialBP )
        {
            bSeenInitialBP = TRUE ;
            dwContinueStatus = DBG_CONTINUE ;
        }
        else
        {
            // Хьюстон, у нас проблема!
            dwContinueStatus =
                DBG_EXCEPTION_NOT_HANDLED ;
        }
    }
    break ;

    // Все остальные исключения передаем
    // отлаживаемой программе.
    default :
    {
        dwContinueStatus =
            DBG_EXCEPTION_NOT_HANDLED ;
    }
    break ;
}

// Для всех остальных событий – просто продолжаем.
default :
{
    dwContinueStatus = DBG_CONTINUE ;
}
break ;
}

// Передаем управление ОС.
#ifdef _DEBUG
    BOOL bCntDbg =
#endif
    ContinueDebugEvent ( stDE.dwProcessId ,
                        stDE.dwThreadId ,
                        dwContinueStatus ) ;
```

см. след. стр.

```

        ASSERT ( TRUE == bCntDbg );
    }
    // Необходимо ли отсоединение?
    if ( TRUE == g_bDoTheDetach )
    {
        // Отсоединение работает только в XP или более поздней версии,
        // поэтому я должен выполнить GetProcAddress, чтобы найти
        // DebugActiveProcessStop.
        bContinue = FALSE ;

        HINSTANCE hKernel32 =
            GetModuleHandle ( _T ( "KERNEL32.DLL" ) );
        if ( 0 != hKernel32 )
        {
            PFNDEBUGACTIVEPROCESSSTOP pfnDAPS =
                (PFNDEBUGACTIVEPROCESSSTOP)
                GetProcAddress ( hKernel32
                                , "DebugActiveProcessStop" );

            if ( NULL != pfnDAPS )
            {
#ifdef _DEBUG
                BOOL bTemp =
                    pfnDAPS ( dwPID );

                ASSERT ( TRUE == bTemp );
            }
        }
    }
}

/*//////////////////////////////////////
// Мониторы обработки Ctrl+Break
//////////////////////////////////////*/
BOOL WINAPI CtrlBreakHandler ( DWORD dwCtrlType )
{
    // Я буду обрабатывать только Ctrl+Break.
    // Все другое убивает отлаживаемую программу.
    if ( CTRL_BREAK_EVENT == dwCtrlType )
    {
        g_bDoTheDetach = TRUE ;
        return ( TRUE );
    }
    return ( FALSE );
}

/*//////////////////////////////////////
// Отображает справку к программе.
//////////////////////////////////////*/

```

```

void ShowHelp ( void )
{
    _tprintf ( _T ( "Start a program to debug:\n" )
               _T ( "    MinDBG <program to debug> " )
               _T ( "<program's command-line options>\n" )
               _T ( "Attach to an existing program:\n" )
               _T ( "    MinDBG -PID\n" )
               _T ( "        PID is the decimal process ID\n" ) );
}

/*//////////////////////////////////////////////////////////////////
// Отображение события создания процесса.
//////////////////////////////////////////////////////////////////*/
void DisplayCreateProcessEvent ( CREATE_PROCESS_DEBUG_INFO & stCPDI )
{
    _tprintf ( _T ( "Create Process Event      :\n" ) );
    _tprintf ( _T ( "    hFile                : 0x%08X\n" ),
               stCPDI.hFile
               );
    _tprintf ( _T ( "    hProcess            : 0x%08X\n" ),
               stCPDI.hProcess
               );
    _tprintf ( _T ( "    hThread             : 0x%08X\n" ),
               stCPDI.hThread
               );
    _tprintf ( _T ( "    lpBaseOfImage       : 0x%08X\n" ),
               stCPDI.lpBaseOfImage
               );
    _tprintf ( _T ( "    dwDebugInfoFileOffset : 0x%08X\n" ),
               stCPDI.dwDebugInfoFileOffset
               );
    _tprintf ( _T ( "    nDebugInfoSize      : 0x%08X\n" ),
               stCPDI.nDebugInfoSize
               );
    _tprintf ( _T ( "    lpThreadLocalBase   : 0x%08X\n" ),
               stCPDI.lpThreadLocalBase
               );
    _tprintf ( _T ( "    lpStartAddress      : 0x%08X\n" ),
               stCPDI.lpStartAddress
               );
    _tprintf ( _T ( "    lpImageName         : 0x%08X\n" ),
               stCPDI.lpImageName
               );
    _tprintf ( _T ( "    fUnicode            : 0x%08X\n" ),
               stCPDI.fUnicode
               );
}

/*//////////////////////////////////////////////////////////////////
// Отображение событий создания потока.
//////////////////////////////////////////////////////////////////*/
void DisplayCreateThreadEvent ( DWORD dwTID ,
                              CREATE_THREAD_DEBUG_INFO & stCTDI )
{
    _tprintf ( _T ( "Create Thread Event      :\n" ) );
    _tprintf ( _T ( "    TID                 : 0x%08X\n" ),
               dwTID
               );
    _tprintf ( _T ( "    hThread             : 0x%08X\n" ),
               stCTDI.hThread
               );
}

```

см. след. стр.

```

    _tprintf ( _T ( "    lpThreadLocalBase      : 0x%08X\n" ),
               stCTDI.lpThreadLocalBase
            );
    _tprintf ( _T ( "    lpStartAddress        : 0x%08X\n" ),
               stCTDI.lpStartAddress
            );
}

/*//////////////////////////////////////////////////////////////////
// Отображение событий завершения потока.
//////////////////////////////////////////////////////////////////*/
void DisplayExitThreadEvent ( DWORD          dwTID ,
                             EXIT_THREAD_DEBUG_INFO & stETDI )
{
    _tprintf ( _T ( "Exit Thread Event          : \n" ) );
    _tprintf ( _T ( "    TID                        : 0x%08X\n" ),
               dwTID
            );
    _tprintf ( _T ( "    dwExitCode                 : 0x%08X\n" ),
               stETDI.dwExitCode
            );
}

/*//////////////////////////////////////////////////////////////////
// Отображение событий завершения процесса.
//////////////////////////////////////////////////////////////////*/
void DisplayExitProcessEvent ( EXIT_PROCESS_DEBUG_INFO & stEPDI )
{
    _tprintf ( _T ( "Exit Process Event          : \n" ) );
    _tprintf ( _T ( "    dwExitCode                 : 0x%08X\n" ),
               stEPDI.dwExitCode
            );
}

/*//////////////////////////////////////////////////////////////////
// Отображение событий загрузки DLL.
//////////////////////////////////////////////////////////////////*/
void DisplayDllLoadEvent ( HANDLE          hProcess ,
                          LOAD_DLL_DEBUG_INFO & stLDDI )
{
    _tprintf ( _T ( "DLL Load Event              : \n" ) );
    _tprintf ( _T ( "    hFile                     : 0x%08X\n" ),
               stLDDI.hFile
            );
    _tprintf ( _T ( "    lpBaseOfDll               : 0x%08X\n" ),
               stLDDI.lpBaseOfDll
            );
    _tprintf ( _T ( "    dwDebugInfoFileOffset     : 0x%08X\n" ),
               stLDDI.dwDebugInfoFileOffset
            );
    _tprintf ( _T ( "    nDebugInfoSize            : 0x%08X\n" ),
               stLDDI.nDebugInfoSize
            );
    _tprintf ( _T ( "    lpImageName                : 0x%08X\n" ),
               stLDDI.lpImageName
            );
    _tprintf ( _T ( "    fUnicode                   : 0x%08X\n" ),
               stLDDI.fUnicode
            );

    static bool bSeenNTDLL = false ;

```

```

TCHAR szDLLName[ MAX_PATH ] ;

// NTDLL.DLL - это специальный случай. В W2K lpImageName равен NULL,
// а в XP он указывает просто на 'ntdll.dll', поэтому я сфабрикую
// загрузочную информацию.
if ( false == bSeenNTDLL )
{
    bSeenNTDLL = true ;
    UINT uiLen = GetWindowsDirectory ( szDLLName , MAX_PATH ) ;
    ASSERT ( uiLen > 0 ) ;
    if ( uiLen > 0 )
    {
        _tcscpy ( szDLLName + uiLen , _T ( "\\NTDLL.DLL" ) ) ;
    }
    else
    {
        _tcscpy ( szDLLName , _T ( "GetWindowsDirectory FAILED!" ) );
    }
}
else
{
    szDLLName[ 0 ] = _T ( '\\0' ) ;

    // Значение в lpImageName является указателем на полный путь
    // загружаемой DLL. Этот адрес находится в адресном пространстве
    // отлаживаемой программы.
    LPCVOID lpPtr = 0 ;
    DWORD dwBytesRead = 0 ;
    BOOL bRet = FALSE ;

    bRet = ReadProcessMemory ( hProcess
                               ,
                               stLDDI.lpImageName ,
                               &lpPtr
                               ,
                               sizeof ( LPCVOID ) ,
                               &dwBytesRead
                               ) ;

    if ( TRUE == bRet )
    {
        // Если имя в отлаживаемой программе задано в UNICODE,
        // я могу копировать его прямо в szDLLName,
        // так как здесь все в UNICODE.
        if ( TRUE == stLDDI.fUnicode )
        {
            // Иногда невозможно сразу считать весь буфер,
            // содержащий имя, поэтому необходимо делать это
            // частями, пока оно не будет считано целиком.
            DWORD dwSize = MAX_PATH * sizeof ( TCHAR ) ;
            do
            {
                bRet = ReadProcessMemory ( hProcess

```

см. след. стр.

```

        lpPtr          ,
        szDLLName      ,
        dwSize         ,
        &dwBytesRead   ) ;

        dwSize = dwSize - 20 ;
    }
    while ( ( FALSE == bRet ) && ( dwSize > 20 ) ) ;
}
else
{
    // Считывание строки ANSI и преобразование ее в UNICODE.
    char szAnsiName[ MAX_PATH ] ;
    DWORD dwAnsiSize = MAX_PATH ;

    do
    {
        bRet = ReadProcessMemory ( hProcess          ,
                                   lpPtr              ,
                                   szAnsiName         ,
                                   dwAnsiSize         ,
                                   &dwBytesRead       ) ;

        dwAnsiSize = dwAnsiSize - 20 ;
    } while ( ( FALSE == bRet ) && ( dwAnsiSize > 20 ) ) ;
    if ( TRUE == bRet )
    {
        MultiByteToWideChar ( CP_THREAD_ACP      ,
                               0                  ,
                               szAnsiName         ,
                               -1                 ,
                               szDLLName          ,
                               MAX_PATH           ) ;
    }
}
}

if ( _T ( '\\0' ) == szDLLName[ 0 ] )
{
    // С этой DLL связано несколько проблем. Попробуйте считать ее
    // с помощью GetModuleHandleEx. Хотя вы и можете думать, что это
    // будет работать, это только кажется, если не может быть получена
    // информация о модуле этим способом. Если невозможно получить имя DLL
    // вышеприведенной функцией, значит, вы в действительности имеете дело
    // с перемещенной DLL.
    DWORD dwRet = GetModuleFileNameEx ( hProcess          ,
                                        (HMODULE)stLDDI.   ,
                                        lpBaseOfDll        ,
                                        szDLLName          ,
                                        MAX_PATH            ) ;

    ASSERT ( dwRet > 0 ) ;
    if ( 0 == dwRet )

```

```

    {
        szDLLName[ 0 ] = _T ( '\\0' );
    }
}

if ( _T ( '\\0' ) != szDLLName[ 0 ] )
{
    _tcsupr ( szDLLName );
    _tprintf ( _T ( "    DLL name                : %s\\n" ),
               szDLLName
               );
}
else
{
    _tprintf ( _T ( "UNABLE TO READ DLL NAME!!\\n" ) );
}
}

/*/////////////////////////////////////////////////////////////////
// Отображение событий выгрузки DLL.
////////////////////////////////////////////////////////////////*/
void DisplayDllUnLoadEvent ( UNLOAD_DLL_DEBUG_INFO & stULDDI )
{
    _tprintf ( _T ( "DLL Unload Event          :\\n" ) );
    _tprintf ( _T ( "    lpBaseOfDll                : 0x%08X\\n" ),
               stULDDI.lpBaseOfDll
               );
}

/*/////////////////////////////////////////////////////////////////
// Отображение событий OutputDebugString.
////////////////////////////////////////////////////////////////*/
void DisplayODSEvent ( HANDLE                hProcess ,
                     OUTPUT_DEBUG_STRING_INFO & stODSI )
{
    _tprintf ( _T ( "OutputDebugString Event :\\n" ) );
    _tprintf ( _T ( "    lpDebugStringData      : 0x%08X\\n" ),
               stODSI.lpDebugStringData
               );
    _tprintf ( _T ( "    fUnicode               : 0x%08X\\n" ),
               stODSI.fUnicode
               );
    _tprintf ( _T ( "    nDebugStringLength    : %d\\n" ),
               stODSI.nDebugStringLength
               );
    _tprintf ( _T ( "    String                 : " ) );

    TCHAR szFinalBuff[ 512 ];
    if ( stODSI.nDebugStringLength > 512 )
    {
        _tprintf ( _T ( "String to large!!\\n" ) );
        return ;
    }

    DWORD dwRead ;

```

```

BOOL bRet ;

// Интересно, что вызовы OutputDebugString независимо
// от того, является ли приложение полностью UNICODE-овым,
// всегда работают со строками ANSI.
if ( false == stODSI.fUnicode )
{
    // Читаем ANSI-строку.
    char szAnsiBuff[ 512 ] ;
    bRet = ReadProcessMemory ( hProcess
                               ,
                               stODSI.lpDebugStringData ,
                               szAnsiBuff
                               ,
                               stODSI.nDebugStringLength ,
                               &dwRead
                               ) ;

    if ( TRUE == bRet )
    {
        MultiByteToWideChar ( CP_THREAD_ACP ,
                               0
                               ,
                               szAnsiBuff
                               ,
                               -1
                               ,
                               szFinalBuff
                               ,
                               512
                               ) ;
    }
    else
    {
        szFinalBuff[ 0 ] = _T ( '\\0' ) ;
    }
}
else
{
    // Читаем UNICODE-строку.
    bRet = ReadProcessMemory ( hProcess
                               ,
                               stODSI.lpDebugStringData
                               ,
                               szFinalBuff
                               ,
                               stODSI.nDebugStringLength *
                               sizeof ( TCHAR )
                               ,
                               &dwRead
                               ) ;

    if ( FALSE == bRet )
    {
        szFinalBuff[ 0 ] = _T ( '\\0' ) ;
    }
}

if ( _T ( '\\0' ) != szFinalBuff[ 0 ] )
{
    _tprintf ( _T ( "%s\n" ) , szFinalBuff ) ;
}
else
{
    _tprintf ( _T ( "UNABLE TO READ ODS STRING!!\n" ) ) ;
}

```



```

    }
}

/*//////////////////////////////////////
// Отображение событий исключений.
////////////////////////////////////*/
void DisplayExceptionEvent ( EXCEPTION_DEBUG_INFO & stEDI )
{
    _tprintf ( _T ( "Exception Event           :\n" ) );
    _tprintf ( _T ( "    dwFirstChance       : 0x%08X\n" ),
               stEDI.dwFirstChance           );
    _tprintf ( _T ( "    ExceptionCode        : 0x%08X\n" ),
               stEDI.ExceptionRecord.ExceptionCode );
    _tprintf ( _T ( "    ExceptionFlags       : 0x%08X\n" ),
               stEDI.ExceptionRecord.ExceptionFlags );
    _tprintf ( _T ( "    ExceptionRecord      : 0x%08X\n" ),
               stEDI.ExceptionRecord.ExceptionRecord );
    _tprintf ( _T ( "    ExceptionAddress     : 0x%08X\n" ),
               stEDI.ExceptionRecord.ExceptionAddress );
    _tprintf ( _T ( "    NumberParameters    : 0x%08X\n" ),
               stEDI.ExceptionRecord.NumberParameters );
}

```

WDBG — настоящий отладчик

Думаю, лучший способ разобраться в работе отладчика — написать его, что я и сделал. Хотя WDBG вряд ли в ближайшее время заменит отладчики Visual Studio .NET и WinDBG, он определенно делает почти все, что должен делать отладчик. WDBG имеется среди файлов, записанных на CD, прилагаемом к книге. На рис. 4-3 вы увидите отладку программы CrashFinder из главы 12 в WDBG. На рисунке CrashFinder застопорен на третьем экземпляре точки прерывания, которую я установил на функции GetProcAddress библиотеки KERNEL32.DLL. Окно Memory в верхнем правом углу отображает второй параметр, строку InitializeCriticalSectionAndSpinCount, передаваемую CrashFinder'ом конкретному экземпляру GetProcAddress. На рис. 4-3 WDBG делает именно все, что вы ожидаете от отладчика, включая отображение регистров, дизассемблированного кода и загруженных в настоящее время модулей и исполняющихся потоков. Выдающимся является окно Call Stack (стек вызовов), показанное в середине правой части рис. 4-3. WDBG не только отображает стек вызовов так, как вы ожидали, но и в полной мере поддерживает отображение локальных переменных и развертывание структур. Что вы не видите на этом рисунке и что должно там быть при первом запуске WDBG, это то, что WDBG поддерживает также точки прерывания, перечисление символов и их отображение в окне Symbols (символы), а также прерывание исполнения приложения в отладчике.

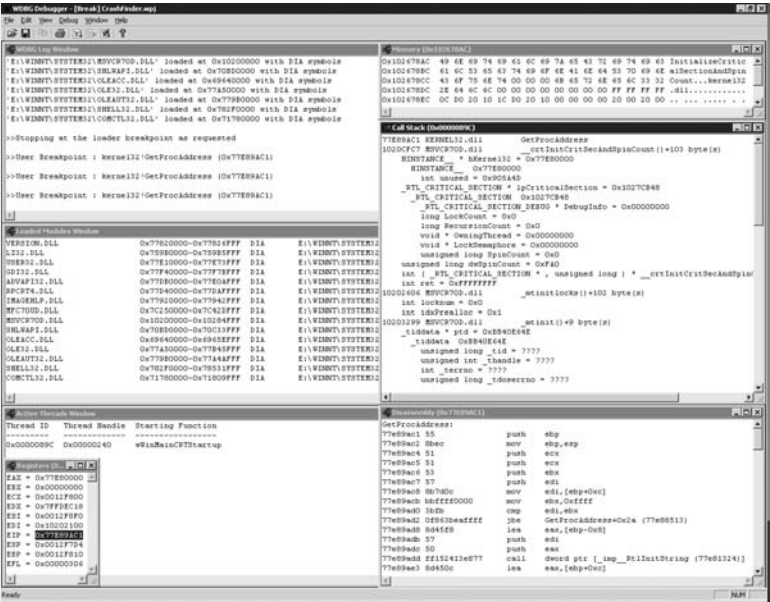


Рис. 4-3. WDBG в действии

В целом WDBG меня радует, так как он является отличным примером, и я горжусь, что WDBG демонстрирует все внутренние приемы, обычно используемые в отладчиках. Однако, глядя на UI, можно заметить, что я не тратил много времени на отдельные его части. На самом деле все окна многооконного интерфейса являются полями ввода. Я сделал это намеренно — оставил пользовательский интерфейс простым, потому что не хотел, чтобы детали интерфейса отвлекали вас от существенно важной части кода отладчика. Я написал пользовательский интерфейс WDBG с применением библиотеки классов Microsoft Foundation Class (MFC), поэтому, если вы знаете ее, для вас не составит большого труда спроектировать более нарядный UI.

Прежде чем перейти к специфическим вопросам отладки, посмотрим поближе на WDBG. В табл. 4-2 перечислены основные подсистемы WDBG. Одной из моих целей при создании WDBG было определить промежуточный интерфейс между UI и циклом отладки. Имея промежуточный интерфейс, если понадобится сделать поддержку удаленной отладки в WDBG в сети, нужно будет просто заменить локальные DLL.

Табл. 4-2. Основные подсистемы WDBG

Подсистема	Описание
WDBG.EXE	Этот модуль содержит весь код UI. Кроме того, здесь производится вся обработка точек прерывания. Основная часть работы отладчика производится в WDBGPROJDOC.CPP.

Табл. 4-2. Основные подсистемы WDBG (продолжение)

Подсистема	Описание
LOCALDEBUG.DLL	Этот модуль содержит цикл отладки. Так как я хотел использовать этот цикл отладки и в других проектах, пользовательский код (WDBG.EXE в данном случае) применяет в цикле отладки класс C++, порожденный от класса CdebugBaseUser (определяемого в DEBUGINTERFACE.H). Цикл отладки будет обращаться к этому классу при возникновении каких-либо событий отладки. Пользовательский класс отвечает за синхронизацию. WDBGUSER.H и WDBGUSER.CPP содержат координирующий класс WDBG.EXE. WDBG.EXE использует простой тип синхронизации с помощью вызовов <code>SendMessage</code> . Иначе говоря, поток отладки посылает сообщение потоку UI и останавливается, пока поток UI не вернет управление. Если событие отладки требует ввода со стороны пользователя, отладочный поток останавливается после отправки сообщения о событии синхронизации. При обработке потоком UI команды Go, он вызывает событие синхронизации, и отладочный поток возобновляет работу.
LOCALASSIST.DLL	Этот модуль — просто оболочка для функций API, манипулирующих памятью отлаживаемой программы и ее регистрами. Благодаря интерфейсу, определяемому в этом модуле, WDBG.EXE и I386CPUHELP.DLL могут управлять также и удаленной отладкой после замены этого модуля.
I386CPUHELP.DLL	Хотя этот вспомогательный модуль для процессоров IA32 (Pentium) специфичен для процессоров Pentium, его интерфейс, определяемый в CPUHELP.H, не зависит от типа процессора. Если вы захотите перенести WDBG на другой процессор, понадобится заменить только этот модуль. Код дизассемблера в этом модуле восходит к примеру кода программы Dr. Watson, поставляемому с Platform SDK. Хотя дизассемблер работает, он требует обновления для поддержки последних версий процессоров Pentium.

Чтение памяти и запись в нее

Чтение из памяти отлаживаемой программы производится очень просто. Это делает `ReadProcessMemory`. Отладчик имеет полный доступ к отлаживаемой программе, если он запустил ее, так как описатель процесса, возвращаемый событием отладки `CREATE_PROCESS_DEBUG_EVENT` имеет права доступа `PROCESS_VM_READ` и `PROCESS_VM_WRITE`. Если ваш отладчик присоединяется к процессу посредством `DebugActiveProcess`, вы должны иметь к процессу, к которому вы присоединяетесь, права `SeDebugPrivileges` для чтения и записи.

Прежде чем я смогу рассказать о записи в память отлаживаемой программы, надо кратко объяснить важную концепцию «копирования при записи» (copy-on-write). Загружая исполняемый файл, Windows разрешает для совместного использования различными процессами столько отображаемых страниц памяти, сколько возможно. Если один из этих процессов выполняется под отладчиком и одна из этих страниц содержит точку прерывания, то очевидно, что она может отсутствовать на некоторых используемых совместно страницах. Как только какой-то

из процессов, исполняющихся вне отладчика, начинает исполнять такой код, возникает аварийное завершение из-за исключения по точке прерывания. Чтобы обойти эту ситуацию, ОС наблюдает, что страница изменена для некоторого процесса, и делает ее копию для этого процесса, который установил на ней точку прерывания. Так что, как только процесс начинает запись в некоторую страницу, ОС копирует ее.

Запись в память отлаживаемой программы почти столь же проста, как и чтение. Однако, так как страницы памяти, в которые вы хотите производить запись, могут быть помечены как «только для чтения», вам сначала следует вызвать `VirtualQueryEx` для получения кода защиты текущей страницы. Зная состояние защиты, можно использовать функцию `API VirtualProtectEx`, чтобы установить состояние `PAGE_EXECUTE_READWRITE` для записи в нее, а Windows подготовилась бы выполнять «копирование при записи». Произведя запись, надо восстановить первоначальное состояние защиты страницы. Если этого не сделать, отлаживаемая программа может случайно произвести успешную запись на этой странице, вместо того чтобы завершиться аварийно. Если исходное состояние защиты было «только для чтения», случайная запись в отлаживаемой программе будет приводить к нарушению доступа. Если не восстановить состояние защиты, при случайной записи исключение вырабатываться не будет, и вы попадете в ситуацию, при которой работа программы под отладчиком будет отличаться от работы программы вне его.

Есть одна интересная деталь работы отладочного API Win32: отладчик отвечает за получение строк для вывода при возникновении события `OUTPUT_DEBUG_STRING_EVENT`. Информация, передаваемая отладчику, включает расположение и длину строки. Когда он получает это сообщение, отладчик читает память отлаживаемой программы. Так как вызовы `OutputDebugString` проходят через отладочный API Win32, задерживающий все потоки каждый раз при появлении события отладки, сообщения трассировки могут легко изменить поведение вашего приложения под отладчиком. Если многопоточность запрограммирована корректно, можете вызывать `OutputDebugString` как угодно без воздействия на ваше приложение. Однако, если у вас есть ошибки в реализации многопоточности, вы можете случайно получить взаимную блокировку потоков из-за незаметных изменений соотношений времен в связи с вызовами `OutputDebugString`.

Листинг 4-2 демонстрирует, как WDBG управляет событием `OUTPUT_DEBUG_STRING_EVENT`. Заметьте: функция `DBG_ReadProcessMemory` является оболочкой функции `ReadProcessMemory` из `LOCALASSIST.DLL`. Хотя отладочный API Win32 предполагает, что вы можете принимать как строки UNICODE, так и строки ANSI в процессе обработки события `OUTPUT_DEBUG_STRING_EVENT`, начиная с Windows XP/Server 2003, она передает только строки ANSI, даже если вызов приходит от `OutputDebugStringW`.

Листинг 4-2. `OutputDebugStringEvent` из `PROCESSDEBUGEVENTS.CPP`

```
static
DWORD OutputDebugStringEvent ( CDebugBaseUser *      pUserClass ,
                              LPDEBUGGEEINFO        pData      ,
                              DWORD                  dwProcessId ,
                              DWORD                  dwThreadId  ,
                              OUTPUT_DEBUG_STRING_INFO & stODSI   )
```

```

{
    // OutputDebugString может выводить огромное количество символов,
    // поэтому я буду выделять память каждый раз.
    DWORD dwTotalBuffSize = stODSI.nDebugStringLength ;

    if ( TRUE == stODSI.fUnicode )
    {
        dwTotalBuffSize *= 2 ;
    }

    PBYTE pODSData = new BYTE [ dwTotalBuffSize ] ;

    DWORD dwRead ;
    // Читать память.
    BOOL bRet = DBG_ReadProcessMemory( pData->GetProcessHandle ( ) ,
                                        stODSI.lpDebugStringData ,
                                        pODSData ,
                                        dwTotalBuffSize ,
                                        &dwRead ) ;

    ASSERT ( TRUE == bRet ) ;
    if ( TRUE == bRet )
    {
        TCHAR * szUnicode = NULL ;
        TCHAR * szSelected = NULL ;
        if ( TRUE == stODSI.fUnicode )
        {
            szSelected = (TCHAR*)pODSData ;
        }
        else
        {
            szUnicode = new TCHAR [ stODSI.nDebugStringLength ] ;
            BSUAnsi2Wide ( (const char*)pODSData ,
                          szUnicode ,
                          stODSI.nDebugStringLength ) ;
            int iLen = (int)strlen ( (const char*)pODSData ) ;
            iLen = MultiByteToWideChar ( CP_THREAD_ACP ,
                                        0 ,
                                        (LPCSTR)pODSData ,
                                        iLen ,
                                        szUnicode ,
                                        stODSI.nDebugStringLength ) ;

            szSelected = szUnicode ;
        }

        LPCTSTR szTemp =
            pUserClass->ConvertCRLF ( szSelected ,
                                      stODSI.nDebugStringLength ) ;

        if ( NULL != szUnicode )

```

```

    {
        delete [] szUnicode ;
    }

    // Послать преобразованную строку пользовательскому классу.
    pUserClass->OutputDebugStringEvent ( dwProcessId ,
                                         dwThreadId  ,
                                         szTemp       ) ;

    delete [] szTemp ;
}
delete [] pODSData ;
return ( DBG_CONTINUE ) ;
}

```

Точки прерывания и одиночные шаги

Многие программисты не понимают, что отладчики негласно активно пользуются точками прерывания для управления отлаживаемой программой. Хотя вы можете и не устанавливать явно некоторые точки прерывания, отладчик сам устанавливает их для выполнения таких функций, как перемещение по шагам через вызовы функций. Отладчик также использует точки прерывания, когда вы выбираете исполнение программы до какой-то строки исходного кода с остановкой на ней. Наконец, отладчик пользуется точками прерывания для прерывания отлаживаемой программы по команде (например, через выбор меню Debug Break в WDBG).

Концепция установки точек прерывания проста. Все, что вам надо сделать, — это иметь адрес памяти, где вы хотите установить точку прерывания, сохранить код операции (ее значение) в этой точке и записать по этому адресу код команды отладочного прерывания. Для семейства процессоров Intel Pentium команда отладочного прерывания имеет мнемонику INT 3 или код операции 0xCC, поэтому вам нужно сохранить только один байт, расположенный по адресу, где вы устанавливаете точку прерывания. Другие процессоры, такие как Intel Itanium, имеют другой размер кода операции, поэтому вам придется сохранять больший объем данных, находящихся по этому адресу.

В листинге 4-3 показан код функции SetBreakpoint. В процессе чтения этого кода имейте в виду, что функции DBG_* определены в LOCALASSIST.DLL и помогают изолировать процедуры манипуляций процессами, помогая упростить добавление удаленной отладки к WDBG. Функция SetBreakpoint иллюстрирует обработку (описанную выше), необходимую для изменения защиты памяти при записи в нее.

Листинг 4-3. Функция SetBreakpoint из I386CPUHELP.C

```

int CPUHELP_DLLINTERFACE __stdcall
SetBreakpoint ( PDEBUGPACKET dp
               , LPCVOID      ulAddr
               , OPCODE *      pOpCode )
{
    DWORD dwReadWrite = 0 ;
    BYTE bTempOp = BREAK_OPCODE ;

```

```

BOOL bReadMem ;
BOOL bWriteMem ;
BOOL bFlush ;
MEMORY_BASIC_INFORMATION mbi ;
DWORD dwOldProtect ;

ASSERT ( FALSE == IsBadReadPtr ( dp , sizeof ( DEBUGPACKET ) ) ) ;
ASSERT ( FALSE == IsBadWritePtr ( pOpCode , sizeof ( OP_CODE ) ) ) ;
if ( ( TRUE == IsBadReadPtr ( dp , sizeof ( DEBUGPACKET ) ) ) ||
    ( TRUE == IsBadWritePtr ( pOpCode , sizeof ( OP_CODE ) ) ) )
{
    TRACE0 ( "SetBreakpoint : invalid parameters\n!" ) ;
    return ( FALSE ) ;
}

// Читать код операции по заданному адресу.
bReadMem = DBG_ReadProcessMemory ( dp->hProcess ,
                                   (LPCVOID)u1Addr ,
                                   &bTempOp ,
                                   sizeof ( BYTE ) ,
                                   &dwReadWrite ) ;

ASSERT ( FALSE != bReadMem ) ;
ASSERT ( sizeof ( BYTE ) == dwReadWrite ) ;
if ( ( FALSE == bReadMem ) ||
    ( sizeof ( BYTE ) != dwReadWrite ) )
{
    return ( FALSE ) ;
}

// Не пытаемся ли мы заменить уже имеющийся код команды прерывания?
if ( BREAK_OPCODE == bTempOp )
{
    return ( -1 ) ;
}

// Получаем атрибуты страницы отлаживаемой программы.
DBG_VirtualQueryEx ( dp->hProcess ,
                    (LPCVOID)u1Addr ,
                    &mbi ,
                    sizeof ( MEMORY_BASIC_INFORMATION ) ) ;

// Заставляем выполнять копирование при записи в отлаживаемой программе.
if ( FALSE == DBG_VirtualProtectEx ( dp->hProcess ,
                                     mbi.BaseAddress ,
                                     mbi.RegionSize ,
                                     PAGE_EXECUTE_READWRITE ,
                                     &mbi.Protect ) )
{
    ASSERT ( !"VirtualProtectEx failed!!" ) ;
}

```

см. след. стр.

```
        return ( FALSE );
    }
    // Сохраняем код операции, которую я собираюсь заменить.
    *pOpCode = (void*)bTempOp ;

    bTempOp = BREAK_OPCODE ;
    dwReadWrite = 0 ;
    // Код операции сохранен, устанавливаем теперь точку прерывания.
    bWriteMem = DBG_WriteProcessMemory ( dp->hProcess      ,
                                          (LPVOID)ulAddr    ,
                                          (LPVOID)&bTempOp   ,
                                          sizeof ( BYTE )   ,
                                          &dwReadWrite      ) ;

    ASSERT ( FALSE != bWriteMem ) ;
    ASSERT ( sizeof ( BYTE ) == dwReadWrite ) ;

    if ( ( FALSE == bWriteMem                ) ||
        ( sizeof ( BYTE ) != dwReadWrite ) )
    {
        return ( FALSE ) ;
    }

    // Восстанавливаем защиту, которая была до моего вмешательства.
    VERIFY ( DBG_VirtualProtectEx ( dp->hProcess      ,
                                    mbi.BaseAddress   ,
                                    mbi.RegionSize    ,
                                    mbi.Protect        ,
                                    &dwOldProtect     ) ) ;

    // Сбрасываем кэш операций, если эта память находится
    // в кэше центрального процессора.
    bFlush = DBG_FlushInstructionCache ( dp->hProcess      ,
                                          (LPCVOID)ulAddr  ,
                                          sizeof ( BYTE ) ) ;

    ASSERT ( TRUE == bFlush ) ;

    return ( TRUE ) ;
}
```

После установки команды прерывания процессор исполнит ее и сообщит отладчику, что произошло исключение `EXCEPTION_BREAKPOINT (0x80000003)`, — то, что надо. Если это обычная точка прерывания, отладчик найдет ее и отобразит ее размещение пользователю. Когда пользователь решит продолжать исполнение, отладчик должен проделать некоторую работу по восстановлению состояния программы. Так как точка прерывания перезаписала часть памяти, то, если вы, как разработчик отладчика, просто разрешите продолжать исполнение процесса, будет исполнен не тот код, и отлаживаемая программа, возможно, завершится аварийно. Вам нужно вернуть указатель команд обратно на адрес точки прерывания и заменить команду прерывания кодом операции, который вы сохранили при уста-

новке точки прерывания. После восстановления кода операции можно продолжить исполнение.

Есть только одна маленькая проблема: как сбросить точку прерывания, чтобы остановиться в этом месте в следующий раз? Если процессор, на котором вы работаете, поддерживает пошаговое исполнение, это сделать легко. При пошаговом исполнении процессор выполняет одну команду и вырабатывает другой тип исключения — `EXCEPTION_SINGLE_STEP` (0x80000004). К счастью, все процессоры, на которых работает Win32, поддерживают пошаговое исполнение. Для семейства Intel Pentium установка пошагового исполнения требует установки бита 8 регистра флагов. Руководство по процессорам Intel называет этот бит TF или флагом ловушки (Trap Flag). Следующий код демонстрирует функцию `SetSingleStep` и что нужно сделать для установки флага TF. После восстановления исходного кода операции на месте точки прерывания отладчик помечает свое внутреннее состояние ожидающим появления исключения пошагового исполнения, переводит процессор в режим пошагового исполнения и продолжает процесс.

```
// SetSingleStep из i386CPUHelp.C
BOOL CPUHELP_DLLINTERFACE __stdcall
SetSingleStep ( PDEBUGPACKET dp )
{
    BOOL bSetContext ;
    ASSERT ( FALSE == IsBadReadPtr ( dp , sizeof ( DEBUGPACKET ) ) ) ;
    if ( TRUE == IsBadReadPtr ( dp , sizeof ( DEBUGPACKET ) ) )
    {
        TRACE0 ( "SetSingleStep : invalid parameters\n!" ) ;
        return ( FALSE ) ;
    }

    // Для i386 надо только установить бит TF.
    dp->context.EFlags |= TF_BIT ;
    bSetContext = DBG_SetThreadContext ( dp->hThread , &dp->context ) ;
    ASSERT ( FALSE != bSetContext ) ;
    return ( bSetContext ) ;
}
```

После освобождения процесса отладчиком путем вызова функции `ContinueDebugEvent`, процесс сразу вырабатывает исключение пошагового исполнения после исполнения единственной команды. Отладчик проверяет свое внутреннее состояние, чтобы убедиться, что это было ожидаемое исключение. Так как отладчик ожидал появления исключения пошагового исполнения, то знает, какую точку прерывания восстанавливать. Выполнение одного шага смещает указатель команд на следующую команду после исходной точки прерывания. Следовательно, отладчик может восстановить код команды прерывания в исходной точке прерывания. ОС автоматически очищает флаг TF при каждом возникновении исключения `EXCEPTION_SINGLE_STEP`, поэтому нет нужды очищать его в отладчике. После установки точки прерывания отладчик освобождает отлаживаемую программу и продолжает ее исполнение.

Если вы хотите увидеть всю обработку точки прерывания в действии, взгляните на метод `CWDBGProjDoc::HandleBreakpoint` в файле `WDBGPROJDOC.CPP` из чис-

ла файлов-примеров к этой книге. Я определил собственно точки прерывания в BREAKPOINT.H и BREAKPOINT.CPP, и эти же файлы содержат парочку классов, управляющих различными типами точек прерывания. Я сделал окно Breakpoints WDBG так, чтобы вы имели возможность установки точек прерывания в то время, когда выполняется отлаживаемая программа, точно так же, как это делается в отладчике Visual Studio .NET. Возможность устанавливать точки прерывания на лету означает, что вам нужно четко отслеживать состояния отлаживаемой программы и точек прерывания. Посмотрите метод `CBreakpointsDlg::OnOK` в файле `BREAKPOINTS.DLG.CPP` из числа примеров, прилагаемых к этой книге, чтобы понять особенности того, как я обрабатываю разрешение и запрещение точек прерывания в зависимости от состояния отлаживаемой программы.

Опция меню `Debug Break`, одна из самых проработанных функций, реализованных мной в WDBG, позволяет прервать исполнение и выйти в отладчик в любое время, когда выполняется отлаживаемая программа. В первом издании этой книги я привел весьма пространное объяснение технологии, реализованной для приостановки потоков отлаживаемой программы, установки одноразовых точек прерывания в каждом потоке и как обеспечить исполнение точек прерывания, посылая сообщения `WM_NULL` потокам (об одноразовых точках прерывания см. ниже раздел «Шаг внутрь, шаг через и шаг наружу»). Для этого потребовалось заставить работать весьма большой объем кода, и он в целом работал вполне прилично. Однако в одном случае, когда он не работал, все потоки отлаживаемой программы взаимно блокировались на объектах режима ядра. Так как потоки приостанавливались в режиме ядра, не было способа вытолкнуть их обратно в пользовательский режим. Я вынужден был оставить все как есть и жить с ограничениями своей реализации, поскольку WDBG должен был работать в Windows 98/Me, а также в ОС на базе Windows NT.

Так как я прекратил поддержку Windows 98/Me, реализация меню `Debug Break` стала совершенно тривиальной и теперь работает всегда. Фокус заключается в замечательной функции `CreateRemoteThread` — ее нет в Windows 98/Me, но есть в Windows 2000 и более поздних ОС. Другая функция, дающая такой же эффект, что и `CreateRemoteThread`, но доступна только в Windows XP и более поздних, — `DebugBreakProcess`. Как можно увидеть из следующего кода, собственно реализация весьма проста. Когда удаленный поток вызывает функцию `DebugBreak`, при обработке исключения я имею дело только с результирующим исключением по точке прерывания, будто была выполнена обычная, определенная пользователем, точка прерывания.

```
HANDLE LOCALASSIST_DLLINTERFACE __stdcall
DBG_CreateRemoteBreakpointThread ( HANDLE   hProcess   ,
                                  LPDWORD  lpThreadId  )
{
    HANDLE hRet = CreateRemoteThread ( hProcess   ,
                                       NULL        ,
                                       0            ,
                                       (LPTHREAD_START_ROUTINE)DebugBreak ,
                                       0            ,
                                       0            ,
                                       0            ,
```

```
                                lpThreadId                ) ;  
    return ( hRet ) ;  
}
```

Хотя запуск потока в отлаживаемой программе может показаться рискованным, я почувствовал себя достаточно безопасно, особенно потому, что именно эта технология применяется в WinDBG для реализации меню Debug Break. Заметьте, однако, что вызов `CreateRemoteThread` имеет побочные эффекты. Когда в процессе запускается поток, по соглашению с ОС он вызовет `DllMain` каждой из загруженных DLL, не вызывавших `DisableThreadLibraryCalls`. Соответственно, когда поток завершается, все функции `DllMain`, вызванные с уведомлением `DLL_THREAD_ATTACH`, будут вызваны также с уведомлением `DLL_THREAD_DETACH`. Все это значит, что, если в ваших функциях `DllMain` имеется ошибка, способность функции `CreateRemoteThread` останавливать отлаживаемую программу может только усложнить проблему. Шансы невелики, но это надо учитывать.

Таблицы символов, серверы символов и анализ стека

Настоящее колдовство написания отладчика связано с серверами символов — кодом, манипулирующим таблицами символов. Отладка непосредственно на уровне языка ассемблера интересна в течение нескольких минут и быстро надоедает. Таблицы символов, называемые также отладочными символами, — это то, что преобразует шестнадцатеричные числа в строки исходного текста, имена функций и переменных. Таблицы символов содержат также сведения о типах, используемых в программе. Эта информация позволяет отладчику, получив необработанные данные, отобразить их в виде структур и переменных, определенных в вашей программе.

Работать с современными таблицами символов трудно. Самый распространенный формат таблицы символов, Program Database (PDB), наконец-то получил документированный интерфейс, но с ним очень трудно работать, и он пока не поддерживает такую полезную функциональность, как анализ стека (stack walking). К счастью, `DBGHELP.DLL` предлагает достаточное количество вспомогательных упрощающих жизнь классов-оболочек, которые мы сейчас и обсудим.

Различные форматы символов

Прежде чем погрузиться в дискуссию о доступе к таблицам символов, познакомимся с форматами символов. Я понял, что людей сбивает с толку наличие разных форматов и их разнообразные возможности, поэтому я хочу навести порядок.

Общий формат объектных файлов (Common Object File Format, COFF) был одним из первоначальных форматов таблиц символов и появился в Windows NT 3.1, первой версии Windows NT. Создатели Windows NT имели большой опыт разработки ОС и хотели добавить в Windows NT некоторые уже существующие средства. Формат COFF является частью большей спецификации, которой следовали поставщики UNIX, пытаясь создать общие форматы двоичных файлов. Visual C++ 6 был последней версией компиляторов Microsoft, поддерживавших COFF.

Формат C7, или CodeView, появился как часть Microsoft C/C++ версии 7 во времена MS-DOS. Если вы ветеран программирования, то, возможно, слышали рань-

ше название CodeView — это название старого отладчика Microsoft. Формат C7 обновлен для поддержки ОС Win32, а Visual C++ 7 является последним компилятором, поддерживающим этот формат символов. Формат C7 был частью исполняемого модуля, так как компоновщик добавлял информацию о символах к исполняемому файлу после его компоновки. Добавление символов к двоичному файлу означает, что ваши отлаживаемые модули могут быть весьма большими; информация о символах может легко оказаться больше, чем исполняемый код.

Формат PDB (Program Database, база данных программы) наиболее распространенный сегодня, поддерживается и Visual C++, и Visual C#, и Visual Basic .NET. Каждый, кто имеет более чем пятиминутный опыт работы, видел, что файлы PDB хранят символьную информацию отдельно от исполняемых модулей. Чтобы увидеть, содержит ли исполняемый файл сведения о символах PDB, запустите для исполняемого файла программу DUMPBIN из комплекта поставки Visual Studio .NET. Ключ /HEADERS, указанный в командной строке DUMPBIN, распечатает информацию заголовка переносимого исполняемого файла (Portable Executable, PE). Часть информации заголовка содержит Debug Directories (каталоги отладки). Если для них указан тип cv с форматом RSDS, значит, это исполняемый файл, созданный Visual Studio .NET с PDB-файлами.

DBG-файлы уникальны, так как в отличие от других форматов символов они создаются не компоновщиком. DBG-файл в основном является файлом, содержащим другие типы отладочных символов, таких как COFF и C7. DBG-файлы используют некоторые из таких же структур, определяемых форматом файла PE — форматом, используемым для исполняемых файлов Win32. REBASE.EXE строит DBG-файлы путем выделения отладочной информации COFF или C7 из модуля. Нет нужды запускать REBASE.EXE для модуля, построенного с использованием файлов PDB, так как при этом символы уже отделены от модуля. Microsoft распространяет DBG-файлы с отладочными символами ОС в формате более ранних отладчиков, чем Visual Studio .NET и последней версии WinDBG, для них будет необходимо найти соответствующий PDB-файл для исполняемых файлов ОС.

Доступ к символьной информации

Традиционный способ обработки символов заключался в применении DBGHELP.DLL, поставляемой Microsoft. Раньше DBGHELP.DLL поддерживала только общую информацию, которая включала имена функций и элементарных глобальных переменных. Однако этой информации более чем достаточно для написания некоторых замечательных утилит. В первом издании этой книги я посвятил несколько страниц тому, как загрузить символы с помощью DBGHELP.DLL, но в последних реализациях DBGHELP.DLL загрузка символов заметно улучшена и действительно работает. Мне нет нужды описывать, как применять функции символов DBGHELP.DLL, — я только отошлю вас к документации MSDN. Обновленная версия DBGHELP.DLL входит в комплект Debugging Tools for Windows, поэтому вы можете как-нибудь посетить www.microsoft.com/ddk/debugging для получения новейшей и наилучшей версии.

Когда вышли первые бета-версии Visual Studio .NET, я был восхищен, так как Microsoft предполагала поставлять интерфейс к PDB-файлам. Сначала я подумал, что SDK интерфейса доступа к отладочным данным (Debug Interface Access, DIA)

может стать весьма полезным для нас, людей, интересующихся разработкой средств разработки: мы могли бы получать доступ к локальным переменным и параметрам, а также предоставлять возможность развертывания структур и массивов — короче, имели бы все, что нужно для работы с символами.

Когда я только взялся за второе издание этой книги, я намеревался написать сервер символов, используя DIA, поставляемый с Visual Studio .NET 2002. Первая возникшая проблема заключалась в том, что сервер символов DIA являлся не более, чем программой чтения PDB. Это не очень важно, но это означало, что я должен буду справиться с большим количеством функций обработки высокого уровня самостоятельно. DIA делает то, что и ему положено делать; я просто считал, что он может больше. Приступив к работе, я заметил, что в DIA, кажется, нет поддержки анализа стека. Функция `API StackWalk64`, о которой я еще расскажу, должна иметь некоторые возможности, помогающие осуществлять доступ к символьной информации, необходимой для анализа стека. К сожалению, DIA в то время не показывал всей нужной информации. Например, функции `StackWalk64` необходим пропуск указателя фрейма (frame pointer omission, FPO).

Похоже, что реализация DIA в Visual Studio .NET 2003 поддерживает интерфейсы, которые должны работать при анализе стека, но это лишь отдельные IDL-интерфейсы, а документации оказалось недостаточно для использования этого нового кода. Я думал, что, располагая этими новыми средствами анализа стека, я должен продолжить работу с DIA, так как казалось, что это перспективная вещь, хотя DIA и выглядел весьма неуклюже. А потом я нарвался на самую большую проблему: интерфейс DIA — это псевдо-COM-интерфейс: он выглядит, как COM, но очень громоздкий, из-за чего получаешь все проблемы, связанные с COM, взамен не получая ничего.

Начав работу над базовой частью кода, я наткнулся на интерфейс, наилучшим образом демонстрирующий, почему так важно правильно проектировать COM. Символьный интерфейс `IDiaSymbol` имеет 95 документированных методов. Увы, почти все в DIA является символами. В действительности в перечислении `SymTagEnum` имеется 31 уникальный символьный тип. Все, что в DIA называется символом, на самом деле больше похоже на массив меток, а не на реальные значения. Огромнейшая проблема интерфейса `IDiaSymbol` в том, что все типы поддерживают только несколько из этих 95 интерфейсов. Так, базовый тип, поддерживающий самые элементарные типы символов, такие как целые, поддерживает только два интерфейса: один для получения собственно базового типа перечисления и еще один для получения длины типа. Остальные интерфейсы возвращают просто `E_NOTIMPLEMENTED` (не реализовано). Иметь плоскую иерархию, в которой почти все делает единственный интерфейс, прекрасно, когда совместно используемые элементы относительно малы, но наличие различных типов в DIA ведет к огромному объему кодирования и хождению по кругу, что, по-моему, вовсе не нужно. Типы, используемые DIA, иерархичны, и интерфейс должен быть спроектирован так же. Вместо использования единственного интерфейса буквально для всего типы должны иметь собственные интерфейсы, так как с ними гораздо проще работать. Начав проектировать свою оболочку над DIA, я быстро понял, что я собираюсь написать море кода для построения иерархии над DIA, которая уже должна быть заложена в интерфейс.

Я начал понимать, что в процессе работы над сервером символов, в основе которой лежит DIA, я изобретаю колесо, и это мне не понравилось. Колесом в данном случае являлся сервер символов DBGHELP.DLL. Я уже познакомился с заголовочным файлом DBGHELP.H и заметил, что в позднейших версиях WinDBG библиотека DBGHELP.DLL поддерживала некоторые формы локальных и параметризованных перечислений, а также развертывания структур и массивов. Единственная проблема была в том, что большие куски локальных и параметризованных перечислений оказались недокументированными. К счастью, так как я продолжал «перемалывание» заголовочных файлов DIA, я начал понимать, что некоторые образчики возвращаемых значений в коде DBGHELP.DLL и то, что я видел в CVCONST.H (одном из заголовочных файлах DIA SDK), несомненно, соответствуют друг другу. Это было большим достижением, так как это позволяло приступить к использованию DBGHELP.DLL для получения сведений о локальных символах. DBGHELP.DLL напоминает оболочку поверх DIA, которой гораздо проще пользоваться, чем непосредственно DIA. Поэтому я решил построить свое решение над DBGHELP.DLL, вместо того чтобы самому реализовать эту библиотеку.

Все, чего я достиг, находится в проекте SymbolEngine на диске с примерами. Этот код обеспечивает WDBG локальными символами, а также реализует диалоговое окно SUPERASSERT. В общем, проект SymbolEngine является оболочкой для функций сервера символов DBGHELP.DLL, упрощающей ее использование и расширяющей управление символами. Чтобы избежать проблем экспорта классов из DLL, я сделал SymbolEngine статической библиотекой. В реализации SymbolEngine вы заметите несколько вариантов компоновки, заканчивающихся на _BSU. Это специальные варианты SymbolEngine для BUGSLAYERUTIL.DLL, так как BUGSLAYERUTIL.DLL является частью SUPERASSERT. Я не хотел, чтобы SUPERASSERT использовалась для утверждений, что вызвало бы проблемы с реентерабельностью кода, и поэтому не компоновал ее с этими версиями. Вы также можете заметить, что при использовании сервера символов DBGHELP.DLL я всегда вызывал функции, чьи имена заканчиваются на 64. Хотя в документации говорится, что применение не-64-битных функций в качестве оболочки для вызова 64-разрядных функций возможно, я несколько раз замечал, что непосредственный вызов 64-разрядных функций работает лучше, чем вызов их оболочек. По этой причине я их использую всегда. Вы, возможно, захотите рассмотреть подробнее проект SymbolEngine, но он слишком велик, чтобы привести его в книге и следить с его помощью за обсуждением. Я хочу объяснить, как пользоваться моим перечислением символов, а также коснуться некоторых главных моментов реализации. Последнее замечание о моем проекте SymbolEngine: символьный механизм DBGHELP.DLL поддерживает только символы ANSI. Мой проект SymbolEngine является частично Unicode-совместимым, поэтому мне нет нужды постоянно преобразовывать строки DBGHELP.DLL в Unicode при использовании SymbolEngine. В процессе работы над проектом я при необходимости расширял параметры формата ANSI в Unicode. Не все было конвертировано, но в большинстве случаев этого достаточно.

Основной алгоритм перечисления локальных переменных показан в следующем примере. Как видите, он очень прост. В этом псевдокоде я использовал реальные имена функций из DBGHELP.DLL.

```

STACKFRAME64 stFrame ;
CONTEXT      stCtx  ;

// Заполнить stFrame.

GetThreadContext ( hThread , &stCtx ) ;

while ( TRUE == StackWalk ( . . . &stFrame . . . ) )
{
    // Настроить контекстную информацию, указав
    // перечисляемые локальные переменные.
    SymSetContext ( hProcess , &stFrame , &stCtx ) ;

    // Перечисляем локальные переменные.
    SymEnumSymbols ( hProcess , // Значение, передаваемое SysInitialize.
                    0           , // Базовый адрес DLL, установлен в 0
                               // для просмотра всех DLL.
                    NULL        , // Маска RegExp для поиска,
                               // NULL означает "все".
                    EnumFunc     , // Функция обратного вызова.
                    NULL         ); // Контекст пользователя,
                               // передаваемый функции обратного вызова.
}

```

Функция обратного вызова, адрес которой передается методу `SymEnumSymbols`, получает структуру `SYMBOL_INFO`, показанную в следующем фрагменте. Если нужна только основная информация о символе, такая как адрес и имя, достаточно структуры `SYMBOL_INFO`. Кроме того, поле `Flags` сообщит вам, чем является символ: локальной переменной или параметром.

```

typedef struct _SYMBOL_INFO {
    ULONG      SizeOfStruct;
    ULONG      TypeIndex;
    ULONG64    Reserved[2];
    ULONG      Reserved2;
    ULONG      Size;
    ULONG64    ModBase;
    ULONG      Flags;
    ULONG64    Value;
    ULONG64    Address;
    ULONG      Register;
    ULONG      Scope;
    ULONG      Tag;
    ULONG      NameLen;
    ULONG      MaxNameLen;
    CHAR       Name[1];
} SYMBOL_INFO, *PSYMBOL_INFO;

```

Как почти все в компьютерах, разница между минимальным и требуемым весьма велика, что и является причиной столь большого объема кода в проекте `Symbol-Engine`. Я добивался функциональности, позволяющей перечислять локальные переменные, показывая типы и значения точно так же, как это делает `Visual Studio`

.NET. Как можно увидеть из рис. 4-3 и снимков экрана с окном SUPERASSERT в главе 3, мне это удалось.

Пользоваться моим кодом для перечисления символов просто. Во-первых, определите функцию, прототип которой показан в следующем фрагменте кода. Эта функция вызывается для каждой из переменных. Строковые параметры полностью расширяемого типа: имя переменной (если есть) и значение этой переменной. Параметр отступа определяет величину сдвига по отношению к предыдущим значениям. Так, если у вас есть локальная структура в стеке с двумя полями-членами и вы задаете моему коду перечисления раскрыть три уровня, ваша функция обратного вызова будет вызвана трижды для этой структуры. Первый вызов будет произведен для имени этой структуры и адреса с уровнем сдвига, равным 0. Каждый член структуры получит собственный вызов с уровнем сдвига, равным 1. Вот так и получаются три обращения к функции обратного вызова.

```
typedef BOOL (CALLBACK *PENUM_LOCAL_VARS_CALLBACK)
( DWORD64 dwAddr      ,
  LPCTSTR szType      ,
  LPCTSTR szName      ,
  LPCTSTR szValue     ,
  int     iIndentLevel ,
  PVOID   pContext    ) ;
```

Чтобы перечислить все локальные переменные в середине просмотра стека, просто вызывайте метод `EnumLocalVariables` — он выполнит все для настройки соответствующего контекста и выполнит перечисление символов. Прототип функции `EnumLocalVariables` показан в следующем фрагменте. Первый параметр — функция обратного вызова, второй и третий сообщают функции перечисления, сколько уровней надлежит раскрыть и требуется ли раскрывать массивы. Как можете представить, чем больше вы раскрываете, тем медленнее работает эта функция. Кроме того, раскрывать массивы может оказаться весьма накладно, так как нет способа узнать, сколько элементов массива используется. Хорошие новости в том, что мой код развертывания корректно обращается с массивами `char *` и `wchar_t *`, развертывая не каждый символ, а всю строку целиком. Четвертый параметр — функция чтения памяти, передаваемая методу `StackWalk`. Если вы укажете `NULL`, моя функция перечисления использует `ReadProcessMemory`. Остальные параметры объяснений не требуют.

```
BOOL EnumLocalVariables
( PENUM_LOCAL_VARS_CALLBACK pCallback      ,
  int iExpandLevel           ,
  BOOL bExpandArrays         ,
  PREAD_PROCESS_MEMORY_ROUTINE64 pReadMem  ,
  LPSTACKFRAME64 pFrame      ,
  CONTEXT * pContext        ,
  PVOID pUserContext        ) ;
```

Чтобы увидеть код перечисления локальных переменных в действии, лучше всего запустить его с тестовой программой `SymLookup` из каталога `SymbolEngine\Tests\SymLookup` на прилагаемом к книге CD. `SymLookup` достаточно мала, чтобы вы смогли увидеть, что в ней происходит. Она также демонстрирует все возмож-

ные типы переменных, генерируемых компилятором C++, и вы можете видеть, как разворачиваются различные переменные.

Код реализации разворачивания всех локальных переменных и структур находится в трех исходных файлах. SYMBOLENGINE.CPP содержит функции верхнего уровня для декодирования переменных и раскрытия массивов. Все декодирование типов сосредоточено в файле TYPEDECODING.CPP, а все декодирование переменных — в файле VALUEDECODING.CPP. Если вы будете читать код, вспомните Капитана Рекурсию! Когда я учился в колледже, профессор, читавший Computer Science 101, пришел в аудиторию в костюме Капитана Рекурсии — в трико и накидке, чтобы обучить нас рекурсии. Это было жутковатое зрелище, но я определенно научился всему в рекурсии всего за один урок. Метод декодирования символов похож на то, как они сохраняются в памяти. Пример на рис. 4-4 показывает, что происходит при разворачивании символа, являющегося указателем на структуру. Значения `SymTag*` имеют типы, определенные в CVCONST.H как тэговые. Значение `SymTagData` имеет тип, указывающий, что для его разбора будет применена последовательность рекурсий. Основное правило таково: рекурсия продолжается, пока не встретится некоторый конкретный тип. Различные типы, такие как типы, определенные пользователем (user-defined types, UDT), и классы, имеющие дочерние классы, всегда нуждаются в проверке на наличие наследников.

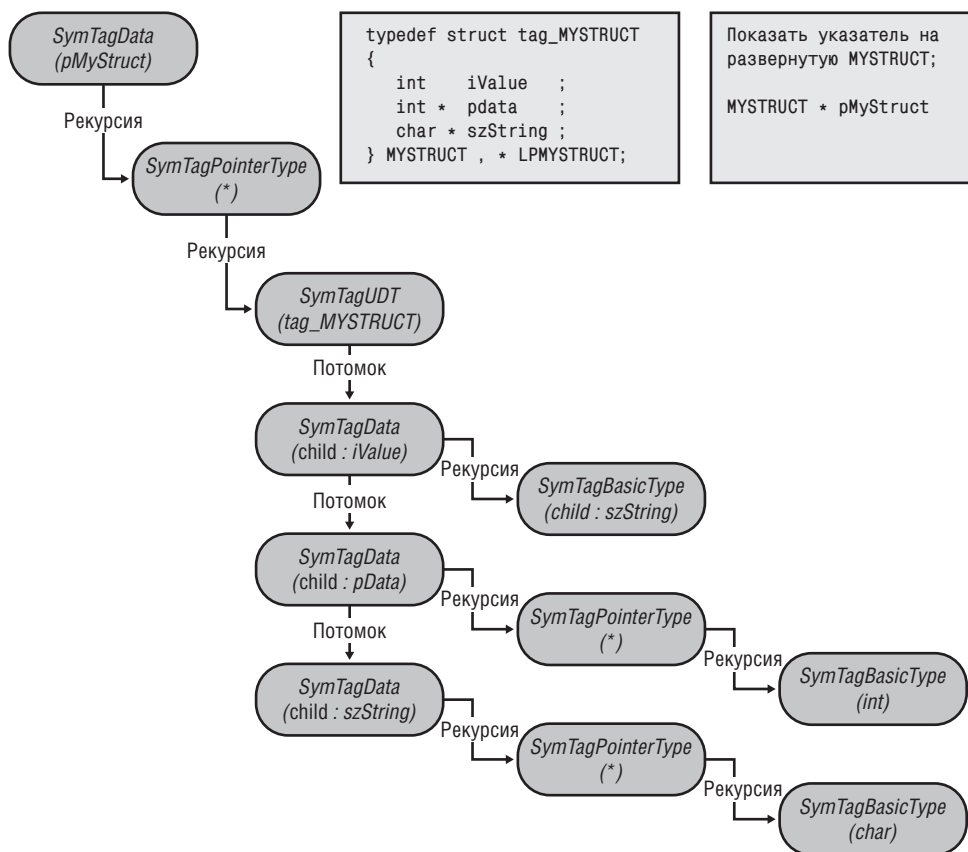


Рис. 4-4. Пример разворачивания символа

Хоть я и пришел к программированию сервера символов DBGHELP.DLL, думая, что это будет относительно просто, оказалось, что Роберт Бернс был прав: «Все лучшие планы мышей и людей часто приводят к печальным итогам»¹. Из всего кода этой книги реализация SymbolEngine и приведение его в рабочее состояние заняло гораздо больше времени, чем что-либо еще. Так как у меня не было ясной картины, какие типы могут появляться при разворачивании, потребовалось совершить множество проб и ошибок, чтобы в конце концов все заработало. Самый большой урок, полученный мной: даже если вы думаете, что вы все понимаете, доказать это можно только в процессе реализации.

Анализ стека

Я уже говорил, что DBGHELP.DLL имеет функцию `API StackWalk64` и поэтому нет нужды писать собственную процедуру анализа стека. Функция `StackWalk64` проста и обеспечивает все потребности в анализе стека. WDBG использует функцию `API StackWalk64` так же, как это делает отладчик WinDBG. Может быть лишь одна загвоздка: в документации нигде явно не говорится, что должно быть указано в структуре `STACKFRAME64`. Этот код демонстрирует поля, которые нужно в ней заполнить:

```
// InitializeStackFrameWithContext из i368CPUHELP.C.
BOOL CPUHELP_DLLINTERFACE __stdcall
InitializeStackFrameWithContext ( STACKFRAME64 * pStack ,
                                CONTEXT *      pCtx      )
{
    ASSERT ( FALSE == IsBadReadPtr ( pCtx , sizeof ( CONTEXT ) ) );
    ASSERT ( FALSE == IsBadWritePtr ( pStack , sizeof ( STACKFRAME64 ) ));
    if ( ( TRUE == IsBadReadPtr ( pCtx , sizeof ( CONTEXT ) ) ) ||
        ( TRUE == IsBadWritePtr ( pStack , sizeof ( STACKFRAME ) ) ) )
    {
        return ( FALSE );
    }

    pStack->AddrPC.Offset      = pCtx->Eip ;
    pStack->AddrPC.Mode        = AddrModeFlat ;
    pStack->AddrStack.Offset   = pCtx->Esp ;
    pStack->AddrStack.Mode     = AddrModeFlat ;
    pStack->AddrFrame.Offset   = pCtx->Ebp ;
    pStack->AddrFrame.Mode     = AddrModeFlat ;

    return ( TRUE );
}
```

Функция `API StackWalk64` отлично справляется со своей работой, поэтому вы можете даже не знать, что анализировать стек оптимизированного кода может быть трудно. Причина трудности в том, что компилятор мог оптимизировать фреймы стека. Компилятор Visual C++ агрессивен при оптимизации кода, и если он может

¹ Та же фраза в переводе С. Я. Маршака: «И рушится сквозь потолок на нас нужда». — *Прим. перев.*

использовать регистр фрейма стека в качестве временного регистра, то делает это. Чтобы облегчить анализ стека в таких ситуациях, компилятор генерирует данные FPO. Данные FPO — это таблица информации, используемой функцией `API StackWalk64` для вычисления, как работать с функциями, для которых нормальный фрейм стека пропущен. Я хотел коснуться FPO, так как вы случайно можете увидеть ссылки на него в MSDN и в разных отладчиках. Для любопытных: структуры данных FPO находятся в `WINNT.H`.

Шаг внутрь, Шаг через и Шаг наружу

Теперь, когда я описал точки прерывания и механизм символов, я хочу объяснить, как отладчики реализуют такую прекрасную функциональность как «Шаг внутрь» (Step Into), «Шаг через» (Step Over) и «Шаг наружу» (Step Out). Я не реализовывал эти функции в WDBG, так как хотел сосредоточиться на центральной части отладчика. Шаг внутрь, Шаг через и Шаг наружу требуют исходного и дизассемблированных текстов, позволяющих отслеживать текущую выполняемую строку текста или оператор. Познакомившись с этим разделом, вы увидите, что архитектура ядра WDBG имеет инфраструктуру, необходимую для реализации этих возможностей, а их добавление является в основном упражнением в программировании UI. Функции Шаг внутрь, Шаг через и Шаг наружу работают на базе одноразовых точек прерывания, которые удаляются отладчиком после того, как они сработают.

Шаг внутрь работает по-разному в зависимости от того, производится отладка на уровне исходного или дизассемблированного текста. В первом случае отладчик должен использовать одноразовые точки прерывания, так как одна строка текста на языке высокого уровня транслируется в одну или более строк языка ассемблера. Если вы переведете центральный процессор в пошаговый режим, то это будет пошаговое исполнение отдельных операторов, а не строк исходного текста.

Во втором случае отладчик знает строку текста, на которой вы сейчас находитесь. Когда вы исполняете команду отладчика Шаг внутрь, отладчик использует сервер символов для нахождения адреса следующей исполняемой строки текста. Отладчик произведет частичное дизассемблирование по адресу следующей строки, чтобы проверить, является ли следующая строка командой вызова функции. Если да, отладчик установит одноразовую точку прерывания на первом адресе функции, которую собирается вызвать отлаживаемая программа. Если по адресу следующей строки находится не команда вызова функции, отладчик устанавливает одноразовую точку прерывания на ней. Установив одноразовую точку прерывания, отладчик освобождает отлаживаемую программу, в результате чего она останется на свежеставленной одноразовой точке прерывания. При срабатывании одноразовой точки прерывания отладчик заменит код операции на ее месте и освободит всю память, ассоциированную с этой одноразовой точкой прерывания. Если пользователь работает на уровне дизассемблера, то Шаг внутрь реализуется гораздо проще, так как отладчик всего лишь переводит процессор в режим пошагового исполнения.

Функция Шаг через похожа на Шаг внутрь тем, что отладчик должен найти следующую строку текста в символьном механизме и произвести частичное диз-

ассемблирование по этому адресу. Разница в том, что при выполнении Шага через отладчик установит точку прерывания после команды вызова функции, если строка является таковой.

Шаг наружу в некотором смысле самый простой: при выборе пользователем этой команды отладчик анализирует стек с целью поиска адреса возврата из выполняемой сейчас функции и устанавливает одноразовую точку прерывания по этому адресу.

Обработка команд Шаг внутрь, Шаг через и Шаг наружу кажется простой, но есть одна особенность, которую надо принимать во внимание. Если вы пишете свой отладчик, обрабатывающий Шаг внутрь, Шаг через и Шаг наружу, то что делать, если вы установили одноразовую точку прерывания для одной из этих команд, а там уже установлена обычная точка прерывания? Как разработчик отладчика, вы имеете два варианта действий. Первый: оставить вашу одноразовую точку прерывания «в одиночестве», пока она не сработает. Второй: удалить вашу одноразовую точку прерывания, когда отладчик уведомит вас о срабатывании обычной точки прерывания. Последний вариант как раз и есть то, что делает отладчик Visual Studio .NET.

Оба метода корректны, но, удаляя одноразовую точку прерывания для команд Шаг внутрь, Шаг через и Шаг наружу, вы избежите путаницы у пользователя. Если вы разрешите срабатывание одноразовой точки прерывания после обычной, пользователь может долго удивляться, почему отладчик останавливается в неправильном месте.

Итак, вы хотите написать свой собственный отладчик

Помнится, я был удивлен количеством программистов, интересующихся написанием отладчиков. Меня не удивляло это, пока я сам жил жизнью разработчика отладчиков. Мы в первую очередь интересовались компьютерами и ПО, потому что хотели знать, как они работали, а отладчики — это волшебное зеркальце, позволяющее вам знать все и вся о них. В результате я получил массу вопросов по электронной почте о том, как приступить к созданию отладчика. Отчасти моя мотивация при написании WDNG заключалась в том, чтобы наконец показать полный пример, чтобы программисты смогли узнать, как работают отладчики.

Первым делом после ознакомления с WDBG надо найти прекрасную книгу Джонатана Розенберга «Как работают отладчики» (Jonathan Rosenberg. How Debuggers Work. — Wiley, 1996). Хотя в книге Джонатана нет кода отладчика, это прекрасное введение и обсуждение реальных проблем, которые необходимо разрешить при создании отладчика. Очень немногие писали отладчики, поэтому она на самом деле поможет.

Вам будет нужно близко познакомиться с форматом PE-файлов и с конкретным процессором, на котором вы работаете. Прочитайте наиболее полные статьи по формату PE-файлов Мэта Питрека (Matt Pietrek) в февральском и мартовском номерах журнала MSDN Magazine за 2002 год. Вы сможете узнать больше о процессорах из руководств по процессорам Intel (www.intel.com).

Прежде чем взяться за полный отладчик, вам, возможно, стоит написать дизассемблер. Это не только научит вас обращаться с процессором, но вы также получите код, который пригодится для отладчика. Дизассемблер в WDBG является кодом только для чтения. Иначе говоря, только разработчик, написавший его, может его читать. Старайтесь сделать свой дизассемблер гибким и расширяемым. Я достаточно программировал на языке ассемблера в прошлом, но, только написав собственный дизассемблер, я стал знать язык ассемблера вдоль и поперек.

Если вы захотите написать собственный дизассемблер, начните со справочных руководств Intel, которые можно загрузить прямо с сервера Intel. Они располагают всей информацией о командах и их кодах. Кроме того, в конце тома 2 есть полная карта кодов операций, которая вам совершенно необходима для преобразования чисел в команды. Исходный код нескольких дизассемблеров болтается в Интернете. Прежде чем погрузиться в работу, вам, возможно, стоит познакомиться с некоторыми из этих дизассемблеров, чтобы узнать, как другие справляются с проблемами.

Что после WDBG?

Само собой разумеется, WDBG делает именно то, что и предполагалось. Однако вы можете расширить его возможности множеством способов. Следующий список должен дать вам некоторые идеи, что можно сделать для расширения возможностей WDBG, если это вам интересно. Если вы расширяете WDBG, я бы хотел знать об этом. Кроме того, как я говорил в главе 1, образцы вашего собственного кода прекрасно подходят для того, чтобы их показать на собеседовании при приеме на работу. Если вы добавите существенную функциональную возможность к WDBG, вы ее должны представить в выгодном свете!

- Пользовательский интерфейс WDBG только достаточен. Первое усовершенствование — это реализация нового пользовательского интерфейса. Вся информация уже есть, вам нужно только спроектировать лучший способ ее отображения.
- WDBG поддерживает только простые точки прерывания по месту. BREAKPOINT.H и BREAKPOINT.CPP готовы для того, чтобы вы добавили интересные виды точек прерывания, такие как счетчик пропусков точек прерывания (исполнить точку прерывания заданное количество раз, прежде чем она сработает) или точки прерывания с выражениями (останавливаться, только если выражение истинно). Обязательно наследуйте свои новые точки прерывания от CLocationBP, чтобы иметь возможность получить код сериализации и не потребовалось бы ничего менять в WDBG.
- Добавьте возможность отсоединяться от процессов при работе под Windows XP/Server 2003 и более поздними версиями ОС.
- Совсем просто добавить в WDBG возможность отладки многопроцессных приложений. Большинство интерфейсов уже настроено для работы со схемой идентификации процессов, поэтому вам потребуется лишь отслеживать процессы, с которыми вы работаете, при обработке уведомлений отладки.
- Интерфейс WDBG уже подготовлен для того, чтобы заглянуть в удаленную отладку и различные процессоры, а UI останется тем же. Напишите DLL уда-

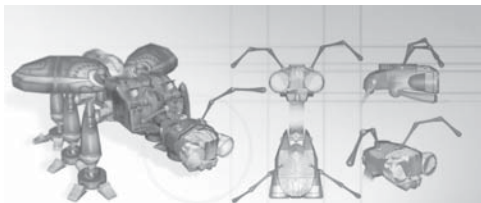
ленной отладки и расширьте WDBG, чтобы пользователи могли выбирать, где отлаживать: на локальной машине или удаленной.

- Вы также могли бы написать дизассемблер получше.
- Код SymbolEngine просто передает декодированные символы функции обратного вызова, и вам надо задавать объем развертывания при первом вызове `EnumLocalVariables`. Если хотите подражать чему-то вроде окна Watch в Visual Studio .NET, вам нужно развертывать локальные перечисления, чтобы возвращать большие двоичные объекты для развертывания «на лету». Вам не понадобится менять что-либо во внутреннем устройстве SymbolEngine — только код верхнего уровня в файле `SYMBOLENGINE.CPP`, управляющий перечислением. Если вы напишете это, возможно, вам захочется расширить `SUPERASSERT`, чтобы вы смогли развертывать символы «на лету».
- В данный момент SymbolEngine нацелен на развертывании локальных переменных на основе области видимости контекста. Посмотрите, как делается развертывание в SymbolEngine, чтобы также разрешить перечисление глобальных переменных. Работа почти сделана — вы должны лишь написать метод, который делает все, что делает `EnumLocalVariables`, кроме вызова `SymSetContext` со стеком, равным `NULL`.

Резюме

Эта глава дала вам общее представление о том, что и как делают отладчики. Было представлено ядро отладочного API Win32 и некоторые вспомогательные системы, используемые отладчиками, например, сервер символов. Вы также познакомились с другими доступными отладчиками, кроме отладчика Visual Studio .NET. Наконец, представлен отладчик WDBG — пример полнофункционального отладчика, иллюстрирующего, как именно работают отладчики.

Если вы никогда ранее не видели, как на этом уровне работают отладчики, вы могли думать, что это чудеса программирования. Однако, поскольку вы видели код WDBG, думаю, вы согласитесь, что отладчики делают ту же самую грязную работу, которой занимаются и прочие программы. Сначала самой большой проблемой при написании отладчика Win32, которую надо было преодолеть, был поиск способа управления локальными переменными, параметрами и типами. К счастью, благодаря серверу символов `DBGHELP.DLL` (а также библиотеке SymbolEngine), мы теперь располагаем прекрасным ресурсом для написания отладчиков или интересного диагностического кода, невозможного ранее.



Эффективное использование отладчика Visual Studio .NET

Сколько диагностического кода ни используй и как тщательно ни планируй, порой приходится применять отладчик. Как я уже много раз говорил, ключевой момент в отладке — стараться обходиться без него, так как он расходует ваше время. Сейчас я знаю: большинство из вас использует отладчик для исправления ошибок своих коллег, а не собственных (ваш код, конечно, совершенен). Просто я хочу убедить вас, что если вам нужен отладчик, то вы можете сделать большую часть работы без него и исправить ошибки максимально быстро. Это значит, что вы захотите делать большую часть работы без отладчика, что позволит вам находить и исправлять ошибки максимально быстро. В этой главе я расскажу, как задействовать преимущества прекрасного отладчика Microsoft Visual Studio .NET. Если вы, как и я, давно занимаетесь разработкой для платформ Microsoft, вы видите явный прогресс в развитии отладчика. На мой взгляд, теперь Visual Studio .NET стала настоящим средством отладки: Windows-программисты теперь имеют один отладчик для сценариев, Microsoft Active Server Pages (ASP), Microsoft ASP.NET, .NET, Web-сервисов XML, неуправляемого кода и отладки SQL в «одном флаконе», и это восхитительно.

Это первая из трех глав, посвященных отладчику Visual Studio .NET. В ней я освещу общие возможности отладки .NET и неуправляемого кода, так как между этими средами много общего. Эти возможности, включая расширенное использование точек прерывания, помогут вам в решении проблем кодирования. Я также дам массу рекомендаций, позволяющих сократить время, проводимое в отладчике. В главе 6 я освещу специфические детали разработки для .NET. В главе 7 мы

обсудим особенности отладки неуправляемого кода. Независимо от того, какой код вы отлаживаете, в этой главе вы найдете много полезного.

Если вы впервые сталкиваетесь с отладчиком Visual Studio .NET, советую ознакомиться с документацией, прежде чем продолжить чтение книги. Я не буду касаться основ отладки, полагая, что при необходимости вы обратитесь к документации. Отладчик обсуждается в документации Visual Studio .NET в разделе Visual Studio .NET\Developing with Visual Studio .NET\Building, Debugging, and Testing или в MSDN Online в разделе Visual Studio .NET\Developing with Visual Studio .NET\Building, Debugging, and Testing (<http://msdn.microsoft.com/library/en-us/vsintro7/html/vxoriBuildingDebuggingandTesting.asp>).

И еще: если вы не прочитали о сервере символов и не настроили его (см. главу 2), вы теряете одну из лучших способностей Visual Studio .NET. Независимо от того, что вы разрабатываете — приложение .NET или неуправляемый код, автоматическая загрузка нужных символов означает, что вы всегда готовы к отладке.

Расширенные точки прерывания

В отладчике Visual Studio .NET просто установить точку прерывания на строке исходного кода, выбрав меню Debug или настроив проект. Просто загрузите исходный файл, поместите указатель на нужную строку и нажмите стандартную клавишу точки прерывания F9. Или же можно щелкнуть левое поле этой строки. Хотя народ, пришедший из Microsoft Visual Basic 6, может и не считать установку точки прерывания на полях выдающимся достижением (в Visual Basic давно можно было так делать), программисты C# и C++ увидят в этом замечательное усовершенствование. Установка точки прерывания таким способом называется установкой точки прерывания по месту. Выход в отладчик происходит при достижении первой команды, реализующей оператор, для которого установлена точка прерывания. Простота установки точки прерывания по месту может дать неверное представление о ее важности: размещение точки прерывания на отдельной строке кода и есть то, что отличает современную отладку от средневековой.

На заре программирования точек прерывания просто не было. Была единственная «стратегия» поиска ошибок — запустить программу, пока она не завершится аварийно, а затем продираться через дебри шестнадцатеричных дампов состояния памяти. Единственными отладчиками в средневековые отладки были трассировочные операторы и вера в Бога. В эпоху ренессанса отладки, наступившую с появлением языков высокого уровня, программисты могли устанавливать точки прерывания, но отлаживать должны были только на уровне языка ассемблера. Языки высокого уровня все еще не были обеспечены средствами просмотра локальных переменных и исходного текста. В процессе эволюции языков в более сложные формы начался век современной отладки, а программисты получили возможность устанавливать точки прерывания на строке исходного кода и видеть свои переменные на дисплее интерпретированными в точном соответствии с их описанием в программе. Такие простые точки прерывания по месту — сверхмощное средство, и только они, я полагаю, позволяют решить 99,46% проблем отладки.

Как бы прекрасно это ни было, установка точек прерывания по месту может очень быстро надоесть. А если вы установите точку прерывания на строке внут-

ри цикла `for`, исполняющегося от 1 до 10000, а ошибка проявляется на 10000-й итерации? Вы не только натрете мозоль на указательном пальце, нажимая клавишу, предназначенную для выполнения команды `Go` (продолжить), но и потратите часы, ожидая наступления итерации, приводящей к ошибке. Не лучше ли было бы как-то сказать отладчику, что вы хотите пропустить прерывание в этой точке 9999 раз, прежде чем программа остановится?

К счастью, есть способ: прошу в царство расширенных точек прерывания. В сущности расширенные точки прерывания позволяют вам программировать «интеллект» точек прерывания, позволяя отладчику управлять рутинными операциями, связанными с выслеживанием ошибок. Пара условий, которые вы можете добавить к расширенным точкам прерывания, — это пропуск точки прерывания определенное число раз и прерывание исполнения, если некоторое выражение принимает значение «истина». Возможности расширенных точек прерывания окончательно переместили отладчики прямо в наше время, позволяя делать за минуты то, что раньше требовало часов, используя простые точки прерывания по месту.

Подсказки к точкам прерывания






Прежде чем перейти к расширенным точкам прерывания, я хочу коснуться четырех вещей, о которых вы, возможно, не догадываетесь. Во-первых, при установке расширенных точек прерывания лучше всего предварительно начать отладку в пошаговом режиме. При запуске приложения без отладки отладчик использует IntelliSense для установки точек прерывания. Однако, когда вы начинаете отладку, в дополнение к IntelliSense вы получаете в распоряжение актуальную базу данных (Program Database, PDB) символов отладки (debugging symbols) программы, помогающую устанавливать ваши точки прерывания. Во всех примерах этой и следующих двух глав я сначала запускал отладчик, а уже потом устанавливал эти точки.

Совет: не делайте окно Breakpoints, отображающее установленные точки прерывания, пристыкованным (docked). Когда вы устанавливаете точки прерывания, Visual Studio .NET иногда устанавливает их, а вы будете удивляться, почему они не работают. Если окно Breakpoints является одним из основных, вы его легко найдете среди тысяч стыкуемых окон интерпретированной среды разработки Visual Studio .NET (IDE). Не знаю, как вас, но меня при работе в Visual Studio .NET тянет установить два 35-дюймовых монитора. Чтобы увидеть окно точек прерывания, нажмите `Ctrl+Alt+B` при стандартной раскладке клавиатуры. Щелкните правой кнопкой заголовок окна точек прерывания или его вкладку и снимите флажок Dockable в появившемся меню. Вам надо проделать это как в нормальном режиме редактирования, так и режиме отладки. Сделав однажды окно точек прерывания одним из основных, щелкните и перетащите его вкладку в первую позицию, чтобы его всегда можно было найти.

В окне Breakpoints отображаются значки, показывающие, установлена ли точка прерывания (табл. 5-1). Значок Warning (предупреждение) — красный кружок с вопросительным знаком — требует дополнительного объяснения. При нормальной отладке вы увидите значок Warning, если установили точку прерывания по

месту в исходном тексте, модуль которого еще не был загружен, поэтому такая точка прерывания в сущности еще не разрешена. Для тех, кто пришел из лагеря Microsoft Visual C++ 6: диалоговое окно Additional DLL (дополнительные DLL) ушло в прошлое. Так как я рекомендую, чтобы вы запустили отладку до начала установки расширенных точек прерывания, значок Warning в окне точек прерывания сигнализирует, что точка прерывания была установлена некорректно.

Табл. 5-1. Коды окна точек прерывания

Значок	Состояние	Значение
	Enabled (Разрешено)	Нормальная активная точка прерывания.
	Disabled (Запрещено)	Точка прерывания игнорируется отладчиком, пока не будет разрешена.
	Error (Ошибка)	Точка прерывания не может быть установлена.
	Warning (Предупреждение)	Точка прерывания не может быть установлена, так как ее место расположения еще не загружено. Если отладчик имеет предположение о точке прерывания, то показывает значок предупреждения.
	Mapped (Отображенная)	Точка прерывания установлена в ASP-коде на странице HTML.

Точку прерывания можно установить в окне Call Stack (стек вызовов), кроме случая отладки SQL. Это полезно, когда вы пытаетесь остановить рекурсию или при глубоко вложенных стеках. Все, что надо сделать, — это подсветить вызов, на котором вы хотите остановиться, и нажать клавишу F9 или щелкнуть правой кнопкой и выбрать Insert Breakpoint (вставить точку прерывания) из появившегося меню.

Однократные точки прерывания можно установить и командой Run To Cursor (запустить до курсора). Их можно установить в окнах редактирования исходного текста, щелкнув правой кнопкой строку и выбрав из меню команду Run To Cursor (она доступна как при отладке, так и при редактировании), и запустится отладка. При раскладке клавиатуры по умолчанию нажатие Ctrl+F10 приведет к тому же результату. Как и с точками прерывания, щелчок правой кнопкой в волшебном окне Call Stack вызывает контекстное меню, также имеющее команду Run To Cursor. Если вы установили точку прерывания до того, как началось выполнение на строке Run To Cursor, отладчик остановится на той точке прерывания и отменит вашу однократную точку прерывания.

В заключение скажу, что в управляемом коде вы можете установить теперь точку прерывания подвыражения (subexpression). Например, если имеется следующее выражение и если при отладке вы щелкнете в этой строке левое поле, красным будет подсвечено только `i = 0` , `m = 0` или инициализирующая часть выражения. Если вы хотели остановиться на подвыражении итератора (в котором происходит увеличение или уменьшение), поместите указатель где-то в районе части выражения `i++` , `m-` и нажмите клавишу F9. В следующем выражении вы можете иметь до трех точек прерывания в одной строке. В окне точек прерывания они отличаются, так как каждая помечена номером строки и позицией в ней. В левом поле будет только одна красная точка, обозначающая точку прерывания. Для сня-

тия всех точек прерывания строки одновременно, щелкните красную точку в левом поле.

```
for ( i = 0 , m = 0 ; i < 10 ; i++ , m- )  
{  
}
```

Быстрое прерывание на функции

Отправной позицией для любой расширенной точки прерывания является диалоговое окно Breakpoint (точка прерывания), доступное по нажатию Ctrl+B при стандартной раскладке клавиатуры. Это окно имеет двойную функцию: New Breakpoint (новая точка прерывания) и Breakpoint Properties (свойства точки прерывания). Во многих отношениях Breakpoint — это просто внешний интерфейс к системе IntelliSense, весьма полезной при написании кода, но используемой и при установке точек прерывания. IntelliSense можно отключить, чтобы проверить точки прерывания в диалоговом окне Options (настройка), папке Debugging (отладка), на странице свойств General (общие), что может ускорить отладку в смешанном режиме, но я настоятельно советую всегда оставлять установленным флажок Use IntelliSense To Verify Breakpoints (использовать IntelliSense для проверки точек прерывания). Вы также сможете работать с точками прерывания IntelliSense, только если открыт исходный текст вашего проекта.

IntelliSense — мощное средство установки точек прерывания, способное сохранить вам массу времени. В пылу отладочных боев, зная имя класса и метода, на котором надо прервать исполнение, я могу ввести его прямо в диалоговом окне Breakpoint на вкладке Function (функция) в поле ввода Function. Я заглядывал через плечо бесчисленному числу программистов, знавших имя метода, но тративших по 20 минут на блуждание по всему проекту, открывая файлы только для того, чтобы установить курсор на строку и нажать F9. Такой метод установки точки прерывания имеет некоторые ограничения, но они необременительны. Во-первых, имя должно вводиться с учетом регистра, если язык программирования чувствителен к регистру (вот где Visual Basic .Net особенно прекрасен!). Во-вторых, в неуправляемом C++ иногда нельзя установить точку прерывания, если имя метода скрыто при определении. Наконец, язык программирования в раскрывающемся списке Language (язык) диалогового окна Breakpoint, должен соответствовать языку текста программы.

Много всего может случиться при попытке установить точку прерывания на функцию в диалоговом окне Breakpoint. Чтобы увидеть результаты, откройте один из ваших проектов или проект из числа примеров к этой книге и пытайтесь устанавливать точки прерывания в диалоговом окне Breakpoint в процессе нашего обсуждения.

Первый случай быстрой установки точки прерывания применим, если вы хотите установить ее на существующий класс и метод. Пусть класс MyThreadClass Visual Basic .NET имеет метод ThreadFunc. При открытии диалогового окна Breakpoint нажатием Ctrl+B все, что нужно сделать, — это ввести mythreadclass.threadfunc (помните: Visual Basic .NET нечувствителен к регистру ввода) и щелкнуть ОК. Для

Visual Basic .NET, J# и C# вы отделяете имя класса от имени метода точкой. Для неуправляемого C++ вы отделяете имя класса от имени метода специфическим для этого языка двойным двоеточием (::). Интересно, что для управляемого C++ для корректной обработки нужно перед выражением добавить `MC::`. Для тех же класса и метода из примера для Visual Basic .NET для управляемого C++ вы должны ввести `MC::MyThreadClass::ThreadFunc`. На рис. 5-1 изображено заполненное диалоговое окно Breakpoint для примера на Visual Basic .NET. Если в момент установки точки прерывания вы не находитесь в режиме отладки, точка, обозначающая точку прерывания, появляется на полях, но красной подсветки `Public Sub ThreadFunc` не будет, так как точка еще не может быть разрешена. После запуска отладки `Public Sub ThreadFunc` подсвечивается красным цветом. Если программа написана на неуправляемом C++, строка кода не подсвечивается. Если вы в режиме отладки, точка прерывания полностью разрешена, на что указывает закрашенная сплошным красным цветом точка в окне Breakpoints, а `Public Sub ThreadFunc` подсвечено красным цветом. В C# и неуправляемом C++ точка прерывания появляется внутри функции, но это первая исполняемая строка кода после пролога функции. Кроме того, только красная точка появляется на полях.

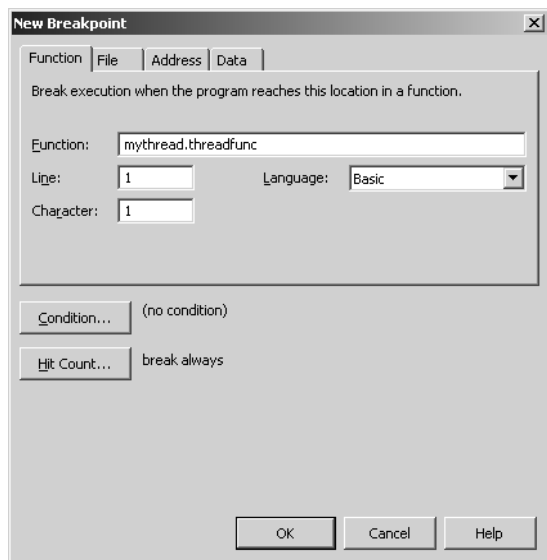


Рис. 5-1. Диалоговое окно Breakpoint для быстрой установки точки прерывания на функцию

Если имя класса и метода заданы в диалоговом окне Breakpoint неверно, вы увидите сообщение: «IntelliSense could not find the specified location. Do you still want to set the breakpoint?» («IntelliSense не может найти указанное место. Вы все же хотите установить точку прерывания?»). Если вы принимаете решение установить точку прерывания на несуществующем методе и работаете с управляемым кодом, то почти наверняка при отладке в точке прерывания будет содержаться вопросительный знак в окне Breakpoints, так как точка не может быть найдена. Как будет показано в главе 7, где мы будем говорить об отладке неуправляемого кода, у вас все же будет шанс установить точку прерывания корректно.

Так как установка точки прерывания путем указания имени класса и метода работает хорошо, я поэкспериментировал, пробуя устанавливать точку прерывания в диалоговом окне Breakpoint, просто указывая имя метода. Для C#, J#, Visual Basic .NET и неуправляемого C++, если имя метода уникально в приложении, окно Breakpoint просто устанавливает точку прерывания, будто вы полностью ввели имя класса и метода. Увы, кажется, управляемый C++ не поддерживает установку точек прерывания простым указанием имени метода.

Так как установка точки прерывания простым указанием имени метода работает очень хорошо и будет экономить ваше время при отладке, вас может удивить то, что случится в большом проекте, когда вам захочется установить точку прерывания на общий или перегруженный метод. Допустим, имеется открытый проект WDBG MFC из главы 4, а вам хочется установить точку прерывания на метод `OnOK`, являющийся методом по умолчанию, который обрабатывает щелчок кнопки ОК. Если вы введете `OnOK` в окне Breakpoint и щелкните ОК, случится нечто весьма приятное.

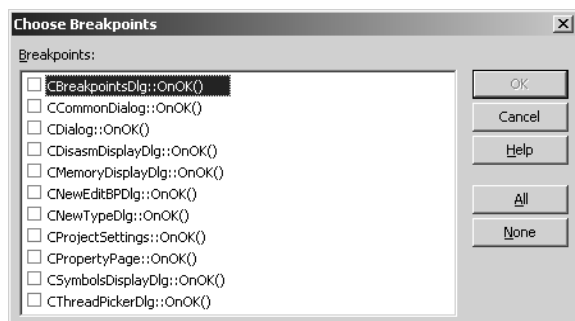


Рис. 5-2. Диалоговое окно *Choose Breakpoints* (выбор точки прерывания)

То, что вы видите на рис. 5-2, есть список IntelliSense всех классов проекта WDBG, имеющих метод `OnOK`. Не знаю, как вы, а я полагаю, что это выдающаяся возможность, особенно тем, что щелчок кнопки All (все), позволяет установить точки прерывания на всех таких методах одним ударом, и это работает во всех языках, поддерживаемых Visual Studio .NET, кроме управляемого C++! Диалоговое окно Choose Breakpoints также отображает перегруженные методы одного и того же класса. Я не знаю, как часто мне приходилось устанавливать точки прерывания и диалоговое окно Choose Breakpoints напоминало мне, что я должен также принять решение о прерывании и на других методах.

Другая моя излюбленная штучка в C++ — просто ввести имя класса в окне Breakpoint. И вдруг возникает диалоговое окно Choose Breakpoints, предлагая все методы класса! Как жаль, что эта сногшибательная функция работает с кодом C++ (а также, что поразительно, с управляемым C++), а с C#, J# и Visual Basic .NET — нет. Но для C++ такая возможность — великое благо. Если вы хотите убедиться, что ваши тестовые примеры охватывают все методы класса, просто введите имя класса в диалоговом окне Breakpoint и щелкните кнопку All в окне Choose Breakpoints. Это сразу установит точки прерывания на все методы, так что вы сможете оценить, все ли методы вызываются вашими тестовыми примерами. Эта возможность также замечательна при поиске неработающего кода, так как если устано-

вить все точки прерывания, а затем снимать по мере их посещения, все оставшиеся точки прерывания соответствуют никогда не исполнявшемуся коду.

Работая над книгой, я использовал Visual Studio .NET 2003 RC2. При проверке, может ли C# и J# устанавливать точки прерывания класса так же, как это делается в неуправляемом C++, я наталкивался на жуткую ошибку, аварийно завершавшую Visual Studio .NET. Надеюсь, эта ошибка исправлена в окончательной версии¹; но если нет, я хочу, чтобы вы знали о ней. В проектах C# и J#, если вы вводите имя класса и точку после него в окне Breakpoint, например Form1., и щелкаете кнопку ОК, появится сообщение об ошибке периода исполнения Visual C++, что приложение попыталось завершиться необычным образом. При щелчке кнопки ОК интегрированная среда разработки (IDE) просто исчезнет. Попытка сделать то же самое в приложении на Visual Basic .NET не закрывает IDE. Хотя это и весьма мерзкая ошибка, только некоторые из вас могут нарваться на нее².

Я просто поражен, что возможность выбора из множества методов в вашем приложении при установке точки прерывания в отладчике не отмечена большим, жирным знаком, как сносшибательная функция Visual Studio .NET. На самом деле в документации MSDN имеется лишь мимолетная ссылка на диалоговое окно Choose Breakpoints. Однако я рад сообщить об этом вместо команды разработчиков отладчика.

Как вы, наверное, догадываетесь, вам теперь нет нужды мучаться с диалоговым окном Choose Breakpoints, если вам известен класс и метод, так как можно просто напечатать их. Конечно же, нужно использовать соответствующий языку программирования разделитель. Если вам нужна полная определенность, можно указать типы параметров перегруженного метода в Visual Basic .NET и C++ с учетом семантических требований этих языков. Что интересно, C# и J# не обращают внимания на параметры, и всегда появляется окно Choose Breakpoints, что, пожалуй, является ошибкой.

Если вы хотите быстро установить точку прерывания при отладке, это еще интереснее, особенно с управляемым кодом. О неуправляемом коде мы поговорим в главе 7, так как во время отладки требуется больше ручной работы для установки точек прерывания. Для управляемого кода я заметил одну очень интересную возможность в окне Breakpoint. Однажды во время утомительной отладки я ввел имя класса и метода библиотеки классов Microsoft .NET Framework вместо класса и метода из своего проекта, и в окне Breakpoints появилось целая куча загадочных точек прерывания. Допустим, имеется консольное приложение, вызывающее Console.WriteLine, а во время отладки вы вводите Console.WriteLine в окне Breakpoint. Вы получаете обычное сообщение, что IntelliSense не знает, что ему делать. Если вы щелкните Yes (да) и попадете в окно Breakpoints, вы увидите нечто вроде того, что изображено на рис. 5-3 (необходимо полностью раскрыть все от корня дерева).

Это дочерние точки прерывания. Документация Visual Studio .NET сообщает, что они есть и что это такое. Например, в документации сказано, что дочерние точки прерывания возникают, когда вы устанавливаете точку прерывания на пе-

¹ В окончательной версии эта ошибка устранена. — *Прим. перев.*

² В окончательной версии эта ошибка также устранена. — *Прим. перев.*

перегруженных функциях, но окно Breakpoints всегда показывает их как точки прерывания верхнего уровня. С дочерними точками прерывания вы можете встретиться при отладке нескольких исполняемых кодов и когда две программы загружают один и тот же элемент управления в свои домены приложения или адресные пространства, а вы устанавливаете точки прерывания в одно и то же место элемента управления в обеих программах. Дико то, что окно Breakpoints на рис. 5-3 изображает единственную исполняющуюся сейчас программу, в которой я устанавливаю точку прерывания на `Console.WriteLine`.

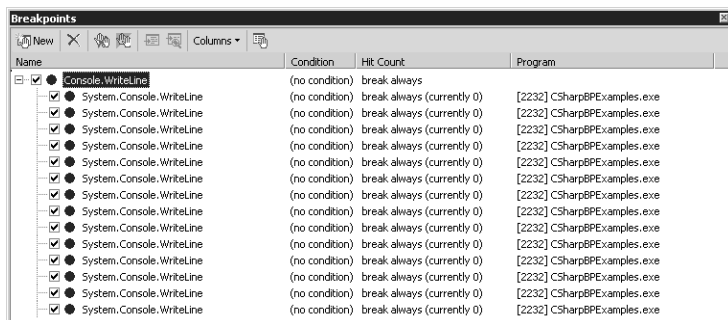


Рис. 5-3. Дочерние точки прерывания в окне Breakpoints

Если щелкнуть правой кнопкой на дочерней точке прерывания и выбрать `Go To Disassembly` (перейти к дизассемблированному тексту), появляется окно Disassembly с дизассемблированным кодом. Однако тайну происхождения можно раскрыть, щелкнув правой кнопкой на дочерней точке прерывания и выбрав из меню Properties (свойства), а в результирующем диалоговом окне Breakpoint — вкладку Address (адрес) (рис. 5-4). Вы увидите, что отладчик сообщает, что точка прерывания установлена на `System.Console.WriteLine` в самом начале функции.

Если это не совсем понятно, вы всегда можете запустить свою программу и заметить, что остановились в глубоком тумане ассемблера x86. Открыв окно Call Stack, вы увидите, что остановились внутри вызова функции `Console.WriteLine` и даже видны переданные параметры. Прелесть таких недокументированных средств обработки точек прерываний в том, что вы всегда можете заставить свое приложение остановиться в заданной точке.

Хотя я сделал лишь один вызов `Console.WriteLine` в своей программе, окно Breakpoints показывает 19 дочерних точек прерывания (рис. 5-3). Методом проб и ошибок я открыл, что количество дочерних точек прерывания связано с количеством перегруженных методов. Иначе говоря, установка точки прерывания путем ввода имени класса и метода .NET Framework или ввода при отсутствии исходного текста приводит к тому, что точка прерывания будет установлена для всех перегруженных методов. Из документации вы узнаете, что `Console.WriteLine` имеет только 18 перегруженных методов, но позвольте заметить, что, посмотрев на класс `Console.WriteLine` с помощью ILDASM, вы увидите, что в действительности имеется 19 перегруженных методов.

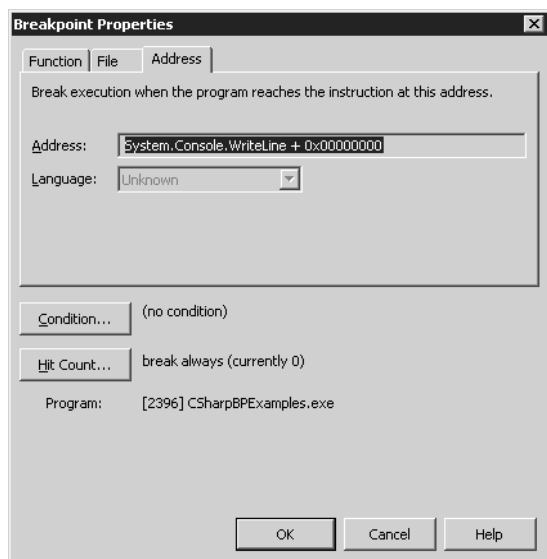


Рис. 5.4. Точка прерывания на любой вызов `Console.WriteLine`

Способность останавливаться на методе, вызванном откуда угодно из моего домена приложения, — вещь крутая. Я немного познакомился с тем, как взаимодействуют классы .NET Framework, останавливаясь на конкретных методах библиотеки классов .NET Framework. Хотя для примера я использовал статический метод, эта технология отлично работает с экземплярными методами и свойствами, если эти методы и свойства вызываются вашим доменом приложения. Имейте в виду: чтобы правильно устанавливать точки прерывания на свойства, нужно указать их префикс `get_` или `set_` в зависимости от того, что вы собираетесь прерывать. Чтобы, скажем, установить точку прерывания на свойство `Console.In`, надо указать `Console.get_In`. Кроме того, важен выбор языка программирования в диалоговом окне Breakpoint. При установке таких точек прерывания для всего домена приложения в местах, где вы не располагаете исходным текстом, мне нравится использовать Basic, чтобы даже при ошибке в регистре ввода точка прерывания все равно была бы установлена.

Есть две проблемы, о которых вы должны знать при установке таких точек прерывания для домена приложения. Первая: точки прерывания, установленные таким способом, не сохраняются при перезапуске приложения. Рассмотрим пример, в котором мы добавили при отладке точку прерывания на `Console.WriteLine`. Окно Breakpoints будет показывать «`Console.WriteLine`» при перезапуске программы, но значок точки прерывания изменится на знак вопроса, а точка прерывания станет найденной и никогда не будет установлена. Вторая: вы можете устанавливать такие точки для всего домена приложения, только если указатель команды находится в коде, для которого вы располагаете PDB-файлом. Если попробовать установить точку прерывания в окне Disassembly при отсутствии исходного текста, она установится как найденная и никогда не будет активизирована.

Хотя вы могли подумать, что я добил тему быстрой установки точек прерывания по месту, все же имеется одно неочевидное, но сверхмощное место, где воз-

можно установка точек прерывания по месту. На панели инструментов есть поле со списком Find (найти) (рис. 5-5). Если в нем напечатать имя класса и метода и нажать F9, на этом методе, если он существует, будет установлена точка прерывания. Если указан перегруженный метод, система автоматически установит точки прерывания на перегруженные методы. Поле со списком Find немного отличается тем, что, если точку прерывания нельзя установить, она и не будет установлена. Кроме того, как и окно Breakpoint при работе с проектом C++, указание имени класса в поле со списком Find и нажатие F9 установит точки прерывания на каждый из методов класса.

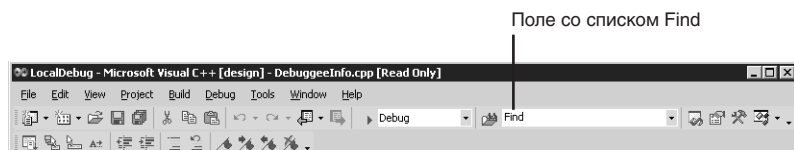


Рис. 5-5. Поле со списком Find

Это маленькое поле со списком Find имеет еще два маленьких секрета. При стандартной раскладке клавиатуры, если вы введете имя файла проекта или include-файла из переменной окружения INCLUDE и нажмете Ctrl+Shift+G, поле со списком Find откроет этот файл для редактирования. В заключение, если вам нравится окно Command Visual Studio .NET, попробуйте следующее: в поле со списком Find введите символ «больше» (>) и наблюдайте превращение этого поля в миниатюрное командное окно со своим собственным IntelliSense. Все эти недокументированные чудеса поля со списком Find часто наводят меня на мысль напечатать «Fix my bugs!» (исправь мои ошибки!) и посмотреть, как этот запрос чудесным образом исполняется.

Модификаторы точек прерывания по месту

Теперь, когда вы умеете расставлять точки прерывания налево и направо, я могу вернуться к некоторым сценариям, обсуждавшимся в начальном разделе о точках прерывания. Основная идея — добавить точкам прерывания мозгов для более эффективного использования отладчика. Наиболее выигрышными являются счетчики выполнений (hit counts) и условные выражения.

Счетчики выполнений

Простейший модификатор, применяемый с точками прерывания по месту, — это счетчик выполнений, иногда также именуемый счетчиком пропусков. Он говорит отладчику поместить точку прерывания, но не останавливаться на ней, пока строка кода не будет исполнена заданное число раз. С этим модификатором прерывание внутри циклов на заданном повторе становится тривиальным.

Добавить счетчик выполнений к точке прерывания по месту очень просто. Во-первых, установите обычную точку прерывания по месту на строке или на подвыражении строки. Для управляемого кода щелкните правой кнопкой в красной области строки для точки прерывания по месту и выберите Breakpoint Properties (свойства точки прерывания) из появившегося меню. Для неуправляемого кода щелкните правой кнопкой красную точку в левом поле. Кроме того, вы можете

выбрать точку прерывания в окне Breakpoints и щелкнуть кнопку Properties или щелкнуть правой кнопкой окно и выбрать Properties. Независимо от того, как вы это делаете, вы попадете в диалоговое окно Breakpoint, где нужно щелкнуть кнопку Hit Count (счетчик выполнений).

В заключительном диалоговом окне Breakpoint Hit Count (счетчик выполнений точки прерывания) вы увидите, что Microsoft усовершенствовала опции счетчика выполнений по сравнению с предыдущими отладчиками. В раскрывающемся списке When The Breakpoint Is Hit (когда достигается точка прерывания) надо выбрать один из вариантов реакции (табл. 5-2) и ввести количество выполнений в поле ввода рядом с выпадающим списком.

Табл. 5-2. Варианты работы счетчика выполнений

При достижении счетчика выполнений	Описание
Break always (всегда прерывать)	Останавливать каждый раз, когда исполняется это место.
Break when the hit count is equal to (прервать, если счетчик выполнений равен)	Остановить, если произошло заданное количество выполнений. Отсчет выполнений начинается с 1.
Break when the hit count is a multiple of (прерывать, если счетчик выполнений кратен)	Прерывать каждые x исполнений.
Break when the hit count is greater than or equal to (прерывать, если счетчик выполнений больше или равен)	Исполнять это место, пока не достигнуто значение счетчика выполнений, и прерывать каждое исполнение после этого. Так счетчик выполнений работал в предыдущих версиях отладчиков Microsoft.

Счетчики выполнений полезны потому, что при прерывании исполнения отладчик показывает, сколько раз исполнялась строка программы, где установлена точка прерывания. Если ваша программа валится или происходит разрушение данных внутри цикла, но вам неизвестно, на какой итерации это происходит, добавьте точку прерывания по месту к строке внутри цикла и добавьте модификатор счетчика прерываний со значением большим, чем количество повторений цикла. При аварийном завершении программы или при разрушении данных активизируйте окно Breakpoints; в колонке Hit Count для этой точки прерывания в скобках вы увидите, сколько раз был исполнен цикл. Окно Breakpoints на рис. 5-6 демонстрирует состояние счетчика выполнений после возникновения разрушения данных. Для неуправляемого кода имейте в виду, что состояние счетчика работает, только если программа запущена «на полной скорости». При пошаговом проходе через точку прерывания счетчик выполнений не обновляется.

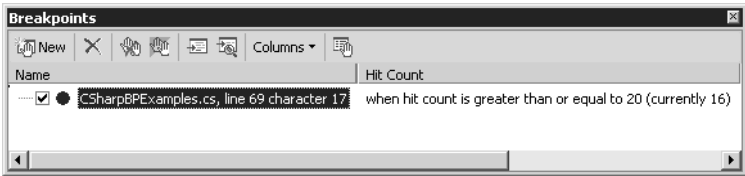


Рис. 5-6. Пример отображения оставшегося числа выполнений

Новым в счетчике выполнений является то, что вы в любое время можете сбросить значение счетчика в 0. Вернитесь к окну Breakpoint, щелкните кнопку Hit Count и щелкните кнопку Reset Hit Count (сбросить счетчик выполнений) диалогового окна Breakpoint Hit Count. Еще одна приятность: вариант работы счетчика выполнений может быть изменен в любой момент времени, а кроме того, можно изменять заданное количество выполнений, не изменяя его текущее состояние.

Условные выражения

Второй модификатор точек прерывания по месту, позволяющий при корректном использовании сэкономить больше времени, чем какой-либо еще тип точки прерывания, — это условное выражение. Точка прерывания по месту, имеющая условное выражение, срабатывает, только если результат вычисления выражения становится верным (истина) или изменяется с момента предыдущего вычисления. Условное выражение получить управление именно тогда, когда это нужно. Отладчик может справиться почти с любым подброшенным ему выражением. Для добавления условного выражения к точке прерывания откройте окно Breakpoint для точки прерывания по месту и щелкните кнопку Condition (условие), в результате чего появится диалоговое окно Breakpoint Condition (условие точки прерывания).

В поле ввода Condition введите условие, подлежащее проверке, и щелкните ОК. Управляемый и неуправляемый коды имеют разную поддержку условий, и у каждой свои «прелести», которые мы обсудим в последующих главах. Вкратце различия заключаются в том, что в управляемом коде вы можете вызывать из условных выражений методы и функции (будьте весьма осторожны) и нет поддержки псевдорегистров и псевдозначений (специальный код, начинающийся с символов @ или \$). В неуправляемом коде вы не можете вызывать функции из своих условных выражений, но имеете доступ к псевдорегистрам и псевдозначениям.

И все же обе среды поддерживают общие выражения, которые можно интерпретировать так: «Что находится в скобках выражения `if`, введенного в строке точки прерывания?» Вы располагаете полным доступом к локальным и глобальным переменным, так как они вычислены в контексте текущей области видимости времени исполнения. Модификатор условного выражения точки прерывания позволяет напрямую проверять предположения, которые вы стремитесь доказать или опровергнуть. Заметьте: синтаксис выражения должен подчиняться правилам языка программирования, для которого установлена точка прерывания; поэтому не путайте `And` с `||`.

По умолчанию обработчик условия прерывает исполнение, если результат вычисления введенного условного выражения равен `true`. Однако, если нужно прервать исполнение при изменении значения условия, вы можете изменить переключатель с `Is True` (является истиной) на `Has Changed` (изменен). Может смутить то, что значение не вычисляется при вводе условия, но после некоторого размышления оказывается, что это имеет смысл. Первый раз вычисление производится, когда вы попадаете на точку прерывания. Так как условие никогда еще не вычислялось, отладчик не видит сохраненных значений во внутреннем поле выражения, поэтому он сохраняет новое значение и продолжает исполнение программы. Таким образом, возможно по меньшей мере два исполнения такой точки до того, как отладчик остановит исполнение программы.

Что еще хорошо в определении Has Changed, так это то, что введенное условие не обязательно должно быть реальным условием. Может быть введена также переменная, доступная в области видимости точки прерывания по месту, при этом отладчик будет вычислять и сохранять значение этой переменной, а вы получаете возможность остановиться при ее изменении. Прочтите материал по точкам прерывания в главах 6 и 7 для понимания отличительных особенностей условных модификаторов точки прерывания для управляемого и неуправляемого кода. Кроме того, для облегчения отладки неуправляемого кода имеются другие типы точек прерывания. Я хочу ответить на вопрос, который вертится у вас на языке (сняв, таким образом, напряженность): при отладке управляемого кода не поддерживаются глобальные точки прерывания.

Несколько точек прерывания на одной строке

Один из наиболее распространенных вопросов, на которые мне доводилось отвечать, когда речь шла об установке точек прерывания, касается установки нескольких точек прерывания в одной строке текста. До Visual Studio .NET это было невозможно. Однако вы не будете ежедневно устанавливать несколько точек прерывания на одной строке текста, так как это не так просто сделать. Вы также будете применять только условные точки прерывания по месту для каждой из этих точек. Как вы догадываетесь, две обычных безусловных точки прерывания по месту в одной строке не очень-то полезны.

После установки первой точки прерывания обычными методами хитрость установки второй таится в чудесных свойствах окна Breakpoint. Проще всего показать это на примере. В следующем фрагменте перед каждой строкой проставлены номера строк в виде комментариев.

```
/* Исходный файл: CSharpBPExamples.cs. */
/*84*/   for ( i = 0 , m = 0 ; i < 10 ; i++ , m- )
/*85*/   {
/*86*/       Console.WriteLine ( "i = {0} m = {0}" , i , m ) ;
/*87*/   }
```

Я устанавливаю первую точку прерывания на строке 86, щелкнув правой кнопкой в левом поле и установив условное выражение в `i==3`. Чтобы установить вторую точку прерывания, переместите окно Breakpoints так, чтобы видеть номер строки исходной точки прерывания, и вызовите окно Breakpoint, нажав Ctrl+B. Щелкните вкладку File (файл) диалогового окна Breakpoint, потому что эту точку прерывания надо устанавливать там. Введите имя файла в поле ввода File и номер строки в поле ввода Line (строка), а в поле ввода Character (символ) оставьте символ 1. В этом примере я использовал файл CSharpBPExamples.CS, а номер строки — 86. Щелкните кнопку Condition и введите нужное условие для второй точки прерывания; я введу `m==--1`.

Окно Breakpoints показывает две точки прерывания, установленные на одной строке кода, а значки слева представляют собой красные кружки, т. е. обе точки активны. Начав отладку, вы увидите, что программа останавливается на обеих точках прерывания при выполнении соответствующих условий. Чтобы узнать, какое из условий вызвало прерывание, посмотрите в окне Breakpoints, какое из них выде-

лено полужирным начертанием. Интересно, что если щелкнуть правой кнопкой точку прерывания в окне с исходным кодом и выбрать из меню Breakpoint Properties, вы всегда увидите свойства первой точки прерывания, установленной на этой строке. Кроме того, если щелкнуть точку в левом поле, будут сняты обе точки прерывания. Хотя приведенный здесь пример слегка надуман — можно было бы объединить оба выражения с помощью оператора «или» (`||`) — множественные точки прерывания в строке весьма полезны при необходимости проверить два сложных выражения.

Окно Watch

Если бы я должен был дать Оскара за технические достижения и общую полезность в Visual Studio .NET, легко победило бы окно Watch (окно наблюдения). Невероятная мощь, предлагаемая окном Watch и его родственниками: диалоговым окном QuickWatch (быстрое наблюдение), Autos (автопеременные), Locals (локальные данные), This/Me (этот класс), — позволяет быстро устранять ошибки, а не шарахаться целыми днями в поисках ответа.

Обращаю внимание: для изменения значений переменных годится любой родственник окна Watch. К сожалению, многие программисты прошли через другие среды разработки, и некоторые из тех, кто много лет программировал для Windows, не осведомлены об этих возможностях. Возьмем окно Autos. Вы только выбираете переменную или дочернюю переменную, значение которой хотите изменить, и щелкаете один раз поле Value. Просто введите новое значение, и переменная изменена.

Многие программисты рассматривают окно Watch как место, предназначенное только для чтения, в которое они притаскивают свои переменные и наблюдают за ними. Окно Watch замечательно тем, что оно содержит полный комплект встроенных средств для вычисления выражений. Если вы хотите видеть что-либо как целое, что на самом деле таковым не является, просто приведите или конвертируйте его тип точно так же, как вы запрограммировали бы это на используемом вами языке программирования. Простой пример: допустим, `CurrVal` объявлена как целое, а вы хотите видеть ее приведенной к булеву типу. В колонке Name (имя) окна Watch в C# и C++ введите `(bool)CurrVal` или для Visual Basic .NET введите `CBool(CurrVal)`. Значение будет отображаться как `true` или `false`.

Замена целого типа на булев не слишком впечатляет, но способность окна Watch вычислять выражения предоставляет вам прекрасные средства тестирования. Как я уже несколько раз говорил, область кода — это одна из целей, за которую надо бороться при блочном тестировании. Если, например, имеется условное выражение внутри функции и трудно на первый взгляд понять, что там вычисляется, и вы должны по шагам пройти ветви, соответствующие значению `true` и `false` соответственно, окно Watch становится вашим спасителем. Встроенная полновесная возможность вычисления выражений позволяет вам просто перетащить выражение вниз в окно Watch и наблюдать, что там вычисляется. Но есть и ограничения. Если ваше выражение вызывает всевозможные функции вместо использования переменных, у вас могут возникнуть проблемы. Взглянув на мой код, вы обнаружите, что я следую такому правилу: если есть три или больше подвыражений в усло-

вии, я использую только переменные, чтобы видеть результат выражения в окне Watch. Некоторые из вас могут быть знатоками таблиц истинности и способны вычислять эти выражения в уме, но я к таким не отношусь.

Разберем следующий пример. Вы можете выделить все внутри внешних скобок и перетащить в окно Watch. Однако, так как здесь три строки, окно Watch интерпретирует все это как три различных строки. Но можно последовательно скопировать в буфер обмена две строки, а затем вставить их в первую и, таким образом, построить полное выражение без необходимости много печатать. Как только выражение введено в строке окна Watch, в колонке Value отображается true или false в зависимости от значений переменных.

```
if ( ( eRunning == m_eState ) ||  
    ( eException == m_eState ) &&  
    ( TRUE == m_bSeenLoaderBP ) )
```

Следующий шаг — поместить каждую из переменных, участвующих в выражении, в отдельные строки окна Watch. В этом случае действительно круто то, что вы можете начать изменение значений индивидуальных переменных и смотреть, как автоматически изменяется полное выражение в первой строке в соответствии с изменением подвыражений. Я люблю эту функцию, так как она не только помогает при работе с областью кода, но и позволяет увидеть область генерируемых данных.

Вызов методов в окне Watch

Иногда меня забавляют программисты, перешедшие в Windows-программирование с ОС UNIX, настойчиво утверждающие, что UNIX лучше. Когда я спрашиваю, почему, они возмущенно отвечают: «В GDB можно вызвать функцию отлаживаемой программы из отладчика!» Я был удивлен, узнав, что оценки ОС вертелись вокруг загадочной функции отладчика. Конечно, такие приверженцы весьма быстро остывали, когда я говорил им, что мы могли вызывать функции из отладчиков Microsoft всегда. Вы можете удивиться, что же здесь так привлекает. Однако, если вы мыслите, как гурю отладки, то поймете, что способность выполнять функции отладчиком позволяет полностью настроить под себя среду отладки. Например, вместо того чтобы потратить 10 минут, разглядывая 10 различных структур данных для определения совместимости данных, вы можете написать функцию, которая проверяет данные, и затем вызвать ее, когда она больше всего будет нужна — когда ваше приложение остановлено в отладчике.

Позвольте предложить два примера из написанных мной методов, которые я вызывал только из окна Watch. Первый пример касается структуры данных, которую нужно было раскрывать в окне Watch, но, чтобы увидеть ее целиком, я вынужден был все время щелкать маленькие плюсы от Канадской границы до Северного полюса. Имея метод, предназначенный только для отладчика, я мог гораздо проще видеть всю структуру данных. Второй пример был реализован, когда я наследовал некий код, который (не смейтесь) имел общие узлы в связанном списке и бинарном дереве. Код был хрупок, и я должен был быть вдвойне уверен, что я ничего не испортил. Имея метод, предназначенный только для отладчика, я по сути получил функцию утверждения, которую мог задействовать как угодно.

В управляемых приложениях интересно то, что, когда вы смотрите свойство объекта в окне Watch, на самом деле вызывается аксессор `get`. Вы можете легко это проверить: поместите следующее свойство в класс, запустите отладку, переключитесь в окно `This/Me` и раскройте значение `this/Me` объекта. Вы увидите, что возвращаемое имя есть имя домена приложения, которому принадлежит свойство.

```
Public ReadOnly Property WhereAmICalled() As String
    Get
        Return AppDomain.CurrentDomain.FriendlyName
    End Get
End Property
```

Конечно же, если есть свойство объекта, копирующее 3-гигабайтную базу данных, автоматическое вычисление может стать проблемой. К счастью, Visual Studio .NET позволяет отключить просмотр свойств в диалоговом окне Options, папка Debugging (отладка), флажок Allow Property Evaluation In Variables Windows (разрешить вычисление свойств в окнах переменных). Еще лучше то, что вы можете включать и отключать вычисление свойств на лету, а отладчик тут же реагирует на это изменение. Вы можете сразу узнать, разрешено ли вычисление свойств, так как семейство окон Watch сообщает об этом в поле Value: «Function evaluation is disabled in debugger windows. Check your settings in Tools.Options.Debugging.General» (Вычисление функций в окнах отладчика запрещено. Проверьте параметры в Tools.Options.Debugging.General).

Неуправляемый код несколько отличается тем, что вы должны сказать окну Watch, что метод надо вызывать. Заметьте, что я использую здесь общий термин «метод». В действительности для неуправляемого кода вы наверняка можете вызывать только функции C или статические методы C++, так как нормальные методы неуправляемого C++ требуют указателя `this`, который вы можете иметь или не иметь в зависимости от контекста. Методы управляемого кода немного более дружелюбны к указателю `this`. Если программа остановлена внутри класса, содержащего метод, подлежащий вызову, окно Watch автоматически предполагает необходимость использования указателя `this`.

Как вы уже увидели, вызов свойств управляемого кода тривиален. Вызов методов в управляемом или неуправляемом коде выполняется так же, как и их вызов из вашего кода. Если методу не нужны какие-либо параметры, просто напечатайте имя метода и добавьте открывающую и закрывающую скобки. Так, если отлаживается метод `MyDataCheck ()`, вы вызовете его из окна Watch строкой `MyDataCheck ()`. Если отлаживаемый метод требует параметров, просто укажите их, как если бы вы вызывали метод обычным образом. Если отлаживаемый метод возвращает значение, оно отображается в колонке Value окна Watch.

Общая проблема вызова функций неуправляемого кода — необходимость убедиться, что они действительны. Для проверки функции введите ее имя в окне Watch и не добавляйте кавычек или параметров. Если она может быть найдена, окно Watch покажет тип и адрес функции в колонке Value. Кроме того, мы можем по желанию также задать расширенный синтаксис точки прерывания (см. главу 7) для функции, сужая область видимости

Некоторые правила применимы как к управляемому, так и неуправляемому коду при вызове методов из окна Watch. Первое правило: метод должен выполняться

не более 20 секунд, так как пользовательский интерфейс отладчика не реагирует до завершения метода. Через 20 секунд управляемый код сообщает: «Evaluation of expression or statement stopped» (вычисление выражения или утверждения прекращено), «error: function '<method name>' evaluation timed out» (ошибка: функция <имя метода> — тайм-аут вычисления) или «Error: cannot obtain value» (Ошибка: невозможно получить значение), а неуправляемый код сообщает: «CXX001: Error: error attempting to execute user function» (Ошибка при попытке выполнить функцию пользователя). Хорошо то, что ваши потоки будут продолжать выполняться. Это прекрасно, так как при вызове методов неуправляемого кода, которые зависали, Visual Studio 6 просто прерывала исполнение текущего исполняющегося потока. Другая хорошая новость: вы можете оставить в покое вызванные методы в окне Watch в многопоточных программах. Предыдущие версии Visual Studio прекращали исполнение любых потоков, которые оказались активными к тому времени, когда вы исполняли свой отладочный метод в другом потоке. Последнее правило имеет общий смысл: для проверки данных обращайтесь к памяти только на чтение. Если вы думаете, что отлаживать программу, меняющую свое поведение из-за побочных эффектов в проверочных функциях, классно, то подождите неприятностей в окне Watch. Кроме того, если вам нужно что-либо выводить, пользуйтесь трассировкой.

Все, что я здесь говорил об окне Watch, является общим как для отладки .NET-приложений, так и приложений неуправляемого кода. Вы также можете автоматически развернуть собственные типы в окне Watch, но для управляемого и неуправляемого кода это делается разными способами. Кроме того, отладка неуправляемого кода имеет массу других возможностей форматирования данных и управления ими (см. главы 6 и 7).

Команда Set Next Statement

Одна из самых крутых скрытых возможностей отладчика Visual Studio .NET — команда Set Next Statement (установить следующий оператор), доступная как в окне исходного текста, так и в окне Disassembly через контекстное меню, но только в режиме отладки. Set Next Statement позволяет установить указатель команд на другое место в программе.

Хороший пример того, когда нужно использовать команду Set Next Statement, — заполнение структуры данных вручную. Вы по шагам проходите метод, вставляющий данные, изменяете значения, передаваемые функции и с помощью Set Next Statement повторяете вызов метода. Таким образом, вы заполняете структуру данных путем изменения порядка исполнения кода.

Нужно отметить, что изменение указателя команд может легко «свалить» вашу программу, если вы не очень аккуратны. Работая с отладочной компоновкой, можно применять команду Set Next Statement в окне исходного текста программы без особых проблем. Если же вы работаете, скажем, с оптимизированной компоновкой неуправляемого кода, безопаснее всего использовать окно Disassembly. Компилятор перемещает код так, что строки исходного текста будут исполняться в другой последовательности. Кроме того, работая с командой Set Next Statement, вы должны знать о создаваемых в стеке временных переменных (см. главу 7).

Если я ищу ошибку и мое предположение заключается в том, что ошибка находится в определенной части кода, я ставлю точку прерывания перед вызывающей подозрение функцией или функциями. Я проверяю данные и параметры, передаваемые функциям, и прохожу их по шагам. Если проблема не повторяется, я использую команду Set Next Statement для возврата точки исполнения обратно к точке прерывания и изменяю данные, передаваемые функциям. Такая тактика позволяет проверять несколько предположений в одном сеансе отладки, сокращая в конечном итоге время. Но делать так можно не всегда, поскольку исполнение некоторого кода один раз, а затем его повторение может разрушить состояние. Команда Set Next Statement лучше всего работает, когда код не сильно меняет состояние программы.

Как я уже говорил, команда Set Next Statement пригодится при блочном тестировании. Например, она удобна для проверки обработчиков ошибок. Скажем, имеется оператор `if`, и нужно проверить, что случится, если его условие не выполняется. Все, что вам необходимо, — это сделать проверку условия и командой Set Next Statement перенести точку исполнения в ветвь, соответствующую невыполнению условия. Кроме Set Next Statement, в меню имеется команда Run To Cursor, доступная также в контекстном меню, вызываемом правым щелчком в окне исходного текста; она позволяет установить «одноразовую» точку прерывания.

Заполнение структур данных, особенно списков и массивов, — другое прекрасное применение команды Set Next Statement при отладке или тестировании. Если есть код, заполняющий структуру данных и добавляющий ее к связанному списку, командой Set Next Statement можно добавить дополнительные элементы к связанному списку и увидеть, как программа работает в этих случаях. Такое применение команды Set Next Statement особенно полезно, если вам нужно обеспечить появление трудновоспроизводимых данных в процессе отладки.

Стандартный вопрос отладки

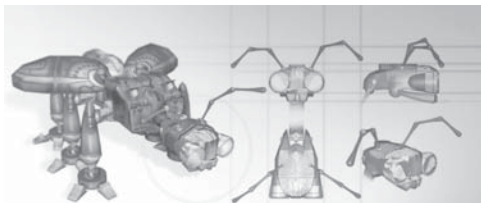
Может ли Visual Studio .NET отлаживать обычные Web-приложения ASP?

Конечно, может, но не сразу, так как сначала нужно добавить некоторые разделы реестра и установить разрешения для DCOM на Web-сервере. Трудно найти правильные шаги, так как они зарыты очень глубоко в документации. Поищите «ASP Remote Debugging Setup» для определения необходимых шагов.

Резюме

Microsoft прислушалась к программистам и выпустила отладчик Visual Studio .NET, существенно упрощающий решение некоторых сверхтрудных проблем отладки. Эта глава познакомила с общими особенностями точек прерывания при работе с управляемым и неуправляемым кодом. Вы должны стараться отдать отладчику Visual Studio .NET как можно больше работы, чтобы сократить время, проводимое в нем.

Расширенные точки прерывания помогают избежать утомительных сеансов отладки, позволяя точно описать условия, при которых срабатывает точка прерывания. Хотя у точек прерывания управляемого и неуправляемого кода свои особенности, модификаторы точек прерывания по месту, счетчики выполнений и условные выражения — ваши лучшие друзья, в основном потому, что сэкономят уйму времени за счет более эффективного использования отладчика. Я настоятельно рекомендую вам поиграть с ним, чтобы избежать необходимости изучать их особенности в критический момент.



Улучшенная отладка приложений .NET в среде Visual Studio .NET

Платформа Microsoft .NET Framework избавила нас от извечных проблем, связанных с утечками и искажениями памяти, однако еще не наступило время, когда «код делает именно то, что я хочу, а не что пишу», а значит, нам все-таки иногда придется испытывать сомнительное удовольствие отладки программ и искать причины ошибок. В этой главе я опишу конкретные стратегии, которые помогут сделать отладку приложений .NET менее тягостной. В предыдущей главе я уже упоминал некоторые методы отладки программ .NET в среде Visual Studio .NET, но в этой главе я опишу их подробнее. Я начну с рассмотрения нескольких факторов, специфических для отладки всех типов приложений .NET именно при помощи Visual Studio .NET, затем перейду к разным хитростям и методам, связанным с отладкой программ .NET в общем. В заключительной части главы я расскажу о дизассемблере промежуточного языка Microsoft (Microsoft Intermediate Language Disassembler, ILDASM) и работе с ним.

В связи с Visual Studio .NET 2003 часто упоминается один новый инструмент (я не буду рассматривать его в данной главе и расскажу про него позднее) — поддержка отладочного расширения SOS (Son of Strike), которая позволяет получать информацию о коде .NET из дампов памяти, а также при отладке неуправляемого кода. Речь о SOS пойдет в главе 8, так как я обнаружил, что интегрировать и использовать его вместе с WinDBG гораздо легче, чем с Visual Studio .NET (странно, но это так!). Если в вашем случае есть хоть какая-то вероятность получения дампов памяти приложений .NET, я настоятельно рекомендую прочитать раздел, посвященный SOS.

Усложненные точки прерывания для программ .NET

В главе 5 я показал, что отладчик Visual Studio .NET заметно облегчает прерывание выполнения программы именно в той части кода, где вы хотите. В случае кода .NET модификаторы условных выражений для точек прерываний по месту имеют некоторые особенности.

Условные выражения

Один из самых частых вопросов про условные точки прерывания, с которым я сталкиваюсь уже много лет, таков: можно ли вызывать функции из модификаторов условных выражений для точек прерываний по месту? При отладке неуправляемого кода — нет, но в случае .NET вполне допустимо. Вызов функций или методов из условных выражений способен облегчить отладку, однако, если делать это без должного внимания, побочные эффекты могут сделать отладку почти невозможной.

Когда я только приступил к изучению .NET, я не понимал эту дополнительную силу условных выражений, потому что их функциональность казалась вполне обычной. Скажем, я использовал такие выражения, как `MyString.Length == 7`, и думал, что отладчик достигает соответствующего места в отлаживаемой программе и получает значение длины строки, читая его прямо из памяти, точно так же, как и при отладке неуправляемого кода Win32. Попробовав выражение, вызывавшее более интересный и сложный аксессор чтения свойства (property get accessor), я начал экспериментировать, чтобы узнать все имеющиеся у меня возможности. В конце концов я обнаружил, что возможны любые допустимые вызовы, кроме вызовов методов Web.

Пожалуй, процесс вызова методов из условных выражений лучше всего продемонстрировать на примере. В программе ConditionalBP (листинг 6-1) из набора примеров к книге, используется класс `Test0`, следящий за числом вызовов метода. Установив условную точку прерывания на строке `Console.WriteLine` в функции `Main` при помощи выражения `(x.Toggle() == true) || (x.CondTest() == 0)`, вы увидите, что выполнение программы будет прерываться, только когда поле `m_bToggle` имеет значение `true`, а поле `m_CallCount` — нечетное значение. Наблюдая при остановках цикла за значениями полей экземпляра `x` класса `Test0`, вы сможете убедиться в том, что они изменяются, а это показывает, что код действительно выполняется.

Листинг 6-1. Пример модификатора условной точки прерывания

```
using System ;
namespace ConditionalBP
{

    class Test0
    {
        public Test0 ( )
        {
```

```
        m_CallCount = 0 ;
        m_bToggle = false ;
    }

    private Int32 m_CallCount ;

    public Int32 CondTest ( )
    {
        m_CallCount++ ;
        return ( m_CallCount ) ;
    }

    private Boolean m_bToggle ;

    public Boolean Toggle ( )
    {
        m_bToggle = !m_bToggle ;
        return ( m_bToggle ) ;
    }
}

class App
{
    static void Main(string[] args)
    {
        Test0 x = new Test0 ( ) ;

        for ( Int32 i = 0 ; i < 10 ; i++ )
        {
            // Точка прерывания: (x.Toggle() == true) || (x.CondTest() == 0 )
            Console.WriteLine ( "{0}" , i ) ;
        }

        x = null ;
    }
}
```

Прежде чем включать вызов свойства или метода в модификатор условного выражения точки прерывания по месту, надо проверить, что свойство или метод делают. Если метод копирует 3-Гбайтную базу данных или приводит к каким-то другим нежелательным изменениям, вам, вероятно, не захочется вызывать его. Еще один интересный аспект вычисления отладчиком методов и свойств: 20-секундный лимит времени окна Watch не имеет отношения к вызову методов из условных выражений. Если у вас есть метод, для вычисления которого нужно несколько дней, отладчик будет покорно ждать. К счастью, пользовательский интерфейс Visual Studio .NET при этом не блокируется, так что, нажав Ctrl+Alt+Break или выбрав в меню Debug пункт Break All (прервать отладку всех программ), вы сможете сразу прекратить отладку.

Эта удивительная возможность вызывать методы и свойства из условных точек прерывания имеет в среде Visual Studio .NET один минус, про который я должен рассказать. Если задать некорректное условие до начала отладки, отладчик сообщит, что он не может вычислить выражение и остановится. Однако, если некорректное условие, которое не может быть вычислено или генерирует исключение, задать после начала отладки, отладчик не остановится. Если вы задаете неверное выражение после начала отладки, вам очень не повезло.

При недопустимом условии отладчик сообщит, что он не смог установить точку прерывания (рис. 6-1), но не остановится, как вы предполагали, а продолжит выполнение отлаживаемой программы. Ничто так не огорчает, как воспроизведение почти неуловимой ошибки только для того, чтобы отладчик просто пропустил ее. Этот недочет имелся в Visual Studio .NET 2002, сохранился он, увы, и в Visual Studio .NET 2003. Я очень надеюсь, что Microsoft решит эту проблему в будущих версиях Visual Studio .NET, чтобы при задании недопустимого условия отладчик останавливался и позволял исправить его.

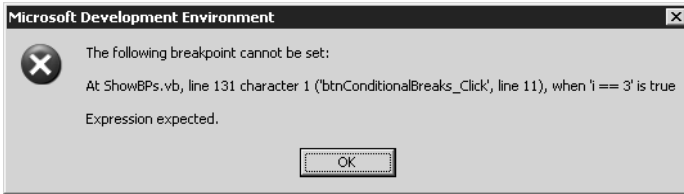


Рис. 6-1. Visual Studio .NET сообщает о невозможности установки точки прерывания

Проблема модификаторов условных выражений точек прерывания так коварна, что я хочу привести несколько ситуаций, в которых с ней можно столкнуться. Первый пример показывает, насколько внимательно нужно относиться к побочным эффектам условных выражений. Как вы думаете, что случится, если в следующем фрагменте кода C# установить условную точку прерывания на строке с вызовом `Console.WriteLine` и ввести условие `i = 3` (заметьте: в условии только один знак равенства)? Если вы думаете, что условие изменит значение `i` и вызовет бесконечный цикл, вы абсолютно правы.

```
for ( Int32 i = 0 ; i < 10 ; i++ )
{
    Console.WriteLine ( "{0}" , i ) ;
}
```

Второй пример. Допустим, у нас есть приложение Microsoft Visual Basic .NET на базе Windows Forms, включающее простой метод:

```
Private Sub btnSkipBreaks_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
    Handles btnSkipBreaks.Click

    Dim i As Integer
```

```
' Очистка поля вывода.
edtOutput.Clear()

m_TotalSkipPresses += 1

edtOutput.Text = "Total presses: " + _
                m_TotalSkipPresses.ToString() + _
                vbCrLf

For i = 1 To 10
    ' Добавление символов в поле вывода.
    edtOutput.Text += i.ToString() + vbCrLf
    ' Обновление поля вывода при каждой итерации цикла.
    edtOutput.Update()
Next
End Sub
```

Если при установке точки прерывания в цикле `For` вы будете думать на C# и введете модификатор условного выражения точки прерывания как `i==3` (при программировании на Visual Basic .NET правильное выражение должно было бы иметь вид `i=3`), то из-за того, что программа выполняется, вы увидите информационное окно (рис. 6-1). Печально то, что в поле будет выведен весь текст, т. е. выполнение кода не прервется. При отладке приложений Windows Forms и консольных приложений условные выражения всегда надо задавать до запуска отладчика. Тогда, если какое-то выражение окажется неправильным, Visual Studio .NET сообщит об этом и остановится на первой строке программы, позволяя исправить проблему. Что до приложений Microsoft ASP.NET и Web-сервисов XML, то в этих случаях средств, позволяющих быстро решить проблему модификаторов условных выражений точек прерываний, просто нет, поэтому при работе над программами таких типов нужно вводить выражения особенно тщательно.

Проблемы с условными выражениями точек прерывания могут возникнуть и тогда, когда выражение почему-либо вызывает исключение. Вы увидите то же простое информационное окно (рис. 6-1). Хорошая новость в том, что исключение не вызовет крах программы, потому что оно никогда не передается вашему приложению, но вы все же не сможете остановить программу и исправить выражение. Используя в выражении переменные, будьте особенно внимательны к тому, чтобы написать его правильно.

Наконец, я хочу напомнить, что языки C# и J# позволяют применять в условных выражениях точек прерываний значения `null`, `true` и `false`. Странно, но в Visual Studio .NET 2002 значение `null` не поддерживается. В Visual Basic .NET при логическом сравнении можно использовать `True` и `False`. Сравнение переменной с `Nothing` можно выполнить оператором `Is` (`MyObject Is Nothing`).

Стандартный вопрос отладки

Как сделать, чтобы программа прерывалась, только когда метод вызывается конкретным потоком?

Чтобы установить точку прерывания для конкретного потока, нужно определить его уникальный идентификатор. Нам повезло: разработчики Microsoft .NET Framework включили в класс `System.Threading.Thread` свойство `Name`, чем сделали задачу идентификации потока тривиальной. Благодаря этому вы можете использовать в условных выражениях точек прерываний что-то вроде `"ThreadIDWantToStopOn" == Thread.CurrentThread.Name`. Конечно, это подразумевает, что, начиная поток, вы всегда задаете его свойство `Name`.

Первый способ получения уникального идентификатора потока — указать имя потока вручную путем изменения значения свойства `Name` в окне Watch. Если у вас есть переменная экземпляра `MyThread`, можете ввести `MyThread.Name` и указать новое имя потока в столбце Value (значение). Если у вас нет переменной потока, задать имя текущему потоку можно через свойство `Thread.CurrentThread.Name`. Указывая имя потока при помощи отладчика, вы не должны задавать его в своем коде. Свойство `Name` может быть задано только раз, и, если вы попытаетесь задать его повторно, будет сгенерировано исключение `InvalidOperationException`. Если вы работаете с собственным кодом, не включая в него много элементов управления сторонних фирм, задавать имя потока из отладчика довольно безопасно, так как библиотека классов .NET Framework не использует свойство `Name`.

Однако, если вы работаете с большим числом элементов управления сторонних фирм или включаете в свою программу код, в котором, как вы подозреваете, свойство `Name` может использоваться, я расскажу про другой метод получения уникального идентификатора потока, также предоставляющий уникальное значение, хотя и в менее удобной форме. Глубоко в недрах класса `Thread` есть закрытая целочисленная переменная — `DONT_USE_InternalThread`, имеющая уникальное значение для каждого потока. (Да, вы можете применять в условных выражениях закрытые переменные.) Чтобы установить точку прерывания для конкретного потока, выберите его в окне Threads (Потоки). В окне Watch введите `Thread.CurrentThread.DONT_USE_InternalThread`, чтобы увидеть значение `DONT_USE_InternalThread`, которое позволит создать подходящее условное выражение точки прерывания. Помните: любая переменная с именем `DONT_USE_xxx` может исчезнуть в будущих версиях Visual Studio .NET.

Окно Watch

Как я упоминал в главе 5, окно Watch — одно из самых лучших отладочных средств в Visual Studio .NET. Мы уже обсудили вызов методов и задание свойств, поэтому мне осталось рассказать об отладке управляемого кода при помощи окна Watch только одно: как настроить окно Watch, чтобы сделать отладку еще быстрее.

Автоматическое развертывание собственных типов

Если вы когда-то занимались отладкой управляемого кода, то, возможно, заметили, что определенные типы выводят в окно Watch больше информации, чем другие. Так, при исключении типа `System.ArgumentException` в столбец Value окна Watch всегда выводится связанное с исключением сообщение, в то время как при исключении типа `System.Threading.Thread` выводится только тип. В первом случае вы можете быстро получить важную информацию о классе, тогда как во втором вам нужно развернуть класс и искать специфическую информацию, например имя, среди огромного списка полей-членов. Я считаю, что, если и в столбце Value, и в столбце Type окна Watch выводится одно и то же значение, пользы от этого мало. На рис. 6-2 показан пример типа, который отчаянно нуждается в авторазвертывании.

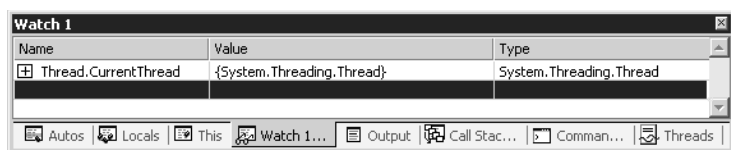


Рис. 6-2. Необходимость авторазвертывания

Добавив в окно Watch собственные типы, а также ключевые классы библиотеки .NET Framework, которых еще там нет, вы сэкономите много времени благодаря быстрому получению нужной информации. Самое лучшее в развертывании то, что оно применяется также к подсказкам для данных, появляющимся при перемещении курсора над переменными в окнах исходного кода, а также при выводе параметров в окно Call Stack (стек вызовов).

Авторазвертывание очень полезно, однако оно неидеально. Самый серьезный его недостаток в том, что авторазвертывание поддерживается только в C#, J# и Managed Extensions for C++ (управляемые расширения для C++). Как и отсутствие комментариев XML в Visual Basic .NET, невозможность авторазвертывания довольно сильно огорчает, потому что это делает отладку кода Visual Basic .NET сложнее, чем она могла бы быть. Кроме того, файл правил авторазвертывания при запуске Visual Studio .NET имеет доступ только для чтения, поэтому при добавлении в него правил авторазвертывания и их тестировании вам придется поработать. Интересно, что в случае авторазвертываний для неуправляемого кода, про которые я рассказываю в главе 7, файл правил читается в начале каждого сеанса отладки. Еще одна небольшая проблема: файл, содержащий правила развертывания, хранится не в каталоге проекта, а в установочном каталоге Visual Studio .NET. Это значит, что при сохранении копии файла авторазвертываний своей группы в системе управления версиями вам придется жестко задавать путь к рабочему каталогу, чтобы отладчик мог найти его.

Первый шаг к получению всей мощи авторазвертываний заключается в нахождении соответствующих файлов правил. Эти файлы (MCEE_CS.DAT для C#, VJSEE.DAT для J# и MCEE_MC.DAT для Managed Extensions for C++) находятся в подкаталоге <установочный каталог Visual Studio .NET>\COMMON7\PACKAGES\DEBUGGER. Изучив эти три файла, вы обнаружите, что в файле для C# несколько больше развертываний, чем в файле для Managed Extensions for C++ (файл для J# содержит больше развертываний, специфических для Java). Чтобы получить дополнитель-

ные правила авторазвертывания при работе над проектом Managed Extensions for C++, возможно, стоит скопировать их из файла MCEE_CS.DAT в MCEE_MC.DAT.

Комментарии в начале обоих файлов, разделенные точками с запятой, описывают операции развертывания типов C# и Managed Extensions for C++. Хотя документация путем умолчания подразумевает, что в правилах авторазвертывания могут применяться только значения действительных полей, на самом деле из правил можно вызывать аксессоры чтения свойств, а также методы. Конечно, стоит убедиться в том, что метод или свойство, которые вы используете, возвращают что-то, что действительно стоит возвращать, например строку или число. При возвращении классов или сложных типов значений вы не увидите никакой полезной информации — только тип. Как обычно, я должен предупредить вас об опасности применения свойств и методов, способных вызвать побочные эффекты.

В качестве примера я добавлю авторазвертывание C# для типов класса `System.Threading.Thread`, чтобы, если имя потока задано, его можно было видеть в столбце Value. Изучая примеры, которые Microsoft приводит в файле MCEE_CS.DAT, вы заметите, что большинство типов специфицированы полными ограничителями пространств имен. Так как в скудной документации, находящейся в начале файла MCEE_CS.DAT, о требованиях к пространствам имен ничего не говорится, для избежания любых проблем я всегда использую полное имя типа.

Документация в файле MCEE_CS.DAT представлена в форме Бэкуса-Наура (Bakus-Naur form, BNR), которая далеко не всегда легка для понимания. Чтобы вам было легче, я приведу более ясный и здравый формат правил авторазвертывания: `<type>=[text]<member/method, format>...>`. Значение каждого поля объясняется в табл. 6-1. Для элемента `type` и раздела `member/method, format` угловые скобки обязательны. Заметьте также, что при авторазвертывании одному элементу `type` могут соответствовать несколько компонентов данных (полей `member`).

Табл. 6-1. Записи авторазвертывания в MCEE_CS.DAT и MCEE_MC.DAT

Поле	Описание
type	Имя типа, которое должно быть полным типом.
text	Любой текст. Обычно в этом поле указывается имя компонента данных или его сокращенный вариант.
member/method	Отображаемый компонент данных или вызываемый метод. В данном поле можно указывать выражения, если вы хотите, чтобы было выведено вычисленное значение. Также допускаются операторы приведения типов.
format	Дополнительные спецификаторы формата переменных для их вывода в той или иной системе счисления. Разрешены значения <code>d</code> (десятичная <code>c/c</code>), <code>o</code> (восьмеричная <code>c/c</code>) и <code>h</code> (шестнадцатеричная <code>c/c</code>).

Например, при развертывании класса `System.Threading.Thread` меня интересует свойство `Name`, поэтому я должен поместить в файл такую запись:

```
<System.Threading.Thread>=Name=<Name>
```

Результат этого развертывания в окне Watch см. на рис. 6-3. На рис. 6-4 вы заметите еще более приятный факт: в подсказках также отображаются авторазвертывания. На CD, прилагаемом к книге, вы найдете мой файл MCEE_CS.DAT, включа-

ющий полезные авторазвертывания, про которые, как я думаю, программисты Microsoft просто забыли, например, развертывание класса `StringBuilder`. Вы можете использовать его в качестве основы для создания собственных файлов авторазвертываний.

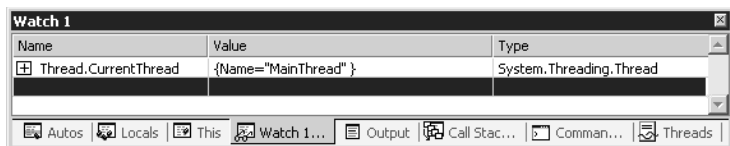


Рис. 6-3. Радость от авторазвертываний

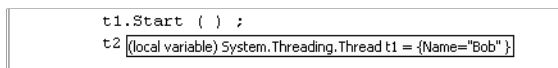


Рис. 6-4. Восторг от авторазвертываний

Стандартный вопрос отладки

Есть ли способ, позволяющий увидеть в окне отладчика поток выполнения управляемого приложения?

Совместную работу компонентов программы иногда лучше всего изучать с помощью иерархии вызовов. Отладчик Visual Studio .NET не обеспечивает такой возможности (может, пока кто-то не напишет соответствующий модуль надстройки), но зато это можно сделать, применив CORDBG.EXE, отладчик из состава .NET Framework (о способе наблюдения за потоком выполнения без помощи отладчика см. главу 11). Если вы желаете познать все радости отладки, какой она была примерно в 1985 году, консольный отладчик CORDBG.EXE — то, что вам нужно.

CORDBG.EXE поддерживает команду `wt`, аналогичную командам `Watch` и `Trace` отладчика WinDBG. Она не только показывает полную иерархию вызовов, но и число машинных команд, выполняемых в результате вызова каждого метода. Используя `wt`, лучше всего устанавливать точку прерывания на первой строке метода, с которого вы хотите начать трассировку. Программа будет выполняться, пока не будет достигнута команда возвращения из метода, в котором была установлена точка прерывания.

Ниже приведен вывод, полученный в результате применения `wt` для приложения, которое вызывает три пустых функции (программа `WT.CS` с CD, прилагаемого к книге). Как вы можете представить, в результате вызова какого-либо метода из библиотеки классов .NET Framework может быть выведен просто огромный текст.

```
(cordbg) wt
1      App::Main
3      App::Foo
3      App::Bar
5      App::Baz
3      App::Bar
```

см. след. стр.

```
3      App: Foo
3      App: Main
21 instructions total
```

Если вы только начинаете работать с CORDBG.EXE, самой важной командой для вас будет `?`, потому что она выводит справочную информацию о других командах. Указав после `?` интересующую вас команду, вы получите подробную информацию о ней, а также примеры ее использования.

Советы и хитрости

Прежде чем перейти к ILDASM и промежуточному языку Microsoft (MSIL), я хочу рассказать о некоторых трюках, которые могут пригодиться при работе с управляемым кодом.

DebuggerStepThroughAttribute и DebuggerHiddenAttribute

Есть два очень интересных атрибута, демонстрирующих реальную силу программирования на основе атрибутов в .NET: `DebuggerStepThroughAttribute` и `DebuggerHiddenAttribute`. Они могут применяться к свойствам, методам и конструкторам, но CLR никогда их не использует. Тем не менее отладчик Visual Studio .NET использует их в период выполнения для контроля пошагового исполнения программы. Запомните: если вы будете неосторожны, эти атрибуты могут сделать отладку кода очень сложной (если не невозможной), так что используйте их на свой страх и риск. Я вас предупредил!

Более полезным является атрибут `DebuggerStepThroughAttribute`. Когда он применяется к классам, структурам, конструкторам или методам, отладчик Visual Studio .NET автоматически «перешагивает» через них, даже если вы используете команду `Step Into` (шаг внутрь). Однако вы можете задавать точки прерывания в элементах программы, которые желаете отладить. Этот атрибут лучше всего использовать в элементах, содержащих только одну строку кода, таких как аксессоры чтения и записи. Вы также увидите, что этот атрибут применяется в случае метода `InitializeComponent`; в код, создаваемый при помощи Visual Basic .NET Windows Forms Designer, он включается автоматически. Возможно, вам следует задействовать его в методе `InitializeComponent` программ Windows Forms, написанных на C#, J# или Managed Extensions for C++.

`DebuggerStepThroughAttribute` по крайней мере позволяет устанавливать точки прерывания, тогда как `DebuggerHiddenAttribute` скрывает метод или свойство, к которым он применен, и не позволяет устанавливать точки прерывания в этом методе или свойстве. Я настоятельно рекомендую не использовать этот атрибут, потому что иначе вы не сможете отлаживать соответствующую часть кода. Однако он может оказаться полезным для полного сокрытия внутренних методов. `DebuggerHiddenAttribute` не является простым антиотладочным приемом, так как разработчики отладчика сами решают, читать ли метаданные для атрибута. На момент написания книги отладчик DBGCLR.EXE для Visual Studio .NET и .NET Framework SDK поддерживает этот атрибут, а CORDBG.EXE — нет.

Отладка в смешанном режиме

Отладка на уровне исходного кода неуправляемой DLL, вызванной управляемым приложением, называется отладкой в смешанном режиме (mixed mode debugging). Для начала нужно выяснить, как этот режим включить. Команды для приложений C# см. на рис. 6-5, а для приложений Visual Basic .NET — на рис. 6-6. Но я никак не возьму в толк, почему для C# и Visual Basic .NET понадобились две абсолютно разные страницы свойств, если все ключи командной строки для компиляторов, а также большинство команд по сути идентичны.

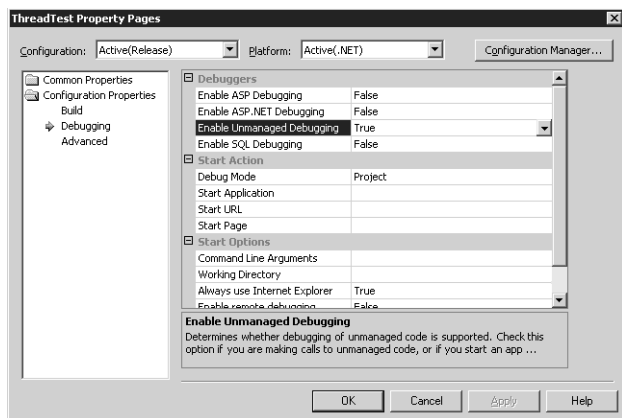


Рис. 6-5. Включение отладки в смешанном режиме для проекта C#

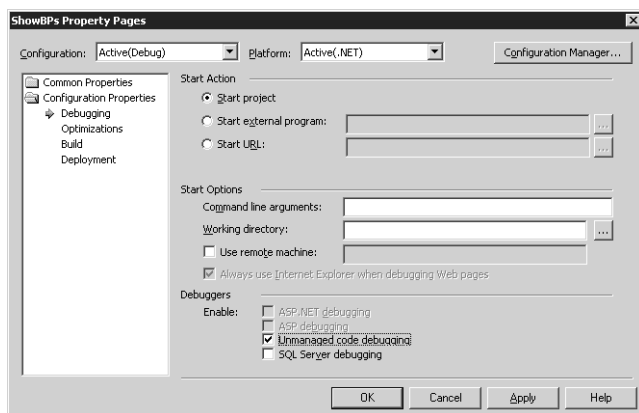


Рис. 6-6. Включение отладки в смешанном режиме для проекта Visual Basic .NET

Самый большой недостаток отладки в смешанном режиме в том, что иногда она может быть очень медленной. Управляемый и неуправляемый код лучше всего отлаживать по отдельности всегда, когда это возможно. Однако, если отладки в смешанном режиме избежать не удастся, сначала нужно отключить вычисление свойств (рис. 6-7), потому что замедление связано главным образом с вычислением значений, выводимых в окна Watch. После отключения вычисления свойств

отладка в смешанном режиме все равно будет медленней, чем отладка только управляемого или неуправляемого кода, но все же ускорится. Большое различие между отладкой управляемого кода и отладкой в смешанном режиме состоит в том, что при отладке в смешанном режиме вы можете увидеть все модули, загружаемые управляемым процессом, если щелкните правой кнопкой в окне Modules (модули) и выберете пункт Show Modules For All Programs (показывать модули для всех программ).

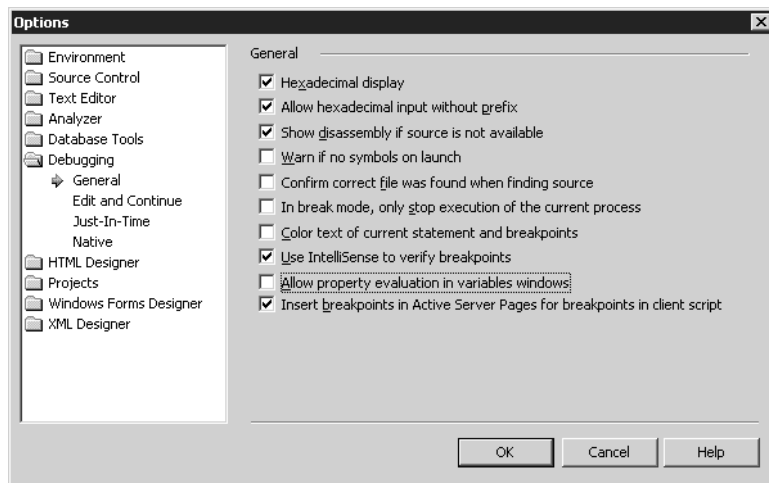


Рис. 6-7. Отключение вычисления свойств

Отладка в смешанном режиме имеет и менее существенные недостатки. Во-первых, хотя вы и отлаживаете процесс в неуправляемом режиме, вы не можете устанавливать точки прерывания по данным (см. главу 7). Во-вторых, вы не можете создавать минидампы процесса, если отладчик не находится в неуправляемой функции.

Удаленная отладка

Microsoft вполне заслужила Нобелевскую премию мира за то, что помогла достичь согласия двум из самых непримиримых групп в истории: разработчикам и сетевым администраторам. До управляемых приложений ситуация была такова, что разработчики должны были входить в группу Administrators на любом компьютере, который собирались использовать для отладки приложений. Однако, если программистам нужно было выполнять отладку на главном сервере (production server), сетевые администраторы очень неохотно шли на это, так как боялись изменения конфигурации системы, что, например, могло потребовать изменения паролей всех пользователей и применения паролей, состоящих из 65 символов. Со мной этого не случилось, но я слышал, что другие умудрялись сделать и такое.

Процесс, который всех примирил, называется удаленной отладкой. Я не буду излагать здесь всю соответствующую документацию, а только проясню некоторые вопросы, связанные с настройкой и применением отладчиков для работы в удаленном режиме. Прежде всего я хочу сказать, что для удаленной отладки не надо

устанавливать на удаленном компьютере Visual Studio .NET полностью; для этого нужны только компоненты удаленной отладки из состава Visual Studio .NET. Установить их можно, щелкнув ссылку «Remote Components Setup» (установка удаленных компонентов) в нижней части окна приложения Visual Studio .NET Setup (рис. 6-8). Следуйте указаниям в появившемся окне под пунктом Full Remote Debugging Support (полная поддержка удаленной отладки) и установите .NET Framework до нажатия на кнопку Install Full (установить все). Забыть про установку .NET Framework SDK очень просто, и, если такое произойдет, вы будете долго чесать голову, удивляясь невозможности отладки или запуска любых управляемых приложений. Кроме того, вам следует знать, что, хотя для удаленной отладки вовсе не требуется входить в группу Administrators, для установки компонентов удаленной отладки быть членом этой группы все-таки нужно.

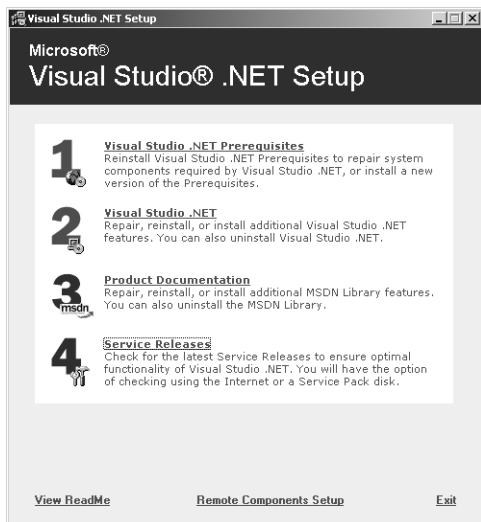


Рис. 6-8. Установка только компонентов удаленной отладки Visual Studio .NET

Ключ к удаленной отладке — добавление вашей учетной записи в группу Debugger Users (Пользователи отладчика) на удаленном компьютере. Конечно, чтобы иметь право добавлять пользователей в эту группу, нужно быть на соответствующем компьютере членом группы Administrators. Наиболее частая проблема при настройке удаленной отладки заключается в том, что разработчики забывают добавить на удаленном компьютере учетную запись в группу Debugger Users.

Стандартный вопрос отладки

При отладке Web-сервисов XML я получаю исключения, утверждающие, что время операции истекло. Что делать?

При отладке Web-сервисов XML эта ошибка встречается очень часто. Укажите бесконечный лимит времени, присвоив в своем коде или в окне Watch значение -1 свойству Timeout объекта Web-сервиса XML.

ILDASM и промежуточный язык Microsoft

Несколько лет назад, начав изучать .NET, я написал классическую программу «Hello World!» и сразу захотел узнать все детали ее работы. Я испытал настоящий шок, когда понял, что .NET — по сути абсолютно новая среда разработки! При изучении новой среды мне нравится опускаться до простейших операций и постепенно подниматься вверх, узнавая детали совместной работы всех элементов.

Так, когда я переходил с MS-DOS на Microsoft Windows 3.0 (ого, неужели я так стар?) и мне было что-то непонятно, я всегда обращался к выполняемому процессором языка ассемблера. Ассемблер (который также иногда называют недвусмысленным языком) хорош тем, что он никогда не лжет. Я спокойно продолжал пользоваться этим методом, пока не начал переход на .NET, и мой мир перевернулся. Я потерял свою ассемблерную опору! Я мог видеть в отладчиках ассемблер Intel, но это почти не помогало. Я видел много ассемблерных команд, вызываемых путем выделенных адресов и других сложных методик, но однозначного отображения, присутствовавшего при разработке программ Win32 на Microsoft C/C++, более не было.

Однако сразу же после написания «Hello World!» я обнаружил в .NET одну наиболее полезную вещь: дизассемблер промежуточного языка Microsoft (ILDASM). Вооружившись им, я наконец почувствовал, что могу приступить к изучению .NET по одному элементу за раз. ILDASM позволяет увидеть язык псевдоассемблера для .NET, называемый промежуточным языком Microsoft (MSIL). Возможно, вы никогда ничего не напишете на MSIL, но в знании ассемблера среды разработки как раз и заключается разница между использованием среды и ее пониманием. Кроме того, хотя документация к .NET и великолепна, нет лучшего способа обучения, чем видеть реализацию приложения.

Прежде чем мы перейдем к ILDASM и MSIL, я хочу пролить свет на один распространенный вопрос о .NET, который часто вызывает путаницу. Мне часто говорили, что .NET нельзя воспринимать всерьез, поскольку написанные для .NET программы очень легко поддаются восстановлению алгоритма (reverse engineering), или декомпиляции, не обеспечивая защиту интеллектуальной собственности. Это абсолютно верно. В обмен на достоинства по-настоящему распределенных объектов, сбора мусора и простоты разработки двоичные файлы .NET должны быть самоописательными. Однако приведенный аргумент ошибочен.

Такие же жалобы высказывались, когда на сцене появился язык Java. Многие поражались тому, что его так просто декомпилировать. Одна сторонняя компания выпустила просто великолепный декомпилятор двоичных файлов Java. Я разговаривал с некоторыми клиентами этой компании, которым результаты декомпиляции нравились настолько, что они компилировали код своих коллег и тут же декомпилировали его, потому что в результате получался улучшенный и более понятный код! В самом начале все эти люди также очень беспокоились по поводу интеллектуальной собственности, однако это беспокойство несколько не замедлило признание Java корпоративными разработчиками.

Программисты, создающие Web-приложения или Web-сервисы XML, могут не беспокоиться о восстановлении алгоритма, потому что клиенты и пользователи не имеют физического доступа к двоичным файлам. Однако многие из вас разрабатывают приложения Windows Forms или консольные приложения, и в этом случае волнение вполне резонно. С Visual Studio .NET 2003 корпорация Microsoft начала

обеспечивать разработчиков «облегченной версией» (community edition) обфускатора Dotfuscator от компании PreEmptive Solutions. Как мне кажется, все, что делает эта версия, заключается в простом переименовании классов и методов, однако и этого вполне хватает. Будьте готовы к тому, чтобы потратить на освоение Dotfuscator некоторое время, потому что его графический пользовательский интерфейс требует основательного исследования.

Для тех, кого волнует интеллектуальная собственность приложений Windows Forms или консольных приложений, Уайз Аул¹ [также известный как Брент Ректор (Brent Rector), мой коллега по работе в Wintellect] написал отличный обфускатор для .NET — Demeanor. Demeanor обеспечивает полную защиту интеллектуальной собственности. Этот инструмент избавит вас от волнений, связанных с внедрением приложений .NET в тех случаях, когда кто-то имеет физический доступ к двоичным файлам. Подробнее о Demeanor для .NET см. по адресу: <http://www.wiseowl.com>.

Если вы собираетесь выполнить обфускацию своего кода для .NET, убедитесь, что он полностью отлажен и протестирован. На данный момент нет способа сопоставить код, выданный обфускатором, с исходными файлами, поэтому отлаживать его можно только на уровне ассемблера x86.

Начинаем работу с ILDASM

ILDASM расположен в подкаталоге <установочный каталог Visual Studio .NET >\SDK\v1.1\Bin, который по умолчанию может быть не указан в переменной среды PATH. Выполнив файл VSVARS.BAT, находящийся в подкаталоге <установочный каталог Visual Studio .NET >\Common7\Tools, вы зададите соответствующие значения всем нужным переменным среды .NET и сможете получать доступ к любым утилитам .NET из командной строки. Когда вы запустите ILDASM и выберете файл для дизассемблирования, его окно будет выглядеть, как на рис. 6-9. То, что вы видите, — это расширения метаданных для модуля. На рис. 6-9 показаны все воз-

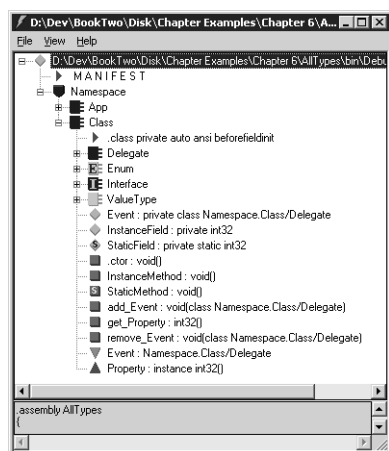














Рис. 6-9. Основное окно ILDASM

¹ Wise Owl — мудрый сын (англ.). — Прим. перев.

возможные значки для типов, однако рисунок ничего не говорит о том, для чего эти значки нужны и каковы их текстовые значения при сохранении дерева в файл. Чтобы вам было легче понять это, я создал табл. 6-2.

Табл. 6-2. Описание элементов дерева ILDASM

Значок	Текстовый вывод	Описание
	[MOD] — заголовок модуля	Информационные директивы, объявления классов и данные из декларации
	[NSP]	Пространство имен
	[CLS]	Класс
	[INT]	Интерфейс
	[ENU]	Перечисление
	[VCL]	Размерный класс
	[MET]	Экземплярный метод (закрытый, открытый или защищенный)
	[STM]	Статический метод
	[FLD]	Экземплярное поле (закрытое, открытое или защищенное); также сборка
	[STF]	Статическое поле
	[EVT]	Событие
	[PTY]	Свойство (получение и/или задание свойства)

Если вам нужна более подробная информация и статистика об открываемых файлах, запустите ILDASM с ключом командной строки `/ADV`. Это включит вывод расширенной информации и добавит в меню View три новых пункта.

- **COR Header** (заголовок COR) позволяет просмотреть информацию, содержащуюся в заголовке файла.
- **Statistics** (статистика) выводит различную статистику о процентном соотношении и категориях всех метаданных в системе.
- **MetaInfo** (метаинформация) содержит подменю, позволяющее выбрать, какую именно информацию вы желаете получить. Чтобы увидеть эту информацию, выберите в подменю пункт **Show!** (показать) (или нажмите `Ctrl+M`), и она появится в отдельном окне MetaInfo. Если не выбрать конкретной информации, при выборе пункта **Show!** будет выведен просто дамп метаданных.

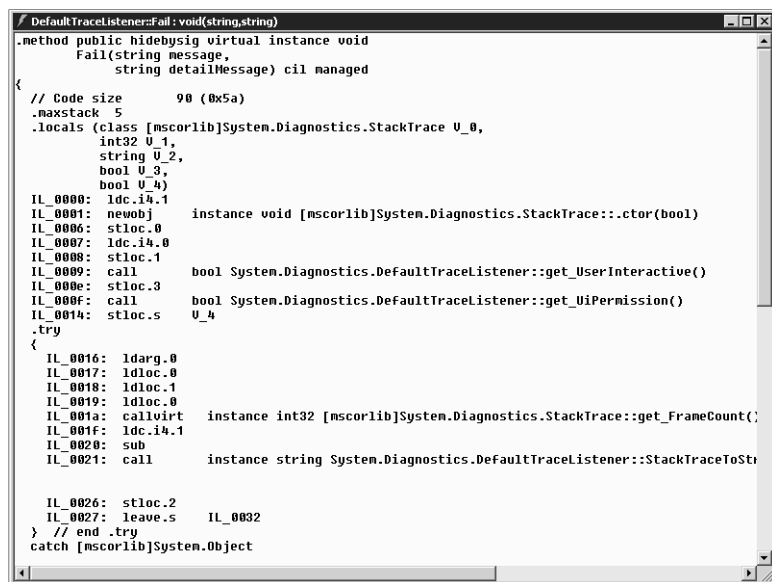
Если для конкретного модуля у вас есть исходный код, вам непременно захочется отметить в меню View пункт **Show Source Lines** (показать исходный код). Для показа исходного кода можно также задать в командной строке ключ `/SOURCE`. В результате этого дизассемблер будет выводить исходный код программы в виде

комментариев над соответствующими инструкциями MSIL. Получить сведения обо всех ключах можно, указав в командной строке ключ `/?`. Для запуска ILDASM со всеми нужными мне ключами я использую командный файл:

```
ildasm /adv /source %1
```

Чтобы увидеть код MSIL, соответствующий конкретному элементу, нужно дважды щелкнуть его значок. В появившемся окне в зависимости от выбранного элемента вы увидите его дизассемблированный код, информацию об объявлениях или общую информацию. Если окно выглядит так, как на рис. 6-10, мы можем приступить к изучению MSIL. Одно из достоинств ILDASM в том, что он поддерживает функциональность буксировки, благодаря чему вы сможете легко переключаться между модулями.

Наконец я хочу рассказать вам, что сделать, чтобы при работе с ILDASM вы видели код MSIL и свой код C#, J#, Managed Extensions for C++ или Visual Basic .NET одновременно. Если при отладке управляемого приложения вы наблюдали в отладчике за окном Disassembly, вы, вероятно, видели код на одном из языков .NET и только ассемблерные команды Intel. Причина в том, что MSIL компилируется по требованию (just-in-time, JIT), так что на самом деле выполняется только машинный язык ассемблера, а не MSIL. Что делает ILDASM действительно интересным, так это то, что по крайней мере в одном аспекте он достиг совершенства: это по-настоящему «обратимый» (round-trip) дизассемблер!



```

DefaultTraceListener.Fail: void(string,string)
.method public hidebysig virtual instance void
  Fail(string message,
    string detailMessage) cil managed
{
  // Code size          90 (0x5a)
  .maxstack 5
  .locals (class [mscorlib]System.Diagnostics.StackTrace U_0,
    int32 U_1,
    string U_2,
    bool U_3,
    bool U_4)
  IL_0000: ldc.i4.1
  IL_0001: newobj          instance void [mscorlib]System.Diagnostics.StackTrace::.ctor(bool)
  IL_0006: stloc.0
  IL_0007: ldc.i4.0
  IL_0008: stloc.1
  IL_0009: call             bool System.Diagnostics.DefaultTraceListener::get_UserInteractive()
  IL_000e: stloc.3
  IL_000f: call             bool System.Diagnostics.DefaultTraceListener::get_UiPermission()
  IL_0014: stloc.s        U_4
  .try
  {
    IL_0016: ldarg.0
    IL_0017: ldloc.0
    IL_0018: ldloc.1
    IL_0019: ldloc.0
    IL_001a: callvirt        instance int32 [mscorlib]System.Diagnostics.StackTrace::get_FrameCount()
    IL_001f: ldc.i4.1
    IL_0020: sub
    IL_0021: call             instance string System.Diagnostics.DefaultTraceListener::StackTraceToString(int32)
    IL_0026: stloc.2
    IL_0027: leave.s         IL_0032
  } // end .try
  catch [mscorlib]System.Object

```

Рис. 6-10. MSIL для метода

Обратимый дизассемблер позволяет дизассемблировать двоичный файл и тут же ассемблировать его, собрав приложение заново. Так как в состав .NET входит ILASM, ассемблер промежуточного языка Microsoft (Microsoft Intermediate Language Assembler), то у вас есть все, чтобы видеть MSIL и код своих программ C#/J#/Managed Extensions for C++/Visual Basic .NET одновременно. Это позволяет лучше узнать

тонкости работы программы. Дизассемблируйте файл при помощи ILDASM, задав ключи командной строки /SOURCE и /OUT= и указав имя файла вывода с расширением .IL. Скомпилируйте файл при помощи ILASM с ключом /DEBUG. Теперь вы сможете просматривать MSIL в отладчике Visual Studio .NET и видеть соответствующий код C#/J#/Managed Extensions for C++/Visual Basic .NET как комментарии. Открыв окно Disassembly, вы увидите, как высокоуровневый язык компилируется в MSIL и как инструкции MSIL компилируются по требованию в язык ассемблера процессоров Intel. Листинг 6-2 представляет собой дизассемблированный метод программы ShowBPs (которую можно найти на CD, прилагаемом к книге) вместе с соответствующим исходным кодом, указанным в комментариях.

Листинг 6-2. Объединенный вывод исходного кода и кода MSIL

```
.method private instance void
    btnConditionalBreaks_Click(object sender,
                                class [mscorlib]System.EventArgs e)
    cil managed
{
    // Code size      139 (0x8b)
    .maxstack 4
    .locals init ([0] int32 i,
                  [1] int32 j,
                  [2] string[] _Vb_t_array_0,
                  [3] class [System.Windows.Forms]
                        System.Windows.Forms.TextBox _Vb_t_ref_0)

//000120:
//000121: Private Sub btnConditionalBreaks_Click _
//                                ( ByVal sender As System.Object, _
//                                ByVal e As System.EventArgs) _
//                                Handles btnConditionalBreaks.Click
    IL_0000: nop
//000122:      Dim i As Integer = 0
    IL_0001: ldc.i4.0
    IL_0002: stloc.0
//000123:      Dim j As Integer = 0
    IL_0003: ldc.i4.0
    IL_0004: stloc.1
//000124:
//000125: ' Очистка поля вывода.
//000126: edtOutput.Clear()
    IL_0005: ldarg.0
    IL_0006: callvirt instance class
                [System.Windows.Forms]System.Windows.Forms.TextBox
                ShowBPs.ShowBPsForm::get_edtOutput()
    IL_000b: callvirt instance void
                [System.Windows.Forms]System.Windows.Forms.TextBoxBase::Clear()
    IL_0010: nop
//000127:
//000128: ' Я записал оба цикла в одной строке, чтобы показать,
//                                как устанавливать точки прерываний для части строки.
```

```

//000129: For i = 1 To 5 : For j = 1 To 5
IL_0011: ldc.i4.1
IL_0012: stloc.0
IL_0013: ldc.i4.1
IL_0014: stloc.1
//000130: ' Вывод текста
//000131: edtOutput.Text += "i = " + i.ToString() + " j = " + _
//          j.ToString() + vbCrLf
IL_0015: ldarg.0
IL_0016: callvirt instance class
           [System.Windows.Forms]System.Windows.Forms.TextBox
           ShowBPs.ShowBPsForm::get_edtOutput()
IL_001b: stloc.3
IL_001c: ldloc.3
IL_001d: ldc.i4.6
IL_001e: newarr [mscorlib]System.String
IL_0023: stloc.2
IL_0024: ldloc.2
IL_0025: ldc.i4.0
IL_0026: ldloc.3
IL_0027: callvirt instance string
           [System.Windows.Forms]System.Windows.Forms.TextBox::get_Text()
IL_002c: stelem.ref
IL_002d: nop
IL_002e: ldloc.2
IL_002f: ldc.i4.1
IL_0030: ldstr "i = "
IL_0035: stelem.ref
IL_0036: nop
IL_0037: ldloc.2
IL_0038: ldc.i4.2
IL_0039: ldloc.s i
IL_003b: call instance string [mscorlib]System.Int32::ToString()
IL_0040: stelem.ref
IL_0041: nop
IL_0042: ldloc.2
IL_0043: ldc.i4.3
IL_0044: ldstr " j = "
IL_0049: stelem.ref
IL_004a: nop
IL_004b: ldloc.2
IL_004c: ldc.i4.4
IL_004d: ldloc.s j
IL_004f: call instance string [mscorlib]System.Int32::ToString()
IL_0054: stelem.ref
IL_0055: nop
IL_0056: ldloc.2
IL_0057: ldc.i4.5
IL_0058: ldstr "\r\n"
IL_005d: stelem.ref

```

см. след. стр.

```

IL_005e:  nop
IL_005f:  ldloc.2
IL_0060:  call      string [mscorlib]System.String::Concat(string[])
IL_0065:  callvirt  instance void
           [System.Windows.Forms]System.Windows.Forms.TextBox::set_Text(string)
IL_006a:  nop
//000132:  ' Обновление текста в поле вывода.
//000133:  edtOutput.Update()
IL_006b:  ldarg.0
IL_006c:  callvirt  instance class
           [System.Windows.Forms]System.Windows.Forms.TextBox
           ShowBPs.ShowBPsForm::get_edtOutput()
IL_0071:  callvirt  instance void
           [System.Windows.Forms]System.Windows.Forms.Control::Update()
IL_0076:  nop
//000134:  Next j
IL_0077:  nop
IL_0078:  ldloc.1
IL_0079:  ldc.i4.1
IL_007a:  add.ovf
IL_007b:  stloc.1
IL_007c:  ldloc.1
IL_007d:  ldc.i4.5
IL_007e:  ble.s     IL_0015

//000135:  Next i
IL_0080:  nop
IL_0081:  ldloc.0
IL_0082:  ldc.i4.1
IL_0083:  add.ovf
IL_0084:  stloc.0
IL_0085:  ldloc.0
IL_0086:  ldc.i4.5
IL_0087:  ble.s     IL_0013

//000136:      End Sub
IL_0089:  nop
IL_008a:  ret
} // end of method ShowBPsForm::btnConditionalBreaks_Click

```

Основы CLR

Прежде чем вы начнете продираться сквозь лес команд MSIL, я должен вкратце пояснить работу CLR. CLR по сути является процессором для команд MSIL. В то время как традиционные процессоры для выполнения всех команд используют регистры и стеки, CLR использует только стек. Это значит, что при сложении чисел CLR должна загрузить оба числа в стек и вызвать команду их сложения. Команда сложения удаляет два числа из стека и помещает результат в его вершину. Если вы по складу характера похожи на меня, возможно, вам захочется изучить дей-

ствительную реализацию CLR. Для знакомства с системой, похожей на CLR и в то же время достаточно небольшой, чтобы ее можно было понять, обратитесь к книге Брайана Кернигана и Роба Пайка (Brian Kernighan and Rob Pike) «The Unix Programming Environment» (Prentice Hall, 1984). В этой книге они разрабатывают калькулятор высшего порядка (higher order calculator, hoc) — нетривиальный, написанный на C пример системы, основанной на стеке. Если вы хотите увидеть реальную реализацию CLR, загрузите из Интернета общезыковую инфраструктуру с открытым кодом (Shared Source Common Language Infrastructure) — также известную как «Rotor», — кросс-платформенную CLR, основанную на стандартах Европейской ассоциации производителей компьютеров (European Computer Manufacturers' Association). Она включает огромный объем кода, но вы сможете узнать, как все работает. Загрузить Shared Source CLI можно по адресу: <http://msdn.microsoft.com/netframework>.

В ячейках стека CLR может храниться любой тип значения. Копирование значений из памяти в стек называется загрузкой (loading), а копирование значений стека в память — сохранением (storing). В отличие от стека процессоров Intel стек CLR не хранит локальных переменных — они хранятся в памяти. Стеки локальны по отношению к работающим методам, и CLR сохраняет их между вызовами методов. Наконец, в стек также помещаются возвращаемые методами значения. Теперь, когда я описал вкратце работу CLR, перейдем прямо к командам.

MSIL, локальные переменные и параметры

Думаю, что прежде чем перейти к более сложным вопросам, нам стоит рассмотреть простейшую программу, написанную на MSIL, т. е. «Hello World!». Так вы сможете увидеть программу MSIL в действии, и я смогу начать объяснение элементов, которые будут встречаться в информации, выводимой ILDASM. Полная программа «Hello World!» приведена в листинге 6-3; вы найдете ее на CD, прилагаемом к книге, под именем HelloWorld.IL. Даже если вы впервые видите MSIL, вы легко поймете, что происходит. Все слова, начинающиеся с точки, являются директивами ассемблеру ILASM.EXE, а для комментариев служат стандартные двойные слэши C#.

Листинг 6-3. HelloWorld.IL

```
// Для запуска программы нужна директива .assembly.
.assembly hello {}

// Объявление main в стиле "C".
.method static public void main() il managed
{
    // Эта директива указывает ядру исполняющей подсистемы, откуда
    // начинается выполнение программы. Нужна одна на программу.
    // Кроме того, она может использоваться и в случае методов.
    .entrypoint

    // ILASM этого не требует, но ILDASM всегда показывает
    // эту директиву, поэтому я включил ее в программу.
```

см. след. стр.

```
.maxstack 1

// Помещаем строку в стек.
ldstr "Hello World from IL!"

// Вызываем метод WriteLine класса System.Console.
call void [mscorlib]System.Console::WriteLine(class System.String)

// Возвращаемся в вызывающую программу. Если про это забыть, файл
// скомпилируется, но ядро исполняющей подсистемы сгенерирует исключение.
ret

}
```

Важными элементами HelloWorld.IL являются три последние строки. Команда `ldstr` помещает строку в стек. Помещение элементов в стек называется загрузкой (loading), поэтому, когда вам будут встречаться команды, начинающиеся с букв «ld», вы сможете сразу догадаться, что они берут значения из памяти и помещают их в стек. Извлечение элементов из стека и размещение их в памяти в программе «Hello World!» не использовалось, однако я все равно скажу, что этот процесс называется сохранением (storing) и все команды такого типа начинаются с букв «st». Вооружившись знанием этих двух фактов и помощью, предоставляемой ILDASM путем помещения жестко закодированных строк в дизассемблированный листинг, вы можете всерьез заняться восстановлением алгоритма.

Теперь, когда я вкратце рассказал об ассемблере MSIL, самое время узнать, какая же информация, предоставляемая ILDASM, поможет вам разобраться в тонкостях работы программы.

Узнать параметры и возвращаемые типы при помощи ILDASM проще простого: дизассемблер указывает их, когда вы дважды щелкаете имя метода. На самом деле все еще лучше, потому что дизассемблер показывает действительные имена параметров. Значения классов изображаются в формате `[module]namespace.class`. Базовые типы, такие как `int`, `char` и т. д., изображаются в виде типа специфического класса. Например, `int` имеет вид `Int32`.

Локальные переменные также очень легко определяются. Если у вас есть символы отладки, то будут указаны действительные имена переменных. Однако дизассемблированные системные классы будут выглядеть примерно так:

```
.locals (class [mscorlib]Microsoft.Win32.RegistryKey V_0,
        class System.Object V_1,
        int32 V_2,
        int32 V_3)
```

После директивы `.locals` в круглых скобках указывается полный список параметров, разделенных запятыми. Вслед за типом располагается поле вида `V_#`, где `#` — номер каждого параметра. Как вы увидите, номер используется в довольно многих командах. В предыдущем фрагменте `[mscorlib]` указывает на конкретную DLL, к которой относится данный класс.

Важные команды

Вместо того чтобы приводить огромную таблицу команд, я расскажу про наиболее важные и приведу примеры их применения. Я начну с команд загрузки и объясню все их варианты. Рассматривая другие типы команд, я не буду касаться тех их аспектов, которые не отличаются от команд загрузки, и просто продемонстрирую их использование. Назначение команд, которые я не описываю, просто понять по их названиям.

ldc Загрузка численной константы

Помещает в стек жестко закодированное число. Она имеет формат `ldc.size[.num]`, где `size` — размер значения в байтах, а `num` — специальный короткий код для 4-байтных целых чисел, находящихся в пределах от -128 до 127 (если `size` имеет значение `i4`). Поле `size` может иметь значения `i4` (4-байтовое целое число), `i8` (8-байтовое целое число), `r4` (4-байтовое число с плавающей точкой) или `r8` (8-байтовое число с плавающей точкой). Эта команда имеет много форм, что призвано уменьшить число кодов операций.

```
ldc.i4.0           // Загрузка в стек 0 с использованием
                   // специальной формы.
ldc.r8 2.1000000000000001 // Загрузка в стек 2.1000000000000001.
ldc.i4.m1          // Загрузка в стек -1. Это специальная
                   // форма команды.
ldc.i4.s -9        // Загрузка в стек -9 с использованием
                   // короткой формы.
```

ldarg Загрузка аргумента

ldarga Загрузка адреса аргумента

Номера аргументов начинаются с 0. Для экземплярных методов аргументом 0 является указатель `this`, а первый аргумент имеет номер 1, а не 0.

```
ldarg.2           // Загрузка в стек аргумента 2. При использовании
                   // этой формы максимальным номером является 3.
ldarg.s 6         // Загрузка в стек аргумента 6. Эта форма применяется
                   // для всех номеров аргументов больше 4 (включительно).
ldarga.s newSample // Загрузка адреса newSample
```

ldloc Загрузка локальной переменной

ldloca Загрузка адреса локальной переменной

Эти команды загружают в стек указанную локальную переменную. Все локальные переменные определяются порядком, в котором они указаны в объявлении `locals`. Команда `ldloca` загружает адрес локальной переменной.

```
ldloc.0           // Загрузка в стек локальной переменной с индексом 0. При
                   // использовании этой формы максимальным индексом является 3.
ldloc.s V_6       // Загрузка в стек локальной переменной с индексом 6.
                   // Эта форма используется для всех переменных
                   // с индексом больше или равным 4.
ldloca.s V_5      // Загрузка в стек адреса локальной переменной с индексом 5.
```


stelem	Сохранение элемента массива
--------	-----------------------------

Если три предыдущих команды помещают в стек одномерный массив, начинающийся с нулевого индекса, индекс и значение (в указанном порядке), `stelem` приводит размерный тип к типу соответствующего массива и помещает значение в массив. Команда `stelem` удаляет все три элемента из стека. Как и при команде `ldelem`, преобразование определяется полем типа. Чаще всего используется `stelem.ref`, выполняющая преобразование размерного типа в объект.

```

.method public hidebysig specialname
instance void  set_MachineName(class System.String 'value') il managed
{
    .maxstack 4
    .locals (class System.String[] V_0)
    :
    ldloc.0                // Загрузка массива в стек.
    ldc.i4.1               // Загрузка индекса: константы 1.
    ldarg.1                // Загрузка аргумента: строки.
    stelem.ref              // Сохранение элемента.
    :
}

```

stfld	Сохранение значения в поле объекта
-------	------------------------------------

Берет значение из вершины стека и помещает его в поле объекта. Как и при загрузке поля, указывается полная ссылка.

```
stfld    int32[] System.Diagnostics.CategoryEntry::HelpIndexes
```

seq Сравнение на равенство

Сравнивает два верхних значения, находящихся в стеке. Оба элемента удаляются из стека, и если их значения равны, в стек заносится 1; в противном случае в стек заносится 0.

```
ldloc.1           // Загрузка локальной переменной с индексом 1.
ldc.i4.0          // Загрузка константы 0.
seq               // Сравнение элементов на равенство.
```

cgt Сравнение по условию «больше чем»

Также сравнивает два верхних значения стека. Оба элемента удаляются из стека, и, если значение, которое было занесено в стек первым, больше второго, в стек заносится 1, в противном случае — 0. Команда `cgt` может также иметь модификатор `.un`, указывающий, что сравнение является беззнаковым или неупорядоченным.

```
// Получение числа элементов в наборе.
call instance int32 System.Diagnostics.
CounterCreationDataCollection::get_Count()
ldc.i4.0 // Загрузка константы 0.
cgt // Число элементов (count) больше 0?
```

clt Сравнение по условию «меньше чем»

Идентична `cgt` за исключением того, что 1 заносится в стек, если первое значение меньше второго.

```
// Получение уровня трассировки.
call instance value class System.Diagnostics.TraceLevel
    System.Diagnostics.TraceSwitch::get_Level()
ldc.i4.1          // Загрузка константы 1.
clt              // Уровень трассировки меньше 1?
```

br Безусловный переход

Эта команда MSIL аналогична оператору `goto`.

```
br.s IL_008d      // Переход к смещению в методе.
```

brfalse Переход, если ложь
brtrue Переход, если истина

Обе команды проверяют значение, находящееся в вершине стека, и выполняют переход, если выполняется соответствующее условие. Команда `brtrue` совершает переход, только когда это значение равняется 1, а `brfalse` — только когда оно равно 0. Обе удаляют значение из вершины стека.

```
ldloc.1          // Загрузить локальную переменную с индексом 1.
brfalse.s IL_006a // Переход, если 0.
ldloc.2          // Загрузка локальной переменной с индексом 2.
brtrue.s IL_006c  // Переход, если 1.
```

beq Переход, если равно
bgt Переход, если больше или равно
ble Переход, если меньше или равно
blt Переход, если меньше
bne Переход, если не равно

Все эти команды берут два значения, находящиеся в вершине стека, и сравнивают самое верхнее значение со вторым по счету. Во всех случаях сначала производится сравнение, после чего следует булев переход. Так, команда `bgt` эквивалентна последовательности команд `cgt` и `brtrue`.

conv Преобразование типа данных

Преобразует значение, находящееся в вершине стека, к новому типу и помещает в вершину преобразованное значение. Тип, к которому преобразуется значение, указывается после команды `conv`. Например, `conv.u4` выполняет преобразование значения к 4-байтовому беззнаковому целому. Если после команды `conv` указан только тип, она не генерирует исключений при всех видах переполнения. Если между `conv` и типом указано поле `.ovf` (например, `conv.ovf.u8`), переполнение приводит к генерированию исключения.

```
ldloc.0          // Загрузка локальной переменной с индексом 0 (массив).
ldlen           // Получение длины массива.
```

```
conv.i4          // Преобразование длины массива в
                 // 4-байтовое значение.
```

newarr Создание одномерного массива, начинающегося с нулевого индекса

Создает новый массив указанного типа с числом элементов, указанным в вершине стека. Число элементов удаляется из стека, и в вершину стека помещается новый массив.

```
ldc.i4.5         // Число элементов создаваемого
                 // массива будет равно 5.
                 // Создание нового массива.
newarr System.ComponentModel.MemberAttribute
```

newobj Создание нового объекта

Создает новый объект и вызывает его конструктор. Перед этим все аргументы конструктора заносятся в стек. Если создание объекта происходит успешно, аргументы удаляются из стека, и в стек заносится ссылка на объект.

```
.method public hidebysig specialname rtspecialname
    instance void .ctor(class [mscorlib]System.IO.Stream 'stream',
                        class System.String name) il managed
{
    :
    ldarg.1         // Загрузка аргумента с индексом 1 (поток).
                   // Создание нового класса.
    newobj instance void [mscorlib]
        System.IO.StreamWriter::.ctor(class
                                        [mscorlib]System.IO.Stream)
```

box Преобразование размерного типа в объектный

Преобразует значение в объект, оставляя после этого объект в стеке. Именно она выполняет упаковку (boxing). При передаче параметров вы будете часто видеть следующий код:

Заметьте: этому методу передается размерный тип INT32.

```
.method public hidebysig specialname
    instance void set_Indent(int32 'value') il managed
{
    :
    ldstr      "Indent"          // Занесение имени метода.
    ldarga.s   'value'          // Загрузка адреса аргумента
                                // первого параметра.
    box        [mscorlib]System.Int32 // Преобразование адреса в объект.
                                // Загрузка сообщения.
    ldstr      "The Indent property must be non-negative."
                                // Создание нового объекта класса ArgumentOutOfRangeException
    newobj     instance void [mscorlib]System.ArgumentOutOfRangeException::
    .ctor(class System.String,
            class System.Object,
            class System.String)
```

unbox Преобразование упакованного размерного типа в обычную форму

Возвращает управляемую ссылку на размерный тип в упакованной форме. Возвращаемая ссылка — не копия, а действительный объект. В откомпилированном коде C# и Visual Basic .NET за командой `unbox` следует команда `ldind` (косвенная загрузка значения в стек) или `ldobj` (копирование размерного типа в стек).

```
// Преобразование значения в System.Reflection.Emit.LocalToken
unbox System.Reflection.Emit.LocalToken
// Затапливание значения в стек
ldobj System.Reflection.Emit.LocalToken
unbox [mscorlib]System.Int16           // Преобразование значения в объект Int16.
ldind.i2                               // Занесение значения объекта в стек.
```

call Вызов метода

callvirt Вызов метода, ассоциированного с объектом в период выполнения

Команда `call` вызывает статические и неvirtуальные нормальные методы. Для вызова виртуальных методов и методов интерфейсов служит команда `callvirt`. Аргументы размещаются в порядке слева направо. Этот порядок противоположен большинству соглашений вызова в мире IA32. Вот пример использования `callvirt`:

```
// Загрузка параметра.
ldfld class System.String
    System.CodeDOM.Compiler.CompilerResults::pathToAssembly
// Вызов виртуального метода set_CodeBase.
callvirt instance void [mscorlib]
System.Reflection.AssemblyName::set_CodeBase
(class System.String)
:
ldarg.0                // Загрузка указателя this, который
                        // всегда является первым параметром.
ldarg.1                // Загрузка первого аргумента.
ldnull                 // Загрузка значения null.
                        // Вызов виртуальной функции.
callvirt instance void
    System.Diagnostics.TraceListener::Fail(class System.String
                                           class System.String)
ret                    // Возвращение в исходную программу.
```

Другие инструменты восстановления алгоритма

ILDASM — великолепный инструмент, но я хочу упомянуть еще два средства, которые считаю просто бесценными. Обе программы имеют «правильную» цену: они бесплатны! Первый инструмент — .NET Reflector (<http://www.aisto.com/roeder/dotnet/>) Лутца Родера (Lutz Roeder), поддерживающий все функции ILDASM и многие другие. Одна из важнейших функций .NET Reflector заключается в том, что он позволяет легко искать в сборке типы. Конечно, хотелось бы надеяться, что все разработчики будут правильно документировать генерируемые исключения, но

это не всегда так. Выберите в .NET Reflector меню Type Search (поиск типов)¹ и введите в поле Type Search слово «except». В результате этого будут показаны все типы, в имя которых входит слово «exception» (исключение).

Иногда очень важно быстро узнать, какие методы вызывает конкретный метод. Выделите в дереве, изображаемом .NET Reflector, интересующий вас метод и выберите в меню View подменю Call Tree (дерево вызовов). Развернув в окне Call Tree дочерние вызовы, вы увидите иерархию вызовов для конкретного метода. Это прекрасный способ изучения совместной работы отдельных элементов программы.

Наконец, .NET Reflector поддерживает более развитые по сравнению с ILDASM возможности вывода дизассемблированного кода. Выберите метод, про который хотите узнать, нажмите Enter и увидите окно Disassembler. Если вас интересует, что делает команда, наведите на нее курсор, и появится подсказка с ее объяснением. Типы параметров и локальных значений, а также методы, вызываемые командами call, подчеркиваются. Просто щелкните интересующий вас элемент, и в основном окне .NET Reflector будет выведен тип или метод, чтобы вы могли получить о нем более подробную информацию.

Вторая программа, которую я хочу упомянуть, называется Anakrino, что на греческом означает «исследовать» или «судить». Anakrino — это декомпилятор для .NET, который показывает для сборки код C# или Managed Extensions for C++. Эту программу, которую написал Джей Фримен (Jay Freeman), можно загрузить по адресу <http://www.saurik.com/net/exemplar/>. В отличие от .NET Reflector исходный код Anakrino доступен. Нельзя сказать, что Anakrino не имеет недостатков, однако это тоже великолепный инструмент для изучения кода .NET Framework. Работать с Anakrino очень просто, поэтому я не буду про это рассказывать. Однако хочу предупредить: исходный код Anakrino довольно «оригинален» и включает массу шаблонов, поэтому если вы захотите улучшить его, вам придется приложить немало усилий. На момент написания книги уже доступны более полезные коммерческие декомпиляторы, но они чрезмерно дороги, поэтому недостатки Anakrino вполне простительны.

Резюме

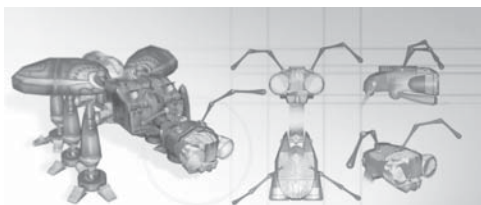
Управляемый код прекрасен, потому что нам больше не нужно беспокоиться об искажениях и утечках памяти, однако нам все равно еще нужно уметь пользоваться различными инструментами отладки. В этой главе я сосредоточился на особенностях, связанных с Visual Studio .NET и отладкой управляемых приложений.

В начале главы я рассмотрел некоторые вопросы, касающиеся улучшенных точек прерывания. Поддерживаемая Visual Studio .NET возможность вызова методов и свойств из условных точек прерывания замечательна, но это подразумевает, что вы должны быть более внимательными для избежания побочных эффектов. Не забывайте, что, если вы укажете некорректное условие прерывания и отладчик не сможет его вычислить, он не остановится.

¹ Работа более поздних версий этой программы отличается от описанной автором. — *Прим перев.*

Крайне полезное окно Watch предоставляет удивительные возможности для отладки управляемых приложений. Его механизм вычисления выражений позволяет легко вызывать методы и свойства, благодаря чему вы можете изменять работу отлаживаемой программы, облегчая ее тестирование. Кроме того, разрабатывая приложения C# и Managed Extensions for C++, вы можете добавлять к правилам авторазвертывания собственные типы, что позволяет сделать отладку еще более быстрой.

Наконец, хотя вы, возможно, никогда не будете программировать на MSIL, его легко освоить, и он поможет вам по-настоящему разобраться во всех тонкостях работы библиотеки классов .NET Framework. Подробнее о MSIL см. файл Partition III CIL.DOC, расположенный в подкаталоге <установочный каталог Visual Studio .NET>\SDK\v1.1\Tools Developers Guide\docs. В этом документе вы найдете сведения обо всех командах и о том, что они делают.



Усложненные технологии неуправляемого кода в Visual Studio .NET

Хотя разработка управляемого кода прекрасно защищает вас от всякого рода проблем, разработка неуправляемого кода (native code) дает вам все шансы не только прострелить себе ногу, но и поранить человека в соседнем отсеке. Разработка неуправляемого кода более трудоемка, но вы получаете при этом максимальные контроль и скорость. Кроме того, хоть, по словам некоторых деятелей маркетинга, вам нужно бросить все и перейти на .NET, такие резкие перемены вряд ли случатся в ближайшем будущем.

В этой главе я расскажу о «продвинутых» технологиях Microsoft Visual Studio .NET, позволяющих отлаживать неуправляемые приложения (native applications). Я начну с обсуждения точек прерывания, поскольку в неуправляемом коде в вашем распоряжении еще больше вариантов остановки процесса. Мы продолжим дополнительными подробностями об окне Watch и удаленной отладкой. В заключение я расскажу о языке ассемблера Intel IA32 (Pentium), чтобы вы всегда могли выяснить что происходит, даже не имея под рукой исходного кода и затерявшись во дебрях чужого кода или ОС.

Усложненные точки прерывания для неуправляемого кода

В главе 5 говорилось об общих точках прерывания для неуправляемого и управляемого кода. В этой главе я обращусь к уникальным особенностям неуправляемых приложений и некоторым проблемам, с которыми вы столкнетесь. Я также

расскажу о чудесных точках прерывания по данным (data breakpoints), доступных в неуправляемых приложениях.

Усложненный синтаксис точек прерывания

В отличие от отладки управляемого кода в отладке неуправляемого кода есть дополнительные возможности управлять местом и временем инициации точек прерывания. В силу природы генерирования неуправляемых символов вам много раз придется предоставлять отладчику дополнительную помощь для правильного размещения точек прерывания в нужных местах. Когда речь идет об отладочных символах, правила гораздо мягче строгих правил C++. Так, вполне обосновано наличие нескольких символов верхнего уровня (top-level symbols) для LoadLibrary. Каждый модуль, импортирующий LoadLibrary, содержит для нее символ (указывающий на импорт), и экспортирующий модуль также содержит символ (указывающий на экспорт). Усложненный синтаксис точек прерывания помогает установить диапазон действия точно на нужный символ.

Усложненный синтаксис точек прерывания интересен тем, что вы привыкли постоянно видеть его в Microsoft Visual C++ 6 и предыдущих версиях, поскольку именно так в старом диалоговом окне Breakpoint отображались установленные вами точки прерывания. В Visual Studio .NET вы больше не увидите отображения усложненного синтаксиса точек прерывания, но вам все-таки следует знать, что это за синтаксис для полноценного управления отладчиком.

Усложненный синтаксис точек прерывания состоит из двух частей. Первая — это контекстная часть (context portion), а вторая представляет местоположение, выражение или переменную. Контекстную часть можно воспринимать так же, как область видимости переменной при программировании. Контекст просто предоставляет отладчику ясное расположение вашей точки прерывания.

В терминах отладчика контекст определяют функция, исходный файл и двоичный модуль, и в усложненном синтаксисе точек прерывания контекст описывается как «{[функция],[исходный файл],[двоичный модуль]}». Надо указать лишь достаточное количество информации о контексте для установки точки прерывания, так что контекстная часть может содержать единственное поле или включать все три. Для заурядной точки прерывания по месту (location breakpoint) отладчику понадобятся только сведения об имени исходного файла. Так, в Visual C++ 6 стандартная точка прерывания по месту из строки 20 файла TEST.CPP отображалась в диалоговом окне Breakpoint как {, TEST.CPP, }.20. Вообще-то, если вы хотите установить такую же точку прерывания в Visual Studio .NET по-настоящему сложным способом, можете ввести {, TEST.CPP, }@20 в поле ввода Function на вкладке Function диалогового окна New Breakpoint. Когда вы щелкнете OK, появится информационное окно «IntelliSense could not find the specified location. Do you still want to set the breakpoint?» («IntelliSense не может обнаружить указанное место. Вы по-прежнему хотите установить точку прерывания?»), потому что IntelliSense не знает усложненного синтаксиса точек прерывания. Щелкните в информационном окне Yes и, запустив программу, вы увидите, что точка прерывания установилась. Если вы уже в отладке, то увидите появившуюся в строке красную точку.

Возможность указывать контекст для точки прерывания по месту позволяет решить весьма гадкую отладочную проблему. Рассмотрим случай, когда у вас есть

исходный файл с диагностической функцией `CheckMyMem`, используемой двумя DLL — `A.DLL` и `B.DLL`, — и в каждую DLL функция вносится статическим подключением (static linking). Поскольку вы широко применяете профилактическое программирование, вы часто вызываете эту функцию из обеих DLL. Однако случайные сбои происходят только в `B.DLL`. Если вы установите стандартную точку прерывания по месту в строке 27 исходного кода `CheckMyMem`, она будет инициироваться в обеих DLL, хотя вы хотите просмотреть вызовы, сделанные только в `B.DLL`. Чтобы указать, что точка прерывания по месту должна инициироваться только в `B.DLL`, надо вручную ввести контекст точки прерывания `{, ЧЕКСМЫМЕМ.CPP, B.DLL}@27`. Хотя вам, возможно, кажется, что это надуманный пример, и вы никогда не разделяете исходный код между модулями подобным способом, вы, вероятно, никогда не задумывались о том, что происходит при использовании подставляемых функций (inline functions) в классах C++!

Во второй части усложненного синтаксиса точек прерывания указывается место, выражение или переменная. Однако, как вы скоро увидите, в Visual Studio .NET, кроме номера строки и имени функции, другие значения устанавливать нельзя. Это не проблема, так как устанавливать усложненные точки прерывания в Visual Studio .NET гораздо проще, чем было в Visual C++ 6.

Точки прерывания в системных и экспортируемых функциях

В главе 5 я рассказывал о замечательных способах, которыми можно просто ввести имя функции или метода и автоматически получить установленную точку прерывания. Однако я не рассказал об установке точки прерывания в функции, импортируемой вашей программой из DLL. Устанавливая точки прерывания в этих экспортируемых из DLL функциях, можно решать чрезвычайно трудные проблемы. Так, можно получать управление обработкой в известной точке для отслеживания вытекающих нарушений целостности памяти. Еще хороший пример: вы хотите взглянуть, какая информация передается в разных параметрах. Интересно, что, попытавшись установить точку прерывания для экспортируемой функции, вы будете разочарованы. Она не работает. С отладчиком все в порядке — просто вам надо дать ему контекстную информацию о том, где найти эту функцию. Кроме того, важна еще одна небольшая деталь: имя функции зависит от того, загружены ли символы для DLL. Прежде чем я стану это обсуждать, установите отладчик Visual Studio .NET на загрузку экспортов (exports) как символов. В диалоговом окне Options на странице свойств Native из папки Debugging установите флажок Load DLL Exports. Причина установки этого параметра в том, что, даже если у вас нет символов, по крайней мере экспортируемые символы для модуля будут сопровождаться таблицей «псевдосимволов», построенной вне экспортируемых функций из DLL. Тогда вместо шестнадцатеричных чисел вы увидите имена этих экспортируемых функций.

Чтобы проиллюстрировать установку точки прерывания в системной DLL, я устанавливаю ее в функции `LoadLibrary` из `KERNEL32.DLL`. Возможно, вы захотите сделать это со мной, чтобы увидеть все этапы в действии. Так как `LoadLibrary` вызывается из реальных программ, для отладки можно выбрать любое приложение. Начнем с пошагового выполнения, чтобы запустить отладчик и инициализировать все таблицы символов. Если вы попытаетесь просто указать `LoadLibrary` в

диалоговом окне New Breakpoint, то увидите, что при щелчке кнопки ОК кажется, что точка прерывания установилась. Однако вам уже должно быть известно, что всегда надо проверять окно Breakpoints и следить, не отображается ли рядом с точкой прерывания значок вопросительного или восклицательного знака, указывающий на то, что точка прерывания не установилась. В приложении, которое я использую для установки точек прерывания (WDBG из главы 4), в окне Breakpoint отображается вопросительный знак рядом с текстом, показывающим тип `LoadLibrary(const unsigned short *)`.

Первый шаг установки точки прерывания в экспортируемой функции — определение наличия загруженных символов для модуля, содержащего экспорт. Поскольку все вы должны были прекратить чтение в конце второй главы и создать сервер символов, чтобы иметь возможность всегда получать все символы ОС, то символы должны быть загружены. Проверить загрузку символов можно двумя способами. Во-первых, если в окне Debug Output вы видите текст «<Program>: Loaded <DLL>», Symbols loaded., символы загружены. Второй способ задействует окна Modules, доступные из подменю Windows меню Debug или нажатием `Ctrl+Alt+U` при использовании сочетаний клавиш, установленных по умолчанию. Самая правая колонка в окне Module — Information — сообщает, загружены ли символы. Выделите интересующий вас модуль и промотайте до упора вправо. Если столбец Information для вашего модуля сообщает Symbols Loaded, символы в вашем распоряжении. Если в ней сказано что-то другое и вы уверены в наличии правильного PDB-файла для DLL, щелкните правой кнопкой элемент в окне Modules и выберите из контекстного меню команду Reload Symbols. Появившееся диалоговое окно Reload Symbols: filename.pdb позволит найти нужный PDB-файл. Поскольку сервер символов превратит установку символов в тривиальную операцию, я настоятельно рекомендую вам избрать этот способ. Если в окне Debug Output или Modules сказано что-то другое, символы не загружены.

Если символы не загружены, строкой местоположения вам послужит имя, экспортируемое из DLL. Имя можно проверить, запустив утилиту DUMPBIN из комплекта поставки Visual Studio .NET для DLL: `DUMPBIN /EXPORTS Имя DLL`. Запустив DUMPBIN для `KERNEL32.DLL`, вы не увидите функции `LoadLibrary`, но увидите две функции с похожими именами: `LoadLibraryA` и `LoadLibraryW`. (`LoadLibraryExA` и `LoadLibraryExW` — это различные API.) Суффиксы указывают набор символов, используемый функцией: суффикс A соответствует ANSI, а W — Wide или Unicode. Все ОС Microsoft Windows, кроме Windows 98/Me, для многоязыковой поддержки применяют встроенный набор Unicode. Если вы откомпилировали вашу программу с определением UNICODE, лучше использовать версию `LoadLibraryW`. Если нет — годится `LoadLibraryA`. Однако `LoadLibraryA` — это просто оболочка, выделяющая память для преобразования строки ANSI в Unicode и вызывающая `LoadLibraryW`, так что формально вы могли бы также применить `LoadLibraryW`. Если вы уверены, что ваша программа будет вызывать только одну из этих функций, можете установить точку прерывания только в ней. Если нет, установите точки прерывания в обеих функциях.

Если ваше приложение предназначено только для Microsoft Windows 2000/XP/.NET Server 2003, всегда используйте Unicode. Вы сможете получить большой при-

рост производительности. В своей колонке «Under the Hood» в «Microsoft Systems Journal» за декабрь 1997 года Мэтт Петрек (Matt Pietrek) сообщил, что ANSI-оболочки приводят к серьезным потерям производительности. Кроме ускорения работы программы, с использованием Unicode вы на несколько шагов приблизитесь к полной интернационализации.

Если символы не загружены, синтаксис точки прерывания для остановки в `LoadLibrary` будет таким: `{, , KERNEL32.DLL}LoadLibraryA` или `{, , KERNEL32.DLL}LoadLibraryW`. Если символы загружены, придется выполнить некоторые вычисления, так как потребуется соответствие расширенному имени символа (decorated symbol name). Вам понадобится знание соглашений вызова экспортируемых функций и прототипов функций. Ниже я разберу соглашения вызова подробнее. Для функции `LoadLibrary` прототип из `WINBASE.H` (с раскрытыми для ясности несколькими макросами) выглядит так:

```
__declspec (dllimport)
HMODULE
__stdcall
LoadLibraryA(
    LPCSTR lpLibFileName
);
```

Макрос `WINBASEAPI` раскрывается в стандартное соглашение вызова — `__stdcall` — которое, кстати, является соглашением вызова для всех функций системного API. Функции стандартного вызова дополняются префиксом из символа нижнего подчеркивания и суффиксом из символа «@», за которым следует число помещаемых в стек байтов. К счастью, его просто вычислить: это сумма количества байтов всех параметров. В семействе процессоров Intel Pentium можно просто сосчитать число параметров и умножить его на 4. В случае с `LoadLibrary`, принимающей один параметр, итоговым именем будет `_LoadLibraryW@4`. Вот несколько примеров, которые дадут вам представление о том, как выглядят итоговые имена: для `CreateProcess`, имеющей 10 параметров, это `_CreateProcessW@40`; а для `TlsAlloc`, не имеющей параметров, — `_TlsAlloc@0`. Даже если функция не имеет параметров, следует сохранять формат «@#». Как в случае, когда символы не загружены, правила ANSI и Unicode сохраняются. Если символы загружены, синтаксис точки прерывания для остановки в `LoadLibrary` будет `{, , KERNEL32.DLL}_LoadLibraryA@4` или `{, , KERNEL32.DLL}_LoadLibraryW@4`.

Определив усложненный синтаксис для установки точки прерывания, вызовите диалоговое окно `New Breakpoint (Ctrl+B)`. На вкладке `Function` в поле ввода `Function` введите соответствующий усложненный синтаксис точки прерывания. После щелчка `OK` вы получите обычное предупреждение о том, что `IntelliSense` не находит точку прерывания. Щелкните `OK`, и отладчик установит точку прерывания. В окне `Breakpoints` вы увидите рядом с точкой прерывания полную красную точку, а в колонке `Name` точка прерывания будет представлена во всем блеске своего синтаксиса. Щелкнув точку прерывания правой кнопкой и выбрав из контекстного меню команду `Go To Disassembly`, вы увидите, где в памяти располагается экспортируемая функция.

Условные выражения

Хотя управляемый код позволяет вызывать методы и свойства через модификаторы условных выражений для точек прерывания, неуправляемый код — нет. Кроме того, условные выражения не могут вычислять значения макросов C++, поэтому, если вы хотите сравнить значение с `TRUE`, придется использовать значение `1` (хотя `true` и `false` вычисляются корректно). В коде C++, как и в некоторых языках управляемого кода, любые условные выражения должны использовать значения C++. Даже с этими ограничениями модификаторы условных выражений для точек прерывания по месту чрезвычайно эффективны, так как, кроме возможности вычислять значения переменных, вы получаете доступ к специальному набору значений — псевдорегистрам (*pseudoregisters*).

По большей части псевдорегистры представляют значения регистров ЦП. В Visual Studio .NET усовершенствованы используемые и отображаемые типы регистров. Кроме обычных регистров ЦП, Visual Studio теперь поддерживает дополнительные, такие как MMX, SSE, SSE2 и 3Dnow (табл. 7-1). Реальные регистры ЦП сопровождаются разграничителем `@`, а два специальных значения начинаются с символа `$`. Полный список значений регистров см. в документации по процессорам Intel и AMD (Advanced Micro Devices). Помните, в Visual C++ 6 вы также могли указывать `@` перед псевдорегистрами? Для обратной совместимости такая возможность сохранена и в Visual Studio .NET 2003, но будущие версии будут поддерживать для псевдорегистров только знак `$`, поэтому привыкайте к нему уже сейчас. Кроме того, чтобы просмотреть значение, некоторые из вас вводили значения регистров без знака `@`, предшествующего имени. Однако я всегда буду показывать регистры с префиксом `@`.

Табл. 7-1. Примеры псевдорегистров

Псевдорегистр	Описание
@EAX	Регистр возвращаемого значения (32-битное значение)
@BL	Младшее слово регистра EBX (16-битное значение)
@MM0	MMX-регистр 0
@XMM1	SSE-регистр 1
\$ERR	Значение последней ошибки (специальное значение)
\$TIB	Блок информации потока (специальное значение)

Два последних значения в табл. 7-1 предоставляют условным точкам прерывания дополнительные возможности. `$ERR` позволяет просмотреть значение последней ошибки в потоке (значение, возвращаемое вызовом API `GetLastError`) и остановиться, только если выполняется условие последней ошибки. Так, если вы хотите остановиться, только если значение последней ошибки, возвращаемое функцией API, равно `ERROR_INSUFFICIENT_BUFFER`, что указывает на недостаточный размер буфера данных, то вначале следует найти `ERROR_INSUFFICIENT_BUFFER` в `WINERROR.H` и узнать, что значение равно 122. Условное выражение для точки прерывания будет таким: `$ERR==122`.

Специальный псевдорегистр `$TIB` открывает решение неприятной проблемы Visual Studio .NET. К сожалению, здесь нет встроенных способов явной установки точки прерывания по месту, которая инициировалась бы только в определенном

потоке. (В главе 8 вы увидите, что в WinDBG такая возможность встроена). При работе с большими серверными приложениями, такими как ISAPI-фильтры, часто встречаются методы, вызываемые из многих потоков, но вы не хотите снашивать указательный палец, нажимая GO миллион раз потому, что отладчик останавливается на каждом вхождении в каждом потоке. Первый шаг к решению этой проблемы — остановиться в отладчике и вызвать окно Threads, чтобы увидеть все идентификаторы потоков. Определите, какой поток вы хотите останавливать, и запомните его идентификатор. Второй шаг — установить точку прерывания по месту в общей подпрограмме и вызвать диалоговое окно New Breakpoint, щелкнув точку прерывания правой кнопкой и выбрав Properties. Щелкните кнопку Condition и введите выражение `*(long*)($TIB+0x24) == <thread id>`, где `thread id` — идентификатор потока. Идентификатор потока располагается в блоке информации потока по смещению `0x24`. (Это можно узнать, разобрав `GetCurrentThreadId`, что мы сделаем ниже.)

И последнее: поскольку в выражениях нельзя вызывать функции, остановиться по строке с заданным значением непросто. В таком случае просто установите выражение, проверяющее каждый символ, вроде этого:

```
(szBuff[0]=='P')&&(szBuff[1]=='a')&&(szBuff[2]=='m')
```

Стандартный вопрос отладки

Можно ли присвоить имя потоку в неуправляемом коде?

Если вы прочли главу 6, то знаете, что можно легко присвоить потоку имя, отображаемое в окне Threads. Microsoft задокументировала способ, которым можно сделать то же самое в неуправляемых приложениях. По умолчанию неуправляемые приложения показывают имя функции, в которой был открыт поток. Чтобы отображать реальное имя, можно использовать специальное значение исключения для передачи нового имени, а класс записи отладчика (debugger writer) должен считывать память по адресу, переданному как часть исключения. Я заключил необходимый код в оболочку из функций в `BUGSLAYERUTIL.DLL`: `BSUSetThreadName` и `BSUSetCurrentThreadName`. Код для `BSUSetThreadNameA` показан ниже. После вызова этой функции окно Threads будет отображать то, что указано в качестве имени. В отличие от управляемого кода в неуправляемом можно менять имя потока как хочется. И еще: в отладчике показывается только первый 31 символ имени.

```
typedef struct tagTHREADNAME_INFO
{
    DWORD dwType      ; // Должен быть 0x1000.
    LPCSTR szName      ; // Указатель на имя (в адресном пр-ве пользователя).
    DWORD dwThreadId   ; // Идентификатор потока (-1=вызывающий поток).
    DWORD dwFlags      ; // Зарезервировано на будущее, должно равняться 0.
} THREADNAME_INFO ;

void BUGSUTIL_DLLINTERFACE __stdcall
    BSUSetThreadNameA ( DWORD dwThreadId ,
                      LPCSTR szThreadName )
```

см. след. стр.


```
{
    THREADNAME_INFO stInfo ;
    stInfo.dwType      = 0x1000 ;
    stInfo.szName      = szThreadName ;
    stInfo.dwThreadID  = dwThreadID ;
    stInfo.dwFlags     = 0 ;

    __try
    {
        RaiseException ( 0x406D1388 ,
                        0 ,
                        sizeof ( THREADNAME_INFO ) /
                          sizeof ( DWORD ) ,
                        (DWORD*)&stInfo ) ;
    }
    __except ( EXCEPTION_CONTINUE_EXECUTION )
    {
    }
}
```

Точки прерывания по данным

Точки прерывания по данным, также называемые глобальными точками прерывания для памяти (global memory breakpoints), — один из самых мощных инструментов. Благодаря им, когда что-либо меняет определенный участок памяти, отладчик сразу останавливается в той точке, перед которой память изменилась. Область действия точек прерывания по данным глобальна, и они не привязаны к конкретному местоположению за исключением того, где производится изменение памяти. Как вы понимаете, точки прерывания по данным — это именно то, что нужно для отслеживания всех видов проблем нарушения целостности и перезаписи памяти.

В Visual C++ 6 точки прерывания по данным устанавливались весьма затейливо, но теперь они очень просты в использовании. Главная проблема была в том, что, если в Visual C++ 6 точка прерывания устанавливалась неправильно, отладчик просто пошагово выполнял каждую машинную команду в приложении и проверял память после каждого выполнения. Не нужно говорить, что наличие исключений и нескольких переходов между процессами в каждом исключении мучительно замедляло работу. Если вы попадались в эту ловушку, то все, что можно было сделать, — завершить работу отладчика. К счастью, теперь в Visual Studio .NET при неправильной установке точек прерывания по данным вы получаете предупреждающее сообщение.

Прелесть точек прерывания по данным в том, что вся связанная с ними тяжелая работа выполняется не отладчиком, а процессором. Процессоры Intel содержат четыре специальных отладочных регистра (DR0–DR3), к которым может обращаться процессор для установки аппаратных точек прерывания на доступ к памяти. Эти отладочные регистры ограничиваются обзором адреса и 1, 2 или 4

байт по этому адресу. Это значит, что в своей программе в каждый момент времени вы можете контролировать не более 16 байт.

Хитрость установки точек прерывания по данным заключается в использовании адреса памяти, за которым следует наблюдать. Диалоговое окно New Breakpoint предполагает использование имени переменной, но часто это приводит к появлению информационного окна (рис. 7-1). Увидев его, всегда щелкайте No, потому что, если вы щелкните Yes, отладчик установит точку прерывания для обнаружения изменения памяти путем пошагового выполнения каждой команды ассемблера.

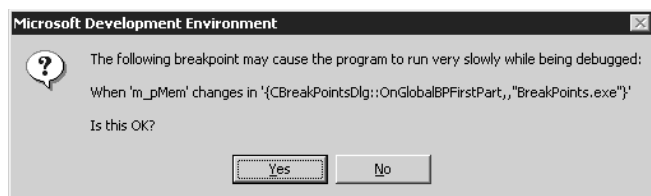


Рис. 7-1. Точка прерывания по данным приведет к пошаговому выполнению

Когда вы получили адрес памяти, за которым надо наблюдать, установить точку прерывания довольно просто. Вызовите диалоговое окно New Breakpoint и перейдите на вкладку Data (рис. 7-2). В поле ввода Variable введите наблюдаемый адрес. При вводе адреса важно помнить о выравнивании данных. Поле Items указывает, сколько байт наблюдать по этому адресу. Если вы хотите отслеживать 4 байта (двойное слово), введенный адрес должен оканчиваться на 0, 4, 8 или С для соответствия выравниванию. Так же, если вы хотите отслеживать только 2 байта (слово), адрес должен заканчиваться на 0, 2, 4, 6, 8, А, С или Е. Если вы попытаетесь установить точку прерывания по данным, не соблюдая выравнивания, скажем, введя адрес памяти, заканчивающийся на 7, и установив в поле Items значение 4, точка прерывания будет казаться установленной, но при ее появлении отладчик не остановится. Интересно, что при попытке установить 4-байтовую точку прерывания по данным по адресу, заканчивающемуся, например, на А, отладчик выровняет ее, сдвинув вперед на 2 байта.

Как видно на рис. 7-2, при установке точки прерывания по данным доступны еще два дополнительных поля. Поле Context используется, когда вы указываете имя переменной в поле Variable, если область видимости переменной отличается от текущего местоположения. Поскольку гораздо лучше использовать адреса, поле Context можно игнорировать. То же относится и к полю Language, поскольку при использовании адресов язык также игнорируется.

Замечательное усовершенствование точек прерывания по данным в Visual Studio .NET по сравнению с Visual C++ 6 в том, что теперь им можно сопоставлять число выполнений и условия. Это позволяет подстраивать нужное состояние для останова в отладчике.

Введя точку прерывания по данным и проверив окно Breakpoints, чтобы убедиться, что точка прерывания полностью допустима, можете запустить приложение. Когда данные по указанному адресу изменяются, в отладчике происходит нечто интересное (рис. 7-3). Появляется информационное окно, указывающее на попадание точки прерывания по данным. Меня часто спрашивали, почему для точек

прерывания по данным созданы такие специальные условия. Причина связана с весьма сложной проблемой пользовательского интерфейса. Поскольку точки прерывания по данным могут инициироваться откуда угодно, остановка отладчика в месте, где на полях нет красной точки, обозначающей точку прерывания, сбивала бы с толку. Увидев информационное окно, вы будете знать, почему отладчик остановился.

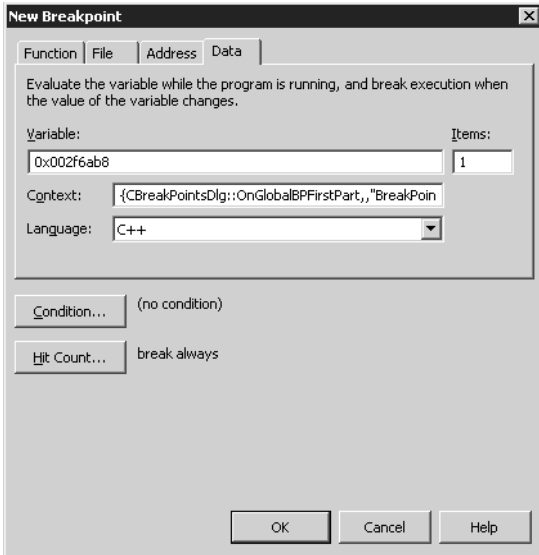


Рис. 7-2. Установка точки прерывания по данным

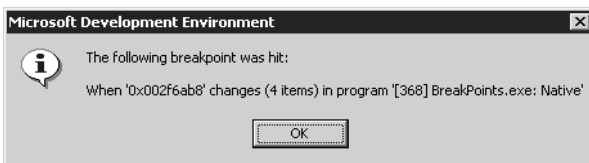


Рис. 7-3. Попадание точки прерывания по данным

При использовании точек прерывания по данным вам по завершении сеанса отладки, наверное, захочется их очистить. Раз я рекомендовал использовать для них адреса, весьма вероятно, что область памяти, за которой следует наблюдать, будет перемещаться от запуска к запуску. Это особенно актуально при наблюдении за памятью стека.

Улучшенные точки прерывания по данным

Это прекрасное усовершенствование отладки неуправляемого кода в Visual Studio .NET. Однако, если вы заглядывали в Intel Architecture Software Developer's Manuals (Руководство разработчика ПО для Intel-архитектуры), согласно документации по отладочным регистрам они могут быть установлены так, что каждый доступ к памяти для чтения или чтения/записи по указанному адресу может инициировать аппаратную точку прерывания. Точки прерывания по данным в Visual Studio .NET

инициируются только когда по указанному адресу изменяются данные. Так что, если записанное по адресу значение не меняет данных по этому адресу, остановки не будет.

Иногда все равно надо знать, кто записывает или читает адрес памяти. Сколько раз я отслеживал проблемы с производительностью, подсчитывая число обращений к памяти! Visual Studio .NET скрывает часть возможностей аппаратных отладочных регистров, но я решил, что должен быть способ заставить отладчик работать в полную мощность.

Обдумывая решение, я получил по электронной почте сообщение от Майка Мориарти (Mike Morearty) о том, что он хотел того же и нашел решение. Впоследствии Майк разработал код, делающий именно то, что мне было нужно, так что я мог даже не думать больше об этой проблеме. Все, что вам надо сделать, — это зайти на <http://www.morearty.com/code/breakpoint> и прочитать об этом. Решение Майка, которое на самом деле является единственно возможным, заключается в добавлении к проекту небольшого класса C++, применяемого для установки улучшенных точек прерывания по данным в собственном коде. Web-страница Майка прекрасно описывает использование его класса `CBreakpoint`, поэтому не буду повторяться. Отмечу, что, раз для создания точек прерывания требуется вручную добавить код в проект, будьте очень осторожны, возвращая код. Если вы оставите класс `CBreakpoint` активным, ваши компоновки за день не запустятся, и вы сразу поймете значение слов «удар по карьере»!

Окно Watch

Как видно из последних двух глав, я очень люблю окно Watch. Для отладки неуправляемого кода оно предлагает больше возможностей, чем когда-либо. Одно из улучшений, которое вы уже могли заметить в отладке неуправляемого кода в Visual Studio .NET, в том, что теперь окно Watch автоматически распознает типы `HRESULTS`, `wchar_t` (символы UNICODE) и `bool`. Вы также могли заметить, что подсказки для данных, появляющиеся в окнах исходного кода, кажется, получили ударную дозу стероидов.

Форматирование данных и вычисление выражений

Первый трюк, который вам следует освоить на пути к мастерству владения окном Watch, — запомнить символы форматирования из табл. 7-2 и 7-3, взятых из документации Visual Studio .NET. Окно Watch обладает потрясающей гибкостью отображения данных, и путь к этой гибкости — в применении форматирующих кодов из этих таблиц. Применять форматирование просто: поставьте за переменной запятую и укажите формат. Самый полезный спецификатор формата для COM-программирования — это `,hr`. Если вы будете держать в окне Watch выражение `@EAX, hr`, то, проходя через вызов COM-метода, сможете видеть результаты вызова в понятной форме. (`EAX` — это регистр процессоров Intel, в котором сохраняются возвращаемые значения.) Спецификаторы формата позволяют легко управлять представлением данных, так что вы сэкономяте массу времени при их интерпретации.

Табл. 7-2. Символы форматирования переменных в окне Watch

Символ	Описание формата	Пример	Отображение
d, i	Десятичное целое со знаком	(int)0xF000F065,d	–268373915
u	Десятичное целое без знака	0x0065,u	101
o	Восьмеричное целое без знака	0xF065,o	0170145
x, X	Шестнадцатеричное целое	61541,X	0x0000F065
l, h	Длинный или короткий префикс для d, i, u, o, x, X	0x00406042,hx	0x0c22
f	Число с плавающей точкой со знаком	3./2.,f	1.500000
e	Число со знаком в экспоненциальном представлении	3./2,e	1.500000e+000
g	Число со знаком с плавающей точкой или со знаком в экспоненциальном представлении, что короче	3./2,g	1.5
c	Одиночный символ	0x0065,c	‘e’
s	Строка ANSI	szHiWorld,s	«Hello world»
su	Строка Unicode	szWHiWorld,su	«Hello world»
hr	HRESULT или код ошибки Win32	0x00000000,hr	S_OK
wc	Флаг класса Windows	0x00000040,wc	WC_DEFAULTCHAR (Хотя он документирован, этот формат не работает в Visual Studio .NET.)
wm	Числа сообщений Windows	0x0010,wm	WM_CLOSE

Табл. 7-3. Символы форматирования дампов памяти в окне Watch

Символ	Описание формата	Пример	Отображение
ma	64 ASCII-символа	0x0012ffac,ma	0x0012ffac .4...0...».0W&.....1W&. 0:W.1....».1JO&.1.2.» ..1...0y...1
m	16 байт в шестнадцатеричном формате, продолжающиеся 16 ASCII-символами	0x0012ffac,m	0x0012ffac b3 34 cb 00 84 30 94 80 ff 22 8a 30 57 26 00 00 .4...0...».0W&..
mb	16 байт в шестнадцатеричном формате, продолжающиеся 16 ASCII-символами	0x0012ffac,mb	0x0012ffac b3 34 cb 00 84 30 94 80 ff 22 8a 30 57 26 00 00 .4...0...».0W&..
mw	8 слов	0x0012ffac,mw	0x0012ffac 34b3 00cb 3084 8094 22ff 308a 2657 0000
md	4 двойных слова	0x0012ffac,md	0x0012ffac 00cb34b3 80943084 308a22ff 00002657
mq	4 учетверенных слова	0x0012ffac,mq	0x0012ffac 8094308400cb34b3 00002657308a22ff
mu	2-байтовые символы (Unicode)	0x0012ffac,mu	0x0012ffac 34b3 00cb 3084 8094 22ff 308a 2657 0000 ??????

Табл. 7-3. Символы форматирования дампов памяти ... *(продолжение)*

Символ	Описание формата	Пример	Отображение
#	Разворачивает указатель на адрес памяти в указанное число значений	pCharArray,10	Развернутый массив из 10 символов с использованием расширителей +/-

Числовой спецификатор формата — ,# — позволяет развернуть указатель на адрес памяти в указанное количество значений. Если у вас есть указатель на массив из 10 чисел типа long, в окне Watch будет показано только первое значение. Чтобы увидеть массив целиком, укажите за переменной число значений, которые вы хотите увидеть. Например, pLong, 10 покажет развернутый массив из 10 элементов. Если у вас большой массив, можно составить указатель на его середину и развернуть только нужные элементы, например, (pBigArray+100), 20, для отображения 20 элементов, начиная со смещения 99. Вы заметите, что при вводе подобного значения индексы всегда начинаются с 0 независимо от позиции первого отображаемого элемента. В примере pBigArray первый индекс будет отображаться как 0, хоть это и сотый элемент массива. Второй индекс — 101-й элемент массива — будет показан как 1 и т. д.

В дополнение к тому, что в окне Watch можно как угодно форматировать данные, оно также позволяет приводить и склонять данные переменных, чтобы увидеть именно то, что нужно. Так, можно использовать выражения BY, WO и DW для задания смещений в указателях. Чтобы увидеть идентификатор текущего потока в окне Watch, можно использовать DW(\$TIB+0x24). Также допускаются оператор вычисления адреса (&) и оператор указателя (*), и оба позволяют получить значение по адресам памяти и увидеть результаты приведения типов в коде.

Отличный фокус, который я люблю использовать в отладке неуправляемого кода, — наблюдение за значениями переменных в стеке. Иногда встречаются локальные переменные, за которыми хотелось бы следить при прохождении через другие функции. С помощью контекстной части усложненного синтаксиса точек прерывания, о которой я рассказывал в разделе «Усложненный синтаксис точек прерывания», можно наблюдать за значением. Так, если в функции CopyDatabaseValue, расположенной в исходном файле FOO.CPP из модуля DB.DLL объявлена переменная szBuff, ее точное значение указывается как {CopyDatabaseValue, FOO.CPP, DB.DLL}szBuff. Теперь, где бы вы ни находились в функциях, вызванных из CopyDatabaseValue, вы легко сможете следить за szBuff.

Хронометраж кода в окне Watch

Это еще один изящный трюк. Псевдорегистр \$CLK может служить простым таймером. Часто требуется лишь примерное представление о времени между двумя точками, например, сколько времени занимает обращение к базе данных. \$CLK позволяет легко выяснить длительность вызова. Помните, что это время включает издержки отладчика. Фокус во введении двух контрольных выражений для \$CLK: первое — просто \$CLK, а второе — \$CLK=0 — обнуляет таймер после запуска. Хотя \$CLK и не идеальный таймер, его достаточно для приблизительных оценок.

Недокументированные псевдорегистры

В Visual Studio .NET появились два новых псевдорегистра. Поскольку в заголовке этого раздела есть слово «недокументированные», я должен предупредить вас, что в будущих версиях отладчика эти значения могут исчезнуть. Первый псевдорегистр — это `$HANDLES`. Он показывает число открытых описателей в текущем процессе. Умопомрачительная идея, позволяющая при отладке следить за утечками описателей. Если сообщаемое `$HANDLES` число постоянно увеличивается, у вас утечка. `$HANDLES`, в свою очередь, занимает постоянную позицию в моем окне Watch из-за своей чрезвычайной полезности.

Второй недокументированный псевдоописатель — `$VFRAME` — прекрасный инструмент для контроля стека в финальных сборках. `$VFRAME` — короткая дорога к виртуальному указателю фреймов (virtual frame pointer). На машинах IA32 `$VFRAME` указывает на следующий фрейм стека, так что он может пригодиться при просмотре стека вручную. Если вы используете стандартные фреймы стека, то `$VFRAME` указывает на значение EBP предыдущего элемента.

Автоматическое разворачивание собственных типов

Хотя управляемый C++ и отладка C# позволяют разворачивать ваши собственные типы в окне Watch, авторазвертывание (autoexpansion), предлагаемое в отладке неуправляемого кода, поднимает эту возможность на новую высоту. Вообще, начиная с Visual Studio .NET 2003, окно Watch и подсказки по данным теперь автоматически пытаются показать несколько первых компонентов структур и классов. Но вы, наверное, видели, как несколько стандартных типов, таких как `CObject`, `RECT` и некоторые типы STL, разворачиваются в окне Watch гораздо подробнее, что достигается правилами авторазвертывания (autoexpand rules). Чудеса творятся в текстовом файле `AUTOEXP.DAT`, расположенном в подкаталоге <установочный каталог Visual Studio .NET>\COMMON7\PACKAGES\DEBUGGER. Вы вправе добавлять в список авторазвертывания собственные типы, вводя их в файл `AUTOEXP.DAT`. (К сожалению, `AUTOEXP.DAT` должен оставаться в этом каталоге, так что вам придется настроить рабочий каталог вашего ПО контроля версий для внесения `AUTOEXP.DAT` в этот каталог.)

В качестве примера я добавлю запись авторазвертывания для структуры `PROCESS_INFORMATION`, которая передается функции `API CreateProcess`. Сначала надо проверить, что отладчик Visual Studio .NET распознает как тип. В созданной для примера программе я поместил переменную `PROCESS_INFORMATION` в окно Watch и посмотрел колонку Type в правой части окна и увидел тип `_PROCESS_INFORMATION`. Если взглянуть на это определение структуры, он соответствует тэгу структуры:

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION
```

Согласно документации на `AUTOEXP.DAT` запись авторазвертывания имеет формат `type=[text]<member[, format]>...`. В табл. 7-4 дано описание полей. Заметьте: при авторазвертывании может отображаться несколько компонентов.

Табл. 7-4. Записи авторазвертывания AUTOEXP.DAT

Поле	Описание
Type	Имя типа. Для типов-шаблонов это поле может завершаться символом <*>, объединяя все производные типы.
Text	Любой буквенный текст. Как правило, в этом поле указывается имя компонента данных или его сокращенная версия.
member	Отображаемый компонент данных. В поле можно указывать выражения, так что, если вам надо добавить смещение к указателям, вы вправе включить их в вычисления. Также допускаются операторы приведения типов.
format	Дополнительные спецификаторы формата для переменных. Это те же спецификаторы, что показаны в табл. 7-2.

В структуре `PROCESS_INFORMATION` меня интересуют значения `hProcess` и `hThread`, так что мое правило авторазвертывания будет таким: `_PROCESS_INFORMATION = hProcess=<hProcess,X> hThread=<hThread,X>`. Я применяю спецификаторы формата `,X`, потому что всегда хочу видеть значения в шестнадцатеричном формате. На рис. 7-4 показано правило авторазвертывания для `_PROCESS_INFORMATION` в подсказке для данных в окне исходного кода.

```
memset ( &pi , NULL , sizeof ( PROCESS_INFORMATION ) ) ;
pi = {hProcess=0x00000000 hThread=0x00000000}
```

Рис. 7-4. Авторазвертывание в подсказке для данных

Вводя свое новое правило аторазвертывания, я должен поместить его в разделе файла `AUTOEXP.DAT`, обозначенном `[AutoExpand]`. Лучшее решение — размещать свои значения сразу после `[AutoExpand]`, чтобы их можно было легко найти и не смешивать с приемами, которые я опишу в следующем разделе. Хорошая новость: в отличие от авторазвертывания в отладке управляемого кода, считываемого только при запуске Visual Studio .NET, файл `AUTOEXP.DAT` читается при каждой отладке, так что создавать правила авторазвертывания для неуправляемого кода намного проще.

В файле вы встретите специальный код форматирования `<,t>`. Он предписывает отладчику внести имя производного типа. Например, если у вас есть базовый класс `A` с производным классом `B` и правило авторазвертывания есть только у `A`, то авторазвертывание переменной типа `B` будет представлять собой имя класса `B`, за которым следует правило авторазвертывания для класса `A`. Формат `<,t>` хорошо помогает поддерживать порядок в классах.

Добавление собственных значений HRESULT

В дополнение к раскрытию ваших типов окно Watch в Visual Studio .NET теперь способно показывать ваши собственные значения `HRESULT` в виде текста вместо каких-то запутанных цифр. Волшебный `AUTOEXP.DAT` также хранит эти значения. В конце файла `AUTOEXP.DAT` добавьте новый раздел `[hresult]` и внесите туда каждый собственный `HRESULT`, используя шаблон: `<«десятичное значение без знака»=<текст HRESULT>`. Следующий код — пример, включающий некоторые значения, не обрабатываемые отладчиком автоматически. Чтобы увидеть реальные значения `HRESULT` для одного из встроенных преобразований или добавленных вами в

раздел [hresult], возьмите переменную HRESULT и добавьте к ней ,и или ,х. Это приведет к отображению переменной в виде целого числа без знака или шестнадцатеричного значения, соответственно.

```
[hresult]
2147500051=CO_E_CANT_REMOTE
2147500056=CO_E_CREATEPROCESS_FAILURE
2147500059=CO_E_LAUNCH_PERMISSION_DENIED
```

Супернастройка отображения в окне Watch

Серьезным усовершенствованием окна Watch в отладке неуправляемого кода стала надстройка Expression Evaluator Add-In (EEAddIn). EEAddIn позволяет отладчику вызывать одну из ваших DLL когда окно Watch оценивает определенный тип. Это открывает прекрасную возможность производить вычисления, отображающие данные в более подходящем виде. Так, окно Watch отображает структуру SYSTEMTIME (представляющую дату и время в Win32) группой шестнадцатеричных чисел, лишая вас возможности определить время. При использовании EEAddIn окно Watch вместо этого отображает читаемую строку, вроде {5/13/2002 12:51 AM}.

Чтобы сообщить окну Watch, что у вас есть DLL для EEAddIn, которую вы хотите загрузить, надо поместить запись для каждого оцениваемого типа в вездесущий файл AUTOEXP.DAT. В разделе [AutoExpand] расширение для типа указывается в следующем синтаксисе:

имя типа=\$ADDIN(имя dll,экспортируемая функция)

Имя типа, как и в правилах авторазвертывания, — это имя, отображаемое для переменной в окне Watch в столбце Type. Имя DLL — имя файла библиотеки DLL. Согласно документации к EEAddIn, которая представляет собой просто пример проекта Visual Studio .NET, названный соответственно EEAddIn, имя DLL должно содержать только имя файла DLL, так как предполагается, что все надстройки EEAddIn хранятся в том же каталоге, что и AUTOEXP.DAT. Однако я обнаружил, что для правильной загрузки надо в имени DLL указывать полный путь к DLL. Экспортируемая функция указывает функцию, которую надо вызывать для обработки пользовательского отображения данного типа.

Так как надстройки запускаются в адресном пространстве отладчика, следует обеспечить корректную обработку возможных исключений, иначе это приведет к краху отладчика. Индивидуальные экспортируемые функции должны соответствовать прототипу CUSTOMVIEWER (листинг 7-1). При вызове функции в качестве параметров она получит адрес типа; указатель на вспомогательную структуру DEBUGHELPER; текущую систему счисления (десятичная или шестнадцатеричная); значение типа Boolean, указывающее, ожидает ли отладчик строки UNICODE (которое в Visual Studio .NET игнорируется, так как здесь всегда ожидается возврат ANSI-символов); строковый буфер (string buffer) для записи результата и максимальную длину строкового буфера. Вспомогательная структура (листинг 7-1) содержит несколько указателей на функции, вызвав которые можно получить сведения о значениях, расположенных по адресу типа. Самые важные — GetRealAddress и ReadDebuggeeMemoryEx. Передав адрес, полученный вашей экспортируемой функ-

цией, функции `GetRealAddress`, вы получите действительный адрес переменной. Передав это значение функции `ReadDebuggeeMemoryEx`, вы получите байты для типа. Вспомогательный класс хорош тем, что полностью скрывает магию получения данных из локальных и удаленных процессов отлаживаемой программы.

Листинг 7-1. Прототип экспорта и вспомогательная структура `EEAddIn`

```
/*-----
    Единственное описание надстроек Expression Evaluator AddIn,
    извлеченное из примера EEAddIn
-----*/

typedef struct tagDEBUGHELPER
{
    DWORD dwVersion ;
    BOOL (WINAPI *ReadDebuggeeMemory)( struct tagDEBUGHELPER * pThis ,
                                       DWORD dwAddr ,
                                       DWORD nWant ,
                                       VOID * pWhere ,
                                       DWORD * nGot );
    // Далее, только если dwVersion >= 0x20000
    DWORDLONG (WINAPI *GetRealAddress)( struct tagDEBUGHELPER *pThis );
    BOOL (WINAPI *ReadDebuggeeMemoryEx)( struct tagDEBUGHELPER *pThis ,
                                         DWORDLONG qwAddr ,
                                         DWORD nWant ,
                                         VOID* pWhere ,
                                         DWORD * nGot );
    int (WINAPI *GetProcessorType)( struct tagDEBUGHELPER *pThis );
} DEBUGHELPER ;

// Прототип, которому должны соответствовать все ваши функции
typedef HRESULT (WINAPI *CUSTOMVIEWER)( DWORD dwAddress ,
                                         DEBUGHELPER * pHelper ,
                                         int nBase ,
                                         BOOL bUniStrings ,
                                         char * pResult ,
                                         size_t max ,
                                         DWORD reserved ) ;
```

Задача вашей экспортируемой функции — преобразовать эти байты, считанные из отлаживаемой программы в нечто отображаемое в окне Watch. Так как можно легко прочитать память отлаживаемой программы, вы будете работать с копией информации. Когда я получил первое представление об архитектуре `EEAddIn`, я сразу представил миллионы отображаемых параметров, которые хотел бы видеть. Первый из них принимал бы `HINSTANCE` или `HMODULE` и показывал значение, за которым следовало бы имя DLL в этом местоположении. Но затем я спустился с небес. Для преобразования `HINSTANCE` или `HMODULE` в имя DLL требуется описатель процесса. Структура `DEBUGHELPER` (листинг 7-1) позволяет считывать память, но не получать описатель процесса. Конечно, позже я осознал, что если бы моя функция `EEAddIn` работала с процессом, отлаживаемым удаленно, то не помогло бы

даже наличие описателя процесса, поскольку я не смог бы ничего с ним сделать на машине, где запущен отладчик. Возможно, будущие версии Visual Studio .NET предложат способы запроса у процесса информации, требующей значения описателей.

Несмотря на ограничение, по которому можно считывать только память отлаживаемой программы, вам открывается масса прекрасных вариантов наблюдаемых параметров для размещения в окне Watch, ускоряющих отладку. В примерах кода к книге содержится моя текущая надстройка EEAddIn — BSU_ExpEval_AddIn. К моменту написания этого абзаца я объединил наблюдаемые параметры `_SYSTEMTIME` и `_FILETIME` из примера Visual Studio, но добавил к ним обработку ошибок и расширения структур `_OSVERSIONINFOA`, `_OSVERSIONINFOW`, `_OSVERSIONINFOEXA` и `_OSVERSIONINFOEXW`. Теперь, имея одну из структур, обрабатываемых `GetVersionEx`, их можно отображать, как показано на рис. 7-5, который демонстрирует часть вывода тестовой программы для BSU_ExpEval_AddIn. В листинге 7-2 показано, что надо сделать для расширения структуры `_OSVERSIONINFOA`.

Один совет, касающийся DLL для EEAddIn: если из своей функции вы возвращаете `E_FAIL`, в окне Watch отображается «???», поэтому лучше возвращать `S_OK` и устанавливать в возвращаемом тексте «...», чтобы ваш вывод совпадал с обычным отображением в окне Watch. Это может помочь и в отладке DLL. Еще советую в отладочных сборках указывать в возвращаемом тексте информацию о сбоях, чтобы ваши расширения для отладчика было легче отлаживать. И еще: стоит нам начать обмениваться своими надстройками EEAddIn, и мы сможем получить гораздо лучшую отладочную информацию, чем когда-либо ранее от IDE. Призываю вас рассматривать любые возможные структуры и классы из Win32, MFC и ATL, решая, нельзя ли предоставить улучшенный вывод.

Name	Value	Type
stOSExA	{Windows .NET Datacenter Server (100) Service Pack 2}	_OSVERSIONINFOEXA
stOSA	{Windows NT 4.0 (100) Service Pack 2}	_OSVERSIONINFOA

Рис. 7-5. EEAddIns в работе

Листинг 7-2. Пример EEAddIn для `_OSVERSIONINFOA`

```
// Затрагивает только первые 5 двойных слов в структурах,
// так что можно передавать и ANSI- и UNICODE-версии
static int ConvertBase0SV ( LP_OSV _pOSVA , char * szStr )
{
    int iCurrPos = 0 ;

    if ( ( _pOSVA->dwMajorVersion == 4 ) && ( _pOSVA->dwMinorVersion ==0))
    {
        if ( _pOSVA->dwPlatformId == VER_PLATFORM_WIN32_NT )
        {
            iCurrPos = wsprintf ( szStr , _T ( "Windows NT 4.0 " ) ) ;
        }
        else
        {
```

```

        iCurrPos = wsprintf ( szStr , _T ( "Windows 95 " ) );
    }
}
else if ( ( pOSVA->dwMajorVersion == 4 ) &&
          ( pOSVA->dwMinorVersion == 10 ) )
{
    iCurrPos = wsprintf ( szStr , _T ( "Windows 98 " ) );
}
else if ( ( pOSVA->dwMajorVersion == 4 ) &&
          ( pOSVA->dwMinorVersion == 90 ) )
{
    iCurrPos = wsprintf ( szStr , _T ( "Windows Me " ) );
}
else if ( ( pOSVA->dwMajorVersion == 5 ) &&
          ( pOSVA->dwMinorVersion == 0 ) )
{
    iCurrPos = wsprintf ( szStr , _T ( "Windows 2000 " ) );
}
else if ( ( pOSVA->dwMajorVersion == 5 ) &&
          ( pOSVA->dwMinorVersion == 1 ) )
{
    iCurrPos = wsprintf ( szStr , _T ( "Windows XP " ) );
}
else if ( ( pOSVA->dwMajorVersion == 5 ) &&
          ( pOSVA->dwMinorVersion == 2 ) )
{
    iCurrPos = wsprintf ( szStr , _T ( "Windows Server 2003 " ) );
}
else
{
    // Сдаюсь!
    iCurrPos = 0 ;
}
return ( iCurrPos ) ;
}

```

// Опять же, эта функция использует общее поле версий A и W,

// так что ее можно использовать для обеих.

static int ConvertBuildNumber (LPOSVERSIONINFOA pOSVA , char * szStr)

```

{
    int iCurrPos = 0 ;
    if ( VER_PLATFORM_WIN32_NT == pOSVA->dwPlatformId )
    {
        iCurrPos = wsprintf ( szStr
                              ,
                              _T ( "(%d) " )
                              ,
                              pOSVA->dwBuildNumber ) ;
    }
    else if ( VER_PLATFORM_WIN32_WINDOWS == pOSVA->dwPlatformId )
    {
        WORD wBuild = LOWORD ( pOSVA->dwBuildNumber ) ;
    }
}

```

см. след. стр.

```

        iCurrPos = wprintf ( szStr , _T ( "(%d) " ) , wBuild ) ;
    }
    return ( iCurrPos ) ;
}

ADDIN_API HRESULT WINAPI
    AddIn_OSVERSIONINFOA ( DWORD          /*dwAddress*/ ,
                           DEBUGHELPER* pHelper ,
                           int            /*nBase*/ ,
                           BOOL          /*bUniStrings*/ ,
                           char *        pResult ,
                           size_t        /*max*/ ,
                           DWORD         /*reserved*/ )
{
    if ( pHelper->dwVersion < 0x20000 )
    {
        // Я не работаю с версиями ниже VS.NET.
        return ( E_FAIL ) ;
    }

    HRESULT hRet = E_FAIL ;

    __try
    {

        DWORDLONG    dwRealAddr = pHelper->GetRealAddress ( pHelper ) ;
        DWORD        nGot      = 0 ;
        OSVERSIONINFOA stOSA ;

        // Пытаемся считать структуру.
        if ( S_OK ==
            pHelper->
                ReadDebuggeeMemoryEx ( pHelper
                                       ,
                                       dwRealAddr
                                       ,
                                       sizeof ( OSVERSIONINFOA ) ,
                                       &stOSA
                                       ,
                                       &nGot
                                       ) )
        {

            // Убеждаемся, что все получено полностью.
            if ( nGot == sizeof ( OSVERSIONINFOA ) )
            {

                // Танцуем...
                char * pCurr = pResult ;
                int iCurr = ConvertBaseOSV ( &stOSA , pCurr ) ;
                if ( 0 != iCurr )
                {
                    pCurr += iCurr ;
                }
            }
        }
    }
}

```

```
        iCurr = ConvertBuildNumber ( &stOSA , pCurr ) ;

        pCurr += iCurr ;
        if ( '\\0' != stOSA.szCSDVersion[0] )
        {
            wsprintf ( pCurr ,
                      _T ( "%s" ) ,
                      stOSA.szCSDVersion ) ;
        }
    }
    else
    {
        _tcscopy ( pResult , _T ( "..." ) ) ;
    }
}
hRet = S_OK ;
}
}
__except ( EXCEPTION_EXECUTE_HANDLER )
{
    hRet = E_FAIL ;
}
return ( hRet ) ;
}
```

Стандартный вопрос отладки

Решена ли проблема ограничения в 255 символов в отладке?

ДА! В Visual Studio до версии Visual Studio .NET отладочная информация для неуправляемого кода была ограничена максимумом в 255 символов. Во времена С это не представляло проблемы, но введение шаблонов совершенно захлестнуло за 255 символов даже для простейших типов. Visual Studio .NET может иметь отладочные символы произвольной длины, так что вы должны увидеть все полностью. Еще это означает, что старое информационное сообщение C4786 (отладочная информация длиннее 255 символов), оставивавшее компиляцию при трактовке предупреждений как ошибок, наконец-то кануло раз и навсегда! Блаженны мы!

Удаленная отладка

Удаленная отладка неуправляемых приложений действует почти так же, как и удаленная отладка управляемых приложений. Просто установите компоненты удаленной отладки (см. главу 6) убедитесь, что ваша учетная запись на удаленном компьютере внесена в группы Administrators и Debugger Users, и можете подключаться и отлаживать все что угодно через новый транспортный уровень DCOM. Это

отличный способ подключаться и отключаться от этих долгоиграющих серверных процессов.

В дополнение к транспортному уровню DCOM, Visual Studio .NET 2003 предлагает еще два варианта удаленной отладки: Pipes (каналы) и TCP/IP. Вариант TCP/IP присутствовал с Visual C++ 6, но он не так безопасен, как Pipes. Удаленная отладка через TCP/IP позволяет любому подключиться к машине, а вот Pipes — указать, каким пользователям разрешено подключаться и отлаживать. Отладка через Pipes теперь установлена по умолчанию, хотя это медленнее чем TCP/IP.

Хотя она не так удобна как DCOM, отладка через Pipes и TCP/IP может стать прекрасным инструментом для решения отладочных задач. Одна, несомненно хорошая особенность в том, что в отладке через Pipes и TCP/IP можно запускать процессы. Кроме того, вы вправе настроить свои решения Visual Studio .NET так, чтобы они всегда запускали процесс для удаленной отладки. Это особенно полезно для «тяжелых» клиентских приложений, таких как DirectX-игры. Весьма востребована новая возможность создания нескольких подключений к удаленной машине, чтобы отлаживать несколько процессов. Еще одна прелесть в том, что, если вы собираетесь отлаживать только неуправляемый код, вам не нужно полностью проходить Remote Components Setup чтобы установить только отладку через Pipes и TCP/IP. Для установки отладки через Pipes и TCP/IP достаточно скопировать двоичные файлы с компьютера, на котором установлена Visual Studio .NET, в каталог на удаленной машине. Эти двоичные файлы и их расположение на компьютере с Visual Studio .NET показаны в табл. 7-5. Помните: версию MSVCMON.EXE из Visual C++ 6 использовать с Visual Studio .NET нельзя.

Табл. 7-5. Компоненты удаленной отладки через Pipe и TCP/IP

Файл	Расположение
MSVCR71.DLL	%SYSTEMROOT%\SYSTEM32
MSVC71.DLL	%SYSTEMROOT%\SYSTEM32
MSVCP71.DLL	%SYSTEMROOT%\SYSTEM32
MSVCMON.EXE	<Установочный каталог Visual Studio .NET>\COMMON7\PACKAGES\DEBUGGER
NATDBGDM.DLL	<Установочный каталог Visual Studio .NET>\COMMON7\PACKAGES\DEBUGGER
NATDBGTLNET.DLL	<Установочный каталог Visual Studio .NET>\COMMON7\PACKAGES\DEBUGGER

Прежде чем начать удаленную отладку, хорошо бы заняться планированием, дабы обеспечить успешность сеанса удаленной отладки. Версии удаленной отладки через Pipes и TCP/IP в Visual Studio .NET не столь темпераментны, как версия Visual C++ 6. Главный фокус в том, чтобы Visual Studio .NET обнаружила символы на локальной машине, где они загружаются. Для символов ОС лучшее решение — в установке сервера символов (см. главу 2). Если вы работаете с локальной сборкой проекта, лучше, чтобы отлаживаемая программа была установлена в одинаковых каталогах на локальной и удаленной машинах, так как не возникнет путаницы с расположением отдельных элементов. Наконец, неплохо убедиться, что вы смо-

жете запустить свою программу на удаленной машине, ведь нет ничего хуже, чем, начав удаленную отладку, обнаружить что не хватает DLL.

Чтобы начать удаленную отладку через подключения Pipe, надо зарегистрироваться в системе на удаленной машине и запустить MSVCMON.EXE. По умолчанию запуск MSVCMON.EXE подразумевает подключение к удаленной машине с машины, на которой запущена IDE Visual Studio .NET, с применением той же учетной записи, под которой вы зарегистрировались на удаленной машине. Если вы хотите сделать удаленную машину несколько более доступной, запустите MSVCMON.EXE с ключом командной строки `-u <домен\группа или пользователь>`, чтобы указать пользователей или группы, которым вы хотите разрешить запускать и отлаживать процессы на этой машине.

Настроить машину, на которой запущена IDE Visual Studio .NET, очень просто. Для этого лишь надо установить несколько элементов на странице свойств Debugging в окне свойств проекта. В разделе Action, поле Command and Working Directory следует заполнить расположением каталогов на удаленной машине. Дополнительно можно указать, что вы хотите подключиться к удаленному процессу, установив Attach в Yes. Последнее, что можно настроить в разделе Action, — это поле Symbol Path, если двоичные файлы не находятся в одинаковых каталогах на обеих машинах.

В разделе Remote Settings установите Connection to Remote Via Pipe (Native Only). В поле Remote Machine укажите имя или IP-адрес машины, хранящей MSVCMON.EXE. Можете попробовать указать имя, но IP-адрес работает безотказно. Неплохо бы проверить подключение к удаленной машине с помощью PING.EXE, чтобы определить, доступна ли она. Если вы можете получить доступ к удаленной машине по ее имени, можете использовать имя, но IP-адрес работает безотказно. Наконец, поле Remote Command должно содержать тот же полный путь и имя, которые указаны в поле Command из раздела Action. На рис. 7-6 показан пример проекта с заполненными полями.

Заполнив поля, вы быстро узнаете, установлено ли хорошее подключение. Консольное окно, в котором запущен MSVCMON.EXE, покажет имя пользователя, осуществляющего подключение, и вы начнете отладку, как обычно. Если возникнет проблема, вы узнаете что нужно сделать, чтобы устранить ее, так как сообщения об ошибках в Visual Studio .NET гораздо лучше, чем в предыдущих версиях.

Если вы занимаетесь отладкой на машине, где работает сервер терминала и выполнять удаленную отладку через Pipes могут несколько пользователей, ключ MSVCMON.EXE `-s <суффикс>` позволит указать уникальный суффикс для именованного канала (named pipe). Поскольку первый пользователь, начинающий удаленную отладку через Pipes, получает стандартное имя канала, следующие пользователи, которые выполняют отладку на той же машине, должны будут указать уникальный идентификатор для экземпляра MSVCMON.EXE, к которому они хотят подключиться. Запустив MSVCMON.EXE с параметром `-s`, укажите суффикс в поле Remote Machine на странице свойств Debugging диалогового окна свойств проекта, добавив суффикс к имени машины и разделив их знаком #. Так, если вы запустили `MSVCMON -s pam` на машине ZENO, надо указать имя машины как `ZENO#pam`.

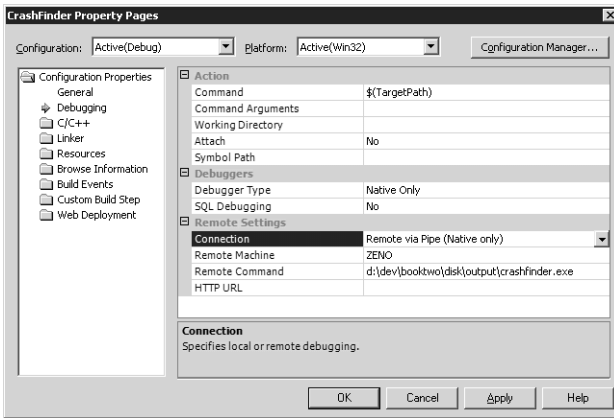


Рис. 7-6. Проект, настроенный для отладки через Pipes

Как я сказал, отладка через Pipes медленнее, хоть и безопаснее, чем отладка через TCP/IP. Если вам нужна скорость, можете включить отладку через TCP/IP с помощью ключа командной строки `-tcpip`. Чтобы уведомить о желании использовать TCP/IP, на странице свойств Debugging диалогового окна свойств проекта выберите Remote Via TCP/IP (Native Only) в разделе Remote Settings.

Для TCP/IP у MSVCMON.EXE есть несколько специальных параметров командной строки, которые, возможно, вас заинтересуют. С помощью первого — `anyuser` — вы можете позволить любому подключиться к машине без защиты, второй — `maxsessions` — указывает максимальное число сеансов отладки, допустимых в каждый момент времени, а третий — `timeout` — сообщает MSVCMON.EXE, как долго вы разрешаете ждать подключения до того, как время истечет.

Советы и уловки

В этом разделе я хочу рассказать о некоторых советах и уловках, необходимых в большинстве случаев отладки неуправляемого кода.

Отладка внедренного кода

Одна из особенностей Visual C++ .NET — новая программная модель с атрибутами. Она способна намного облегчить разработку COM, так как позволяет комбинировать IDL-атрибуты с исходным файлом, так что для создания COM-объекта требуется только один файл. Если хотите посмотреть на реальный пример COM-программирования с атрибутами, взгляните на объект Tester из главы 16. Кроме того, программирование с атрибутами предлагает интегрированный способ унифицированной обработки сообщений (unified message handling) для ваших приложений. Все атрибуты действуют путем внедрения исходного кода в ваш исходный файл.

Есть несколько способов отладки этого внедренного кода. Чтобы увидеть исходный код, находясь в отладчике, перейдите в окно Disassembly, щелкните правой кнопкой и выберите из контекстного меню команду Show Source Code. Но проще всего увидеть, что происходит с внедренным кодом, — провести компиля-

цию с ключом /Fх для CL.EXE. Его можно включить из среды Visual Studio, открыв диалоговое окно Property Pages, развернув папку C/C++, выбрав страницу свойств Output Files и установив Expand Attributed Source в Yes. Это приведет к созданию файла sourcename.MRG.CPP (где sourcename — имя исходного файла) в том же каталоге, где находится CPP-файл. Открыв этот файл, вы увидите внедренный исходный код (код слияния). При желании вы вправе также откомпилировать файл слияния, чтобы при отладке увидеть в окне исходного кода, как все это работает.

Окно Memory и автоматическое обновление

Серьезные усовершенствования в отладке неуправляемого кода представляет окно Memory. Одно из них — их больше, но главное, в окне Memory больше нет таинственного Искусственного Интеллекта, следящего за движением ваших глаз, чтобы понять, на какой адрес вы смотрели, и сдвинуть его в следующий раз, когда вы посмотрите на окно Memory. Оно также получило все виды дополнительных форматов отображения памяти, так что у вас не останется проблем с представлением памяти в том виде, который вам нужен. Чтобы выбрать формат отображения, щелкните правой кнопкой в окне Memory.

Наконец, окно Memory облегчает автоматическое обновление при изменении в просматриваемых блоках памяти. В окне Memory щелкните кнопку справа от адресного поля, и отладчик будет обновлять окно Memory, отображая последние значения. Это особенно ценно при работе с ESP (указателем стека), чтобы наблюдать за стеком по мере его изменения.

Контроль исключений

Одна из серьезнейших угроз производительности неуправляемых приложений — ненужные исключения. Поскольку исключения в неуправляемом коде требуют перехода в режим ядра при каждой инициации, их следует избегать любой ценой. Хотя переход из пользовательского режима в режим ядра выполняется относительно быстро, вся дополнительная работа, связанная с обработкой исключения в режиме ядра, занимает массу времени. Чтобы помочь сгладить эти проблемы с производительностью, окно Exception отладчика Visual Studio .NET позволяет контролировать обработку отладчиком любых исключений. Правильно поняв, как использовать это окно, вы сможете быстрее отслеживать ненужные исключения.

Прежде чем приступить к обсуждению диалогового окна Exception, я должен объяснить, что происходит, когда отладчик неуправляемого кода сталкивается с исключением. При возникновении исключения ОС приостанавливает процесс (т.е. все потоки останавливаются), указывает на точку возникновения исключения и уведомляет отладчик о его возникновении. Это первый случай исключения (first chance exception), поскольку здесь отладчик впервые получает возможность обработать его. У отладчика есть два пути: он может обработать исключение, и отлаживаемая программа ничего о нем не узнает, или он может передать исключение отлаживаемой программе. Мысль о том, что отладчик может обработать (или поглотить) исключение, может показаться странной. Однако, как вы видели в главе 4, установка точки прерывания в неуправляемом коде влечет за собой установ-

ку в этом месте команды INT 3. В случае с точкой прерывания отладчик инициирует исключение в отлаживаемой программе, так что отладчик должен обрабатывать эти исключения. Если исключение инициировано не отладчиком, он сообщает ОС, что не желает обрабатывать исключение, и оно передается отлаживаемой программе. Отладчик также выводит сообщение в окне Output, указывающее на возникновение первого случая исключения. Отлаживаемое приложение вновь запускается, и, если в нем установлена обработка исключений, исключение обрабатывается, и отлаживаемая программа радостно продолжает выполнение. Если в отлаживаемом приложении не установлена обработка исключений, исключение передается конечным обработчикам исключений (final exception handlers) из NTDLL.DLL. На этом этапе ОС вновь приостановит отлаживаемое приложение и сообщит отладчику, что возник второй случай исключения. Это значит, что процесс будет завершен из-за необрабатываемого исключения.

По обработке исключений в неуправляемом коде важно отметить, что, когда вы видите сообщение о первом случае исключения в окне Output, в вашем процессе инициировалось исключение. Как я сказал, исключения представляют угрозу производительности, поэтому если при выполнении процесса вы видите множество сообщений «First-chance exception at...», то у вас проблемы с производительностью. Коварство в том, что обработка исключений C++ реализована через механизм структурной обработки исключений (SEH) за кадром, поэтому применение исключений C++ может свести производительность на нет. Исключения предназначены для исключительных условий. В общем случае следует избегать исключений C++ в неуправляемых приложениях.

Чтобы отследить проблемы с производительностью, вызванные исключениями, можно искать исключения при анализе кода. Однако для большой кодовой базы это может стать жуткой задачей. Диалоговое окно Exceptions в Visual Studio .NET (доступное из меню Debug или нажатием Ctrl+Alt+E при использовании сочетаний клавиш по умолчанию) может сделать остановку прямо в месте возникновения исключения и поиск места его обработки тривиальной задачей (рис. 7-7).



Рис. 7-7. Диалоговое окно Exceptions

Диалоговое окно Exceptions слегка сбивает тем, что неуправляемые исключения разделены между двумя узлами верхнего уровня: C++ Exceptions и Win32 Exceptions. Параметры по умолчанию предписывают отладчику при отладке консольных приложений останавливаться только по исключениям Control-C (0x40010005) и Control-Break (0x40010008). Чтобы отладчик останавливался при инициации определенного исключения, выберите исключение в дереве и в группе When The Exception Is Thrown (т. е. в первом случае исключения) выберите Break Into The Debugger. Изображение выделенного элемента изменится на большой красный шар с крестом внутри. В диалоговом окне серые шары меньшего размера обозначают исключения, наследующие свои параметры от предков. Большой серый шар обозначает продолжение работы при первом случае исключения. Наконец, маленький красный шар говорит о том, что родительский узел прерывает работу при первом случае исключения и узел-потомок наследует от предка. Параметры исключений сохраняются для каждого решения отдельно.

Я хочу установить в узлах Win32 Exceptions и C++ Exceptions параметр Break Into The Debugger для обоих вариантов: When The Exception Is Thrown и If The Exception Is Not Handled. Тогда при инициации неуправляемого исключения любого типа процесс будет остановлен, позволив мне определить, допустимо ли это исключение. Если все или какой-то один тип исключения настроен на останов, вы увидите диалоговое окно (рис. 7-8), показывающее первый случай исключения C++. Щелкнув кнопку Break, вы окажетесь в первой функции из стека, имеющей исходный код, что обычно является именно тем местом в коде, где инициировано исключение. Если в этой точке щелкнуть Step Over или Step Into, отладчик попросит подтверждения информационным окном, спрашивая, хотите ли вы передать исключение отлаживаемой программе. Щелкните Yes, и вы сразу остановитесь в обработчике исключения. Превосходно для определения, где обрабатываются ваши исключения. Есть масса ошибок, порожденных обработкой исключений неверным обработчиком.

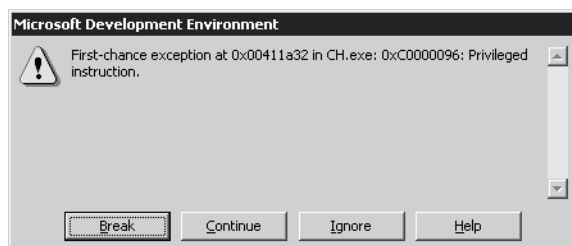


Рис. 7-8. Диалоговое окно первого случая исключения

Щелчок в диалоговом окне первого случая исключения кнопки Continue передает исключение отлаживаемой программе и продолжает выполнение. Кнопка Ignore несколько отличается и зависит от типа исключения, а также того, содержится ли оно в списке диалогового окна Exceptions. Если это серьезное исключение, сгенерированное центральным процессором (вроде нарушения доступа), щелчок кнопки Ignore приведет к попытке перезапуска команды-нарушителя, которая вновь покажет диалоговое окно первого случая исключения. Если исключение инициировано вызовом RaiseException (такое как «0xC0000008, Invalid HANDLE was specified»), выполнение продолжится, как если бы исключение не возникало

вовсе. Поскольку исключения C++ иницируются через вызов `RaiseException`, выполнение продолжится, будто `throw` не было.

Дополнительные советы по обработке символов

Как я уже говорил, усложненная обработка символов Visual Studio .NET с новым сервером символов и технологией хранения символов совершенно великолепна (см. главу 2). В отладке неуправляемого кода вы можете установить дополнительные пути для символов внутри проекта и получить для каждого проекта свое местоположение символов за пределами обычного сервера символов.

В диалоговом окне проекта `Property Pages\Configurations Properties\` на странице свойств `Debugging` есть поле `Symbol Path`, где можно указать свой путь для символов в этом проекте. Хорошая новость: он присоединяется к любым параметрам, указанным в переменной окружения `_NT_SYMBOL_PATH`, таким образом, не перезаписывая их.

Отключение от процессов Windows 2000

Сейчас вы уже знаете, что вправе отключаться от процессов при отладке в Windows XP/Server 2003. Однако, если вы еще поддерживаете Windows 2000, вы попались: начав отладку, вы будете отлаживать этот процесс до конца, что особенно утомляет при отладке прикладного серверного ПО. К счастью, в Microsoft поняли, что не все собираются разом переходить на новейшие и лучшие ОС, и выдали хорошее решение для отключения от процессов Windows 2000.

Как часть Visual Studio .NET и Remote Components Setup, устанавливается служба Win32 с именем `DBGPROXY`, представляющая прокси отладчика. Она будет запускаться в Windows 2000 как отладчик. А значит, в Windows 2000 вы сможете подключаться и отключаться от чего угодно! Запустив однажды `DBGPROXY` командой `NET START DBGPROXY`, вам больше не придется ничего делать. Visual Studio .NET автоматически творит чудо, а вы просто занимаетесь отладкой, и вам доступны функции отключения. Разумеется, если почему-либо `DBGPROXY` останавливается, то все отлаживаемые ею процессы завершаются. Настоятельно рекомендую перевести службу `DBGPROXY` на автоматический запуск, чтобы использовать ее преимущества!

Обработка дамп-файлов

Вернемся в главу 3, где я рассказывал о диалоговом окне `SUPERASSERT` и его кнопке `Create Mini Dump`, позволяющей сохранять на диск текущее состояние процесса, чтобы иметь возможность загрузить его в отладчик позже. Visual Studio .NET позволяет легко читать любые созданные дамп-файлы. Открыть дамп-файл в Visual Studio .NET так же просто, как открыть обычное решение.

Запустив Visual Studio .NET, выберите `Open Solution` из меню `File`. В диалоговом окне `Open Solution` перейдите в каталог, где хранится ваш дамп-файл. Дамп-файлы обычно имеют расширение `DMP`, которое есть в поле со списком `Files Of Type`, так что можете выбрать его или ввести `*.DMP` в поле ввода `File Name`. Выберите `DMP`-файл и щелкните кнопку `Open`. Как всегда, Visual Studio .NET создаст

вездесущий файл решения, необходимый для любых действий внутри среды. Нажав любую из клавиш отладчика — Step Into, Step или Debug, вы получите приглашение для записи решения и загрузки дамп-файла.

Если вы работаете на той машине, где был создан дамп-файл, и ваши двоичные файлы скомпилированы, Visual Studio .NET автоматически найдет исходный код и символы, соответствующие дамп-файлу. Чтобы присоединить операционные символы, установите `_NT_SYMBOL_PATH`, включив туда место хранения символов, или, запустив отладку, откройте окно Modules, щелкните правой кнопкой модули без символов и укажите нужные символы.

Чтобы помочь отладчику определить где найти модули, есть два способа. Проще всего — указать каталоги модулей в переменной окружения `MODPATH`: добавьте каждый каталог, отделяя его точкой с запятой, к переменной окружения `MODPATH`, как в переменной окружения `PATH`. Если хотите установить путь поиска модулей глобально, укажите его в `SZ_REG`-параметре `GlobalModPath` в любом из следующих разделов реестра. Если хотите сделать путь доступным всем пользователям этой машины, используйте раздел `HKEY_LOCAL_MACHINE`.

```
HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\7.1\NativeDE\Dumps\  
HKEY_LOCAL_MACHINE\Software\Microsoft\VisualStudio\7.1\NativeDE\Dumps\
```

К сожалению, отладчик Visual Studio .NET не считывает двоичные файлы напрямую у сервера символов в отличие от WinDBG. Так что, если вы работаете с дамп-файлами с клиентских сайтов, вам, вероятно, лучше использовать WinDBG (см. главу 8).

Кроме открытия дамп-файлов, Visual Studio .NET может их создавать. На первый взгляд, это не так уж важно, но с точки зрения Bugslayer, мы получаем еще одну прекрасную технологию решения проблем. Создавая дамп-файлы на разных этапах напряженного сеанса отладки, вы получаете посмертный след, по которому можно выйти к проблеме. Это дает прекрасную возможность фиксировать состояния программы для демонстрации другим членам команды и оценки поведения во времени. Я дошел до того, что в процессе отладки делаю снимки так часто, что могу быстро заполнить все жесткие диски. Но жесткие диски — это небольшая плата в сравнении с возможной стоимостью неустраненной ошибки.

Записывать дамп-файлы в ходе сеанса отладки очень просто. Из нижней части меню Debug выберите команду Save Dump As. В появившемся диалоговом окне File Save можно указать точное место создания дамп-файла. Visual Studio .NET позволяет записывать два вида дамп-файлов. Первый в Visual Studio .NET называется *минидамп* (minidump). Он содержит информацию о версии ОС, о стеке всех потоков и о версии файлов каждого модуля, загруженного в процесс. Обычные минидамп-файлы довольно малы. При написании этого материала я подключил отладчик к Word 2002 и размер минидамп-файла составил всего 38 Кб, тогда как рабочий набор занимал около 16 Мб.

Второй тип записываемого дамп-файла — *минидамп с кучей* (minidump with heap) — сохраняет ту же информацию, что и минидамп, вместе со всей выделенной памятью в процессе. С таким форматом дамп-файла можно отслеживать значения указателей по всему адресному пространству. Разумеется, за эту дополни-

тельную информацию приходится платить гораздо больше. Минидамп с кучей для Microsoft Word 2002, в который загружена вся эта глава, занимает 96 Мб! Как правило, я придерживаюсь обычной версии минидампа, потому что мне не часто приходится проследивать разные уровни указателей. Но приятно знать, что он есть.

Увы, Visual Studio .NET не позволяет записывать дампы в самом полезном формате — *минидамп с описателями* (minidump with handles). Это можно объяснить тем, что, раз Visual Studio .NET не включает средств просмотра информации описателей, нет причин записывать эту информацию. Однако, как вы увидите в главе 8, WinDBG позволяет просматривать информацию описателей. В наличии информации описателей заключается разница между отслеживанием многопоточной взаимной блокировки и неспособностью решить эту проблему. Поскольку эта информация так важна, диалоговое окно SUPERASSERT сохраняет ее.

Не забудьте прочитать о работе с дампы-файлами в WinDBG. Хотя WinDBG сложнее в использовании, этот отладчик лучше приспособлен к считыванию дампы-файлов, полученных от клиентов, в основном потому, что может загружать двоичные файлы вне хранилища символов. С помощью его улучшенной обработки символов и более информативных команд вам будет проще определить причину проблем у клиентов.

Стандартный вопрос отладки

Как устанавливать точки прерывания в еще не загруженных DLL?

Большая проблема Visual Studio 6 заключалась в том, что попытка установить точку прерывания в DLL, которая была динамически загружена в диалоговом окне Additional DLL, была, мягко говоря, катастрофой. Особенно затруднялась удаленная отладка. Однако вы могли даже не заметить отличия в Visual Studio .NET, так как Microsoft исправила точки прерывания так, что они автоматически взводятся при входе модуля, содержащего исходный файл, в адресное пространство. В окне Breakpoint невзведенные точки прерывания обозначаются белым знаком вопроса в красной точке. Диалоговое окно Additional DLL пропало, и скатертью дорога!

Язык ассемблера x86

Когда неуправляемое приложение терпит крах, истинная разница между исправлением ошибок и причитаниями очень часто определяется тем, насколько хорошо вы понимаете язык ассемблера. Всем нам хотелось бы, чтобы ошибки имели место только в тех модулях, для которых у нас есть исходный код и полный стек вызовов, но это далеко не всегда так. При возникновении ошибки нам обычно остается только смотреть в окно Disassembly (дизассемблированный код) отладчика Visual Studio .NET и пытаться хотя бы определить, в каком месте программы мы находимся, не говоря уж о поиске причин проблемы.

Я ни в коем случае не утверждаю, что вам нужно знать ассемблер настолько хорошо, чтобы писать все программы с использованием Microsoft Macro Assembler (MASM) — его нужно знать на таком уровне, чтобы легко его понимать. Цель дан-

ного раздела — предоставить вам всю информацию, необходимую для получения рабочего знания языка ассемблера. По окончании чтения этого раздела и нескольких часов практики ваши знания ассемблера будут более чем достаточными. Потраченное время может оказаться тем самым фактором, который позволит вам на самом деле исправлять ошибки, а не блуждать в отладчике, практикуясь в нецензурном словоупотреблении. Если вы ранее программировали на ассемблере, помните, что вся информация, представленная в этом разделе, касается того, что вы будете видеть в окне Disassembly. Возможно, вам известны более эффективные способы выполнения некоторых действий, но это не имеет особого значения. Важно познакомиться с тем, как язык ассемблера выглядит в отладчике Visual Studio .NET.

Программисты иногда относятся к ассемблеру с настороженностью: им кажется, что над ним потрудились темные силы. Однако на самом деле в ассемблере нет ничего мистического; просто нужно помнить, что одна ассемблерная команда выполняет одно и только одно действие. Стоит изучить основы ассемблера и понять, как процессор выполняет команды, и вы осознаете, что на самом деле этот язык довольно элегантен. Если вам захочется увидеть настоящую черную магию, изучите любую программу, интенсивно работающую с STL. Магические встраиваемые функции STL могут приводить к вызову 30-40 других функций и невероятно большому числу допущений. Лично мне STL иногда кажется гораздо более загадочной, чем ассемблер.

Познакомив вас с ассемблером, я вернусь к отладчику Visual Studio .NET и расскажу, как выжить в окне Disassembly. Например, я опишу способы просмотра параметров в стеке и навигацию в окне Disassembly. Кроме того, я объясню связь между окнами Memory и Disassembly и познакомлю вас с советами и уловками, которые помогут отлаживать программы на уровне ассемблера.

Прежде чем мы рассмотрим ассемблер, я должен вас кое о чем предупредить. Возможно, некоторым из вас ассемблер понравится настолько, что вы захотите писать на нем свои программы. Это прекрасно, но так вы поставите под угрозу свою карьеру. Ваши начальники уже говорили со мной и просили, чтобы вы не начинали разрабатывать на ассемблере все, что можно. Ассемблер не является платформенно-независимым языком, что может заметно затруднить сопровождение написанных на нем программ.

Основы архитектуры процессоров

Набор команд процессоров Intel берет начало от процессора 8086, представленного компанией Intel в 1978 году. Во времена MS-DOS и 16-разрядных ОС Microsoft Windows язык ассемблера был несколько странным и сложным в использовании, потому что с памятью можно было работать только посредством 64-килобайтных блоков — сегментов. К счастью, современные ОС Windows предоставляют процессору прямой доступ ко всему адресному пространству, а значит, использовать ассемблер стало гораздо легче.

В этом разделе я опишу базовый набор 32-разрядных команд, поддерживаемый всеми процессорами Intel и AMD с архитектурой x86; иногда его еще обозначают как IA32. Расширенные возможности процессоров Intel Pentium, например набор команд MMX, применяются в Windows редко, и вам почти не придется сталкиваться с ними. Я не буду рассматривать по-настоящему мудреные аспекты

ассемблерных команд, такие как байты ModR/M и SIB, определяющие способы доступа к памяти. В этой главе под доступом к памяти я понимаю просто доступ к памяти и ничего больше. Я также не буду описывать команды работы с числами с плавающей точкой. Операции с устройством для выполнения команд с плавающей точкой (Floating-point unit, FPU) процессоров Intel аналогичны обычным командам. Основные различия между ними в том, что FPU имеет свой набор регистров и для выполнения команд с плавающей точкой используется архитектура, основанная на регистровом стеке. Если эта глава пробудит в вас интерес к процессорам семейства Intel (а я надеюсь, что так оно и случится), загрузите с сайта www.intel.com три тома документации Intel Architecture Software Developer's Manual (Руководство по разработке ПО для процессоров с архитектурой Intel), представленной в формате PDF. В нашем случае самым важным является том 2, Instruction Set Reference (Описание набора команд). В томах 1 и 3 приводится базовая информация об архитектуре процессоров и сведения для разработчиков ОС соответственно. Вы даже можете бесплатно получить эти тома в бумажном формате, просто позвонив по телефону. Скорее всего они вам не понадобятся, но, поставив их на полку, вы, несомненно, станете чувствовать себя гораздо более умным!

Необходимо помнить, что процессоры x86 очень гибки и предоставляют много способов выполнения похожих операций. К нашему счастью, компиляторы Microsoft достаточно успешно выбирают самый быстрый способ и используют соответствующую конструкцию во всех возможных случаях, так что понять назначение отдельных разделов кода становится гораздо легче. Ниже я опишу команды языка ассемблера, которые чаще всего встречаются в коде программ. Если вас заинтересуют другие ассемблерные команды, обратитесь к руководству Intel.

Регистры

Сначала мне хотелось бы рассказать про регистры процессора. Каждый бит данных программы рано или поздно попадает в регистры, поэтому, если вы будете понимать назначение каждого регистра, вам будет легче узнать, когда программа начинает плохо себя вести. У процессоров x86 восемь регистров общего назначения (EAX, EBX, ECX, EDX, ESI, EDI, ESP и EBP), шесть сегментных (CS, DS, ES, SS, FS и GS), регистр указателя команд (EIP) и флагов (EFLAGS). В процессоре есть и другие регистры, например, отладочные и управляющие регистры, но они имеют специальное назначение, и при отладке в обычном пользовательском режиме вы с ними не столкнетесь. Схема регистра общего назначения показана на рис. 7-9. Помните: некоторые 32-разрядные регистры позволяют обращаться к отдельным их частям. Описание всех регистров общего назначения приведено в табл. 7-6. Единственный сегментный регистр, представляющий для нас интерес, — FS — содержит адрес блока информации о потоке (Thread information block, TIB), выполняющемся в данный момент. Конечно, ОС использует и другие сегментные регистры, но она настраивает их так, чтобы они были «прозрачны» для обычных операций. Регистр указателя команд содержит адрес выполняемой в данный момент команды.

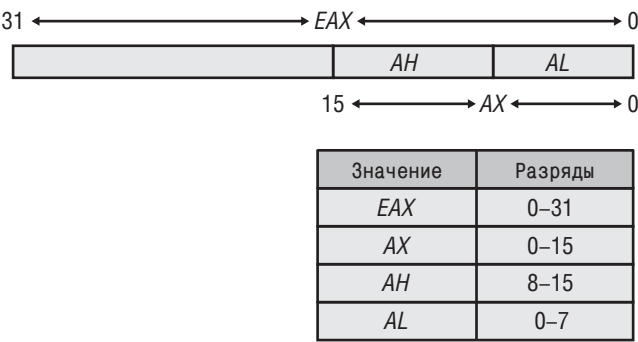


Рис. 7-9. Схема регистра общего назначения

Регистр флагов EFLAGS содержит флаги статуса и флаги управления. Биты регистра EFLAGS характеризуют результат выполнения команд. Так, флаг нуля ZF (Zero Flag) устанавливается в 1, если в результате выполнения команды было получено значение 0. В главе 4 я рассказывал, как перевести процессор в пошаговый режим; если помните, это было связано с установкой флага ловушки TF (Trap Flag) в регистре EFLAGS. На рис. 7-10 показано окно Registers (регистры) отладчика Visual Studio .NET. В этом окне регистр EFLAGS называется EFL. Заметьте: я не показываю в окне Registers регистры для работы с числами с плавающей точкой и другие регистры специального назначения, такие как MMX или 3DNow! Чтобы выбрать интересующие вас регистры, щелкните в окне Registers правой кнопкой и укажите соответствующий пункт в появившемся контекстном меню.

Табл. 7-6. Регистры общего назначения

32-разрядный регистр	Доступ к 16 битам	Доступ к младшему байту (биты 0-7)	Доступ к старшему байту (биты 8-15)	Специальные функции
EAX	AX	AL	AH	В этом регистре сохраняются возвращаемые функциями целочисленные значения.
EBX	BX	BL	BH	Команда Loop использует этот регистр для подсчета итераций цикла.
ECX	CX	CL	CH	
EDX	DX	DL	DH	В этом регистре хранятся 32 старших разряда 64-разрядных значений.
ESI	SI			При выполнении команд перемещения или сравнения блоков памяти в этом регистре хранится адрес источника.
EDI	DI			При выполнении команд перемещения или сравнения блоков памяти в этом регистре хранится адрес назначения.

см. след. стр.

Табл. 7-6. Регистры общего назначения *(продолжение)*

32-разрядный регистр	Доступ к 16 битам	Доступ к младшему байту (биты 0-7)	Доступ к старшему байту (биты 8-15)	Специальные функции
ESP	SP			Указатель стека. Этот регистр неявно изменяется при вызове функций, возвращении из функций, выделении в стеке места для локальных переменных и очистке стека.
EBP	BP			Указатель базы/кадра стека. Этот регистр содержит адрес кадра стека процедуры.

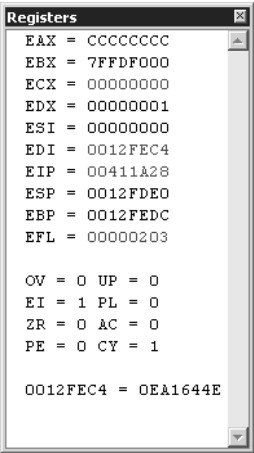


Рис. 7-10. Окно Registers среды Visual Studio .NET

Значения флагов, показанных в окне Registers, описаны в табл. 7-7. В документации к Visual Studio .NET названия флагов в окне Registers не обсуждаются, поэтому вполне возможно, что их обозначения вам незнакомы. К сожалению, мнемонические обозначения этих флагов в Visual Studio .NET не соответствуют обозначениям Intel, поэтому при чтении документации Intel вам придется выполнять некоторые сопоставления. Отмечу также, что окно Registers в Visual Studio .NET стало поддерживать одну очень полезную функцию: измененные в результате выполнения команды флаги выделяются красным цветом. В предыдущих версиях Visual Studio измененные флаги ничем не отличались от остальных, поэтому следить за изменениями было довольно сложно. К нашему всеобщему удовольствию, при отладке машинного кода наблюдать за флагами почти не приходится.

Хотя окно Registers выглядит, как обычное текстовое окно, такое как Output, вы можете редактировать содержащиеся в нем значения. Просто наведите указатель мыши на значение нужного вам регистра, расположенное справа от знака равенства, и введите измененный вариант. Ввод нового значения начинается с места расположения курсора. Окно Registers также поддерживает функцию Undo.

Табл. 7-7. Значения флагов окна Registers

Флаг в окне Registers	Значение	Обозначение флага в документации Intel	Описание
OV	Флаг переполнения	OF	Устанавливается в 1, если команда привела к целочисленному переполнению.
UP	Флаг направления	DF	Имеет значение 1, если команды работы со строками обрабатываются в порядке от старшего адреса к младшему (автодекремент), 0 — если команды работы со строками выполняются в порядке от младшего адреса к старшему (автоинкремент). При генерировании кода C/C++ всегда реализуется второй вариант.
EI	Флаг разрешения прерывания	IF	Значение 1 показывает, что прерывания разрешены. При отладке в пользовательском режиме этот флаг всегда равен 1, так как запрещение прерываний, кроме всего прочего, блокирует клавиатуру и обновление экрана.
PL	Флаг знака	SF	Содержит значение старшего бита результата выполнения команды. Значение 0 показывает, что результат положителен, 1 — отрицателен.
ZR	Флаг нуля	ZF	Имеет значение 1, если в результате выполнения команды был получен 0. Этот флаг очень важен для команд сравнения.
AC	Вспомогательный флаг переноса	AF	Устанавливается в 1 при возникновении переноса или заема в результате операций в двоично-десятичном формате (binary-coded decimal, BCD).
PE	Флаг четности	PF	Устанавливается в 1, если младший байт результата содержит четное число единичных битов.
CY	Флаг переноса	CF	Имеет значение 1, если арифметическая операция привела к переносу или заему старшего бита результата. Также устанавливается в 1 при переполнении в случае беззнаковых целочисленных арифметических операций.

Формат команд и адресация памяти

Все команды процессоров Intel имеют такой базовый формат:

[префикс] команда [операнды]

Как правило, вы будете встречать префиксы только перед некоторыми командами работы со строками (эти случаи я опишу в разделе «Манипуляции со строками»). Представленный формат операндов указывает на направление операции: от источника к приемнику, поэтому читайте операнды в порядке справа налево.

Команда с одним операндом: XXX источник

Команда с двумя операндами: XXX приемник, источник

Чтобы вам было легче привыкнуть к этому формату, укажите пальцем на второй операнд (источник) и переместите палец влево, пока он не укажет на операнд-приемник. Перемещая палец, говорите «от источника к приемнику». Именно так я делаю, когда читаю ассемблерный код. Источник и приемник путают очень многие люди, начинающие изучать ассемблер Intel, так что трюк с пальцем на самом деле может оказаться полезным. Кое-кто говорил мне, что чтение ассемблерного кода облегчается, если вместо запятой между источником и приемником представлять знак равенства. Однако порядок выполнения операций все равно воспринимается как обратный, если только вы не привыкли читать книги, написанные на арабском языке или иврите.

Операнд источника может быть регистром, ссылкой на память или непосредственным — т. е. «жестко закодированным» — значением. Операнд приемника может быть регистром или ссылкой на память. Процессоры Intel не позволяют, чтобы и источник, и приемник являлись ссылками на память.

Ссылки на память представляют собой операнды, заключенные в квадратные скобки. Так, ссылка [0040129Ah] говорит «получить значение ячейки памяти, расположенной по адресу 0x0040129A». Буква «h» в ассемблере свидетельствует о том, что число представлено в шестнадцатеричном виде. Ссылка [0040129Ah] аналогична разыменованию указателя на целое число в языке C. Ссылаться на память можно через регистры: например, операнд [EAX] означает «получить значение памяти по адресу, указанному в регистре EAX». Очень часто адрес обращения к памяти вычисляется иначе: путем сложения значения регистра и некоторого смещения. Так, операнд [EAX+0Ch] означает «добавить 0xC к значению в регистре EAX и получить значение памяти, расположенное по получившемуся адресу». Некоторые ссылки на память могут быть довольно сложными, например [EAX+EBX*2], в формировании которой участвуют два регистра.

Чтобы процессору было ясно, к какому объему памяти обратиться, в ряде случаев ссылке предшествует спецификатор указателя. Спецификаторы указателя обозначаются как BYTE PTR, WORD PTR и DWORD PTR для байта, слова и двойного слова соответственно. Можете считать их аналогами приведения типов в C++. Если в дизассемблированном коде спецификатор указателя отсутствует, значит, выполняется обращение к двойному слову.

Иногда ссылка на память проста, и вы можете легко узнать, к какому адресу она обращается. Например, ссылка [EBX] — это просто ссылка на ячейку памяти, адрес которой содержится в регистре EBX, поэтому, чтобы увидеть значение памяти, можно просто открыть окно Memory и ввести EBX. Однако иногда для вычисления ссылки на память приходится проводить сложное умножение шестнадцатеричных чисел. К счастью, окно Registers покажет, на какую ячейку памяти ссылается команда.

Обратите внимание на строку 0012FEC4 = 0EA1644E в нижней части рис. 7-10. Эта строка отображает эффективный адрес. Текущая команда, в данном случае расположенная по адресу 0x00411A28, ссылается на адрес 0x0012FEC4, указанный в левой части строки. Справа находится значение памяти по этому адресу, равное 0x0EA1644E. В окне Registers отображаются эффективные адреса только тех команд, которые осуществляют доступ к памяти. Так как процессоры x86 не позволяют, чтобы оба операнда были ссылками на память, то, наблюдая за строкой эффективного адреса, можно легко узнать, к какому адресу памяти обращается программа и какое значение там находится.

Если доступ к памяти невозможен, процессор генерирует или общую ошибку защиты (General Protection Fault, GPF), или ошибку страницы. GPF говорит о том, что вы пытаетесь обратиться к области памяти, к которой не имеете доступа. Ошибка страницы возникает при попытке доступа к отсутствующей в памяти странице. Если какая-то строка ассемблерного кода вызывает ошибку, стоит поинтересоваться, к какой области памяти она обращается (если, конечно, вообще обращается). Это укажет вам на неправильные значения. Так, если ссылка на память равна [EAX], нужно посмотреть, какое значение содержится в регистре EAX. Если EAX содержит неверный адрес, нужно пролистать ассемблерный код вверх и определить, какая команда поместила в регистр EAX это ошибочное значение. Возможно, для обнаружения этой команды придется возвращаться на несколько вызовов. О перемещении по стеку вручную я расскажу ниже.

Кое-какие сведения о встроенном ассемблере Visual C++ .NET

Прежде чем я начну описывать ассемблерные команды, мне хотелось бы рассказать немного о встроенном ассемблере Visual C++. Как и большинство профессиональных компиляторов C++, компилятор Visual C++ позволяет включать ассемблерные команды прямо в код C и C++. Использовать встроенный ассемблер обычно не рекомендуется, так как это ухудшает машинную независимость кода, но иногда это единственный выход. В главе 15 я покажу, как при помощи встроенного ассемблера устанавливать ловушки (hook) для импортируемых функций.

Выше я уже говорил, что вам не обязательно уметь писать программы на ассемблере, и я себе не противоречу. Изучение встроенного ассемблера — это не то же самое, что обучение разработке всей программы на MASM: инфраструктуру приложения все равно будет обеспечивать код C/C++. Можете считать встроенный ассемблер программным эквивалентом функции Zoom (увеличить). Например, в начале работы над растровым изображением вы рисуете крупными мазками; когда дело доходит до заключительных штрихов, вы увеличиваете изображение, чтобы можно было контролировать отдельные пиксели. То же самое имеет место и в случае встроенного ассемблера: вы «рисуете» программу крупными мазками C/C++ и увеличиваете ее, когда нужен контроль над отдельными ассемблерными командами. Я рассматриваю именно встроенный ассемблер потому, что для описания странного синтаксиса директив MASM могло потребоваться бы более 100 страниц; к тому же встроенный ассемблер гораздо проще для понимания. Наконец, встроенный ассемблер позволяет вам попрактиковаться с описываемыми мной командами и своими глазами увидеть, что они делают.

Чтобы проиллюстрировать формат встроенного ассемблера, я должен познакомить вас с вашей первой командой:

NOP Нет операции

Команда `NOP` не делает ничего. Компилятор иногда включает ее в функции, чтобы выполнить их выравнивание по определенной границе памяти.

Для вызова встроенного ассемблера используется ключевое слово `__asm`. После него ввести ассемблерную команду, которую нужно выполнить. Если вы не хотите заработать туннельный синдром, то можете просто написать `__asm` и включить в фигурные скобки столько команд, сколько душе угодно. Формат команд встроенного ассемблера я проиллюстрирую на примере двух следующих функций. Эти фрагменты кода функционально эквивалентны.

```
void NOPFuncOne ( void )
{
    __asm NOP
    __asm NOP
}
```

```
void NOPFuncTwo ( void )
{
    __asm
    {
        NOP
        NOP
    }
}
```

Для пояснения ассемблерных операций, таких как доступ к параметрам и переменным, в этой главе я буду использовать встроенный ассемблер. Если вы желаете увидеть, как работает каждая команда, запустите программу `ASMer`, входящую в число файлов к книге. Эта программа включает все последующие ассемблерные примеры.

Команды, которые нужно знать

Процессоры Intel поддерживают массу команд; так, глава документации Intel Instruction Set Reference для процессора Pentium Xeon занимает 854 страницы. Это не означает, что существует 854 команды, просто их описание занимает 854 страницы. К счастью, большинство этих команд не встречается в программах, работающих в пользовательском режиме, поэтому вам не придется изучать их. Я опишу только те, что часто используются генератором кода Microsoft, и ситуации, в которых вы будете обычно сталкиваться с ними. Рассказывая о командах, я буду описывать какую-нибудь их группу и приводить поясняющий их фрагмент кода. Кроме того, весь код на ассемблере будет представлен в том виде, в каком он выводится в окно Disassembly среды Visual Studio .NET. Так вы сможете привыкнуть к реальному ассемблеру, с которым будете работать.

Манипуляции со стеком

PUSH Поместить в стек слово или двойное слово
POP Извлечь значение из стека

Процессоры Intel интенсивно используют стек. Другие процессоры, имеющие гораздо больше регистров, могут передавать параметры в функции в регистрах, но процессоры Intel вынуждены передавать большинство параметров через стек. Стек начинается в старшей области памяти и «растет» вниз. Обе эти команды неявно изменяют регистр ESP, содержащий адрес текущей вершины стека. После выполнения команды PUSH значение регистра ESP уменьшается. В результате команды POP значение регистра ESP увеличивается.

Вы можете помещать в стек значения регистров, ячеек памяти и жестко закодированные числа. При извлечении значения из стека оно обычно помещается в регистр. Стек процессора в первую очередь характеризуется тем, что представляет собой структуру данных типа «последним вошел — первым вышел» (last in, first out; LIFO); если вы помещаете в стек три регистра, чтобы сохранить их значения, извлекать их нужно в обратном порядке:

```
void PushPop ( void )
{
    __asm
    {
        // Сохранение значений регистров EAX, ECX и EDX.
        PUSH EAX
        PUSH ECX
        PUSH EDX

        // Некоторые действия, которые могут
        // изменить значения указанных регистров.

        // Восстановление ранее сохраненных регистров.
        // Заметьте: они извлекаются из стека в порядке LIFO.
        POP EDX
        POP ECX
        POP EAX
    }
}
```

Есть гораздо более эффективные способы обмена значений регистров, однако команды PUSH и POP также позволяют сделать это. Просто нужно выполнить команды POP в обратном порядке:

```
void SwapRegistersWithPushAndPop ( void )
{
    __asm
    {
        // Обмен значений регистров EAX и EBX при помощи стека. Обратите
        // внимание на последовательность команд PUSH и POP.
        PUSH EAX
        PUSH EBX
```

```

        POP EAX
        POP EBX
    }
}

```

PUSHAD Поместить в стек все регистры общего назначения
POPAD Извлечь из стека все регистры общего назначения

Время от времени вы будете сталкиваться с этими командами при отладке системного кода. Вместо того чтобы писать целый ряд команд PUSH для сохранения всех регистров общего назначения и аналогичной последовательности команд POP для их извлечения, процессоры Intel позволяют сохранить и восстановить все регистры общего назначения при помощи команд PUSHAD и POPAD.

Очень распространенные простые команды

MOV Перемещение данных

Команду MOV процессор выполняет чаще всего, потому что она позволяет переместить значение из одного места в другое. Только что я показал, как поменять местами значения двух регистров командами PUSH и POP; сейчас я сделаю то же самое командой MOV:

```

void SwapRegisters ( void )
{
    __asm
    {
        // Регистр EAX выполняет функцию временного хранилища, поэтому
        // я помещаю его в стек, чтобы не потерять его значение.
        // Обмен значений между регистрами ECX и EBX.
        PUSH EAX
        MOV EAX , ECX
        MOV ECX , EBX
        MOV EBX , EAX
        POP EAX
    }
}

```

SUB Вычитание

Команда SUB выполняет вычитание. При этом операнд источника вычитается из операнда приемника, и результат помещается в операнд приемника.

```

void SubtractExample ( void )
{
    __asm
    {
        // Задаем значения регистров и выполняем вычитание. Этому примеру
        // соответствует формула: EAX = Значение(EAX) - Значение(EBX).
        MOV EAX , 5
        MOV EBX , 2
        SUB EAX , EBX
    }
}

```


После выполнения этого фрагмента кода в регистре EAX будет находиться значение 3, а в EBX — 2.

ADD Сложение

Команда ADD прибавляет операнд источника к операнду приемника и сохраняет результат в операнде приемника.

INT 3 Точка прерывания

INT 3 является в процессорах Intel командой точки прерывания. Компиляторы Microsoft используют эту команду для заполнения пространства между функциями в файле. Это нужно для выравнивания разделов PE-файлов (portable executable) в соответствии с ключом компоновщика /ALIGN, который по умолчанию имеет значение 4 кб. Идентификатор операции, т. е. шестнадцатеричное число, соответствующее команде INT 3, равен 0xCC, вот почему она используется для заполнения, а также для инициализации переменных в стеке при помощи ключа /RTCS.

LEAVE Высокоуровневый выход из процедуры

Команда LEAVE восстанавливает при выходе из функции состояние процессора. Подробнее я расскажу об этом в следующем разделе.

Частая последовательность команд: вход в функцию и выход из функции

Вход и выход из большинства функций Windows и ваших программ выполняется одинаково. Код входа в функцию называется прологом, выхода — эпилогом; оба фрагмента компилятор генерирует автоматически. Код пролога выполняет действия, обеспечивающие доступ к локальным переменным и параметрам. Область памяти для хранения локальных переменных и параметров называется кадром стека. Процессоры x86 не навязывают никакой схемы кадра стека, однако из-за особенностей процессоров и некоторых команд ОС удобнее всего хранить указатель кадра стека в регистре EBP.

```
__asm
{
    // Стандартный пролог
    PUSH EBP           // Сохранение регистра кадра стека.
    MOV  EBP, ESP      // Локальный кадр стека функции начинается по адресу ESP.
    SUB  ESP, 20h      // В стеке выделяются 0x20 байт для хранения локальных
                        // переменных. Команда SUB встречается, только когда
                        // функция имеет локальные переменные.
}
```

Такой фрагмент часто встречается и в отладочных, и в заключительных компоновках. Однако в некоторых функциях заключительных компоновок между командами PUSH и MOV могут располагаться и другие команды. Процессоры с несколькими конвейерами — например, Pentium — могут декодировать несколько команд одновременно, поэтому оптимизатор пытается так организовать поток команд, чтобы задействовать все преимущества этой возможности.

В зависимости от вида оптимизации, заданного при компиляции программы, в некоторых функциях в качестве указателя кадра стека может применяться не EBP, а другой регистр. Такие процедуры поддерживают то, что называется данными о кадре стека с отсутствующим указателем (frame pointer omission, FPO). Дизассемблированный код функций с данными FPO выглядит так, будто в них выполняется работа с данными. В следующем разделе я расскажу, как узнать такие функции.

Следующий распространенный эпилог аннулирует операции пролога, и именно с этим вариантом вы будете чаще всего встречаться в отладочных компоновках. Этот эпилог соответствует приведенному ранее прологу.

```
__asm
{
    // Стандартный эпилог
    MOV ESP , EBP    // Восстановление регистра стека.
    POP EBP         // Восстановление сохраненного регистра кадра стека.
}
```

Описанная выше команда LEAVE выполняется быстрее, чем пара MOV/POP, поэтому в заключительных компоновках эпилог может состоять только из нее. Команда LEAVE идентична последовательности MOV/POP. В отладочные компоновки компиляторы по умолчанию включают пару MOV/POP. Интересно отметить, что процессоры x86 имеют аналогичную команду ENTER для пролога, но она выполняется медленнее, чем последовательность PUSH/MOV/ADD, поэтому компиляторы ее не используют.

Генерирование кода компиляторами во многом зависит от типа оптимизации. Если вы оптимизируете программу для получения минимального размера (см. главу 2), то во многих функциях будут задействованы стандартные кадры стека. Оптимизация для скорости приведет к генерированию более замысловатых функций с данными FPO.

Манипуляции с указателями

LEA Загрузить эффективный адрес

Команда LEA загружает в регистр приемника адрес операнда источника; это почти всегда свидетельствует о доступе к локальной переменной. В следующем фрагменте приведены два примера использования LEA. В первом случае показано, как указателю присваивается адрес целочисленной переменной, во втором при помощи команды LEA осуществляется получение адреса локального массива символов (тип char), после чего этот адрес передается в API-функцию GetWindowsDirectory.

```
void LEAExamples ( void )
{
    int * pInt ;
    int iVal ;

    // Следующая пара команд идентична.
    // коду pInt = &iVal; языка C.
    __asm
    {
```

```

    LEA EAX , iVal
    MOV [pInt] , EAX
}

////////////////////////////////////

char szBuff [ MAX_PATH ] ;

// Другой пример доступа к указателю при помощи LEA. Эта
// последовательность команд идентична вызову на языке C
// функции GetWindowsDirectory ( szBuff , MAX_PATH ) ;.
__asm
{
    PUSH 104h          // Помещаем в стек MAX_PATH как второй параметр.
    LEA ECX , szBuff    // Получаем адрес переменной szBuff.
    PUSH ECX           // Помещаем в стек адрес szBuff в качестве
                        // первого параметра.
    CALL DWORD PTR [GetWindowsDirectory]
}
}

```

Вызов процедур и возврат из них

CALL **Вызов процедуры**
RET **Возврат из процедуры**

Прежде чем я смогу начать обсуждение того, как и по каким адресам получать доступ к параметрам и локальным переменным, мне нужно рассказать о вызове функций и возврате из них. Команда **CALL** очень проста. При выполнении **CALL** в стек неявно помещается адрес возврата из процедуры, поэтому если вы остановитесь в отладчике на первой команде вызванной процедуры и узнаете при помощи регистра **ESP** значение, находящееся на вершине стека, оно и будет адресом возврата.

Операндом команды **CALL** может быть почти все, что угодно, и если вы получите изучите информацию в окне **Disassembly**, то сможете найти вызовы процедур при помощи регистров, ссылок на память, параметров и глобальных смещений. Если операндом **CALL** является ссылка на память, вызываемую процедуру можно определить по полю эффективного адреса окна **Registers**.

Вызов локальной функции — это прямой вызов по адресу. Однако довольно часто вам будут встречаться и вызовы через указатели; обычно это вызовы виртуальных функций или функций, импортируемых через таблицу адресов импорта (**import address table, IAT**). Если для изучаемого бинарного файла загружены символы, вы увидите что-то похожее на первую команду **CALL** из приведенного ниже фрагмента **CallSomeFunctions**. Этот код показывает, что вызов осуществляется именно через **IAT**. Префикс **__imp__** — вот что об этом говорит. Пример **CallSomeFunctions** также содержит вызов локальной функции. В комментариях я указываю, что может быть выведено в окне **Disassembly** в зависимости от того, загружены символы или нет.

```

void CallSomeFunctions ( void )
{
    __asm
    {
        // Вызов импортируемой функции GetLastError, не принимающей никаких
        // параметров. Возвращаемое значение будет находиться в регистре EAX.
        // Вызов выполняется через IAT, поэтому используется указатель.
        CALL DWORD PTR [GetLastError]

        // Если символы загружены, в окне Disassembly будет выведено:
        // CALL DWORD PTR [__imp__GetLastError@0 (00402000)].

        // Если символы не загружены, в окне Disassembly будет выведено:
        // CALL DWORD PTR [00402000].

        //////////////////////////////////////
        // Вызов функции, находящейся в том же файле (локальной функции).
        CALL NOPFuncOne

        // Если символы загружены, в окне Disassembly будет выведено:
        // CALL NOPFuncOne (00401000).

        // Если символы не загружены, в окне Disassembly будет выведено:
        // CALL 00401000.
    }
}

```

Команда RET выполняет возврат в вызывающую функцию при помощи адреса, находящегося на вершине стека, не выполняя при этом никакой проверки. Этот недочет используется в атаках типа «переполнение буфера», при которых адрес возврата изменяется так, чтобы функция возвращалась в код злоумышленника. Как вы можете представить, искаженный стек может содержать любые адреса возврата. За командой RET иногда следует число. Оно показывает, сколько байт извлечь из стека, чтобы достичь соответствия числу параметров, помещенных в стек при вызове функции.

Соглашения вызова

При обсуждении команд CALL и RET я слегка затронул параметры. Для понимания параметров нужно разобраться с соглашениями вызова. Описанные мной в предыдущем разделе команды помогут вам освоить некоторые полезные отладочные приемы, но, чтобы перейти к подробному рассмотрению работы с окном Disassembly, мне нужно объединить темы вызова процедур и соглашений вызова.

Соглашение вызова описывает процесс передачи параметров в функцию и очистку стека при возврате из функции. Соглашение вызова определяется программистом при написании функции, и его надо придерживаться при всех вызовах данной функции. Процессор не обязывает использовать какое-то конкретное соглашение вызова. Если вы разберетесь в соглашениях вызова, вам будет гораздо легче работать с параметрами в окне Memoгу и определять поток выполнения ассемблерного кода в окне Disassembly.

Всего существует пять соглашений вызова, но часто употребляются только три: стандартный (`__stdcall`), объявление C (`__cdecl`) и вызов `this`. Стандартный вызов и объявление C вы можете указать сами, тогда как вызов `this` применяется автоматически к функциям C++, определяя передачу указателя `this`. Два других соглашения вызова — это быстрый вызов (`__fastcall`) и имеющее провокационное название соглашение `naked`¹. По умолчанию ОС Win32 не применяют соглашение быстрого вызова в программах, работающих в пользовательском режиме, потому что такой код не обладает машинной независимостью. Соглашение `naked` применяется, когда программист желает сам контролировать генерирование пролога и эпилога, о чем я расскажу в главе 15.

Информация обо всех соглашениях вызова приведена в табл. 7-8. Если вы помните, в начале этой главы я описывал схему расширения имен для установки точек прерываний в системных функциях. В табл. 7-8 вы увидите, что схема расширения имен определяется соглашением вызова.

Изменить соглашение вызова можно при объявлении и определении функции (см. пример). Кроме того, существуют ключи компилятора CLEXE, позволяющие изменить соглашение вызова, используемое по умолчанию, но я не рекомендую этого делать и советую явно указывать соглашение вызова для каждой функции. Так вы облегчите жизнь программистам, которые будут отвечать за сопровождение приложения.

```
// Объявление функции с соглашением __stdcall:  
void __stdcall ImAStandardCallFunction ( void ) ;
```

Если вы раньше не сталкивались с разными соглашениями вызова, вы, возможно, удивляетесь, почему их несколько. Различия между объявлением C и стандартным вызовом довольно тонки. При стандартном соглашении вызова стек очищает вызванная функция, поэтому она должна знать, сколько параметров ей ожидать. Следовательно, функция, использующая соглашение стандартного вызова, не может иметь переменное число аргументов, как, скажем, `printf`. При вызове функции в стиле C стек очищается вызывающей функцией, поэтому переменный список аргументов вполне допустим. Кроме того, при стандартном соглашении вызова программа имеет меньший размер, чем при объявлении C. При объявлении C компилятор должен сгенерировать код очистки стека для каждого вызова функции. Если функция, объявленная в стиле C, вызывается из многих мест программы, то после каждого вызова будет располагаться код очистки стека, в результате чего размер программы увеличится. В то же время при стандартном соглашении вызова функции сами выполняют очистку стека (т. е. соответствующий код находится в теле функции), поэтому после их вызовов компилятор не включает дополнительного кода. Именно поэтому абсолютное большинство системных функций Win32 использует соглашение стандартного вызова. Если вам захочется смутить какого-нибудь программиста, утверждающего, что он досконально знает платформу Win32, спросите его, какие две функции Win32 не используют стандартный вызов и какие соглашения применяются в их случае. Поначалу я не собирался приводить ответ на этот вопрос, чтобы вы поискали его сами, но по-

¹ В переводе с английского — «голый». — *Прим. перев.*

том решил, что это будет некрасиво с моей стороны: это функции `wsprintfA` и `wsprintfW`.

Табл. 7-8. Соглашения вызова

Соглашение вызова	Порядок передачи аргументов	Очистка стека	Расширение имени	Примечания
<code>__cdecl</code>	Справа налево.	Аргументы из стека удаляются вызывающей функцией. Только это соглашение поддерживает переменное число аргументов функций.	К имени функции добавляется префикс из символа подчеркивания: например, <code>_Foo</code> .	Применяется по умолчанию функциями C и C++.
<code>__stdcall</code>	Справа налево.	Вызванная функция сама удаляет свои аргументы из стека.	К имени функции добавляется префикс из символа подчеркивания и суффикс из символа <code>@</code> , за которым следует десятичный размер списка аргументов в байтах: например, <code>_Foo@12</code> .	Используется почти всеми системными функциями; используется по умолчанию внутренними функциями Visual Basic.
<code>__fastcall</code>	Первые два параметра типа <code>DWORD</code> передаются в регистрах <code>ECX</code> и <code>EDX</code> ; остальные параметры передаются справа налево.	Вызванная функция сама удаляет свои аргументы из стека.	К имени функции добавляется префикс из символа <code>@</code> и суффикс из этого же символа, за которым следует десятичный размер списка аргументов в байтах: например, <code>@Foo@12</code> .	Поддерживается только процессорами с архитектурой Intel. Это соглашение используется по умолчанию компиляторами Borland Delphi.
<code>this</code>	Справа налево. Параметр <code>this</code> передается в регистре <code>ECX</code> .	Аргументы из стека удаляются вызывающей функцией.	Нет.	Автоматически применяется к методам классов C++, пока вы не укажете стандартное соглашение вызова. При объявлении методов COM используется стандартное соглашение вызова.
<code>naked</code>	Справа налево.	Аргументы из стека удаляются вызывающей функцией.	Нет.	Используется, когда программисту нужно написать собственные пролог и эпилог.

Примеры соглашений вызова

Чтобы связать вместе описанные мной команды и соглашения вызова, я включил в книгу листинг 7-3, содержащий примеры всех соглашений вызова в том виде, в каком они отображаются в окне Disassembly отладчика Visual Studio .NET. Если вам захочется взглянуть на исходный код этого фрагмента, вы найдете его в файле CALLING.CPP на CD, прилагаемом к книге.

Листинг 7-3 — это отладочная компоновка, при создании которой с целью упрощения кода все дополнительные ключи, такие как /RTCs и /GS, были заблокированы; кроме того, код фактически ничего не делает. Я просто по очереди вызываю функции с использованием каждого соглашения вызова. Обратите особое внимание на порядок передачи параметров в каждую функцию и на то, как очищается стек. Чтобы сделать листинг понятнее, я вставил между функциями команды NOP.

Листинг 7-3. Примеры соглашений вызова

```
1: /*-----
2: "Отладка приложений для Microsoft .NET и Microsoft Windows"
3: Copyright (c) 1997-2003 John Robbins - All rights reserved.
4: -----*/
5: #include "stdafx.h"
6:
7: // Строки, передаваемые в каждую функцию.
8: static char * g_szStdCall = "__stdcall" ;
9: static char * g_szCdeclCall = "__cdecl" ;
10: static char * g_szFastCall = "__fastcall" ;
11: static char * g_szNakedCall = "__naked" ;
12:
13: // Объявление extern "C" полностью отключает расширение имен C++.
14: extern "C"
15: {
16:
17: // Функция с соглашением вызова __cdecl:
18: void CDeclFunction ( char *          szString ,
19:                     unsigned long ullong ,
20:                     char          chChar ) ;
21:
22: // Функция с соглашением вызова __stdcall:
23: void __stdcall StdCallFunction ( char *          szString ,
24:                                unsigned long ullong ,
25:                                char          chChar ) ;
26: // Функция с соглашением вызова __fastcall:
27: void __fastcall FastCallFunction ( char *          szString ,
28:                                  unsigned long ullong ,
29:                                  char          chChar ) ;
30:
31: // Функция, использующая соглашение вызова naked. В данном случае
32: // соглашение указывается в определении, но не в объявлении функции.
33: int NakedCallFunction ( char *          szString ,
```

см. след. стр.

```

34:                unsigned long ullong    ,
35:                char          chChar    ) ;
36: }
37:
38: void main ( void )
39: {
00401000 push        ebp
00401001 mov         ebp,esp
40:    // Вызовы функций, необходимые для генерирования кода компилятором.
41:    // Чтобы сделать дизассемблированный код читабельнее,
42:    // я вставил между функциями по две команды NOP.
43:    __asm NOP __asm NOP
00401003 nop
00401004 nop
44:    CDeclFunction ( g_szCdeclCall , 1 , 'a' ) ;
00401005 push        61h
00401007 push        1
00401009 mov         eax,dword ptr [g_szCdeclCall (403028h)]
0040100E push        eax
0040100F call       CDeclFunction (401056h)
00401014 add         esp,0Ch
45:    __asm NOP __asm NOP
00401017 nop
00401018 nop
46:    StdCallFunction ( g_szStdCall , 2 , 'b' ) ;
00401019 push        62h
0040101B push        2
0040101D mov         ecx,dword ptr [g_szStdCall (40301Ch)]
00401023 push        ecx
00401024 call       StdCallFunction (40105Dh)
47:    __asm NOP __asm NOP
00401029 nop
0040102A nop
48:    FastCallFunction ( g_szFastCall , 3 , 'c' ) ;
0040102B push        63h
0040102D mov         edx,3
00401032 mov         ecx,dword ptr [g_szFastCall (403038h)]
00401038 call       FastCallFunction (401066h)
49:    __asm NOP __asm NOP
0040103D nop
0040103E nop
50:    NakedCallFunction ( g_szNakedCall , 4 , 'd' ) ;
0040103F push        64h
00401041 push        4
00401043 mov         edx,dword ptr [g_szNakedCall (403044h)]
00401049 push        edx
0040104A call       NakedCallFunction (40107Ah)
0040104F add         esp,0Ch
51:    __asm NOP __asm NOP
00401052 nop

```



```

00401053  nop
52:
53: }
00401054  pop          ebp
00401055  ret
54:
55: void CDeclFunction ( char *          szString ,
56:                     unsigned long  ullong  ,
57:                     char           chChar   )
58: {
00401056  push        ebp
00401057  mov         ebp,esp
59:  __asm NOP __asm NOP
00401059  nop
0040105A  nop
60: }
0040105B  pop          ebp
0040105C  ret
61:
62: void __stdcall StdCallFunction ( char *          szString ,
63:                                unsigned long  ullong  ,
64:                                char           chChar   )
65: {
0040105D  push        ebp
0040105E  mov         ebp,esp
66:  __asm NOP __asm NOP
00401060  nop
00401061  nop
67: }
00401062  pop          ebp
00401063  ret         0Ch
68:
69: void __fastcall FastCallFunction ( char *          szString ,
70:                                  unsigned long  ullong  ,
71:                                  char           chChar   )
72: {
00401066  push        ebp
00401067  mov         ebp,esp
00401069  sub         esp,8
0040106C  mov         dword ptr [ebp-8],edx
0040106F  mov         dword ptr [ebp-4],ecx
73:  __asm NOP __asm NOP
00401072  nop
00401073  nop
74: }
00401074  mov         esp,ebp
00401076  pop          ebp
00401077  ret         4
75:
76: __declspec(naked) int NakedCallFunction ( char *          szString ,

```

см. след. стр.

```

77:                                     unsigned long ullong    ,
78:                                     char          chChar      )
79: {
80:     __asm NOP __asm NOP
0040107A nop
0040107B nop
81:     // Функции с соглашением naked должны выполнять возврат ЯВНО.
82:     __asm RET
0040107C ret

```

Доступ к переменным: глобальные переменные, параметры и локальные переменные

Давайте перейдем к рассмотрению доступа к переменным. К глобальным переменным обращаться легче всего, потому что они находятся в памяти по фиксированному адресу. Если у вас есть символы для конкретного модуля, вы увидите имя глобальной переменной. В следующем примере показано, как обратиться к глобальной переменной при помощи встроенного ассемблера, который позволяет использовать переменные и как операнды источника, и как операнды приемника, точно так же, как и при обычном программировании на языке C.

```

int g_iVal = 0 ;

void AccessGlobalMemory ( void )
{
    __asm
    {
        // Присвоение глобальной переменной значения 48 059.
        MOV g_iVal , 0BBBBh

        // Если символы загружены, в окне Disassembly вы увидите:
        // MOV DWORD PTR [g_iVal (4030B4)],0BBBBh.

        // Если символы не загружены, в окне Disassembly вы увидите:
        // MOV DWORD PTR [4030B4],0BBBBh.
    }
}

```

В функциях со стандартным кадром стека параметры располагаются по положительным смещениям от регистра EBP. Если вы не изменяете EBP на протяжении всей функции, к параметрам можно обращаться по тем же положительным смещениям, потому что параметры помещаются в стек до вызова процедуры. Доступ к параметрам показан в следующем фрагменте:

```

void AccessParameter ( int iParam )
{
    __asm
    {
        // Помещаем в регистр EAX значение iParam.
        MOV EAX , iParam
    }
}

```

```

// Если символы загружены, в окне Disassembly вы увидите:
// MOV EAX,DWORD PTR [iParam].

// Если символы не загружены, в окне Disassembly вы увидите:
// MOV EAX,DWORD PTR [EBP+8].
}
}

```

Ранее я уже упоминал одну очень важную фразу «от источника к приемнику», которую нужно проговаривать, перемещая палец от второго операнда к первому. Вот еще одна не менее важная фраза: «Параметры располагаются по положительным смещениям!» То, что в стандартных кадрах стека доступ к параметрам осуществляется при помощи постоянных смещений от регистра EBP, позволяет легко узнать, к какому параметру обращается конкретная команда: первый параметр всегда располагается по адресу [EBP+8], второй — по адресу [EBP+0Ch], третий — по адресу [EBP+10h] и т. д. Если вам нравятся формулы, то адрес n-ого параметра можно вычислить так: [EBP + (4 + (nx4))]. Чуть позже, обсудив локальные переменные, я проиллюстрирую стандартные кадры стека на примере и расскажу, почему значения смещений жестко закодированы.

Если при отладке оптимизированного кода вы видите ссылки по положительным смещениям от регистра стека ESP, знайте, что вы находитесь в функции с данными FPO. Регистр ESP в функции может изменяться, поэтому в данном случае для слежения за параметрами придется приложить чуть больше усилий. При работе с оптимизированным кодом надо внимательно следить за элементами, помещаемыми в стек, так как ссылка на адрес [ESP+20h] вполне может оказаться той же, что и более ранняя ссылка [ESP+8h]. Отлаживая оптимизированный код на уровне ассемблера в пошаговом режиме, я всегда слежу за расположением параметров в стеке.

В функциях со стандартными кадрами стека локальные переменные всегда располагаются по отрицательным смещениям от регистра EBP. Как было показано в разделе «Частая последовательность команд: вход в функцию и выход из функции», пространство в стеке резервируется командой SUB. Вот как присваивается новое значение локальной переменной:

```

void AccessLocalVariable ( void )
{
    int iLocal ;

    __asm
    {
        // Присваивание локальной переменной значения 23.
        MOV iLocal , 017h

        // Если символы загружены, в окне Disassembly вы увидите:
        // MOV DWORD PTR [iLocal],017h.

        // Если символы не загружены, в окне Disassembly вы увидите:
        // MOV [EBP-4],017h.
    }
}

```

Если вы имеете дело с функцией не со стандартным кадром стека, обнаружить локальные переменные может оказаться сложно, если не невозможно. Проблема в том, что в таких функциях обращение и к локальным переменным, и к параметрам выполняется при помощи положительных смещений от регистра ESP. В этом случае нужно попытаться найти команду SUB, чтобы увидеть, сколько байт отведено для хранения локальных переменных. Если в какой-то команде смещение от регистра ESP превышает число выделенных байт, то скорее всего это смещение ссылается на параметр.

При первом знакомстве кадры стека немного непонятны, поэтому я приведу еще один заключительный пример и несколько рисунков. Следующий фрагмент — очень простая функция C — покажет, почему в стандартных кадрах стека параметры располагаются по положительным, а локальные переменные — по отрицательным смещениям от регистра EBP. В этом примере определяется функция C AccessLocalsAndParamsExample, после чего следует код ее вызова и значения параметров. В конце примера приводится дизассемблированный код функции в том виде, какой он имеет в программе ASMer.

```
// Сама функция C:
void AccessLocalsAndParamsExample ( int * pParam1 , int * pParam2 )
{
    int iLocal1 = 3 ;
    int iLocal2 = 0x42 ;

    iLocal1 = *pParam1 ;
    iLocal2 = *pParam2 ;
}

// Код, вызывающий функцию AccessLocalsAndParamsExample:
void DoTheCall ( void )
{
    int iVal1 = 0xDEADBEEF ;
    int iVal2 = 0xBADDF00D ;
    __asm
    {
        LEA EAX , DWORD PTR [iVal2]
        PUSH EAX
        LEA EAX , DWORD PTR [iVal1]
        PUSH EAX
        CALL AccessLocalsAndParamsExample
        ADD ESP , 8
    }
}

// Дизассемблированный код функции AccessLocalsAndParamsExample:
void AccessLocalsAndParamsExample ( int * pParam1 , int * pParam2 )
{
0040107A push        ebp
0040107B mov         ebp,esp
0040107D sub         esp,8
    int iLocal1 = 3 ;
```

```

00401080  mov          dword ptr [iLocal1],3
          int iLocal2 = 0x42 ;
00401087  mov          dword ptr [iLocal2],42h

          iLocal1 = *pParam1 ;
0040108E  mov          eax,dword ptr [pParam1]
00401091  mov          ecx,dword ptr [eax]
00401093  mov          dword ptr [iLocal1],ecx
          iLocal2 = *pParam2 ;
00401096  mov          edx,dword ptr [pParam2]
00401099  mov          eax,dword ptr [edx]
0040109B  mov          dword ptr [iLocal2],eax
}
0040109E  mov          esp,ebp
004010A0  pop          ebp
004010A1  ret
    
```

Если вы установите точку прерывания в начале функции `AccessLocalsAndParamsExample` по адресу `0x0040107A`, то увидите такие значения стека и регистров (рис. 7-11):

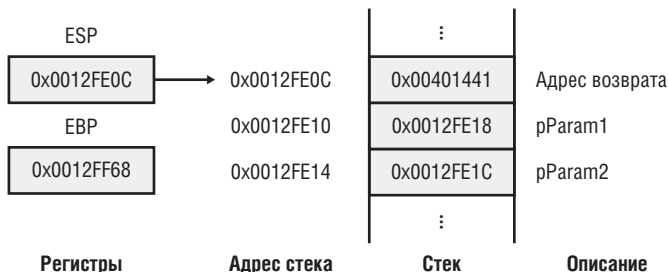
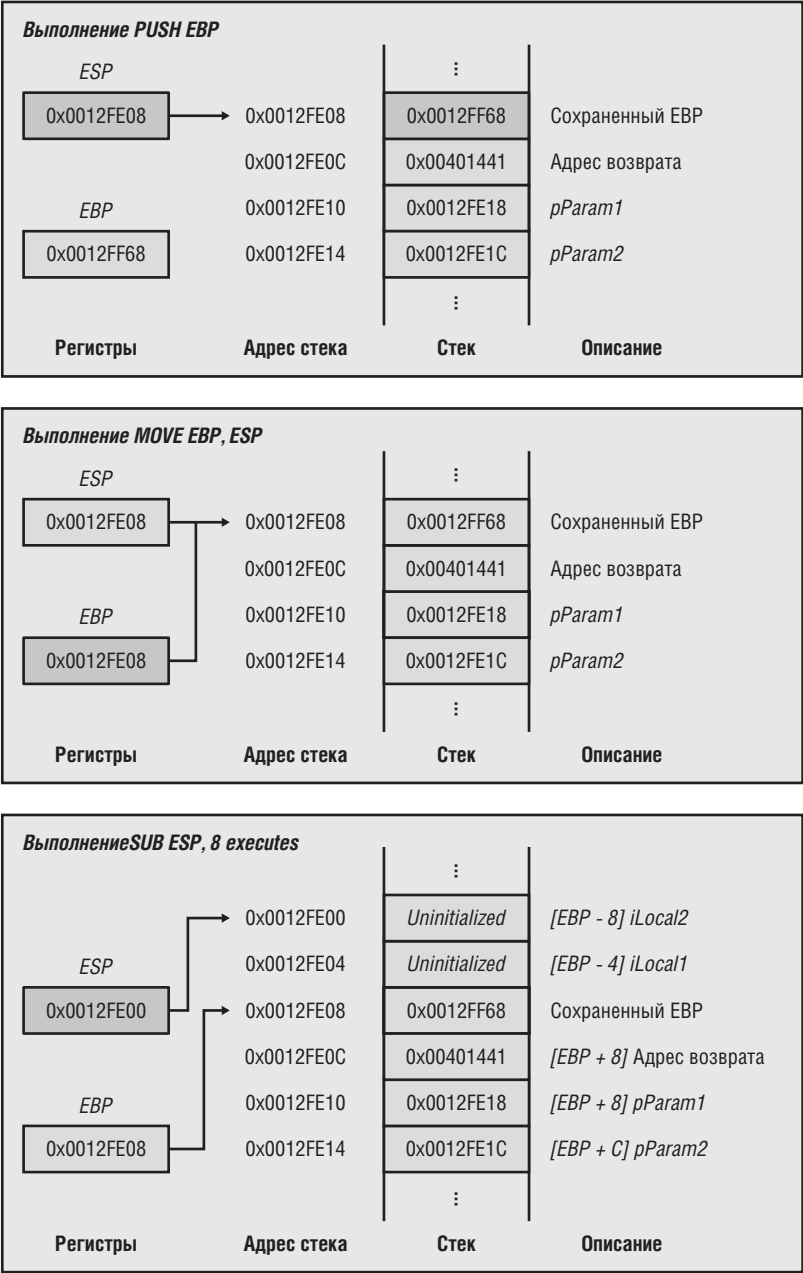


Рис. 7-11. Стек перед прологом функции `AccessLocalsAndParamsExample`

Первые три ассемблерных команды в функции `AccessLocalsAndParamsExample` представляют собой пролог функции. В результате выполнения пролога указатели стека и базы настраиваются так, что параметры будут располагаться по положительным, а локальные переменные — по отрицательным смещениям от регистра `EBP`. На рис. 7-12 показаны значения стека и указателя базы после выполнения каждой команды пролога. Я советую вам изучить этот пример как в книге, так и при помощи программы `ASMER.CPP`, расположенной на CD.



Дополнительные команды, которые нужно знать

В этом разделе описаны команды манипуляции с данными и указателями, сравнения и проверки, безусловных и условных переходов, организации циклов и работы со строками.

Манипуляции с данными

AND **Логическое И**
OR **Логическое ИЛИ**

Команды **AND** и **OR** выполняют логические побитовые операции, которые наверняка известны всем программистам, потому что они лежат в основе манипуляций с отдельными битами.

NOT **Логическое НЕТ**
NEG **Изменение знака числа**

Команды **NOT** и **NEG** иногда вызывают некоторое замешательство, потому что, несмотря на все сходство, они выполняют совершенно разные операции. Команда **NOT** — это побитовая операция, изменяющая все нулевые биты на единичные и наоборот, **NEG** эквивалентна вычитанию операнда из 0 (она изменяет все биты и прибавляет к результату единицу). Различия между этими двумя командами иллюстрирует следующий код:

```
void NOTExample ( void )
{
    __asm
    {
        MOV EAX , 0FFh
        MOV EBX , 1
        NOT EAX      // В регистре EAX теперь содержится 0FFFFFF00h.
        NOT EBX      // В регистре EBX теперь содержится 0FFFFFFFh.
    }
}

void NEGExample ( void )
{
    __asm
    {
        MOV EAX , 0FFh
        MOV EBX , 1
        NEG EAX      // В EAX теперь содержится 0FFFFFF01h ( 0 - 0FFh ).
        NEG EBX      // В EBX теперь содержится 0FFFFFFFh ( 0 - 1 ).
    }
}
```

XOR **Логическое исключающее ИЛИ**

Вы будете сталкиваться с командой **XOR** очень часто, но не потому, что люди очень интересуются операцией «исключающее ИЛИ», а потому что это самый быстрый способ обнуления значения. Выполнение **XOR** на двух операндах устанавливает бит

операнда приемника в 1, если соответствующие биты операндов имеют разные значения. Если все соответствующие биты одинаковы, операнд приемника станет равен 0. Команда `XOR EAX, EAX` выполняется быстрее (за меньшее число тактов процессора), чем `MOV EAX, 0`, поэтому именно ее компиляторы Microsoft генерируют для обнуления регистров.

INC	Инкремент
DEC	Декремент

Эти команды крайне просты, и об их функциях легко догадаться по названиям. Обе команды выполняются за один такт процессора, поэтому компилятор часто использует их для оптимизации определенных последовательностей кода. Стоит также отметить, что эти команды соответствуют непосредственно арифметическим операциям `++` и `--` языка C.

SHL	Сдвиг влево/умножение на 2
SHR	Сдвиг вправо/деление на 2

Команды битового сдвига выполняются процессорами x86 быстрее, чем соответствующие команды умножения и деления. Эти команды аналогичны побитовым операторам `<<` и `>>` языка C соответственно.

DIV	Деление без знака
MUL	Умножение без знака

Эти, казалось бы, простые команды на самом деле не так просты. Обе они выполняют беззнаковые операции над регистром `EAX`. Однако для хранения результата неявно используется и регистр `EDX`. В регистр `EDX` помещаются старшие байты результата умножения двойных слов и более объемных операндов. Команда `DIV` сохраняет остаток в `EDX`, а частное в `EAX`. Обе команды требуют, чтобы исходное число находилось в регистре `EAX`, а делителем/множителем могут быть значения, содержащиеся в других регистрах или в ячейках памяти.

IDIV	Деление со знаком
IMUL	Умножение со знаком

Эти команды аналогичны командам `DIV` и `MUL` за исключением того, что они рассматривают операнды как значения со знаком. Результат обрабатывается так же, как и в случае команд `DIV` и `MUL`. `IMUL` может иметь три операнда: первый — приемник, а два оставшихся — источники. В наборе команд x86 `IMUL` — единственная команда, допускающая применение трех операндов.

LOCK	Префикс блокировки шины (сигнал LOCK#)
-------------	---

Это не «настоящая» команда, а префикс для других команд, который говорит процессору, что доступ к памяти, выполняемый следующей командой, должен быть атомарной операцией, в результате чего процессор блокирует шину памяти, запрещая доступ к памяти другим процессорам системы. Если вы хотите увидеть префикс `LOCK` в действии, дизассемблируйте функцию `InterlockedIncrement` Windows XP или более поздней версии ОС Microsoft.

MOV SX **Перемещение значений с расширением знакового бита**
MOV ZX **Перемещение значений с обнулением старших битов**

Эти две команды копируют значения из меньших операндов в большие, определяя правило заполнения старших битов. При выполнении команды **MOV SX** старшие биты регистра приемника становятся равны знаковому биту операнда источника. Команда **MOV ZX** заполняет старшие биты регистра приемника нулями. На эти команды следует обращать особое внимание, если программа неправильно работает со знаками чисел.

Сравнение и проверка

CM P **Сравнение двух операндов**

Сравнивает операнды: вычитает второй операнд из первого и отбрасывает результат, устанавливая при этом соответствующие флаги в регистре **EFLAGS**. Можете считать **CM P** условной частью оператора **if** языка **C**. В табл. 7-9 показаны флаги и их значения, соответствующие выполнению команды **CM P**.

Табл. 7-9. Значения результата и устанавливаемые флаги

Результат	Флаги в окне Registers	Флаги из руководства Intel
Операнды равны	ZR = 1	ZF = 1
Первый операнд меньше	PL != 0V	SF != 0F
Первый операнд больше	ZR = 0 и PL = 0V	ZF = 0 и SF = 0F
Операнды не равны	ZR = 0	ZF = 0
Первый операнд больше, или операнды равны	PL = 0V	SF = 0F
Первый операнд меньше, или операнды равны	ZR = 1 или PL != 0V	ZF = 1 или SF != 0F

TEST **Логическое сравнение**

Выполняет над операндами побитовое логическое **И**, устанавливая соответствующим образом флаги **PL**, **ZR** и **PE** (флаги **SF**, **ZF** и **PF** в руководстве **Intel**). Эта команда позволяет проверить, равен ли бит единице.

Команды безусловного и условного переходов

JMP **Безусловный (абсолютный) переход**

Как говорит само название, **JMP** передает управление по абсолютному адресу.

JE **Переход, если равно**
JL **Переход, если меньше**
JG **Переход, если больше**
JNE **Переход, если не равно**
JGE **Переход, если больше или равно**
JLE **Переход, если меньше или равно**

Команды **CM P** и **TEST** не имели бы особого смысла, если бы у нас не было способа использования их результатов. Команды перехода позволяют перейти к нужному

месту программы согласно установленным флагам. С этими командами вы будете чаще всего сталкиваться в окне Disassembly. В документации к процессору Pentium Xeon II указаны 62 команды условного перехода, однако многие из них образуют идентичные пары, отличающиеся только тем, что функция одной из команд характеризуется при помощи частицы «не». Так, команда JLE (переход, если меньше или равно) имеет тот же идентификатор операции, что и команда JNG (переход, если не больше). Применяя какой-то другой дизассемблер, а не отладчик Visual Studio .NET, вы можете встретить и другие команды условного перехода. Чтобы узнать про все команды перехода, изучите в документации Intel коды «Jcc».

Я привел команды условного перехода в том же порядке, в каком указаны условия в табл. 7-9, чтобы вы могли их сопоставить. Обычно за командами CMP или TEST сразу же следует один из условных переходов. В оптимизированном коде между командами сравнения или проверки и перехода могут иметься и другие команды, но при этом гарантируется, что они не изменяют флаги.

Изучая дизассемблированный код, вы заметите, что команды условного перехода обычно противоположны условию, указанному в коде высокого уровня. Это поясняется в первом разделе следующего кода:

```
void JumpExamples ( int i )
{
    // Это оператор C. Заметьте: условие имеет вид "i > 0".
    // Тем не менее компилятор генерирует противоположное условие. Указанный
    // мной ассемблерный код очень похож на код, генерируемый компилятором.
    // При различных методах оптимизации может генерироваться различный код.
    // if ( i > 0 )
    // {
    //     printf ( "%i > 0\n" );
    // }
    char szGreaterThan[] = "%i > 0\n" ;
    __asm
    {
        CMP i , 0           // Сравнение i с нулем путем вычитания (i - 0).
        JLE JE_LessThanOne // Если i меньше или равно 0, выполняется переход
                           // к метке JE_LessThanOne.

        PUSH i              // В стек помещается параметр i.
        LEA EAX , szGreaterThan // В стек помещается адрес строки формата.
        PUSH EAX
        CALL DWORD PTR [printf] // Вызов функции printf. Можно догадаться,
                               // что printf скорее всего находится в DLL,
                               // так как она вызывается при помощи указателя.

        ADD ESP , 8         // printf вызывается по соглашению __cdecl, поэтому
                           // очистка стека выполняется в вызывающей функции.

JE_LessThanOne:           // При помощи встроенного ассемблера можно
                           // выполнять переход к любой метке языка C.
    }
```

////////////////////////////////////

```
// Вычисление абсолютного значения параметра и его проверка.
// Код на языке C:
// int y = abs ( i ) ;
// if ( y >= 5 )
// {
//     printf ( "abs(i) >= 5\n" ) ;
// }
// else
// {
//     printf ( "abs(i) < 5\n" ) ;
// }

char szAbsGTEFive[] = "abs(i) >= 5\n" ;
char szAbsLTFive[] = "abs(i) < 5\n" ;
__asm
{
    MOV  EBX , i           // Значение i помещается в регистр EBX.
    CMP  EBX , 0           // Сравнение EBX с нулем (EBX - 0).
    JG   JE_PosNum        // Если результат больше 0, то
                          // EBX содержит положительное значение.
    NEG  EBX              // Преобразование отриц. числа в положит.
```

JE_PosNum:

```
    CMP  EBX , 5           // Сравнение EBX с 5.
    JL   JE_LessThan5     // Если EBX < 5, переход к метке.
    LEA  EAX , szAbsGTEFive // Загрузка указателя на верную строку
                          // в регистр EAX.
    JMP  JE_DoPrintf      // Переход к вызову функции printf.
```

JE_LessThan5:

```
    LEA  EAX , szAbsLTFive // Загрузка указателя на верную строку в EAX.
```

JE_DoPrintf:

```
    PUSH EAX              // Адрес строки помещается в стек.
    CALL DWORD PTR [printf] // Печать строки.
    ADD  ESP , 4           // Восстановление стека.
```

```
}
```

Команды цикла

L00P Цикл согласно значению регистра ECX

В коде, сгенерированном компиляторами Microsoft, команда L00P встречается редко. И все же в некоторых участках кода ядра ОС (которые, похоже, написаны на ассемблере) она время от времени попадает. Использовать команду L00P просто. Загрузите в регистр ECX число итераций цикла и выполните блок кода. Сразу же

после блока укажите команду `L00P`, которая каждый раз будет уменьшать `ECX` на 1 и, если `ECX` не равен 0, передавать управление на начало блока. Когда `ECX` станет равным 0, начнет выполняться код, расположенный после команды `L00P`.

Большинство циклов, с которыми вы столкнетесь, будут представлять собой комбинацию команд условного и абсолютного перехода. Они очень похожи на представленный чуть выше код оператора `if` за исключением того, что в конце блока `if` располагается команда `JMP`, передающая управление в начало блока. Вот типичный пример цикла:

```
void LoopingExample ( int q )
{
    // Код на языке C:
    // for ( ; q < 10 ; q++ )
    // {
    //     printf ( "q = %d\n" , q ) ;
    // }

    char szFmt[] = "q = %d\n" ;
    __asm
    {
        JMP  LE_CompareStep    // При первой итерации цикла сразу
                               // переходим к проверке условия.

LE_IncrementStep:
        INC  q                // Увеличение q на 1.

LE_CompareStep:
        CMP  q , 0Ah          // Сравнение q с 10.
        JGE  LE_End            // Если q >= 10, цикл завершен.

        MOV  ECX , DWORD PTR [q] // Значение q помещается в ECX,
        PUSH ECX                // а затем – в стек.
        LEA  ECX , szFmt        // Загрузка адреса строки формата в ECX.
        PUSH ECX                // Адрес строки формата помещается в стек.
        CALL DWORD PTR [printf] // Печать номера итерации цикла.
        ADD  ESP , 8            // Очистка стека.

        JMP  LE_IncrementStep   // Увеличение q и переход к началу цикла.

LE_End:                                // Цикл завершен.

    }
}
```

Манипуляции со строками

Процессоры Intel предоставляют развитые средства работы со строками. Под этим понимается то, что одна команда процессора может выполнить некоторые действия над большими блоками памяти. Все описываемые мной команды работы со строками имеют несколько мнемонических обозначений, которые можно узнать

в документации Intel, однако в окне Disassembly среды Visual Studio .NET они всегда будут выглядеть так, как я покажу. Все эти команды могут работать с блоками памяти объемом в байт, слово и двойное слово.

MOVSB Перемещение данных из одной строки в другую

Перемещает значение, содержащееся в ячейке памяти по адресу ESI, в ячейку по адресу EDI. Она позволяет использовать только регистры ESI и EDI. Можете считать ее аналогом функции `memcpy` языка C. В окне Disassembly среды Visual Studio .NET всегда указывается спецификатор размера операции, так что вы сразу сможете определить, какой объем памяти перемещается. После перемещения одного значения регистры ESI и EDI увеличиваются или уменьшаются в зависимости от флага направления в регистре EFLAGS (в окне Registers среды Visual Studio .NET этот флаг обозначается как UP). Если поле UP равно 0, значения регистров увеличиваются, если 1 — регистры уменьшаются. Все компиляторы Microsoft всегда генерируют код, перемещающий строки в порядке уменьшения адресов (UP = 1). Значение, на которое увеличиваются или уменьшаются регистры, зависит от размера операции: 1 для байтов, 2 для слов и 4 для двойных слов.

SCASB Сканирование строки

Сравнивает значение ячейки памяти, на которую указывает регистр EDI, со значением, содержащимся в AL, AX или EAX, что зависит от указанного размера. Сравнение приводит к установке флагов регистра EFLAGS в те же значения, что указаны в табл. 7-9. Сканируя строку при помощи команды SCASB на предмет наличия терминального символа NULL, можно воспроизвести функцию `strlen` языка C. Как и MOVSB, команда SCASB автоматически увеличивает или уменьшает значения регистра EDI.

STOSB Сохранение значения в строке

Сохраняет значение из AL, AX или EAX (в зависимости от заданного размера) по адресу памяти, на который указывает регистр EDI. Эта команда аналогична функции `memset` языка C. Как и MOVSB и SCASB, команда STOSB также автоматически увеличивает или уменьшает значение регистра EDI.

CMPSB Сравнение строк

Поочередно сравнивает значения двух строк и устанавливает соответствующие значения флагов регистра EFLAGS. В то время как SCASB сравнивает все символы с одним значением, команда CMPSB «перебирает» по очереди значения обеих строк. CMPSB аналогична функции `memcmp` языка C. Как и все остальные команды работы со строками, CMPSB сравнивает значения заданного размера и автоматически увеличивает или уменьшает указатели на обе строки.

REP	Повтор согласно счетчику ECX
REPE	Повтор, пока равно или пока ECX не равен 0
REPNE	Повтор, пока не равно или пока ECX не равен 0

Команды работы со строками удобны, но они не заслуживали бы внимания, если бы работали только с одним элементом за раз. Префиксы повторения позволяют выполнять строковые команды заданное (в ECX) число раз или пока не будет достигнуто указанное условие. Выбрав функцию Step Into (шаг внутрь), когда в окне Disas-

sembly выполняется команда с префиксом повтора, вы останетесь на той же строке, потому что команда просто выполнится еще раз. Чтобы пройти все итерации, используйте функцию Step Over (шаг через). Отлаживая программу, вы можете при помощи функции Step Into проверить, на какие строки указывают регистры ESI и EDI. Если ошибка вызывается строковой командой с префиксом повтора, нужно также следить за значением регистра ECX: так вы узнаете, какая итерация заканчивается неудачей.

Описывая команды работы со строками, я говорил, на какие функции стандартной библиотеки C они похожи. Вот как могут выглядеть их эквиваленты на языке ассемблера, не включающие проверку ошибок:

```
void MemCPY ( char * szSrc , char * szDest , int iLen )
{
    __asm
    {
        MOV ESI , szSrc          // Загрузка в ESI адреса строки источника.
        MOV EDI , szDest        // Загрузка в EDI адреса строки назначения.

        MOV ECX , iLen           // Указание числа копируемых элементов.

                                // Скопировать!
        REP MOVSB BYTE PTR [EDI] , BYTE PTR [ESI]
    }
}

int StrLEN ( char * szSrc )
{
    int iReturn ;
    __asm
    {
        XOR EAX , EAX           // Обнуление EAX (Ищем символ NULL).
        MOV EDI , szSrc         // Загрузка адреса проверяемой строки в EDI.
        MOV ECX , 0FFFFFFFFh     // Максимальное число проверяемых символов.

        REPNE SCASB BYTE PTR [EDI] // Сравнение, пока ECX не станет равным 0
                                // или пока не будет найден символ NULL.

        CMP ECX , 0              // Если ECX=0, значит, символ NULL
        JE StrLEN_NoNull         // в строке обнаружен не был.

        NOT ECX                  // ECX уменьшался, поэтому его нужно
                                // преобразовать в положительное число.
        DEC ECX                  // Символ NULL в строку не входит.
        MOV EAX , ECX            // Возвращение числа символов.
        JMP StrLen_Done          // Переход на возвращение из функции.
    }
}

StrLEN_NoNull:
```

```
MOV EAX , 0FFFFFFFh    // NULL не был обнаружен – возвращаем -1.
```

```
StrLEN_Done:
```

```
    }
    __asm MOV iReturn , EAX ;
    return ( iReturn ) ;
}
```

```
void MemSET ( char * szDest , int iVal , int iLen )
```

```
{
    __asm
    {
        MOV EAX , iVal          // В EAX заносится символ заполнения.
        MOV EDI , szDest        // Загрузка в EDI адреса строки.
        MOV ECX , iLen           // Загрузка в ECX числа итераций.

        REP STOS BYTE PTR [EDI] // Заполнение памяти указанным значением.
    }
}
```

```
int MemCMP ( char * szMem1 , char * szMem2 , int iLen )
```

```
{
    int iReturn ;
    __asm
    {
        MOV ESI , szMem1        // В ESI – адрес первого блока памяти.
        MOV EDI , szMem2        // В EDI – адрес второго блока памяти.
        MOV ECX , iLen           // Макс. число сравниваемых байтов.

        // Сравнение блоков памяти.
        REPE CMPS BYTE PTR [ESI], BYTE PTR [EDI]

        JL  MemCMP_LessThan      // Переход, если szSrc < szDest
        JG  MemCMP_GreaterThan   // Переход, если szSrc > szDest

        // Блоки памяти идентичны.
        XOR EAX , EAX            // Возвращаем 0.
        JMP MemCMP_Done
    }
}
```

```
MemCMP_LessThan:
```

```
    MOV EAX , 0FFFFFFFh    // Возвращаем -1.
    JMP MemCMP_Done
```

```
MemCMP_GreaterThan:
```

```
    MOV EAX , 1            // Возвращаем 1.
    JMP MemCMP_Done
```

```
MemCMP_Done:
```

```
    }
```

```
__asm MOV iReturn , EAX
return ( iReturn ) ;
}
```

Распространенные ассемблерные конструкции

До этого момента я просто описывал основные команды языка ассемблера. Теперь я рассмотрю ассемблерные конструкции и объясню, как их определить и преобразовать в операции более высокого уровня.

Доступ к памяти при помощи регистра FS

В ОС Win32 регистр FS играет специальную роль: в нем хранится указатель на блок информации о потоке (TIB), который еще иногда называют блоком переменных окружения потока (Thread environment block, TEB). Этот блок содержит все специфичные для потока данные, нужные для того, чтобы ОС могла легко предоставить доступ к потоку. Они включают все цепочки структурной обработки исключений (Structured exception handling, SEH), локальную память потока, стек потока и другую необходимую ОС информацию. Подробнее о SEH см. главу 13. Пример локальной памяти потока я приведу при обсуждении программы MemStress (см. главу 17).

Блок TIB хранится в специальном сегменте памяти, и, когда ОС нужно получить доступ к TIB, она преобразует сумму значения регистра FS и смещения в нормальный линейный адрес. Если вы видите команду, использующую регистр FS, знайте, что выполняется одна из операций: создание или уничтожение кадра SEH, доступ к блоку TIB или доступ к локальной памяти потока.

Создание или уничтожение кадра SEH

Первые команды после создания кадра стека часто похожи на указанный ниже фрагмент, стандартный для начала блока `__try`. Первый узел цепи обработчиков SEH располагается в TIB по смещению 0. В приведенном ниже дизассемблированном коде компилятор помещает в стек данные и указатель на функцию `__except_handler3` ОС Windows 2000. В Windows XP эту роль играет функция `_SEH_prolog`. Первая команда `MOV` получает доступ к TIB; смещение 0 показывает, что узел добавляется на вершину цепи исключений. Две последних команды определяют место цепи, в которое узел будет перемещен.

```
PUSH 004060d0
PUSH 004014a0
MOV EAX , FS:[00000000]
PUSH EAX
MOV DWORD PTR FS:[0] , ESP
```

Этот пример прост и ясен, но компилятор не всегда генерирует такой чистый код. Иногда он распределяет создание кадра SEH по большому участку кода. В зависимости от установленных флагов генерирования кода и оптимизации компилятор упорядочивает команды так, чтобы лучше использовать конвейеры процессора. В следующем дизассемблированном фрагменте, для которого загружены символы `KERNEL32.DLL`, показано начало функции `IsBadReadPtr` Microsoft Windows 2000:


```
PUSH EBP
MOV EBP , ESP
PUSH 0FFFFFFFh
PUSH 77E86F40h
PUSH OFFSET __except_handler3
MOV EAX , DWORD PTR FS:[00000000h]
PUSH EAX
MOV DWORD PTR FS:[0] , ESP
```

Windows XP имеет один интересный аспект: похоже, код этой ОС включает собственный механизм обработки исключений, генерирующий вызов функции `_SEH_prolog`, в которой выполняется большая часть предыдущего кода. Это приводит к значительному уменьшению кода, и на уровне ассемблера кажется, что функции Windows XP, работающие с SEH, используют для колдовства собственные пролог и эпилог.

А вот уничтожение кадра SEH гораздо менее интересно, чем его создание; нужно только помнить, что любой доступ по адресу `FS:[0]` означает работу с SEH:

```
MOV ECX , DWORD PTR [EBP-10h]
MOV DWORD PTR FS:[0] , ECX
```

Доступ к TIB

Значение `FS:[18]` — линейный адрес структуры TIB. Вот как функция `GetCurrentThreadId` ОС Windows XP получает сначала линейный адрес TIB, а затем — идентификатор потока, расположенный по смещению `0x24`:

```
GetCurrentThreadId:
MOV EAX , FS:[00000018h]
MOV EAX , DWORD PTR [EAX+024h]
RET
```

Доступ к локальной памяти потока

Локальная память потока — это механизм Win32, позволяющий каждому потоку иметь собственные экземпляры глобальных переменных. По смещению `0x2C` в структуре TIB располагается указатель на массив локальной памяти потока. Вот как получить доступ к указателю локального хранилища потока:

```
MOV ECX , DWORD PTR FS:[2Ch]
MOV EDX , DWORD PTR [ECX+EAX*4]
```

Ссылки на структуры и классы

При разработке приложений для Windows приходится иметь дело со структурами и классами, поэтому я хочу вкратце рассмотреть доступ к занимаемой ими памяти. Со структурами и классами удобно работать на языках высокого уровня, но на уровне ассемблера они на самом деле не существуют. В языках высокого уровня структуры и классы — это просто удобные способы указания смещений в блоке памяти.

Обычно компиляторы организуют память для структур и классов именно так, как вы указываете. Иногда компилятор расширяет поля, чтобы они были выровнены по естественным границам блоков памяти, которые для процессоров x86 кратны 4 или 8 байтам.

Для ссылок на структуры и классы используется сочетание регистра и смещения. Ниже я привожу пример структуры `MyStruct` и указываю в комментариях к каждому ее члену его смещение от начала структуры. После определения структуры `MyStruct` я демонстрирую способы доступа к ее полям.

```
typedef struct tag_MyStruct
{
    DWORD dwFirst ;           // Смещение равно 0.
    char  szBuff[ 256 ] ;     // 4-байтовое смещение.
    int   iVal ;              // 260-байтовое смещение.
} MyStruct , * PMyStruct ;

void FillStruct ( PMyStruct pSt )
{
    char szName[] = "Pam\n" ;

    __asm
    {
        MOV  EAX , pSt // В EAX помещается pSt. Ниже я использую
                        // непосредственные смещения, чтобы показать,
                        // как они выглядят в дизассемблированном коде.
                        // Встроенный ассемблер позволяет использовать
                        // нормальные ссылки вида <структура>.<поле>.

        // Код на языке C: pSt->dwFirst = 23 ;
        MOV  DWORD PTR [EAX] , 17h

        // Код на языке C: pSt->iVal = 0x33 ;
        MOV  DWORD PTR [EAX + 0104h] , 0x33

        // Код на языке C: strcpy ( pSt->szBuff , szName ) ;
        LEA  ECX , szName      // Адрес szName помещается в стек.
        PUSH ECX

        LEA  ECX , [EAX + 4]    // Адрес szBuff помещается в стек.
        PUSH ECX

        CALL strcpy
        ADD  ESP , 8            // Функция strcpy использует соглашение __cdecl.

        // Код на языке C: pSt->szBuff[ 1 ] = 'A' ;
        MOV  BYTE PTR [EAX + 5] , 41h

        // Код на языке C: printf ( pSt->szBuff ) ;
        MOV  EAX , pSt          // В EAX снова помещается pSt, так как
                                // регистр EAX был изменен при вызове strcpy.
```

```

    LEA ECX , [EAX + 4]
    PUSH ECX
    CALL DWORD PTR [printf]
    ADD ESP , 4           // Функция printf использует соглашение __cdecl.
}
}

```

Полный пример

Я описал все важные аспекты языка ассемблера процессоров Intel и, прежде чем перейти к рассмотрению окна Disassembly, хочу привести полный пример функции Win32 API. Листинг 7-4 представляет собой подробно описанный дизассемблированный код функции `lstrcpyA` из библиотеки `KERNEL32.DLL` системы Windows 2000 с установленным пакетом обновлений 2 (Service Pack 2). Функция `lstrcpyA` копирует одну строку в другую. Я выбрал ее потому, что она поясняет все описанные мной в этой главе вопросы и потому что ее цель понятна. После точек с запятой я привожу комментарии, которые попытался сделать максимально подробными.

Листинг 7-4. Полный пример на языке ассемблера: функция `lstrcpyA`

```

; Прототип функции:
; LPTSTR __stdcall lstrcpy ( LPTSTR lpString1 , LPCTSTR lpString2 )

_lstrcpyA@8:

; Стандартный пролог кадра стека.
PUSH      EBP
MOV       EBP , ESP

; Конфигурирование блока SEH __try. По адресу 0x77E88000 содержится
; значение -1. Это стандартный способ указания блока __except.
; с EXCEPTION_EXECUTE_HANDLER.
PUSH      0FFFFFFFFh
PUSH      77E88000h
PUSH      OFFSET __except_handler3 (77E8615Bh)
MOV       EAX , DWORD PTR FS:[00000000h]
PUSH      EAX
MOV       DWORD PTR FS:[0] , ESP

; Вместо того чтобы зарезервировать в стеке пространство для дополнительных
; относящихся к SEH элементов при помощи "SUB ESP 8", генератор кода решил
; выполнить две команды PUSH. Идентификатор операции команды "PUSH ECX"
; состоит из одного байта(0x51), поэтому в данном случае это самый
; быстрый вариант.
PUSH      ECX
PUSH      ECX

; Сохранение используемых в функции регистров. EBX в функции не применяется,
; однако, возможно, он помещается в стек для оптимизации работы конвейера.

```

см. след. стр.

```
PUSH     EBX
PUSH     ESI
PUSH     EDI
```

; В конце процесса настройки SEH сохраняется адрес начала
; блока try в стеке, после чего код входит в блок try.

```
MOV      DWORD PTR [EBP-18h] , ESP
AND      DWORD PTR [EBP-4] , 0
```

; После выполнения всех параметров функция получает длину
; копируемой строки, являющейся вторым параметром функции.

; В EDI помещается адрес второго параметра копируемой строки.

```
MOV      EDI , DWORD PTR [EBP+0Ch]
```

; Функция lstrcpy будет просматривать 4 294 967 295 байт,
; пока не обнаружит терминальный символ NULL.
; Для подсчета итераций цикла команда REPNE SCAS использует регистр ECX.

```
OR       ECX , 0FFFFFFFFh
```

; Обнуление EAX (поиск символа NULL).

```
XOR      EAX , EAX
```

; Команда SCAS просматривает строку, пока не найдет символ NULL.

```
REPNE SCAS BYTE PTR [EDI]
```

; Регистр ECX отсчитывался вниз от максимального значения, поэтому для
; нахождения длины строки нужно инвертировать все биты. Полученное число
; будет длиной строки с учетом символа NULL.

```
NOT      ECX
```

; Команда REPNE SCAS увеличивала и EDI, поэтому, чтобы он снова
; указывал на начало строки, из EDI нужно вычесть ее длину.

```
SUB      EDI , ECX
```

; Длина строки помещается в регистр EDX. Он не был сохранен в начале
; этой функции, так как при вызове функций C/C++ это не требуется.

```
MOV      EDX , ECX
```

; Второй параметр помещается в регистр ESI. Именно этот регистр
; в командах работы со строками является операндом источника.

```
MOV      ESI , EDI
```

; В регистр EDI помещается первый параметр – адрес строки приемника.

```
MOV      EDI , DWORD PTR [EBP+8]
```

; Второй параметр сохраняется в регистре EAX. Этот регистр
; также не требуется сохранять при вызове функций.

```
MOV      EAX , EDI
```

```
; Длина строки была получена в байтах. Чтобы получить длину
; строки в двойных словах, надо разделить ее на 4. Если число
; символов нечетно, команда REP MOVSB не сможет скопировать их все.
; Все оставшиеся байты копируются после REP MOVSB.
SHR          ECX , 2

; Копирование второй строки-параметра в первую строку-параметр.
REP MOVSB   DWORD PTR [EDI] , DWORD PTR [ESI]

; Загрузка сохраненной длины строки в регистр ECX.
MOV         ECX , EDI

; В результате операции "логическое И" над длиной строки и числом 3
; будет найдено число байтов, которые еще не были скопированы.
AND         ECX , 3

; Копирование оставшихся байт из одной строки в другую.
REP MOVSB   BYTE PTR [EDI] , BYTE PTR [ESI]

; Присваивание локальной переменной значения -1 показывает,
; что функция покидает этот блок try/except.
OR          DWORD PTR [EBP-4] , 0FFFFFFFFh

; Получение предыдущего кадра SEH.
MOV         ECX , DWORD PTR [EBP-10h]

; Уничтожение кадра SEH.
MOV         DWORD PTR FS:[0] , ECX

; Восстановление регистров, сохраненных в стеке в начале функции.
POP         EDI
POP         ESI
POP         EBX

; Команда LEAVE эквивалентна следующей паре команд:
; MOV ESP , EBP
; POP EBP
LEAVE

; Удаление 8 байтов из стека (параметры)
; и возвращение в главную функцию.
; Функция lstrcpw использует соглашение __sdtrcall.
RET        8
```

Окно Disassembly

Теперь вы имеете некоторые сведения об ассемблере, поэтому окно Disassembly отладчика Visual Studio .NET уже не должно вас пугать. Окно Disassembly обеспечивает много возможностей, которые помогут вам в отладке. Ниже я поговорю о некоторых из них и о том, как сократить время, проводимое в окне Disassembly.

Навигация

Если вы работали с отладчиком, который не поддерживал функции навигации по дизассемблированному коду, вы знаете, насколько это печально. К счастью, окно Disassembly предлагает несколько способов, позволяющих попасть в нужное место отлаживаемой программы.

Первый — использовать поле Address (адрес) в левом верхнем углу окна Disassembly. Если вам известен нужный адрес, просто введите его и окажетесь в соответствующем месте кода. Поле Address может также интерпретировать символы и контекстную информацию, так что вы сможете попасть в нужное место, даже не зная его точного адреса.

Одна небольшая проблема заключается в том, что необходимость форматирования символов, описанную в разделе «Усложненный синтаксис точек прерывания», никто не отменял. Вы должны будете выполнять то же расширение имен, которое требуется при установке точки прерывания для системной или экспортируемой функции. Например, если у вас загружены символы для KERNEL32.DLL и вы хотите перейти в окне Disassembly к функции LoadLibrary, то, чтобы очутиться в нужном месте, вам надо ввести в поле Address выражение `{, , kernel32}_LoadLibraryA@4`.

Вам обязательно понравится то, что окно Disassembly поддерживает функцию «drag and drop». Если вы работаете с ассемблерным кодом и желаете быстро узнать, к какой области памяти происходит обращение, вы можете выбрать адрес и перетащить его в поле Address. Когда вы отпустите кнопку, окно Disassembly автоматически перейдет к этому адресу.

Увлечшись окном Disassembly, можно легко забыть дорогу назад и заблудиться. Чтобы вернуться к указателю команд, просто щелкните в окне Disassembly правой кнопкой и выберите пункт Show Next Statement (Показать следующий оператор). Окна исходного кода также поддерживают эту функцию. Очень полезной может оказаться еще одна возможность, отсутствовавшая в предыдущих версиях Visual Studio: поле Address теперь запоминает все адреса, к которым вы переходили, благодаря чему вы всегда сможете вернуться назад.

Просмотр параметров стека

В разделе «Усложненный синтаксис точек прерывания» я рассказал, как устанавливать точки прерывания для системных и экспортируемых функций. Важность этих точек прерывания объясняется тем, что они позволяют узнать параметры, передающиеся в конкретную функцию. Чтобы показать, как просматривать элементы стека, я хочу привести вполне реальный и простой пример.

При загрузке программы из Интернета вам хотелось бы иметь гарантию, что она не запускает тайком какие-то другие программы. Для этого нужно отследить все вызовы функции CreateProcess. В этом примере я буду использовать программу CMD.EXE (командная строка), однако ничто не мешает вам аналогично изучить любое другое приложение.

Сначала я запустил программу DUMPBIN.EXE из состава Visual Studio .NET с ключом командной строки `/IMPORTS`, чтобы узнать, что именно вызывает CMD.EXE. Вывод показал все неявно импортируемые DLL и вызываемые функции. Изучая

импортируемые функции, я увидел, что CMD.EXE импортирует `CreateProcessW` из `KERNEL32.DLL`. Затем из среды Visual Studio .NET я открыл CMD.EXE как новое решение, выбрав в меню File пункт Open Solution (открыть решение) (CMD.EXE находится в каталоге `%SYSTEMROOT%\System32`). У меня были загружены символы (после прочтения книги у вас тоже всегда будут загружены нужные символы), поэтому мне нужно было установить точку прерывания для функции `{, , kernel-32}_CreateProcessW@40`. После запуска CMD.EXE я написал в командной строке `S0L.EXE` и нажал Enter.

В Windows 2000 с пакетом обновлений 2 точка прерывания для `_CreateProcessW@40` останавливает программу по адресу `0x77E96F60` на команде `PUSH EBP`, участвующей в создании стандартного кадра стека. Так как программа останавливается на первой команде функции `CreateProcess`, значит, в этот момент вершина стека содержит параметры и адрес возвращения из функции. Далее я открыл одно из четырех окон Memory, выбрав подменю Debug\Windows\Memory. Открыв окно Memory, я щелкнул в нем правой кнопкой, выбрал в контекстном меню пункт 4-byte Integer (4-байтовые целочисленные значения) и настроил размер окна так, чтобы в нем помещались только два столбца: адреса стека и значения для каждого адреса. Наконец, чтобы узнать, что находится в стеке, я ввел в поле Address «ESP», регистр указателя стека.

Отображение стека в окне Memory в начале функции `CreateProcess` показано на рис. 7-13. Первое значение, `0x4AD0728C`, — это адрес возвращения; следующие 10 — параметры функции `CreateProcess` (табл. 7-10). Параметры `CreateProcess` занимают 40 байт: каждый по 4 байта. Стек растет от старших адресов к младшим, и параметры помещаются в него справа налево, поэтому в окне Memory они появляются в том же порядке, что и в определении функции.

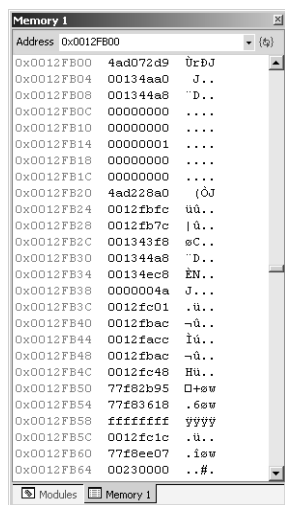


Рис. 7-13. Стек в окне Memory отладчика Visual Studio .NET

Табл. 7-10. Параметры, передаваемые программой CMD.EXE функции CreateProcessW

Значение	Тип	Параметр
0x00134AA0	LPCWSTR	lpApplicationName
0x001344A8	LPWSTR	lpCommandLine
0x00000000	LPSECURITY_ATTRIBUTES	lpProcessAttributes
0x00000000	LPSECURITY_ATTRIBUTES	lpThreadAttributes
0x00000001	BOOL	bInheritHandles
0x00000000	DWORD	dwCreationFlags
0x00000000	LPVOID	lpEnvironment
0x 4AD228A0	LPCWSTR	lpCurrentDirectory
0x0012FBFC	LPSTARTUPINFO	lpStartupInfo
0x0012FB7C	LPPROCESS_INFORMATION	lpProcessInformation

Значения первых двух параметров можно узнать двумя способами. Первый — открыть второе окно Memory и указать адрес параметра в поле Address. Щелкните в окне правой кнопкой, чтобы выбрать желаемый формат данных в верхней части контекстного меню. Более важные функции расположены в нижней части этого меню, где можно выбрать отображение значений, расположенных по адресам памяти, в текстовом формате ANSI или Unicode. Второй и более простой способ — перетащить интересующий вас адрес в окно Watch. Для просмотра значения в окне Watch служит оператор приведения типа. Так, для просмотра параметра lpApplicationName нужно ввести в окне Watch выражение (wchar_t*)0x00134aa0.

Получить параметры функции оказалось в этом примере так легко потому, что я остановил функцию на первой команде, до того как функция смогла поместить в стек дополнительные элементы. Если вы желаете просмотреть параметры, находясь в теле функции, придется поработать. Облегчить эту задачу можно, поислав положительные смещения от регистра EBP. И все же иногда лучше просто открыть окно Memory и узнать все самому.

Команда Set Next Statement

Как и окна исходного кода, окно Disassembly поддерживает функцию Set Next Statement, доступную из контекстного меню, так что вы можете изменить EIP для выполнения другой команды. Работая над отладочными компоновками в окнах исходного кода, к функции Set Next Statement можно относиться более-менее небрежно, однако в окне Disassembly она требует пристального внимания.

Главное условие корректного изменения EIP (чтобы это не привело к ошибке) — должное внимание ко стеку. Число помещенных в стек элементов должно соответствовать числу извлеченных элементов; если это не так, программа в конечном счете потерпит крах. Я не отпугиваю вас от изменения потока выполнения программы — я поощряю вас на эксперименты с ним. Изменение потока выполнения при помощи функции Set Next Statement — очень мощный метод, позволяющий заметно ускорить отладку. Если вы позаботитесь о стеке, стек позаботится о вас.

Например, если вы хотите заново выполнить функцию, не вызвав немедленную ошибку, убедитесь, что изменение потока сохраняет стек в сбалансирован-

ном состоянии. В следующем примере я хочу дважды вызвать функцию, находящуюся по адресу 0x00401005:

```
00401032 PUSH EBP
00401033 MOV EBP , ESP
00401035 PUSH 404410h
0040103A CALL 00401005h
0040103F ADD ESP , 4
00401042 POP EBP
00401043 RET
```

При повторной отладке этого дизассемблированного кода нужно гарантировать выполнение команды `ADD` по адресу 0x0040103F, чтобы стек был сбалансированным. Как я указывал при обсуждении различных соглашений, в данном фрагменте вызывается функция с соглашением `__cdecl`, о чем говорит команда `ADD` сразу же после вызова. Поэтому для повторного вызова функции мне нужно установить указатель команд на адрес 0x00401035, чтобы выполнялась команда `PUSH`.

Исследование стека «вручную»

Окна `Memory` и `Disassembly` тесно связаны. Пытаясь определить, что делает фрагмент ассемблерного кода при помощи окна `Disassembly`, нужно держать открытым окно `Memory`, чтобы видеть адреса и значения, над которыми производятся действия. Ассемблерные команды изменяют память, а значения памяти определяют выполнение ассемблерной программы; окна `Disassembly` и `Memory` позволяют наблюдать эту связь в динамике.

Само по себе содержимое окна `Memory` — просто набор чисел, особенно при крахе программы. Однако, объединив информацию двух окон, вы непременно начнете понимать причины неприятных ошибок. Совместное использование этих окон особенно важно при отладке оптимизированного кода, когда отладчик не может легко перемещаться по стеку. Для исправления ошибки вам придется исследовать стек вручную.

Прежде всего для этого нужно знать, в какие области памяти загружены ваши двоичные файлы. Новое окно `Modules` (модули) делает это элементарным: оно показывает имя модуля, путь к модулю, порядок загрузки и, что важнее всего, диапазон занимаемых модулем адресов памяти. Сравнивая элементы стека со списком диапазонов адресов, вы легко поймете, какие элементы являются адресами в ваших модулях.

После этого нужно открыть окна `Memory` и `Disassembly`. Введите в поле `Address` окна `Memory` название регистра `ESP` и прикажите отладчику отображать значения в формате двойных слов, щелкнув в окне правой кнопкой и выбрав пункт `4-byte Integer`. Кроме того, лучше не «прикреплять» окно `Memory` к остальным блокам интерфейса IDE и уменьшить его, чтобы оно вмещало только адреса стека и один столбец 32-разрядных значений, потому что именно так представляет себе стек большинство людей. Когда вам встретится число, похожее на адрес одного из ваших загруженных модулей, перетащите его в поле `Address` окна `Disassembly`. Окно `Disassembly` покажет ассемблерный код по этому адресу, и если вы создавали свое приложение с полным набором отладочной информации, то сможете определить вызывающую функцию.

Если дампы по регистру ESP не содержат ничего похожего на адрес модуля, можно отобразить в окне Мемогу дампы памяти, начиная с регистра EBP, и выполнить то же самое. По мере освоения языка ассемблера вы станете замечать в дизассемблированном коде и другую полезную информацию. На месте преступления часто можно обнаружить улики, подсказывающие, какой регистр использовать для поиска адреса возврата: ESP или EBP.

У просмотра элементов стека вручную есть и обратная сторона медали: иногда для нахождения нужных адресов приходится проделать довольно значительный путь. Однако если вы представляете, где загружены модули, то найдете интересующую вас информацию достаточно быстро.

Отладка: фронтовые очерки

Что не так с GlobalUnlock? Всего лишь разыменование указателя.

Боевые действия

Меня попросили помочь одной группе в отладке очень неприятной проблемы — достаточно серьезной, чтобы это не позволяло выпустить программу. Члены группы провели примерно месяц, пытаясь воспроизвести ошибку, и все еще не имели никакого понятия о ее причинах. Единственной подсказкой было то, что крах программы наступал только после вызова диалогового окна Print и изменения некоторых параметров. Вскоре после закрытия окна Print возникала ошибка в элементе управления сторонней фирмы. Стек вызовов показывал, что ошибка происходила в функции GlobalUnlock.

Исход

Поначалу я сомневался, что при программировании для Win32 кто-то еще использует функции, работающие с памятью через описатели (GlobalAlloc, GlobalLock, GlobalFree и GlobalUnlock). Однако, взглянув на дизассемблированный код элемента управления сторонней фирмы, я увидел, что его автор, очевидно, портировал элемент с 16-разрядного кода. Я сразу предположил, что в элементе управления неправильно используются API-функции, работающие с памятью через описатели.

Для проверки этой гипотезы я установил точки прерывания на GlobalAlloc, GlobalLock и GlobalUnlock, чтобы найти в элементе управления места распределения и обработки памяти. Как только выполнение программы прервалось в элементе управления, я начал изучать, как он работает с описателями памяти. Все казалось нормальным, пока я не попробовал воспроизвести ошибку.

Через некоторое время после закрытия окна Print я заметил, что функция GlobalAlloc начинает возвращать описатели, заканчивающиеся на нечетные числа, например на 5. В ОС Win32 для доступа к памяти через описатель выполняется разыменование указателя, поэтому я сразу же понял, что нахожусь у самой сути проблемы. Все описатели памяти в Win32 должны заканчиваться на 0, 4, 8 или С в шестнадцатеричной системе, потому что области памяти выравниваются по границам двойных слов. Значения опи-

сателей, возвращаемые `GlobalAlloc`, указывали на наличие какой-то серьезной проблемы.

Вооруженный этой информацией, руководитель проекта уже готов был звонить по телефону разработчикам элемента управления и требовать его исходный код, так как он был уверен, что именно в этом причина ошибки и задержки выпуска программы. Успокоив его, я объяснил, что это ничего не доказывает и что, пока мы не испортили жизнь ни в чем не повинным людям, мне нужно убедиться в том, что причиной ошибки является именно элемент управления. Я продолжил изучать элемент управления и провел несколько следующих часов, отслеживая все выполняемые им манипуляции с описателями памяти. Работал он с описателями памяти правильно, поэтому у меня зародилась новая гипотеза: я предположил, что истинная проблема кроется в основном приложении. Обнаружение ошибки в элементе управления было просто случайным стечением обстоятельств.

Просмотрев код главной программы, я пришел в полное замешательство, потому что приложение было написано по всем правилам Win32 и совершенно не работало с описателями памяти. После этого я перешел к изучению кода вывода на печать. Все выглядело правильным.

Тогда я начал сужать диапазон случаев возникновения ошибки. Спустя некоторое время я обнаружил, что для воспроизведения ошибки нужно открыть окно `Print` и изменить ориентацию листа бумаги. Закрыв окно `Print`, нужно было открыть его заново, потом закрыть опять, и через несколько секунд после этого возникала ошибка. На тот момент я был очень доволен точным воспроизведением ошибки, так как понял, что проблема была скорее всего обусловлена изменением какого-то байта памяти в результате переориентации листа.

При первоначальном просмотре код выглядел отлично, тем не менее я еще раз проанализировал и сверил с документацией MSDN каждую его строку. Через 10 минут ошибка была найдена. Разработчики сохраняли структуру `PRINTDLG`, применявшуюся для инициализации диалогового окна `Print`, при помощи API-функции `PrintDlg`. Третье поле этой структуры, `hDevMode`, представляет собой описатель памяти, выделяемой окном `Print`. Ошибка заключалась в том, что программисты работали с этим значением, как с обычным указателем, не разыменовывая описатель так, как нужно, и не вызывая для него функцию `GlobalLock`. Когда они изменяли значения в структуре `DEVMODE`, они по сути выполняли запись в таблицу глобальных описателей процесса. Эта таблица представляет собой блок памяти, в котором хранится информация обо всех описателях памяти, выделенной в куче. Из-за записи в таблицу глобальных описателей функция `GlobalAlloc` использовала неправильные смещения, получала в таблице неправильные значения и возвращала неверные указатели.

Полученный опыт

Первый урок: внимательно читайте документацию. Если в ней сказано, что структура данных является «глобальным перемещаемым объектом памяти», значит, работа с памятью осуществляется через описатели, и вам нужно

см. след. стр.

правильно выполнять их разыменование или вызывать для них функцию `GlobalLock`. Даже если 16-разрядная ОС Microsoft Windows 3.1 уже кажется древностью, некоторые архаизмы еще сохранились в Win32 API, и об этом нужно помнить.

Кроме того, я узнал, что таблица глобальных описателей хранится в памяти, разрешенной для записи. Если бы меня спросили об этом раньше, я, наверное, сказал бы, что такая важная структура ОС должна находиться в памяти с доступом только для чтения. Я могу только догадываться, что заставило Microsoft пойти на это. С технической точки зрения, память, подерживающая описатели, нужна только для обратной совместимости, и все приложения Win32 должны использовать типы памяти, специфичные для Win32. Если бы таблица глобальных описателей была защищенной, то при каждом вызове функции, работающей с описателями, нужно было бы выполнить два переключения контекста между пользовательским режимом и режимом ядра. Это очень накладно в плане ресурсов процессора, поэтому, наверное, Microsoft не защитила таблицу глобальных описателей именно по этой причине.

Наконец, я понял, что слишком долго изучал элемент управления. Всего для обнаружения ошибки мне понадобилось около семи часов. Однако я мог бы найти ее быстрее, если бы обратил должное внимание на то, что ошибка возникала только при вызове окна Print, код которого находился в основном приложении.

Советы и хитрости

Выше я уже привел некоторые советы и хитрости, способные облегчить отладку неуправляемых приложений. Завершить главу я хочу некоторыми советами, которые помогут выполнять отладку на уровне ассемблера.

Порядок размещения значений в памяти

Порядок размещения значений (endianness) указывает, какой байт хранится по младшему адресу. Процессоры Intel располагают значения в памяти в прямом порядке, т. е. младшие байты хранятся по младшим адресам. Так, значение `0x1234` хранится в памяти как `0x34 0x12`. Глядя на представление памяти в отладчике, очень важно выполнять соответствующие преобразования. Если вы изучаете узлы связанного списка и один из указателей имеет на самом деле значение `0x12345678`, то в окне Мемогу оно будет выведено в формате `0x78 0x56 0x34 0x12`.

Термин «endian» впервые использовал Джонатан Свифт в «Путешествиях Гулливера», а его компьютерное значение появилось благодаря Дэнни Коэну (Danny Cohen) в RFC (Request for Comments) 1980 года, посвященном порядку байтов. Если вас заинтересовала эта история, то работу Дэнни можно найти по адресу: http://www.rdrop.com/~cary/html/endian_faq.html#danny_cohen.

«Мусорный» код

При изучении дампов в окне Disassembly нужно определить, действительно ли вы имеете дело с реальным кодом, но иногда это довольно затруднительно. Вот не-

которые советы, которые помогут вам узнать, на что вы смотрите: на исполняемый код или на что-то другое.

- Я обнаружил, что эту задачу можно облегчить, выбрав в контекстном меню окна Disassembly пункт Code Bytes (байт-коды), включающий отображение идентификаторов операций. Знание некоторых идентификаторов очень часто помогает выяснить, что же вы видите в окне Disassembly.
- Если вы видите последовательность одинаковых команд ADD BYTE PTR [EAX], AL, это не исполняемый код. Это просто ряд нулей.
- Если вы видите символы с очень большими значениями смещений, обычно более 0x1000, вы скорее всего находитесь вне раздела кода. Однако очень большие значения могут также означать, что вы отлаживаете модуль, для которого отсутствуют частные символы.
- Если вы видите ряд команд, которые я в этой главе не описывал, вероятно, вы смотрите на данные.
- Если дизассемблер Visual Studio .NET не может дизассемблировать команду, вместо идентификатора операции он выводит «???».
- Если команда некорректна, дизассемблер выведет «db», после чего будет следовать число. Аббревиатура «db» означает «байт данных» (data byte) и не является корректной командой. Это значит, что согласно картам процессоров Intel идентификатор(ы) операции по этому адресу не соответствует никаким командам.

Регистры и окно Watch

Окно Watch отладчика Visual Studio .NET умеет преобразовывать все регистры в значения. Это позволяет ввести в окне Watch интересующий регистр и привести его к нужному вам типу. Например, если вы изучаете команду работы со строками, то, введя в окне Watch выражение `(char*)@EDI` или `(wchar_t*)@EDI`, вы увидите данные в формате, более легком для понимания.

Изучение файлов ASM

Если вы хотите лучше изучить параллели между ассемблером и исходным кодом, можете приказать Visual Studio .NET генерировать ассемблерные листинги для исходных файлов. Откройте страницу Property Pages, папку C/C++, страницу Output Files и выберите в поле Assembler Output пункт Assembly With Source Code (/FAs) (ассемблерный и исходный код). После этого компилятор будет генерировать для каждого вашего исходного файла соответствующий файл ASM. Возможно, вам не захочется создавать ASM-файлы для каждой компоновки, но они очень наглядно иллюстрируют работу компилятора. Файлы ASM избавят вас от необходимости загружать приложение при каждом приступе интереса к ассемблеру.

Генерируемые файлы почти готовы к компиляции при помощи Microsoft Macro Assembler (MASM), поэтому в них иногда очень сложно разбираться. Большую часть файлов занимают директивы MASM, однако в основных разделах вы увидите код C соответствующими ассемблерными командами под каждой конструкцией. После прочтения этой главы у вас не должно возникать никаких проблем с пониманием файлов ASM.

Резюме

Отладчик Visual Studio .NET поддерживает много мощных возможностей, и в этой главе я познакомил вас с рядом методов отладки неуправляемого кода. Наиболее важный вывод заключается в том, что отладчик может проделать за вас очень большую работу, но для этого нужно владеть эффективными приемами его использования. Вам следует постараться освоить отладчик неуправляемого кода Visual Studio .NET на максимально высоком уровне, чтобы минимизировать проводимое с ним время.

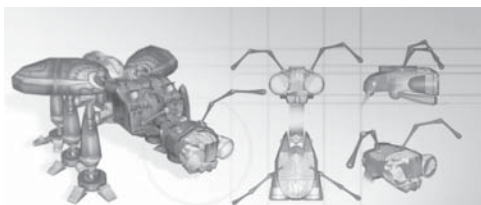
Усложненные точки прерывания для неуправляемого кода помогут вам избежать нудной работы путем указания точного места и условий срабатывания точки прерывания. Для указания отладчику точной области видимости и расположения точек прерывания используется контекстная часть усложненного синтаксиса точек прерывания. Псевдореестры, особенно \$TIB и \$ERR, наделяют условные выражения еще большей силой. Глобальные точки прерывания по данным поддерживают установку прерывания на доступ к памяти; когда интересующие вас данные изменятся, программа остановится. Написанный Майком Мориарти класс аппаратных точек прерывания позволяет устанавливать в своем коде точки прерывания на чтение и запись данных.

Удивительно гибкое окно Watch предоставляет прекрасные возможности, ускоряющие отладку: оно не только разрешает изменять значения переменных, но и поддерживает функции форматирования, чтобы вы могли просматривать данные в наиболее удобном виде. Окно Watch также позволяет вызывать из отладчика функции вашей программы. Благодаря этому можно создавать и использовать специальные отладочные функции, автоматизирующие большинство утомительных отладочных задач. Кроме того, сэкономить время можно, применив быстрое развертывание собственных типов и отображение значений HRESULT. Наконец, новая модель EEAddIn, позволяющая вам использовать собственные правила отображения информации путем вызова ваших DLL окном Watch, открывает новый мир возможностей отображения данных.

Высокой эффективности отладки неуправляемых приложений можно достичь путем удаленной отладки DCOM, а также при помощи Pipes, решения, поддерживающего отладку только неуправляемого кода. Удаленная отладка через Pipes позволяет запускать в отладчике процессы на удаленных машинах, что может оказаться очень полезным разработчикам крупных клиентских приложений. Кроме того, Pipes позволяет подключаться и отлаживать несколько процессов на удаленном компьютере.

Вторая половина этой главы была посвящена тем аспектам языка ассемблера процессоров Intel, которые необходимы для выживания в окне Disassembly. Я начал с основ архитектуры процессоров Intel, таких как регистры и флаги статуса, после чего рассмотрел команды работы со стеком, данными, указателями и строками; команды сравнения и проверки операндов; безусловного и условного переходов и циклов. Я также привел некоторые советы и хитрости, которые помогут сделать отладку на уровне ассемблера максимально эффективной.

Решающим фактором при поиске причины краха программы может оказаться понимание ассемблера. Хотя некоторые люди боятся его, как чумы, на самом деле ассемблер не так уж и сложен, и, конечно, в нем нет ничего мистического.



Улучшенные приемы для неуправляемого кода с использованием WinDBG

Хоть я и потратил время, как мне кажется, на миллион страниц документации к отладчику Microsoft Visual Studio .NET, все же есть и другой отладчик от Microsoft, о котором надо поговорить, — Microsoft WinDBG. Я часто удивляюсь, почему в Microsoft две разные команды работают над отладчиками, но я доволен, что они приложили столько усилий, так как WinDBG имеет некоторые сверхмощные возможности для усмирения ошибок. Когда я спрашивал людей из Microsoft, почему у них два отладчика, они приводили разумные аргументы. Visual Studio .NET прекрасна для разработки приложений, но тем, кто работает над ОС, нужно что-то более наращиваемое для автоматизации задач поиска трудных ошибок, для отслеживания проблем, встречающихся в более чем 40 миллионах строк кода.

WinDBG — мощнейшая вещь. Если Visual Studio .NET предлагает прекрасные расширяемые средства для управления средой (как вы увидите в главе 9), WinDBG — это мускулы, необходимые для пинков и толчков в отлаживаемой программе. Конечно, этой силе можно противопоставить и альтернативные варианты, которые мы обсудим чуть позже.

Многие думают, что WinDBG нужен только разработчикам драйверов, но его сила распространяется и на неуправляемые приложения пользовательского режима. WinDBG может предоставить столько информации о процессах, что в Visual Studio .NET об этом можно лишь мечтать. Чтобы соблазнить вас, замечу, что WinDBG предоставляет вам точки прерывания в реальной памяти и потрясающие методы управления двоичными данными в минидампах, а также позволяет увидеть все кучи ОС и всю информацию об описателях ваших процессов.

Моя цель в этой главе — помочь вам преодолеть кое-какие препятствия, с которыми вы столкнетесь, когда начнете применять WinDBG. Кроме того, я хочу показать некоторые мощные команды и объяснить, как их использовать. Я также помогу вам обойти проблемы, ошибки и прочие странности, с которыми вы встретитесь в WinDBG. Наконец, как я обещал в главе 6, я освещу отладчик Son of Strike (SOS) для работы с управляемыми приложениями и файлами дампов.

На CD, прилагаемом к книге, вы найдете пакет Debugging Tools for Windows (средства отладки для Windows), включающий в себя WinDBG. CD содержит последнюю версию на момент выхода книги в свет. Вам также следует проверить страничку <http://www.microsoft.com/ddk/debugging>, где Microsoft выкладывает последние версии и наиболее важную информацию о Debugging Tools for Windows. Команда разработчиков постоянно обновляет WinDBG, обеспечивая поддержку все большего набора функций и новейших ОС. Для материала этой главы я использовал последнюю версию WinDBG, доступную мне на момент написания книги, — 6.1.0017.0.

Прежде чем начать

Прежде чем нырнуть с головой в WinDBG, я хочу кратко осветить некоторые ключевые моменты. Первое: если у вас появится намек на то, чтобы написать расширение WinDBG, проверьте, что у вас установлен SDK из состава Debugging Tools for Windows. Этот SDK включает заголовочные файлы и примеры, демонстрирующие идеи, заложенные в расширения. При установке укажите выборочную (Custom) установку на соответствующем экране. Экран выборочной установки (рис. 8-1) по умолчанию имеет только один элемент, помеченный как «не устанавливать», — собственно SDK. Щелкните узел SDK и выберите Entire Feature Will Be Installed On Local Hard Drive (все функции будут установлены на локальный диск). Установочный каталог SDK называется также SDK и размещен в основном каталоге Debugging Tools for Windows.

Что делает WinDBG немного странным, так это то, что вы и я полагаем, что UI находится совсем не там, где выполняется основная работа. Этот UI просто является внешним представлением собственно механизма отладки, именуемого DBGENG.DLL. Многие в командах разработчиков ОС Microsoft привыкли к отладчику NTSD (Microsoft NT Symbolic Debugger, символический отладчик Microsoft NT). NTSD — это консольное приложение, являющееся в Debugging Tools for Windows консольным уровнем над механизмом отладки, входит в состав установочного комплекта Debugging Tools for Windows. Это значит, что, зная, как работать с WinDBG, вы легко привыкнете и к NTSD. Кроме того, я предпочитаю простоту графического интерфейса, а вы, возможно, захотите присмотреться к NTSD, так как он будет отпугивать некоторых блуждающих руководителей, любящих сидеть в вашем офисе и заглядывать вам через плечо, наблюдая, как вы работаете. В табл. 8-1 перечислены наиболее интересные программы для разработчиков пользовательского режима, которые устанавливаются в составе Debugging Tools for Windows.

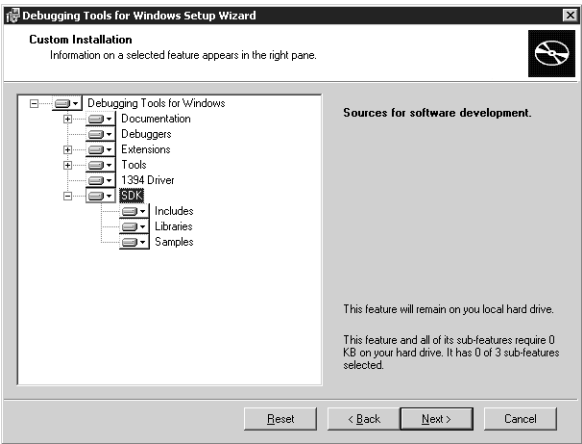


Рис. 8-1. *Выбрана установка SDK в составе Debugging Tools for Windows*

Табл. 8-1. **Дополнительные средства, устанавливаемые в составе Debugging Tools for Windows**

Программы	Описание
CDB.EXE	Такой же отладчик, как и NTSD, за исключением того, что при запуске будет задействована существующая командная оболочка вместо создания новой.
LOGGER.EXE, LOGVIEWER.EXE	Система регистрации для записи всех вызовов API, параметров и возвращаемых значений для выявления проблем взаимодействия.
LIST.EXE	Консольная утилита вывода файлов.
UMDH.EXE	Утилита формирования дампа кучи пользовательского режима.
TLIST.EXE	Выводит список исполняющихся в данное время процессов в консольном окне. Чтобы увидеть другие интересные возможности командной строки утилиты TLIST.EXE, введите -?.
KILLEXE	Убийца процессов, полностью убирающий любой процесс пользовательского режима из памяти.
BREAKIN.EXE	Заставляет выполнить вызов DebugBreak в процессе, указанном в командной строке.

Самый большой плюс WinDBG даже не в том, что он делает больше, автоматически выслеживая непокорные ошибки, а в том, что к нему прилагается действительно хорошая документация. Перед первым запуском WinDBG надо открыть файл DEBUGGER.CHM и почитать его. Имейте при этом в виду, что большая часть сильных сторон WinDBG относится к отладке в режиме ядра, поэтому вы можете спокойно пропустить эти разделы, пока не займетесь разработкой драйверов. Ознакомившись с разделом «Installation and Setup» (установка и настройка) и всем, что относится к пользовательскому режиму в разделе «Debugger Operation» (процесс отладки), как я уже говорил в главе 2, нужно полностью прочитать раздел «Symbols» (символы). В разделе «Debugging Techniques» (приемы отладки) обязательно про-

читите «Elementary Debugging Techniques» (элементарные приемы отладки), «Stack Traces» (трассировка стека) и «Processor Architecture» (архитектура процессора). Так как окно Command (команды) является в WinDBG всем, потратьте время на чтение содержимого раздела «Reference» (ссылки), в особенности «Debugger Commands» (команды отладчика) и «Debugger Extension Commands» (команды расширения отладчика).

Вы, возможно, удивлены: если документация WinDBG столь хороша, зачем писать эту главу? Беда документации в том, что она предполагает, что либо рядом с вами сидит коллега, показывающий вам, как работать с WinDBG, либо телефон, по которому вы можете связаться с разработчиками. Не думаю, что это злой умысел, — просто Microsoft создавала WinDBG в первую очередь для разработчиков ОС. Моя цель в этой главе — ускорить процесс для вас, так как WinDBG выглядит несколько пугающе, если вы им никогда не пользовались. Кроме того, я покажу вам кое-какие тонкости, которые могут сделать отладку намного проще.

Основы

Самые большие проблемы WinDBG — его настройка и огромный объем выходных данных. Я помогу вам преодолеть кое-какие преграды, чтобы изучение WinDBG не было слишком болезненным. В заключение я освещу некоторые странности, чтобы они вас не удивляли.

Первым делом запомните, что WinDBG — это разновидность ретро-отладчика. Черт, консольный NTSD должен вызывать слезы на глазах старых программистов! Но это немного вам поможет. Visual Studio .NET поможет вам при поиске символов и работе с исходным кодом, тогда как для WinDBG вы должны точно сообщить, куда смотреть, чтобы найти то, что нужно. При отладке программы, собранной на той же машине, компилятор и компоновщик Microsoft Visual C++ .NET «знают» размещение символов, а внутри PDB «прописаны» пути ко всем исходным файлам, в результате чего у вас не будет никаких проблем. Однако для сбора информации о символах и строках текста, если программа собиралась не на той машине, где производится отладка, потребуются усилия.

В главе 2 я представлял технологию сервера символов, являющуюся одним из важнейших достижений в отладке для Windows за все время. К этому моменту у вас должен быть настроен собственный сервер символов и установлена переменная среды `_NT_SYMBOL_PATH`, чтобы Visual Studio .NET использовал сервер символов. WinDBG будет автоматически вызывать переменную `_NT_SYMBOL_PATH` в качестве базового пути символов. WinDBG использует рабочие пространства для сохранения нужной информации о каждом отлаживаемом процессе, такой как точки прерывания, размещение окон и пути символов. Сразу после запуска, еще до открытия процесса, WinDBG позволяет изменить параметры базового рабочего пространства, откуда все остальные будут наследовать свои параметры. Вы узнаете, что находитесь в базовом рабочем пространстве по тому, что дочерние окна MDI открыты во фрейме WinDBG. Используя параметры базового рабочего пространства с общими данными, необходимыми для всех процессов, вы избавите себя от огромного количества препятствий.

Настроив переменную среды `_NT_SYMBOL_PATH`, надо сообщить WinDBG, где искать общие исходные файлы. Хотя и можно задать пути к исходным файлам в переменной среды `_NT_SYMBOL_PATH` (детали см. в файле `DEBUGGER.CHM`), есть способ попроще. Откройте базовое рабочее пространство в WinDBG, щелкните в меню `File` (файл), `Source File Path` (путь к исходным файлам), чтобы открылось диалоговое окно `Source File Path`, в котором вы введете пути к размещению общих исходных файлов. По минимуму вам всегда будет нужен путь к исходным файлам библиотеки времени выполнения C и исходным кодам MFC/ATL, поэтому надо ввести что-то похожее на такую последовательность (заметьте: я отделил пути друг от друга, чтобы они легче воспринимались, но вводить их нужно в одной строке):

```
<Visual Studio .NET Installation Dir>\vc7\crt\src;  
<Visual Studio .NET Installation Dir>\vc7\crt\src\intel;  
<Visual Studio .NET Installation Dir>\vc7\atlmfc\include;  
<Visual Studio .NET Installation Dir>\vc7\atlmfc\src\mf;  
<Visual Studio .NET Installation Dir>\vc7\atlmfc\src\atl;  
<Visual Studio .NET Installation Dir>\vc7\atlmfc\src\atl\atls;  
<Visual Studio .NET Installation Dir>\vc7\atlmfc\src\atl\atlmincrt;
```

Каталоги разделяются точкой с запятой — так же, как это делается при обычной записи. Если вы и впрямь горите желанием использовать NTSD, установите такое же значение переменной среды `_NT_SOURCE_PATH`.

Последнее, что надо настроить, — это путь к образу исполняемого файла, который нужен WinDBG для поиска двоичных файлов. Если вы производите отладку в реальном времени, то WinDBG автоматически найдет файлы и загрузит их. Если же вы собираетесь отлаживать минидампы, в которых WinDBG превосходен, надо указать WinDBG, где искать двоичные файлы. Если вы следовали моим рекомендациям в главе 2 при настройке сервера символов, то вы уже поместили символы ОС, вашей разработки и двоичные файлы в ваш сервер символов. Путь к исполняемым файлам можно задать в базовом рабочем пространстве путем выбора из меню `File`, `Image File Path` (путь к двоичным файлам) и ввода той же строки, что применяется в качестве пути к символам, или значения переменной среды `_NT_SYMBOL_PATH`. WinDBG достаточно умен, чтобы корректно работать, получая информацию о двоичных файлах непосредственно от вашего сервера символов.

WinDBG может работать с минидампами независимо от того, откуда они получены: от заказчика или с машины начальника. Так что, даже если вы настроили переменную среды `_NT_EXECUTABLE_IMAGE_PATH`, чтобы указать Visual Studio .NET, где искать исполняемые файлы, Visual Studio .NET их не загрузит. Так как минидампы столь важны для поиска проблем у заказчика (в промышленных условиях), WinDBG жизненно важен, чтобы убить все ошибки.

Открывая процессы для «живой» отладки или отладки минидампов, вы сможете обновить каждое из этих рабочих пространств символами, путями к исходным текстам и двоичным файлам для каждого проекта в отдельности. Каждый раз, когда вы меняете что-либо в рабочем пространстве, включающем в себя установленные точки прерывания, пути к символьным, исходным и двоичным файлам, а также расположение окон, WinDBG будет предлагать сохранить рабочее пространство, прежде чем его закрыть. Поэтому, возможно, в ваших интересах будет всегда сохранять рабочее пространство. Вы можете удалить неиспользуемые рабочие про-

странства или очистить конкретные данные, сохраняемые с рабочим пространством путем выбора соответствующих пунктов обслуживания рабочего пространства меню File.

В заключение вам может понадобиться выделять в окне Command важную информацию цветом. Если вы никогда раньше не запускали WinDBG, то заметите, что он крайне разговорчив. Все происходит в окне Command, и очень легко пропустить что-то важное. Просто загрузка большого процесса может в результате изрыгнуть более 100 строк! К счастью, WinDBG сейчас позволяет раскрасить выходную информацию, так что вы можете отделить зерна от плевел. Цветовые параметры находятся в нижней части диалогового окна Options (настройки), вызываемого выбором Options из меню View (вид).

Плохие новости в том, что значения цветовых элементов недостаточно документированы. Некоторые из этих раскрашиваемых элементов, такие как Enabled Breakpoint Background (фон действующей точки прерывания), не требуют объяснений, но другие, скажем, Error Level Command Window Text (текст уровня ошибки в окне команд), только кажутся понятными: я никогда не видел выбранного мной цвета. На самом деле главное — «подсветить» все вызовы TRACE и OutputDebugString, производимые вашей программой. Вы можете настроить вывод этих важных данных другим цветом, установив цвет Debuggee Level Command Window Text (текст уровня отлаживаемой программы в окне команд). Лично я всегда выбираю зеленый, так как он показывает, что все хорошо.

Чтобы избавить вас от скачков кровяного давления, хочу отметить несколько странностей WinDBG. Первая странность относится к тому, что происходит при завершении вашего процесса. В Visual Studio .NET при завершении процесса нажатие F5 перезапустит отладку. В WinDBG случаются одна или две вещи. Если вы открываете процесс, выбрав Open Executable (открыть исполняемый файл) из меню File, нажатие F5, возможно, предложит вам сохранить рабочее пространство. После щелчка кнопки в информационном окне с запросом рабочее пространство чудесным образом закрывается, а вы видите WinDBG, в котором нет открытого рабочего пространства. Если вам повезет запустить WinDBG для отладки программы из командной строки, нажатие F5 по завершении процесса снова предложит вам сохранить рабочее пространство. Теперь, после подтверждения сохранения рабочего пространства, WinDBG работает! Да, это совершенно противоестественно, но так устроен WinDBG. Если вам надо перезапустить отладку, выберите Restart (перезапустить) из меню Debug (отладка) или нажмите Ctrl+Shift+F5.

И, наконец, WinDBG — это полный бред, что касается положения окон. Он размещает дочернее окно там, где нравится ему, а не вам! Если вы устали от выскакивающих окон при попытке переместить дочернее окно сообщений, отключите пункт Auto-Arrange (размещать автоматически) в меню Window (окно). Хотя WinDBG местами слегка груб, я прощаю это из-за той силы, которую он привносит в отладку.

Прежде чем заняться окном Command, я предлагаю вам день-два попользоваться WinDBG как обычным графическим отладчиком. С ним можно работать, как с обычным отладчиком уровня исходного кода, поэтому вы можете открыть исходный файл, установить курсор на строку и нажать F9 для установки точки прерывания. Так как WinDBG не загружает файлы символов, до того как они понадобятся-

ся, возможно, вы увидите предложение загрузить символы. Всегда щелкайте Yes, и все должно быть хорошо. О проблемах загрузки символов мы еще поговорим. В дополнение к окнам Source (исходный текст) меню View (вид) предлагает различные типы окон. WinDBG имеет полный комплект окон отладки, таких как Registers (регистры), Memory (память) и Locals (локальные переменные). Интересно, что он имеет также окно Scratch Pad (блокнот для заметок), если вы столь ленивы, чтобы нажимать Alt+Tab для доступа к Notepad (Блокнот) для вставки в него отладочной информации или записи заметок. Как вы увидите, начав пользоваться WinDBG, блестящим интерфейсом Visual Studio .NET он не располагает и все же определенно полезен.

Общий вопрос отладки

Как изменить аргумент командной строки своего процесса, если он открыт в WinDBG?

Увы, никак. После того как вы открыли процесс, единственный способ запустить отлаживаемую программу еще раз с другими параметрами командной строки — это закрыть рабочее пространство и либо открыть процесс заново с новыми аргументами через диалоговое окно Open Executable, либо перезапустить WinDBG с новыми параметрами.

Параметры командной строки можно задать одним из двух способов. Первый: в диалоговом окне Open Executable. На рис. 8-2 показано диалоговое окно Open Executable; выделенная область (строка ввода Arguments) — это то место, где вы вводите параметры, передаваемые в командной строке отлаживаемой программе.

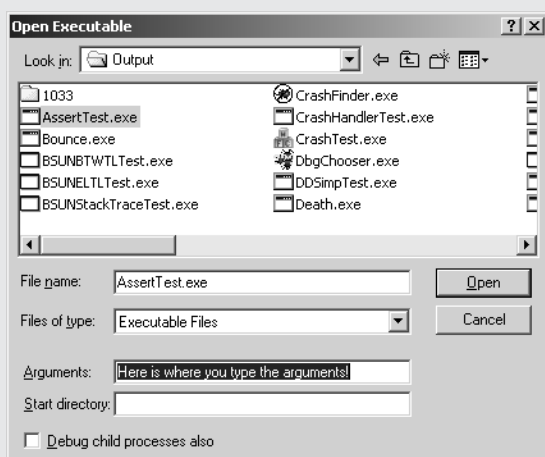


Рис. 8-2. Диалоговое окно WinDBG Open Executable

Другой вариант задания параметров командной строки — ввести параметры отлаживаемой программы после ее имени в командной строке WinDBG.

Что случается при отладке

Теперь, когда у вас есть общее представление о том, что может делать WinDBG и как избежать некоторые присущие ему проблемы, я хочу объяснить, как оседлать его с помощью окна Command. Окно Command является вершиной всего в WinDBG и, хотя его труднее познать, чем воспользоваться графическим интерфейсом, вы сможете отлаживать гораздо быстрее, познакомившись с его командами. Все определяется тем, сколь много усилий вы хотите приложить.

Прежде чем ринуться в чащу команд, надо выяснить парочку вопросов. Во-первых, напомним, что нажатие Alt+I в процессе отладки выводит окно Command наверх и центрирует его. Во-вторых, это синтаксис вычисления адреса, так как очень многие команды базируются на нем. Главный способ указания одиночного адреса, основанного на символе, — использование формата *module!symbol* (модуль/символ), при этом module и symbol чувствительны к регистру. Так, для получения адреса LoadLibraryW вводим kernel32!LoadLibraryW. Для задания адреса, основанного на имени исходного файла и номере строки, правила ввода — '[module!]filename[: linenumber]' ('[[имя модуля] имя файла][: номер строки]'). Обратите особое внимание на разделители — это гравис ('.'). Имя модуля и имя файла необязательны. Если вы пропустили имя модуля ('foo.cpp:23'), WinDBG просмотрит символы во всех загруженных модулях. Пропуск имени файла (':23') предполагает использование файла, которому принадлежит текущий выполняемый оператор.

WinDBG имеет три типа команд. *Стандартные команды* (regular commands) управляют отлаживаемой программой. Например, трассировка (tracing), исполнение по шагам (stepping) и просмотр памяти — это стандартные команды. *Мета-команды* (meta commands; их также называют точка-командами) в основном управляют отладчиком и действиями отладки. Например, создание файлов регистрации, присоединение к процессам и запись файлов дампа это мета-команды. *Команды расширения* (extension commands) производят некоторые действия в отлаживаемой программе и производят анализ ситуаций или состояний. Примеры команд расширения включают вывод описателей, анализ критических секций и анализ аварийного завершения.

Получение помощи

Когда вы начинаете с мерцающего курсора в нижней строке окна Command, думая, какая же нужна команда, обратитесь к справочной системе. Если вам достаточно получить подсказку об имени стандартной команды или ее синтаксисе, введите ? (Command Help — справка по командам), и появится пара страниц со списком, где можно познакомиться с информацией о стандартных командах. Некоторые стандартные команды имеют поддержку ввода -? в качестве параметра, для них вы получите быструю помощь по применению параметров. Вам придется использовать метод проб и ошибок для выяснения, какие из них поддерживают -. Чтобы увидеть список мета-команд, введите .help (Meta-command Help — справка по мета-командам). Так как справочная система по расширенным командам выполнена, как будто ее писали от случая к случаю, то синтаксис этих команд

выглядит несколько иначе. Я опишу справочную систему по этим командам в разделе «Магические расширения». И еще: стандартные и мета-команды не чувствительны к регистру, но все команды расширения вводятся в нижнем регистре.

Возможно, мета-команда `.hh` (Open HTML Help File — открыть HTML-файл справочной системы) самая важная. Указание имени команды в качестве параметра `.hh` откроет файл справочной системы `DEBUGGER.CHM` на вкладке `Index` (указатель), где указанная команда будет подсвечена. Просто нажмите `Enter`, и появится справочная информация по этой команде. Надеюсь, в будущих версиях разработчики WinDBG исправят команду `.hh` так, чтобы она автоматически открывала раздел справочной системы, относящийся к заданной команде.

При ознакомлении со справочной системой по командам обратите особое внимание на раздел `Environment` (среда), появляющийся в каждой команде. Таблица этого раздела описывает ситуации, в которых WinDBG может исполнить команду. Очевидно, что при отладке пользовательского режима, поле `Modes` (режимы) должно будет распознать пользовательский режим. Почти все команды пользовательского режима работают как при выполнении живой отладки, так и при работе с минидампами.

Единственное, что еще не очень ясно в справочной системе для всех команд, — это почему полностью отсутствует логика, когда вы переходите к параметрам, передаваемым командам. Одни команды принимают параметры, которые должны отделяться символом «минус», другие требуют слэша, а третьи вообще не имеют разделителей. Отнеситесь внимательно к документации, чтобы правильно указывать параметры для каждой команды.

Обеспечение корректной загрузки символов

WinDBG превосходит в обработке символов. В то время как Visual Studio .NET не предоставляет вам ни единого шанса увидеть, что же все-таки загружено, WinDBG показывает все. Отлаживая неуправляемый код, я использую WinDBG, чтобы быть уверенным, что все мои символы настроены, прежде чем перейти к Visual Studio .NET. В итоге я точно знаю, где нахожусь, что не хожу по комментариям и что нет символов ОС, указывающих никуда.

Важнейшая возможность обработки символов в WinDBG в том, что в любой момент можно заставить его перезагрузить все символы. В нижней части окна `Symbol Search Path` (путь поиска символов) есть флажок `Reload` (перезагрузить). Изменив каталог символов, установите его и щелкните кнопку `OK`, и WinDBG выгрузит все загруженные символы и загрузит их вновь, исходя из нового каталога символов. Это позволяет выбрать наилучшие символы из возможных. Есть и команды для перезагрузки символов, но сначала я должен показать, как удостовериться, какие символы вы загрузили.

Когда окно `Command` активно, команда `LM` (List Loaded Modules — список загруженных модулей) покажет список модулей и соответствующие им файлы символов. В качестве примера я загрузил программу `ASSERTTEST.EXE`, которая помогает проверить `SUPERASSERT` (имеется в файлах примеров к этой книге), в WinDBG. Если WinDBG остановить на точке прерывания загрузчика, запуск `LM` покажет:


```
0:000> lm
start      end          module name
00400000 0040a000    AssertTest (deferred)
10200000 10287000    MSVC71D    (deferred)
10480000 1053c000    MSVCP71D   (deferred)
60000000 6004a000    BugslayerUtil (deferred)
6d510000 6d58d000    dbghelp    (deferred)
70a70000 70ad4000    SHLWAPI     (deferred)
71950000 71a34000    COMCTL32    (deferred)
77c00000 77c07000    VERSION    (deferred)
77c10000 77c63000    msvcrt      (deferred)
77c70000 77cb0000    GDI32       (deferred)
77d40000 77dc6000    USER32     (deferred)
77dd0000 77e5d000    ADVAPI32    (deferred)
77e60000 77f46000    kernel32    (deferred)
77f50000 77ff7000    ntdll       (pdb symbols)
              \\zeno\WebSymbols\ntdll.pdb\3D6DE29B2\ntdll.pdb
78000000 78086000    RPCRT4      (deferred)
```

Так как загрузка символов занимает огромный объем памяти, WinDBG использует отложенную загрузку символов, т. е. загружает символы, только когда они нужны. Поскольку исходное предназначение WinDBG — отлаживать ОС, загрузка всех символов ОС при первом присоединении к ядру системы сделает WinDBG бесполезным. Таким образом, только что приведенный пример показывает, что я загрузил только символы NTDLL.DLL. Остальные помечены как «deferred» (отложены), потому что у WinDBG нет причин получать доступ к ним. Если б я загрузил файл исходного текста ASSERTTEST.EXE и нажал F9 для установки точки прерывания на строке, WinDBG начал бы загрузку этих символов, пока не нашел бы нужный в этом файле. Вот зачем нужно информационное окно с запросом необходимости загрузки символов. Однако на уровне командной строки вы можете более тонко управлять выбором загружаемых символов.

Чтобы заставить загрузить символ, команда LD (Load Symbols — загрузить символы) делает небольшой трюк. LD принимает только имя файла в командной строке, поэтому, чтобы загрузить символы программы ASSERTTEST.EXE, я ввел `ld asserttest` и получил такой результат:

```
0:000> ld asserttest
*** WARNING: Unable to verify checksum for AssertTest.exe
Symbols loaded for AssertTest
```

WinDBG весьма обстоятелен при работе с символами и сообщает о символах все, что может быть потенциально ошибочным. Так как я использую отладочную версию ASSERTTEST.EXE, то у меня не был задан ключ `/RELEASE` при сборке программы, отключающий инкрементальную компоновку. Как я говорил в главе 2, ключ `/RELEASE` называется неправильно, он должен бы называться `/CHECKSUM`, так как он лишь добавляет контрольную сумму к двоичному файлу и PDB-файлу.

Чтобы загрузить все символы, укажите символ «звездочка» как параметр команды LD: `ld *`. Порывшись в документации WinDBG, вы увидите другую команду — `RELOAD` (Reload Module — перезагрузить модуль), которая в сущности делает то же, что и LD. Для загрузки всех символов с помощью `.RELOAD`, задайте параметр `/f: .RELOAD /f.`

Если вы отлаживаете большую программу, `.RELOAD` может оказаться немного полезнее, так как она будет сообщать только о тех модулях, у которых имеются проблемы с символами, тогда как `LD` покажет результат загрузки каждого модуля. В любом случае вы сразу узнаете, какие символы некорректны.

Вы также можете проверить правильность загрузки символов командой `LM`. После загрузки всех символов `LM` выводит следующее (я перенес последний элемент каждой строки на следующую строку, чтобы все поместилось по ширине на странице):

```
0:000> lm
start  end      module name
00400000 0040a000  AssertTest  C (pdb symbols)
D:\Dev\BookTwo\Disk\Output\AssertTest.pdb
10200000 10287000  MSVCR71D    (pdb symbols)
e:\winnt\system32\msvcr71d.pdb
10480000 1053c000  MSVCP71D    (pdb symbols)
e:\winnt\system32\msvcp71d.pdb
60000000 6004a000  BugslayerUtil  C (pdb symbols)
D:\Dev\BookTwo\Disk\Output\BugslayerUtil.pdb
6d510000 6d58d000  dbghelp     (pdb symbols)
\\zeno\WebSymbols\dbghelp.pdb\
819C4FBAB64844F3B86D0AEEDDCE632A1\dbghelp.pdb
70a70000 70ad4000  SHLWAPI     (pdb symbols)
\\zeno\WebSymbols\shlwapi.pdb\3D6DE26F2\shlwapi.pdb
71950000 71a34000  COMCTL32    (pdb symbols)
\\zeno\WebSymbols\MicrosoftWindowsCommon-Controls-
60100-comctl32.pdb\3D6DD9A81\
MicrosoftWindowsCommon-Controls-
60100-comctl32.pdb
77c00000 77c07000  VERSION     (pdb symbols)
e:\winnt\symbols\dll\version.pdb
77c10000 77c63000  msvcrt      (pdb symbols)
\\zeno\WebSymbols\msvcrt.pdb\3D6DD5921\msvcrt.pdb
77c70000 77cb0000  GDI32       (pdb symbols)
\\zeno\WebSymbols\gdi32.pdb\3D6DE59F2\gdi32.pdb
77d40000 77dc6000  USER32      (pdb symbols)
\\zeno\WebSymbols\user32.pdb\3DB6D4ED1\user32.pdb
77dd0000 77e5d000  ADVAPI32    (pdb symbols)
\\zeno\WebSymbols\advapi32.pdb\3D6DE4CE2\advapi32.pdb
77e60000 77f46000  kernel32    (pdb symbols)
\\zeno\WebSymbols\kernel32.pdb\3D6DE6162\kernel32.pdb
77f50000 77ff7000  ntdll       (pdb symbols)
\\zeno\WebSymbols\ntdll.pdb\3D6DE2982\ntdll.pdb
78000000 78086000  RPCRT4      (pdb symbols)
\\zeno\WebSymbols\rpcrt4.pdb\3D6DE2F92\rpcrt4.pdb
```

Буква «С» после имени модуля указывает, что в модуле или файле символов отсутствует контрольная сумма символов. Символ «решетка» после имени модуля указывает, что символы в файле символов и исполняемом файле не соответствуют друг другу. Да, WinDBG загрузит символы посвежее, даже если это неправильно. В предыдущем примере жизнь хороша, и все символы корректны. Однако со-

вершено нормально, что «решетка» стоит рядом с COMCTL32.DLL. Это потому, что он, видимо, меняется с каждым пакетом обновления, исправляющим ошибку защиты в Microsoft Internet Explorer, и шансы получить в распоряжение корректную таблицу символов для COMCTL32.DLL почти нулевые. Чтобы поточнее узнать, какие модули и соответствующие файлы символов загружены, укажите *v* в команде *LM*. Чтобы показать единственный модуль в следующем примере, я задал параметр *m* для выбора конкретного модуля.

```
0:000> lm v m gdi32
start      end          module name
77c70000 77cb0000  GDI32          (pdb symbols)
              \\zeno\WebSymbols\
              gdi32.pdb\3D6DE59F2\gdi32.pdb
Loaded symbol image file: E:\WINNT\system32\GDI32.dll
Image path: E:\WINNT\system32\GDI32.dll
Timestamp: Thu Aug 29 06:40:39 2002 (3D6DFA27) Checksum: 0004285C
File version:      5.1.2600.1106
Product version:   5.1.2600.1106
File flags:        0 (Mask 3F)
File OS:           40004 NT Win32
File type:         2.0 Dll
File date:         00000000.00000000
CompanyName:      Microsoft Corporation
ProductName:       Microsoft® Windows® Operating System
InternalName:      gdi32
OriginalFilename:  gdi32
ProductVersion:    5.1.2600.1106
FileVersion:       5.1.2600.1106 (xpsp1.020828-1920)
FileDescription:   GDI Client DLL
LegalCopyright:    © Microsoft Corporation. All rights reserved.
```

Чтобы точно узнать, где WinDBG загружает символы и почему, расширенная команда *!sym* предлагает параметр *noisy*. Вывод в окнах Command показывает, через что проходит сервер символов WinDBG, чтобы найти и загрузить символы. Вооружившись этими результатами, вы сможете решить всевозможные проблемы загрузки символов, с которыми столкнетесь. Чтобы отключить многословный вывод, исполните команду *!sym quiet*.

И последнее о символах. WinDBG имеет встроенный браузер символов. Команда *X* (Examine Symbols — проверить символы) позволяет просматривать символы глобально, применительно к модулю или в локальном контексте. Указав формат *module!symbol*, вы избавите себя от отслеживания места хранения символа. Кроме того, команда *X* не чувствительна к регистру, что упрощает жизнь. Чтобы увидеть адрес символа *LoadLibraryW* в памяти, введите:

```
0:000> x kernel32!LoadLibraryw
77e8a379  KERNEL32!LoadLibraryW
```

Формат *module!symbol* поддерживает «звездочку», поэтому, если вы хотите, например, увидеть в модуле *KERNEL32.DLL* что-либо, имеющее «lib» в имени символа,

ла, введите `x kernel! *Lib*`, что хорошо работает и тоже не чувствительно к регистру. Чтобы увидеть все символы модуля, напишите «звездочку» вместо имени символа. Использование «звездочки» в качестве параметра приведет к выводу локальных переменных в текущей области видимости, что идентично команде `DV` (Display Variables — отобразить переменные), которую мы обсудим в разделе «Просмотр и вычисление переменных».

Процессы и потоки

Разобравшись с символами, можно перейти к запуску процессов под управлением WinDBG. Подобно Visual Studio .NET, WinDBG способен отлаживать одновременно любое количество процессов. Немного интереснее его делает то, что вы располагаете лучшим контролем над отлаживаемыми процессами, порожденными из отлаживаемого процесса.

Отладка дочерних процессов

В самом низу диалогового окна Open Executable (рис. 8-2) имеется флажок Debug Child Processes Also (отлаживать также и дочерний процесс). Установив его, вы сообщаете WinDBG, что вы также хотите отлаживать любые процессы, запущенные отлаживаемым процессом. При работе под Microsoft Windows XP/Server 2003, если вы забыли установить этот флажок при открытии процесса, вы можете изменить этот параметр «на лету» командой `.CHILDBG` (Debug Child Processes — отлаживать дочерний процесс). Собственно `.CHILDBG` сообщит вам текущее состояние. Команда `.CHILDBG 1` включит отладку дочерних процессов, а `.CHILDBG 0` отключает ее.

Чтобы показать возможности работы со многими процессами и потоками, я приведу несколько результирующих распечаток отладки процессора командной строки (CMD.EXE). После того как CMD.EXE начнет исполняться, я запущу `NOTEPAD.EXE`. Если вы проделаете те же шаги при разрешенной отладке дочерних процессов, как только загрузите `NOTEPAD.EXE`, WinDBG остановится на точке прерывания загрузчика для `NOTEPAD.EXE`. То, что WinDBG остановил `NOTEPAD.EXE`, — логично, но это останавливает и `CMD.EXE`, так как оба процесса теперь работают совместно в одном цикле отладки.

Чтобы увидеть в графическом интерфейсе исполняющиеся сейчас процессы, выберите Processes And Threads (процессы и потоки) из меню View. Вы увидите нечто вроде того, что изображено на рис. 8-3. В окне Processes And Threads процессы изображены как корневые узлы, а потоки процессов — как дочерние. Числа рядом с `CMD.EXE (000:9AC)` являются номером процесса WinDBG, после которого указан идентификатор процесса Win32. Для `CMD.EXE` поток `000:9B0` обозначает идентификатор потока WinDBG и идентификатор потока Win32. Номера процессов и потоков WinDBG уникальны в течение всего времени работы WinDBG. Это значит, что никогда не может появиться другой процесс с номером 1, пока я не перезапущу WinDBG. Номера процессов и потоков WinDBG важны, так как они служат для установки точек прерывания для процессов и потоков, а также могут использоваться в качестве модификаторов в командах.

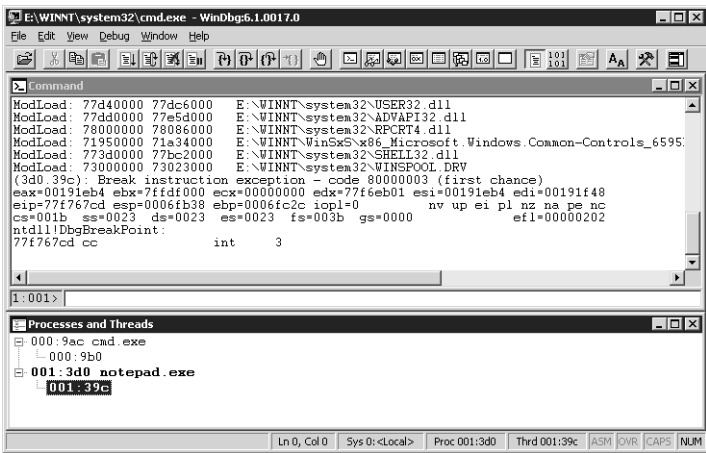


Рис. 8-3. Окно Processes And Threads

Просмотр процессов и потоков в окне Command

Все, что WinDBG отображает в окне, позволяет просмотреть соответствующая команда окна Command. Для просмотра процессов и потоков служит команда | (Process Status — состояние процесса). Результат работы для двух процессов, показанных на рис. 8-3, выглядит так:

```
0:001> |
0      id: 9ac      create   name: cmd.exe
. 1     id: 3d0     child    name: notepad.exe
```

Точка в левой позиции индицирует активный процесс, т. е. все вводимые вами команды будут работать с этим процессом. Другое интересное поле показывает, как был запущен процесс в отладчике. «Create» означает, что процесс создан WinDBG, а «child» — процесс, порожденный родительским процессом.

Перегруженная команда \$ имеет два варианта: !\$ (Set Current Process — установить текущий процесс), а ~\$ (Set Current Thread — установить текущий поток) изменяет текущий активный процесс. К вашим услугам также окно Processes And Threads (процессы и потоки), вызываемое двойным щелчком процесса, который вы хотите сделать активным. Полужирным начертанием выделен активный процесс. Используя команду \$, необходимо задать процесс в виде префикса команды. Так, для переключения со второго процесса на первый, нужно ввести !0s. Чтобы выяснить, какой процесс активен, взгляните на крайние слева номера строки ввода окна Command. При смене процессов номера меняются. В примере с CMD.EXE и NOTEPAD.EXE при переключении на первый процесс путем повторной выдачи команды | результат выглядит немного иначе:

```
0:000> |
. 0      id: 9ac      create   name: cmd.exe
# 1     id: 3d0     child    name: notepad.exe
```

Разница — в символе «#» перед процессом NOTEPAD.EXE. Символ «#» указывает процесс, вызвавший исключение, остановившее его в WinDBG. Так как NOTEPAD.EXE находится на точке прерывания, то последняя и является причиной исключения.

Просмотр потоков почти идентичен просмотру процессов. Я собираюсь запустить NOTEPAD.EXE, поэтому я в WinDBG нажимаю F5. В NOTEPAD.EXE я открою диалоговое окно File Open (открыть файл), так как оно создаст целый букет потоков, а в WinDBG нажму Ctrl+Break для прерывания внутри отладчика. Если вы проделываете то же самое и у вас открыто окно Processes And Threads, вы увидите, что NOTEPAD.EXE имеет четыре потока, а CMD.EXE — два.

Команда ~ (Thread Status — состояние потока) показывает активные потоки текущего процесса. Переключение к процессу NOTEPAD.EXE и ввод команды ~ выводит следующую информацию:

```
1:001> ~
.  1  Id: 3d0.39c Suspend: 1 Teb: 7ffde000 Unfrozen
   2  Id: 3d0.1a4 Suspend: 1 Teb: 7ffdd000 Unfrozen
   3  Id: 3d0.8f0 Suspend: 1 Teb: 7ffdc000 Unfrozen
   4  Id: 3d0.950 Suspend: 1 Teb: 7ffdb000 Unfrozen
```

Как и в случае с |, команда ~ использует точку для индикации текущего потока, а символ «#» — для обозначения потока, который либо вызвал исключение, либо был активен при подключении к нему отладчика. В следующем столбце отображается номер потока WinDBG. Так же, как и с номерами процессов, может быть только один поток с номером 2 за все время жизни экземпляра WinDBG. Далее идут значения ID — идентификаторы процессов Win32, за которыми следуют идентификаторы потоков. Счетчик приостановок (suspend count) немного сбивает с толку. Значение счетчика 1 указывает на то, что поток не приостанавливался. Справочная система по команде ~ показывает значение счетчика приостановок, равное 0, которого я никогда не видел. После счетчика приостановок идет линейный адрес (linear address) блока переменных окружения потока (Thread Environment Block, TEB). TEB — это то же, что и блок информации о потоке (Thread Information Block, TIB), обсуждавшийся в главе 7, который в свою очередь является адресом блока данных потока, содержащего информацию экземпляра потока, такую как стек и параметры инициализации COM. Наконец, Unfrozen (размороженный) индицирует, использовали ли вы команду ~F (Freeze Thread — заморозить поток) для «замораживания» потока. Замораживание потока в отладчике сродни вызову `SuspendThread` для этого потока из вашей программы. Это остановит поток до его «разморозки».

По умолчанию команда работает для текущего потока, но иногда хочется увидеть информацию и о другом потоке. Скажем, чтобы увидеть регистры другого потока, надо использовать модификатор потока перед командой R (Registers — регистры): ~R. Если у вас открыто несколько процессов, нужно также добавлять к командам модификатор процесса. Команда |0~0r показывает регистры для первого процесса и первого потока независимо от того, какие процесс и поток активны.

Создание процессов из окна Command

Теперь, когда вы научились просматривать процессы и потоки, я могу перейти к некоторым более продвинутым приемам запуска процессов под WinDBG. Коман-

да `.CREATE` (Create Process — создать процесс) позволяет вам запускать произвольные процессы. Это весьма полезно, если необходимо отлаживать различные аспекты COM+ или других кросс-процессных приложений. Основные параметры `.CREATE` — полный путь к процессу, который надо запустить, и параметры командной строки этого процесса. Так же, как и при обычном запуске любого процесса, лучше заключить путь и имя процесса в кавычки, дабы избежать проблем с пробелами. Ниже показано применение `.CREATE` для запуска программы Solitaire на одной из моих машин для программирования:

```
.create "e:\winnt\system32\sol.exe"
```

После нажатия клавиши Enter WinDBG сообщает, что процесс будет создан для дальнейшего исполнения. Это значит, что WinDBG должен разрешить «раскрутиться» схеме отладчика, чтобы обработать уведомление о создании процесса. WinDBG уже сделал вызов `CreateProcess`, но отладчик его еще не видит! Нажав F5, вы освободите цикл отладки. Появляется уведомление о создании процесса, и WinDBG остановится на точке прерывания загрузчика. Если вы применяете команду `|` для просмотра процессов, WinDBG покажет процессы, запущенные `.CREATE` с пометкой «create», как будто вы запустили сеанс отладчика, указав этот процесс.

Присоединение к процессам и отсоединение от них

При отладке уже работающего процесса вам пригодится команда `.ATTACH` (Attach to Process — присоединиться к процессу). Сейчас мы обсудим все аспекты присоединения к процессу. В следующем разделе мы обсудим неразрушающее присоединение, при котором процесс не работает в цикле отладчика.

Команда `.ATTACH` требует указания ID процесса для присоединения к процессу. Если вы располагаете физическим доступом к машине, на которой выполняется процесс, можно увидеть ID процесса в диспетчере задач (Task Manager), но при удаленной отладке это сделать трудновато. К счастью, разработчики WinDBG добавили команду `.TLIST` (List Process Ids — вывести ID процессов) для вывода списка исполняющихся на машине процессов. Если вы отлаживаете сервисы Win32, укажите параметр `-v` команды `.TLIST`, чтобы увидеть, какие сервисы в каких процессах выполняются. Вывод `.TLIST` выглядит так:

```
0n1544 e:\winnt\system32\sol.exe
0n1436 E:\Program Files\Windows NT\Pinball\pinball.exe
0n2120 E:\WINNT\system32\winmine.exe
```

Впервые увидев этот вывод, я подумал, что в этой команде ошибка и кто-то случайно напечатал «0n» вместо «0x». Однако позже я узнал, что `0n` — такой же стандартный префикс ANSI для десятичных значений, как `0x` для шестнадцатиричных.

Располагая десятичным значением ID процесса, вы передаете его как параметр команде `.ATTACH` (если, конечно, вы используете префикс `0n`, или это не будет работать). Так же, как и при создании процесса, WinDBG что-либо скажет о том, что подключение произойдет при следующем исполнении, поэтому вам нужно нажать F5 для запуска цикла отладки. С этого момента вы отлаживаете процесс, к которому присоединились. Разница только в том, что `|` пометит процесс как «attach» в своем выводе.

При отладке под Windows XP/Server 2003 для освобождения отладчика служит команда `.DETACH` (Detach from Process — отсоединиться от процесса). Так как это работает только в текущем процессе, вам нужно переключиться на процесс, от которого хотите отсоединиться, прежде чем исполните команду `.DETACH`. В любой момент вы можете снова присоединиться к процессу для полной его отладки.

Если вы просто хотите присоединиться к процессу сразу после запуска WinDBG, когда еще не открыто окно Command, нажмите F6 либо выберите из меню File Attach To A Process (присоединиться к процессу). В появившемся диалоговом окне Attach To Process (присоединиться к процессу) можно раскрыть узлы дерева для просмотра командных строк процессов. Если, как случается, процесс содержит сервисы Win32, вы их тоже увидите. Выбрав процесс, щелкните ОК, и вы погрузитесь в отладку.

Неразрушающее присоединение

Только что описанное полное присоединение прекрасно, так как вы располагаете доступом ко всем способам отладки, например, к точкам прерывания. Однако в Microsoft Windows 2000 процесс, запущенный однажды под отладчиком, будет работать под ним вечно. Это не всегда удобно, если вы пытаетесь отлаживать рабочие серверы, так как вам придется оставлять кого-то зарегистрированным на этом сервере с полными правами администратора, чтобы мог работать WinDBG, не говоря уж о замедлении процессов отладчиком. К счастью, в Windows XP/Server 2003 можно отсоединяться от отлаживаемых процессов (то, о чем я просил еще во времена Microsoft Windows 3.1!).

Чтобы сделать промышленную отладку под Windows 2000 попроще, WinDBG предлагает неразрушающее присоединение. WinDBG приостанавливает процесс, чтобы вы могли исследовать его с помощью команд, но вы не можете осуществлять обычные задачи отладки, скажем, устанавливать точки прерывания. Это приемлемый компромисс: вы можете получить полезную информацию, например, состояние описателей, причем затем процесс будет работать на полной скорости.

Возможно, самый лучший вариант неразрушающей отладки — использование отдельного экземпляра WinDBG. Как вы скоро увидите, для продолжения процесса, возобновляющего все потоки, рабочее пространство нужно закрыть. Если вы уже отлаживаете процессы, WinDBG должен будет сразу остановить эти процессы. Прежде чем выбрать отлаживаемый процесс, в нижней части диалогового окна Attach To Process (рис. 8-4) установите флажок Noninvasive (неразрушающее), и вы не попадете в полную отладку.

Когда вы щелкнете ОК, WinDBG будет готов к нормальной отладке. Однако предупреждение в верхней части окна Command, показанное здесь, поможет вам вспомнить, что вы делаете:

```
WARNING: Process 1612 is not attached as a debuggee
         The process can be examined but debug events will not be received
```

Внимание: Процесс 1612 не присоединен как отлаживаемый процесс.

Процесс доступен для исследования, но события отладки не обрабатываются.

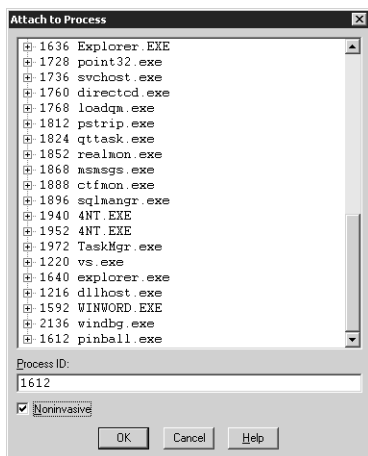


Рис. 8-4. Подготовка к неразрушающей отладке

Присоединившись, можно исследовать в процессе что угодно. Завершив исследование, надо освободить процесс, чтобы продолжить его исполнение. Лучший способ освободить отлаживаемую программу — дать команду **Q** (Quit — завершить). Она закроет рабочее пространство, но WinDBG продолжит работать — потом вы сможете опять присоединиться. **.DETACH** тоже работает, но вам придется завершить WinDBG, так как нет способа присоединиться к процессу снова в этом же сеансе.

Общие вопросы отладки в окне Command

В этом разделе я объясню, как начать отладку с помощью WinDBG, и расскажу о ключевых командах, позволяющих эффективно выполнять отладку из окна Command. Вы узнаете также о некоторых хитростях. Пересказывать документацию я не буду, но прочитать ее настоятельно рекомендую.

Просмотр и вычисление переменных

Просмотр локальных переменных — это вотчина команды **DV** (Display Local Variables — отобразить локальные переменные). Единственное, что слегка путает при работе с WinDBG, — это просмотр локальных переменных вверх по стеку. На самом деле эта команда исполняется в виде нескольких команд, которые делают то, что происходит автоматически при щелчке в окне Call Stack (стек вызовов).

Первый шаг — дать команду **K** (Display Stack Backtrace — отобразить обратную трассировку стека) с модификатором **N**, чтобы увидеть стек вызовов с номерами фреймов в самой левой колонке каждого элемента стека (между прочим, моя любимая команда отображения стека — **KP** — показывает стек со значениями параметров, передаваемых функциям в каждом элементе стека). Номера фреймов обычны в том смысле, что вершина стека всегда имеет номер 0, следующий элемент — 1 и т. д. Эти номера фреймов понадобятся вам, чтобы указать команде **.FRAME** (Set Local Context — установить локальный контекст) переместиться вниз по стеку. Значит, чтобы просмотреть локальные переменные функции, которая вызвала

текущую функцию, вы используете последовательность команд, приведенную ниже. Для перемещения контекста обратно к вершине стека дайте команду `.frame 0`:

```
.frame 1
dv
```

Команда `DV` возвращает достаточно информации, чтобы предоставить вам суть происходящего с локальными переменными. Следующий вывод получен в результате исполнения команды `DV` при отладке программы `PDB2MAP.EXE` из главы 12.

```
cFuncFMT = CResString
cIM = CImageHlp_Module
szBaseName = Array [260]
pMark = cccccccc
dwBase = 0x400000
bEnumRet = 0xc0000000
argc = 2
argv = 00344e18
fileOutput = 00000000
szOutputName = Array [260]
iRetValue = 0
bRet = 1
hFile = 000007c8
cRS = CResString
```

Увидеть больше позволяет команда `DT` (`Display Type` — отобразить тип), которая может выполнять проход по связанным спискам и перемалывание массивов. К счастью, вы можете задать в `DT` параметр `/?`, чтобы быстро получить справку, находясь в центре боевых действий.

Еще `DT` может производить поиск типов символов. Вместо передачи ей имени или адреса переменной вы указываете параметр в формате `module! type`, где `type` — либо полное имя типа, либо содержит звездочку для поиска подвыражений. Так, увидеть типы, начинающиеся с «`IMAGE`» в `PDB2MAP`, позволяет `dt pdb2map! IMAGE*`. Указав тип полностью, вы увидите все поля этого типа, если это класс или структура, либо лежащий в основе базовый тип, если это `typedef`.

Последняя из команд вычисления — `??` (`Evaluate C++ Expression` — вычислить выражение C++) — служит для проверки арифметики указателей и управления другими потребностями в вычислениях C++. Внимательно прочтите документацию по работе с выражениями, так как этот процесс не так прост, как кажется. Теперь, когда вы можете просмотреть и вычислить все свои переменные, самое время обратиться к исполнению, проходу по шагам и остановке программ.

Исполнение, проход по шагам и трассировка

Как вы, наверное, уже поняли, нажатие `F5` продолжает исполнение после его прерывания в WinDBG. Вы не могли этого заметить, но нажатие `F5` просто выполняет команду `G` (`Go` — дальше). Совершенно ясно, что в качестве параметра команды `G` вы можете указать адрес команды. WinDBG использует этот адрес как одно-разовую точку прерывания, и, таким образом, вы можете запустить исполнение

до этого места. Замечали ли вы, что нажатие Shift+F11 (команда Step Out), выполняет команду G, дополненную адресом (иногда в форме выражения)? Этот адрес есть адрес возврата на вершину стека. Вы можете проделать то же самое в окне Command, но вместо ручного вычисления адреса возврата можно использовать псевдорегистр \$ra в качестве параметра, чтобы WinDBG сам вычислил адрес возврата. Имеются и другие псевдорегистры, но не все из них применимы в пользовательском режиме. Задайте «Pseudo-Register Syntax» в справочной системе WinDBG, чтобы найти остальные псевдорегистры. Заметьте: эти псевдорегистры WinDBG характерны только для WinDBG и не используются в Visual Studio .NET.

Для управления трассировкой и движением по шагам служат команды T (Trace) и P (Step) соответственно. Напомню, что трассировка будет проходить внутри любой встреченной функции, тогда как прохождение по шагам — сквозь вызовы функций. Один аспект, отличающий WinDBG от Visual Studio .NET, состоит в том, что WinDBG не переключается автоматически между движением по шагам в тексте исходного кода и в командах ассемблерного кода только потому, что вы случайно переключаете фокус ввода между окнами Source (исходный код) и Disassembly (дизассемблированный код). По умолчанию WinDBG движется по шагам в строках исходного текста, если они загружены из места размещения исполняемого файла. Если вы хотите идти шагами по ассемблерным командам, то либо снимите флажок Source Mode (режим исходного текста) в меню Debug, либо дайте команду .LINES (Toggle Source Line Support — переключить поддержку исходного кода) с параметром -d.

Как и G, команды T и P делают то же самое, что и нажатие кнопок F11 (или F8) и F10 в окне Command. Вы можете также указать или адрес, до которого должна выполняться трассировка, или движение по шагам, или количество шагов, которое необходимо выполнить. Это пригодится, так как иногда это проще, чем устанавливать точку прерывания. В сущности это команда «run-to-cursor» (исполняй до курсора), выполняемая вручную.

Две относительно новые команды для движения по шагам и трассировки: TC (Trace to Next Call — трассировать до следующего вызова) и PC (Step to Next Call — шаг до следующего вызова). Разница между ними в том, что они выполняют движение по шагам или трассировку, пока не попадетсся следующий оператор CALL. При выполнении PC, если указатель команд находится на команде CALL, исполнение будет продолжаться, пока не произойдет возврат из подпрограммы. TC сделает шаг внутрь подпрограммы и остановится на следующей команде CALL. Я нахожу TC и PC полезными, когда хочу пропустить часть функции, но не выходить из нее.

Трассировка данных и наблюдение за ними

Одна из самых больших проблем при выявлении проблем производительности программ (быстродействия) в том, что почти невозможно прочесть и точно увидеть, что происходит на самом деле. Так, код Standard Template Library (стандартной библиотеки шаблонов, STL) создает одну из самых больших проблем быстродействия при отладке приложений других программистов. В результате в код впихивается столько inline-функций (а код STL вообще почти невозможно читать), что анализ путем чтения просто нереален. Но, так как STL негласно выделяет для себя столько памяти и производит всякие блокировки там и сям, жизненно важ-

но иметь способ увидеть, что на самом деле вытворяют функции, использующие STL. К счастью, WinDBG имеет ответ на эту головоломку — и в этом главное отличие WinDBG от Visual Studio .NET — команду WT (Trace and Watch Data — трассировать данные и наблюдать за ними).

WT показывает в иерархическом виде вызовы всех функций в вызове одной функции. В конце трассировки WT показывает точно, какие функции вызывались и сколько раз вызывалась каждая. Кроме того (и это важно при решении проблем быстродействия), WT показывает, сколько было сделано переходов в режим ядра. Для повышения быстродействия главное исключить побольше переходов в режим ядра, поэтому то, что WT — один из немногих способов увидеть такую информацию, делает ее ценной вдвойне.

Как вы догадываетесь, вся эта трассировка может генерировать в окне Command тонны хлама, которые, возможно, вы захотите сохранить в виде файла. К счастью, программисты WinDBG удовлетворили требования полного сохранения всей системы регистрации. Открыть файл регистрации очень просто — укажите имя файла регистрации как параметр команды .LOGOPEN (Open Log File — открыть файл регистрации). Вы также можете добавлять к существующему файл регистрации командой .LOGAPPEND (Append Log File — добавить файл регистрации). При завершении отладки вызовите .LOGCLOSE (Close Log File — закрыть файл регистрации).

Эффективное использование WT для получения поддающегося интерпретации вывода без всего лишнего, через что придется продираться, требует планирования. WT трассирует, пока не попадется адрес возврата из текущей подпрограммы. А значит, вам нужно тщательно позиционировать указатель команд за один-два шага до применения WT. Первое место — непосредственный вызов функции, которую вы хотите исполнить. Это нужно делать на уровне ассемблерного кода, поэтому вам понадобится установить точку прерывания прямо на команде вызова подпрограммы или установить движение по шагам на уровне ассемблерного кода и дойти поэтапно до команды вызова. Второе место — на первой команде функции. Вы можете шагать до команды PUSH EBP или установить точку прерывания на открывающей фигурной скобке функции в окне Source (исходный код).

Прежде чем перейти к параметрам WT, я хочу обсудить ее вывод. Для простоты я написал WExample — маленькую программу с несколькими функциями, вызывающими самих себя (вы найдете ее среди примеров на CD). Я устанавливаю точку прерывания на первую команду в wmain и даю WT для получения результатов под Windows XP SP1, как показано в листинге 8-1 (заметьте: я сократил некоторые пробелы и перенес некоторые строки, чтобы листинг поместился на странице).

Листинг 8-1. Вывод команды wt WinDBG

```
0:000> wt
Tracing WExample!wmain to return address 0040139c
3    0 [ 0] WExample!wmain
3    0 [ 1]   WExample!Foo
3    0 [ 2]     WExample!Bar
3    0 [ 3]      WExample!Baz
3    0 [ 4]       WExample!Do
3    0 [ 5]        WExample!Re
```

см. след. стр.

```

3      0 [ 6]          WTEExample!Mi
3      0 [ 7]          WTEExample!Fa
3      0 [ 8]          WTEExample!So
3      0 [ 9]          WTEExample!La
3      0 [10]          WTEExample!Ti
6      0 [11]          WTEExample!Do2
3      0 [12]          kernel32!Sleep
3      0 [13]          kernel32!SleepEx
18     0 [14]          kernel32!_SEH_prolog
15    18 [13]          kernel32!SleepEx
16     0 [14]          ntdll!
                        RtlActivateActivationContextUnsafeFast
20    34 [13]          kernel32!SleepEx
15     0 [14]          kernel32!BaseFormatTimeOut
26    49 [13]          kernel32!SleepEx
3      0 [14]          ntdll!ZwDelayExecution
2      0 [15]          SharedUserData!
                        SystemCallStub
1      0 [14]          ntdll!ZwDelayExecution
31    55 [13]          kernel32!SleepEx
3      0 [14]          kernel32!SleepEx
14     0 [15]          ntdll!
                        RtlDeactivateActivationContextUnsafeFast
4      14 [14]          kernel32!SleepEx
36     73 [13]          kernel32!SleepEx
9      0 [14]          kernel32!_SEH_epilog
37    82 [13]          kernel32!SleepEx
4     119 [12]          kernel32!Sleep
8     123 [11]          WTEExample!Do2
2      0 [12]          WTEExample!_RTC_CheckEsp
11   125 [11]          WTEExample!Do2
2      0 [12]          WTEExample!_RTC_CheckEsp
13   127 [11]          WTEExample!Do2
5     140 [10]          WTEExample!Ti
2      0 [11]          WTEExample!_RTC_CheckEsp
7     142 [10]          WTEExample!Ti
5     149 [ 9]          WTEExample!La
2      0 [10]          WTEExample!_RTC_CheckEsp
7     151 [ 9]          WTEExample!La
5     158 [ 8]          WTEExample!So
2      0 [ 9]          WTEExample!_RTC_CheckEsp
7     160 [ 8]          WTEExample!So
5     167 [ 7]          WTEExample!Fa
2      0 [ 8]          WTEExample!_RTC_CheckEsp
7     169 [ 7]          WTEExample!Fa
5     176 [ 6]          WTEExample!Mi
2      0 [ 7]          WTEExample!_RTC_CheckEsp
7     178 [ 6]          WTEExample!Mi
5     185 [ 5]          WTEExample!Re

```

```
2      0 [ 6]           WTEsample!_RTC_CheckEsp
7    187 [ 5]           WTEsample!Re
5    194 [ 4]           WTEsample!Do
2      0 [ 5]           WTEsample!_RTC_CheckEsp
7    196 [ 4]           WTEsample!Do
5    203 [ 3]           WTEsample!Baz
2      0 [ 4]           WTEsample!_RTC_CheckEsp
7    205 [ 3]           WTEsample!Baz
5    212 [ 2]           WTEsample!Bar
2      0 [ 3]           WTEsample!_RTC_CheckEsp
7    214 [ 2]           WTEsample!Bar
5    221 [ 1]           WTEsample!Foo
2      0 [ 2]           WTEsample!_RTC_CheckEsp
7    223 [ 1]           WTEsample!Foo
6    230 [ 0] WTEsample!wmain
2      0 [ 1]           WTEsample!_RTC_CheckEsp
8    232 [ 0] WTEsample!wmain
```

240 instructions were executed in 239 events (0 from other threads)

Function Name	Invocations	MinInst	MaxInst	AvgInst
SharedUserData!SystemCallStub	1	2	2	2
WTEsample!Bar	1	7	7	7
WTEsample!Baz	1	7	7	7
WTEsample!Do	1	7	7	7
WTEsample!Do2	1	13	13	13
WTEsample!Fa	1	7	7	7
WTEsample!Foo	1	7	7	7
WTEsample!La	1	7	7	7
WTEsample!Mi	1	7	7	7
WTEsample!Re	1	7	7	7
WTEsample!So	1	7	7	7
WTEsample!Ti	1	7	7	7
WTEsample!_RTC_CheckEsp	13	2	2	2
WTEsample!wmain	1	8	8	8
kernel32!BaseFormatTimeOut	1	15	15	15
kernel32!Sleep	1	4	4	4
kernel32!SleepEx	2	4	37	20
kernel32!_SEH_epilog	1	9	9	9
kernel32!_SEH_prolog	1	18	18	18
ntdll!				
RtlActivateActivationContextUnsafeFast	1	16	16	16
ntdll!				
RtlDeactivateActivationContextUnsafeFast	1	14	14	14
ntdll!ZwDelayExecution	2	1	3	2

1 system call was executed

```
Calls  System Call
      1  ntdll!ZwDelayExecution
```

Начальная часть вывода (отображение иерархического дерева) — это информация о вызовах. Перед каждым вызовом WinDBG отображает три числа: первое — количество ассемблерных команд, исполняемых функцией до вызова следующей функции, второе не документировано, но похоже на полное число исполненных ассемблерных команд при трассировке до возврата, последнее число в скобках — это текущий уровень вложенности иерархического дерева.

Вторая часть вывода — отображение итогов — немного менее понятна. В дополнение к подведению итогов вызовов каждой функции она отображает счетчик вызовов каждой функции, а также минимальное количество ассемблерных команд, вызываемых при выполнении функции, максимальное количество команд, вызываемых при выполнении функции, и среднее количество вызванных команд. Последние строки итогов показывают количество системных вызовов. Вы можете увидеть, что WTEExample иногда вызывает `Sleep` для обращения к режиму ядра. Сам факт, что вы располагаете количеством обращений к ядру, потрясюще крут.

Как вы можете себе представить, `WT` может дать огромный вывод и замедлить ваше приложение, так как каждая строка вывода требует парочки межпроцессных передач информации между отладчиком и отлаживаемой программой. Если вы хотите увидеть крайне важную итоговую информацию, то параметр `-nc` команды `WT` подавит вывод иерархии. Конечно, если вы интересуетесь только иерархией, укажите параметр `-ns`. Чтобы увидеть содержимое регистра возвращаемого значения (EAX в языке ассемблера x86), задайте `-or`, а чтобы увидеть адрес, исходный файл и номер строки (если это доступно) для каждого вызова — `-oa`. Последний параметр — `-l` — позволяет установить максимальную глубину вложенности отображаемых вызовов. Параметр `-l` полезен, если вы хотите увидеть только главные моменты того, что исполняется, или сохранить в выводе только функции вашей программы.

Я настоятельно советую вам посмотреть ключевые циклы и операции в своих программах с помощью `WT`, чтобы точно знать, что происходит за кулисами. Не знаю, сколько проблем производительности, неправильного использования языков и технологий я выследил с ее помощью!

Общий вопрос отладки

Некоторые имена в моих программах на C++ огромны. Как использовать WinDBG, чтобы не заработать туннельного синдрома?

К счастью, WinDBG теперь поддерживает текстовые псевдонимы (aliases). Определить пользовательский псевдоним и эквивалент расширения позволяет команда `AS` (Set Alias — установить псевдоним). Например, команда `as LL kernel32!LoadLibraryW` назначит строку «LL» для расширения ее до `kernel32!LoadLibraryW` везде, где вы ее вводите в командной строке. Увидеть назначенные вами псевдонимы позволяет команда `AL` (List Aliases — список псевдонимов), а удалить — `AD` (Delete Alias — удалить псевдоним).

Есть еще одно место, указанное в документации, где вы можете определить псевдонимы с фиксированными достаточно странными именами, — это команда `R` (Registers — регистры). Псевдонимы с фиксированными именами — `$u0`, `$u1`, ..., `$u9`. Чтобы определить псевдоним с фиксированным

именем, надо ввести точку перед `u: r $u0=kernel32!LoadLibraryA`. Увидеть, какие псевдонимы назначены фиксированным именам, позволяет лишь команда `.ЕCHO` (Echo Comment — вывести комментарий): `.echo $u0`.

Точки прерывания

WinDBG предлагает те же виды точек прерывания, что и Visual Studio .NET, плюс несколько уникальных. Важно, что WinDBG дает гораздо больше возможностей в момент срабатывания точек прерывания и позволяет увидеть, что происходит после этого. Прежние версии WinDBG имели хорошее диалоговое окно, где очень просто было устанавливать точки прерывания. Увы, это диалоговое окно отсутствует в переписанной версии WinDBG, которой мы располагаем сейчас, поэтому при установке точки прерывания мы все должны делать вручную.

Общие точки прерывания

Первое, за что я хочу взяться в точках прерывания, — это две команды, устанавливающие точки прерывания: `BP` и `BU`. Обе имеют одинаковые параметры и модификаторы. Можно считать, что версия команды `BP` — это строгая точка прерывания, всегда ассоциируемая в WinDBG с адресом. Если модуль, содержащий такую точку, выгружен, WinDBG исключает точку `BP` из списка точек прерывания. С другой стороны, точки прерывания `BU` ассоциированы с символом, поэтому WinDBG отслеживает символ, а не адрес. Если символ перемещается, точка `BU` также перемещается. А значит, точка `BU` будет активна, но заблокирована, если модуль выгружается из процесса, но будет немедленно реактивирована, как только модуль вернется в процесс, даже если ОС переместит модуль. Основная разница между точками прерывания `BP` и `BU` в том, что WinDBG сохраняет точки `BU` в рабочих пространствах WinDBG, а `BP` — нет. Наконец, при установке точки прерывания в окне Source путем нажатия F9 WinDBG устанавливает точку `BP`. Я рекомендую использовать точки прерывания `BU` вместо `BP`.

Имеется ограниченное диалоговое окно Breakpoints (точки прерывания) — щелкните меню Edit, затем Breakpoints, — но я управляю точками прерывания из окна Commands, так как, по-моему, это проще. Команда `BL` (Breakpoint List — список точек прерывания) позволяет увидеть все активные сейчас точки прерывания. Вы можете прочитать документацию к выводу команды `BL`, но я хочу заметить, что первое поле — это номер точки прерывания WinDBG, а второе — буква, обозначающая статус точки прерывания: `d` (disabled — запрещена), `e` (enabled — разрешена) и `u` (unresolved — неразрешима). Вы можете разрешить или заблокировать точки прерывания командами `BE` (Breakpoint Enable — разрешить точку прерывания) и `BD` (Breakpoint Disable — заблокировать точку прерывания). Указание звездочки (*) в любой из этих команд будет разрешать/блокировать все точки прерывания. Наконец, вы можете разрешить/заблокировать конкретные точки прерывания, указанием номера точки прерывания в командах `BE` и `BD`.

Синтаксис команды для установки точки прерывания пользовательского режима x86 таков:

```
[~Thread] bu[ID] [Address [Passes]] ["CommandString"]
```

Если вы просто вводите `BU`, WinDBG устанавливает точку прерывания на месте текущего указателя команд. Модификатор потока (`~Thread`) — это просто номер потока WinDBG, который делает установки точки прерывания в конкретном потоке тривиальной. Если вы пожелаете указать номер точки прерывания WinDBG, укажите его сразу после `BU`. Если точка прерывания с таким номером уже существует, то WinDBG заменит имеющуюся точку прерывания новой, которую вы устанавливаете сейчас. Поле адреса (Address) может содержать любое допустимое адресное выражение, которые я описывал в начале раздела «Ситуации при отладке». В поле проходов (Passes) вы указываете, сколько раз вы хотели бы пропустить эту точку останова до того, как произойдет останов. Сравнение в этом поле действует по правилу «больше или равно», максимальное значение — 4 294 967 295. Так же как при отладке неуправляемого кода в Visual Studio .NET, поле Passes уменьшается только в случае исполнения программы «на полной скорости», а не при проходе по шагам или трассировке.

Последнее поле, которое можно использовать при установке точки прерывания, — это чудесная командная строка (CommandString). Это правда, что вы можете ассоциировать команды с точкой прерывания! Наверное, лучший способ продемонстрировать эту удивительную штуку — рассказать, как я применил эту технологию для устранения почти неразрешимой ошибки. Одна ошибка проявлялась только после нескольких длинных последовательностей данных, прошедших через определенный участок кода. Как обычно, это занимало уйму времени, чтобы выполнились все условия, а я не мог просто тратить день или неделю на просмотр состояний переменных при каждой остановке программы на точке прерывания (увы, я работал не на условиях почасовой платы!). Мне нужен был способ регистрировать все значения переменных, чтобы изучить поток данных в системе. Так как можно объединять массу команд с помощью точки с запятой, я постепенно построил огромную команду, которая выводила все переменные путем вызова `DT` и `??`. Я также разбросал несколько команд `.ESNO`, чтобы видеть, где я был, и иметь общую строку, которая появлялась бы каждый раз, когда срабатывала точка прерывания. Командную строку я завершил командой `«;G»`, чтобы исполнение программы продолжалось после точки прерывания после полного дампа значений переменных. Я, конечно же, включил регистрацию и просто запустил процесс на исполнение, пока он не завершился аварийно. Просмотрев весь файл регистрации, я сразу увидел образчик данных и быстренько исправил ошибку. Не будь в WinDBG такой прекрасной возможности расширения точек прерывания, я бы никогда не нашел эту ошибку.

Команда `J` (Execute If — Else — выполнить если — то) особенно хороша для применения в командной строке точки прерывания. Она позволяет исполнять команды по условию, основанному на частном выражении. Иначе говоря, `J` предоставляет возможность использовать условные точки прерывания в WinDBG. Формат команды:

```
j expression 'if true command' ; 'if false command'
```

Выражение (expression) — это любое выражение, с которым может справиться вычислитель WinDBG. Текст в одиночных кавычках — это командные строки для истинного (true) и ложного (false) значения выражения. Всегда заключайте командные строки в одиночные кавычки, так как вы получаете возможность вклю-

чать в командные строки точки с запятой для построения больших выражений. И, конечно же, вы можете включать подкоманды J в командные строки для true и false. Из документации не совсем ясно, что делать, если нужно оставить одно из условий (true или false) пустым, т. е. если вы не хотите исполнять никакие команды для этого условия, просто введите два символа одиночной кавычки рядом для пропущенного условия.

Точки прерывания по обращению к памяти

Помимо блестящих точек прерывания исполнения, WinDBG имеет феноменальную команду BA (Break On Access — прервать в случае обращения), позволяющую остановиться, если фрагмент памяти считывается или записывается вашим процессом. Visual Studio .NET предлагает только точки прерывания по изменению состояния памяти, а вам нужно пользоваться классом аппаратных точек прерывания Майка Морепарти (Mike Morearty) для доступа ко всей мощи, предлагаемой аппаратными точками прерывания Intel x86. Однако WinDBG сам располагает всей этой мощью.

Формат точек прерывания по обращению к памяти для пользовательского режима Intel x86 таков:

```
[~Thread] ba[ID] Access Size [Address [Passes]] ["CommandString"]
```

Как видите, BA предлагает гораздо больше возможностей, чем просто остановка при обращении к памяти. Так же, как и в случае команд BP и BU, вы можете приказать останавливаться, если только указанный поток «прикасается» к памяти, установить счетчик проходов и ассоциировать эту удивительную командную строку с определенным типом обращения. Поле типа обращения (Access) — это одиночный символ указывающий, хотите ли вы остановиться в случае чтения (r), записи (w) или исполнения (e). Поле размера (Size) указывает, сколько байт вы собираетесь поставить под надзор. Так как BA использует Intel Debug Registers (регистры отладки Intel) для реализации своего волшебства, вы ограничены только возможностью надзора за 1, 2 или 4 байтами в каждый момент, вы также ограничены четырьмя точками прерывания BA. Как и при установке точек прерывания по данным в Visual Studio .NET, надо помнить о проблемах выравнивания памяти, поэтому, если вы хотите надзирать за 4 байтами памяти, адрес этой памяти должен заканчиваться на 0, 4, 8 или C. Поле адреса (Address) — это адрес памяти, обращение к которой должно вызвать прерывание. Хотя WinDBG менее требователен к переменным, я все же предпочитаю использовать реальные шестнадцатеричные адреса, чтобы быть уверенным, что точка прерывания установлена именно там, где я хочу.

Чтобы увидеть BA в действии, можете воспользоваться программой MemTouch из числа файлов-примеров к этой книге. Программа просто размещает локальный фрагмент памяти szMem, передаваемый одной из функций, которая «прикасается» к памяти, а другая функция просто читает эту память. Установите точку прерывания на адрес szMem, чтобы указать место прерывания. Чтобы получить адрес локальной переменной, дайте команду DV. Получив этот адрес, вы можете указать это значение в BA. Чтобы узнать, что делать с командами, возможно, стоит вызвать «kr;g», в результате чего вы увидите время доступа, а затем продолжите исполнение.

Исключения и события

WinDBG предлагает продвинутые средства управления исключениями и событиями. Исключения — это все аппаратные исключительные ситуации, такие как нарушение доступа, приводящее к аварийному завершению программы. События — это все стандартные события, передаваемые отладчикам средствами Microsoft Win32 Debugging API. Это значит, например, что вы можете установить прерывание WinDBG, когда загрузился модуль, и, таким образом, получить управление еще до того, как будет исполнена точка входа модуля.

Для управления исключениями и событиями из окна Command служат команды `SXE`, `SXI`, `SXN` (Set Exceptions — установить исключения), но они сильно сбивают с толку. К счастью, в WinDBG есть диалоговое окно Event Filters (фильтры событий) (рис. 8-5), доступное из меню Debug.

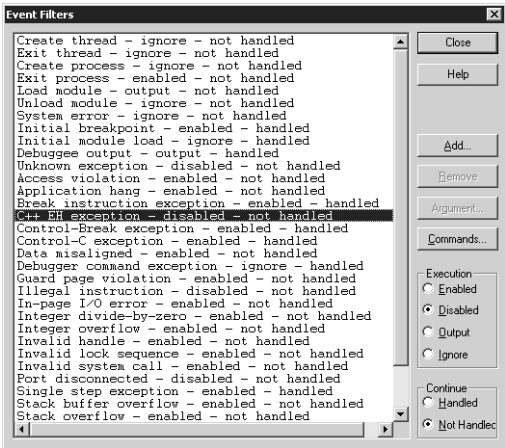


Рис. 8-5. Диалоговое окно Event Filters

Но даже оно все еще чуточку путает при попытке понять, что происходит при исключении, так как WinDBG использует странноватую терминологию в командах `SX*` и диалоговом окне Event Filters. Групповое поле Execution (исполнение) в нижнем правом углу указывает, как WinDBG будет управлять исключением (табл. 8-2). Поскольку поле Exceptions указывает, что вы хотите передавать функции API `ContinueDebugEvent`, напомним, что мы обсуждали ее в главе 4.

Табл. 8-2. Состояния прерываний по исключению

Состояние	Описание
Enabled (разрешено)	При возникновении исключения (исключительной ситуации) оно исполняется и происходит прерывание в отладчик.
Disabled (блокирована)	При первом возникновении исключения отладчик игнорирует его, при повторном исполнении останавливается, и осуществляется выход в отладчик.
Output (вывод сообщения)	При возникновении исключения прерывание в отладчик не производится. Однако выводится информационное сообщение об этом исключении.

Табл. 8-2. Состояния прерываний по исключению (продолжение)

Состояние	Описание
Ignore (игнорируется)	При возникновении исключения отладчик его игнорирует. Сообщение не отображается.

Вы можете игнорировать группу элементов управления Continue в нижнем правом углу. Она важна, только если вы хотите производить различную обработку точки прерывания, одиночного шага и недопустимых исключений. Если вы добавляете к списку собственную структурную обработку исключений, сохраните параметры группы Continue по умолчанию, Not Handled — без обработки. В результате каждый раз при возникновении исключения WinDBG будет корректно передавать его прямо отлаживаемой программе. Вы же не хотите, чтобы отладчик съедал исключения кроме тех, которые он сам вызывает, таких как точка прерывания и одиночный шаг.

После выбора собственно исключения самой важной кнопкой в этом диалоговом окне является Commands (команды). Только имя может подсказать вам, что она делает. Щелчок этой кнопки выводит окно Filter Command (команда фильтра) (рис. 8-6). Первое поле ввода названо неправильно — оно должно называться First-Chance Exception.

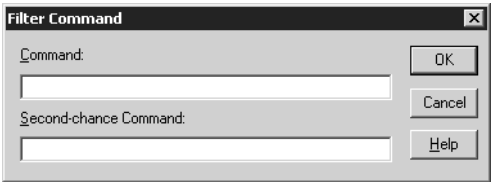


Рис. 8-6. Диалоговое окно Filter Command

В окне Filter Command можно вводить команды WinDBG, исполняемые при возникновении в отлаживаемой программе конкретного исключения. Когда мы в разделе «Контроль исключений» главы 7 обсуждали диалоговое окно Exception в Visual Studio .NET, я показал, как устанавливать исключения C++, чтобы остановиться на первом исключении, чтобы было можно контролировать, где ваши программы вызывают throw, а после нажатия F10 — и catch. Проблема в том, что Visual Studio .NET останавливается всякий раз, когда вырабатывается исключительная ситуация C++, а вам приходится сидеть и каждый раз на этом месте нажимать F5, в то время как ваше приложение обрабатывает множество команд throw.

Что хорошо в WinDBG и в возможности ассоциировать команды с исключениями, так это то, что вы можете применять эти команды для регистрации всей важной информации и эффективно продолжать исполнение без вашего вмешательства в ход исполнения. Чтобы настроить обработку исключений C++, выберите C++ EH Exception из списка исключений в диалоговом окне Event Filter и щелкните кнопку Commands. В диалоговом окне Filter Command (команда фильтра) введите в поле ввода Command `kp; g`, чтобы WinDBG зарегистрировал состояние стека и продолжил выполнение. Теперь у вас будет состояние стека вызовов при каждом выполненном throw, а WinDBG продолжит корректное исполнение. И все же, чтобы увидеть последнее событие или исключение, происшедшее в процессе, дайте команду `.LASTEVENT` (Display Last Event — отобрази последнее событие).

Управление WinDBG

Теперь, когда вы познакомились с важными командами отладки, я хочу обратиться к нескольким мета-командам, которые я использую ежедневно в процессе отладки с помощью WinDBG.

Простейшая, но чрезвычайно полезная команда — `.CLS` (Clear Screen — очистить экран) — позволяет очистить окно Command, чтобы начать вывод сначала. Так как WinDBG способен изрыгать огромные объемы информации, требующей место для хранения, время от времени полезно очищать «рабочее поле».

Если ваше приложение работает со строками Unicode, вам захочется настроить отображение указателей `USHORT` как строк Unicode. Команда `.ENABLE_UNICODE` (Enable Unicode Display — разрешить отображение Unicode), введенная с параметром 1, настроит все так, чтобы команда `DT` корректно отображала ваши строки. Если нужно настроить национальные параметры для корректного отображения строк формата Unicode, то команда `.LOCALE` (Set Locale — установить местный диалект) в качестве параметра принимает локализирующий идентификатор. Если приходится работать с битами и вы хотите видеть значения битов, команда `.FORMATS` (Show Number Formats — отобразить форматы чисел) покажет значения передаваемых параметров всех числовых форматов, в том числе двоичных.

А вот команда `.SHELL` (Command Shell) позволяет запустить программу MS-DOS из отладчика и перенаправить ее вывод в окно Command. Отлаживая на той же машине, на которой выполняется отлаживаемая программа, конечно же, проще переключиться с помощью `Alt+Tab`, но красота `.SHELL` в том, что при выполнении удаленной отладки, программа MS-DOS выполняется на удаленной машине. `.SHELL` можно использовать также для запуска единственной внешней программы, с перенаправлением ее вывода в окно Command. После выдачи `.SHELL` окно Command в строке ввода будет отображать `INPUT>` для обозначения, что программа MS-DOS ожидает ввода. Для завершения программы MS-DOS и возврата к окну Command, используйте либо команду `MS-DOS exit`, либо, что предпочтительнее, `.SHELL_QUIT` (Quit Command Prompt), так как она прекратит исполнение программы MS-DOS, даже если она заморожена.

Последнюю мета-команду, о которой я упомяну, я искал в отладчике много лет, но обнаружил лишь теперь. При написании обработчиков ошибок вы обычно знаете, что к моменту обработки ошибок в вашем процессе возникли серьезные неприятности. Вам также известно в 9 случаях из 10, что если происходит обработка какой-то ошибки, то, вероятно, вам нужно посмотреть значения каких-то переменных или состояние стека вызовов, а кроме того, вы захотите записать конкретную информацию. Я всегда хотел иметь способ закодировать то, что нужно выполнить прямо в моем процессе обработки ошибки. Сделав это, команды будут исполняться, позволяя программистам службы сопровождения и мне отлаживать быстрее. Моя идея была такова: так как вызовы `OutputDebugString` проходят через отладчик, можно было бы встроить команды в `OutputDebugString`. Вы могли бы сказать отладчику, что искать в начале текста `OutputDebugString`, а что все остальное после этого будут команды, подлежащие исполнению.

Именно так работает команда WinDBG `.OCCOMMAND` (Expect Commands from Target — ожидать команды от отлаживаемой программы). Вы вызываете `.OCCOMMAND`, указывая искомый префикс строки в начале вызовов `OutputDebugString`. Если этот пре-

фикс присутствует, WinDBG исполнит остальную часть текста как командную строку. Очевидно, что вам нужно быть осторожным при использовании строк, а то WinDBG сойдет с ума, пытаясь исполнить вызовы `OutputDebugString` во всей вашей программе. В качестве такой строки мне нравится `WINDBGCMD`: — я разбрасываю командные строки WinDBG во всех своих программах.

При использовании `.oscommand` необходимо в конце каждой команды добавлять «;g», иначе WinDBG остановится по завершении команды. В следующей функции все команды завершаются «;g», чтобы выполнение продолжалось. Чтобы они начали работать, я выдаю команду `.oscommand WINDBGCMD`: при запуске программы:

```
void Baz ( int )
{
    // Чтобы это воспринималось как команды WinDBG, выполните команду
    // ".oscommand WINDBGCMD:" внутри WinDBG.
    OutputDebugString ( _T ( "WINDBGCMD: .echo \"Hello from WinDBG\";g" ) );
    OutputDebugString ( _T ( "WINDBGCMD: kp;g" ) );
    OutputDebugString ( _T ( "WINDBGCMD: .echo \"Stack walk is done\";g" ) );
}
```

Магические расширения

Теперь вы знаете достаточно команд (представляющих лишь малую толику возможных), чтобы у вас голова пошла кругом, и вы, возможно, удивляетесь, почему я трачу так много времени на обсуждение WinDBG. WinDBG труднее в использовании, чем Visual Studio .NET, и кривая обучения не только крута — она почти вертикальна! Вы увидели, что WinDBG предлагает классные возможности по точкам прерывания, но вы все еще, возможно, удивляетесь, почему игра стоит свеч.

WinDBG — достойная вещь благодаря командам расширения. Эти команды позволяют увидеть то, что по-другому увидеть невозможно. Microsoft предложила целый букет замечательных расширений, которые мастера отладки используют для разрешения самых неприятных проблем.

Я хочу сосредоточиться на наиболее важных. Найдите время на чтение документации об остальных расширениях. В разделе `Reference\Debugger Extension Commands` документации к WinDBG имеются два ключевых раздела: `General Extensions` (общие расширения) и `User-Mode Extensions` (расширения пользовательского режима).

Физически расширения являются файлами DLL, экспортирующими особые имена функций для своей работы. В каталоге `Debugging Tools For Windows` есть несколько каталогов, таких как `W2KFRE` (Windows 2000 Free Build — свободная поставка для Windows 2000) и `WINXP`. Эти каталоги содержат команды расширения для разных ОС. Как писать свои расширения, вы можете прочитать в файле `README.TXT`, прилагаемом к примеру `EXTS` в каталоге `<Debugging Tools for Windows Dir>\SDK\SAMPLES\EXTS`.

Загрузка расширений и управление ими

Прежде чем рассмотреть команды расширения, надо поговорить о том, как увидеть, какие расширения вы уже загрузили, как загрузить ваше собственное и как

получить справку из расширения. Загруженные расширения покажет команда `.CHAIN` (List Debugger Extensions — список расширений отладчика). Она выведет и порядок поиска команд сверху вниз на дисплее, и то, как WinDBG ищет библиотеки DLL расширений. Под Windows 2000 отображение для четырех библиотек расширений пользовательского режима (DBGHELP.DLL, EXT.DLL, UEXT.DLL и NTSDEXTS.DLL) выглядит так (зависит от расположения каталога Debugging Tools for Windows):

```
0:000> .chain
```

```
Extension DLL search Path:
```

```
G:\windbg\winext;G:\windbg\pri;G:\windbg\WINXP;G:\windbg;
```

```
Extension DLL chain:
```

```
dbghelp: image 6.1.0017.1, API 5.2.6, built Sat Dec 14 15:32:30 2002
```

```
[path: G:\windbg\dbghelp.dll]
```

```
ext: image 6.1.0017.0, API 1.0.0, built Fri Dec 13 01:46:07 2002
```

```
[path: G:\windbg\winext\ext.dll]
```

```
exts: image 6.1.0017.0, API 1.0.0, built Fri Dec 13 01:46:07 2002
```

```
[path: G:\windbg\WINXP\exts.dll]
```

```
uext: image 6.1.0017.0, API 1.0.0, built Fri Dec 13 01:46:08 2002
```

```
[path: G:\windbg\winext\uext.dll]
```

```
ntsdexts: image 5.2.3692.0, API 1.0.0, built Tue Nov 12 14:16:20 2002
```

```
[path: G:\windbg\WINXP\ntsdexts.dll]
```

Загрузка расширений проста — укажите имя библиотеки DLL (без расширения .DLL) как параметр команды `.LOAD` (Load Extension DLL — загрузить DLL расширения). Для выгрузки укажите имя библиотеки DLL в качестве параметра команде `.UNLOAD` (Unload Extension DLL — выгрузить DLL расширения).

Принято, что все команды расширения вводятся в нижнем регистре и в отличие от обычных и мета-команд они чувствительны к регистру. Кроме того, команды в библиотеке расширения называются так же: например, команда `help` предназначена для того, чтобы быстро информировать, что имеется в этой библиотеке DLL расширения. При загруженных расширениях по умолчанию ввод команды `!help` не показывает всю доступную справку. Чтобы вызвать команду расширения конкретной библиотеки DLL расширения, добавьте имя DLL и точку для команды расширения: `!dllname.command`. Следовательно, чтобы увидеть справку о NTSD-EXTS.DLL, нужно ввести `!ntsdexts.help`.

Важные команды расширения

Теперь, когда вы вооружены некоторыми основами работы с расширениями, я хочу обратиться к командам расширения, которые облегчат вашу жизнь. Все эти расширения являются частью набора расширений по умолчанию, загружаемого всегда, поэтому, пока вы специально не выгрузите что-то из этих расширений, они будут всегда доступны.

Первая важная команда — `!analyze -v` — позволяет быстро проанализировать текущее исключение. Я специально показал эту команду с параметром `-v`, потому что без него вы не увидите большую часть информации. Команда `!analyze` не разрешает все ваши ошибки — ее идея заключается в том, что она предоставляет вам ту информацию, которую вы обычно хотите видеть во время аварийного завершения, такую как запись исключения и стек вызовов.

Так как критические секции являются облегченными объектами синхронизации, многие программисты пользуются ими. WinDBG предлагает две команды расширения для заглядывания внутрь критической секции, чтобы узнать состояние блокировок объектов и какие потоки владеют ими. Если у вас есть адрес критической секции, можно применить команду `!critsec`, передавая ей как параметр адрес секции. А увидеть все заблокированные критические секции позволяет `!locks`. Все критические секции процесса она покажет с параметром `-v`. В Windows XP/Server 2003 дополнительный параметр `-o` покажет сиротские критические секции.

Если вы программируете защищенные приложения Win32, очень трудно понять, какая текущая информация безопасности применена к текущему потоку. Команда `!token` (Windows XP/Server 2003) или `!threadtoken` (Windows 2000) покажет состояние заимствования прав текущего потока и остальную информацию безопасности, такую как идентификация пользователя и групп, плюс отобразит в текстовом виде все привилегии, ассоциированные с потоком.

Есть одна команда, которая сохранила мне бесчисленное количество часов отладки, — `!handle`. Как можно понять из названия, она делает что-то с описателями в процессе. Если просто ввести `!handle`, вы увидите значения описателей, тип объекта, содержащегося в описателе, и секцию, подводящую итоги, сколько объектов каждого типа имеется в процессе. Некоторые из этих типов могут показаться вам бессмысленными, если вы не программировали драйверы или не читали книгу Дэвида Соломона и Марка Руссиновича «Inside Microsoft Windows 2000»¹. Табл. 8-3 предлагает переводы (трансляцию) с языка команды `!handle` на язык терминов пользовательского режима некоторых типов.

Табл. 8-3. Трансляция (перевод) типов описателей

Термин команды	Термин пользовательского режима
<code>!handle</code>	
Desktop	Win32 desktop (рабочий стол Win32)
Directory	Win32 object manager namespace directory (каталог рабочего пространства менеджера объектов Win32)
Event	Win32 event synchronization object (объект синхронизации события Win32)
File	Disk file, communication endpoint, or device driver interface (диск-овый файл, конечная точка коммуникационной связи или интерфейс драйвера устройства)
IoCompletionPort	Win32 IO completion port (порт завершения ввода/вывода Win32)
Job	Win32 job object (объект задания Win32)
Key	Registry key (раздел реестра)
KeyedEvent	Non-user-creatable events used to avoid critical section out of memory conditions (созданные не пользователем события, используемые предотвращения выхода критической секции за пределы памяти)
Mutant	Win32 mutex synchronization object (объект синхронизации мьютекс Win32)

см. след. стр.

¹ Соломон Д., Руссинович М. Внутреннее устройство Microsoft Windows 2000. — М.: «Русская Редакция», 2001. — *Прим. перев.*

Табл. 8-3. Трансляция (перевод) типов описателей (продолжение)

Термин команды	Термин пользовательского режима
!handle	
Port	Interprocess communication endpoint (конечная точка межпроцессного взаимодействия)
Process	Win32 process (процесс Win32)
Thread	Win32 thread (поток Win32)
Token	Win32 security context (контекст защиты Win32)
Section	Memory-mapped file or page-file backed memory region (отображаемый на память файл или страничный файл выгрузки региона памяти)
Semaphore	Win32 semaphore synchronization object (объект синхронизации семафор Win32)
SymbolicLink	NTFS symbolic link (символьная связь NTFS)
Timer	Win32 timer object (объект таймер Win32)
WaitablePort	Interprocess communication endpoint (конечная точка межпроцессного взаимодействия)
WindowStation	Top level of window security object (объект защиты окна верхнего уровня)

Даже просмотр описателей замечателен, но, указав параметр `-?` в `!handle`, вы увидите, что команда способна на большее. Чтобы появилось больше информации об описателе, можно задать в первом параметре значение описателя, а во втором — битовое поле, указывающее, что вы хотите узнать об этом описателе. В качестве второго параметра вы всегда должны задавать `F`, так как в результате вам будет показано все. Например, я отлаживаю программу `WDBG` из главы 4, описатель `0x1CC` является событием. Вот как получить детальную информацию об этом описателе:

```
0:006> !handle 1cc f
Handle 1cc
Type          Event
Attributes    0
GrantedAccess  0x1f0003:
               Delete,ReadControl,WriteDac,WriteOwner,Synch
               QueryState,ModifyState
HandleCount   3
PointerCount  6
Name          \BaseNamedObjects\WDBG_Happy_Synch_Event_614
Object Specific Information
  Event Type  Manual Reset
  Event is    Waiting
```

Вы видите не только предоставленные права, но также имя и, что важнее, что событие находится в состоянии ожидания (т. е. в занятом состоянии). Так как `!handle` покажет эту информацию для всех типов, теперь вы легко увидите взаимные блокировки, поскольку вы можете проверить состояния всех событий, семафоров и мьютексов, чтобы понять, кто из них заблокирован, а кто нет.

Вы можете посмотреть подробную информацию для всех описателей процесса, передавая два параметра 0 и F. Если вы работаете над большим процессом, вывод может занять кучу времени на перемалывание всех деталей. Чтобы узнать о конкретном классе описателей, укажите два первых параметра 0 и F, а третий — имя класса. Например, чтобы увидеть все события, введите `!handle 0 f Event`.

Выше я касался применения `!handle` для просмотра состояний событий, чтобы сделать вывод, почему ваше приложение взаимоблокируется. Другое замечательное использование `!handle` — оценка потенциальной утечки ресурсов. Так как `!handle` показывает общее количество всех текущих описателей процесса, вы можете легко сравнить результаты `!handle` до и после. Если вы видите, что общее количество описателей изменилось, вы точно скажете, утечка какого типа описателей происходит. Так как отображается детальная информация, такая как разделы реестра и имя описателя, вы легко видите, утечка какого из описателей происходит.

Я выследил массу утечек ресурсов и взаимных блокировок с помощью `!handle` — это единственный способ получить информацию об описателях в процессе отладки, так что стоит потратить немного времени на ознакомление с ней и выводимыми с ее помощью данными.

Общий вопрос отладки

Функции Win32 API, такие как `CreateEvent`, создающие описатели, имеют необязательный параметр «имя». Должен ли я назначать имена моим описателям?

Абсолютно, безусловно, ДА! Команда `!handle` может показать состояния каждого из ваших описателей. Но это лишь малая часть того, что необходимо для поиска проблем. Если описатели не именованы, очень трудно сопоставить сами описатели с происходящим в отладчике, скажем, при взаимных блокировках. Не дав имена своим описателям, вы делаете свою жизнь заметно сложнее, чем она должна быть.

Однако вы можете просто пойти и начать давать сногсшибательные имена в этом необязательном поле. Когда вы создаете событие, например, имя, даваемое этому событию, такое как `«MyFooEventName»`, глобально для всех процессов, выполняемых на машине. Хотя можно подумать, что второй процесс, вызывающий `CreateEvent`, дает ему уникальное имя внутренними средствами, на самом деле `CreateEvent` вызывает `OpenEvent` и возвращает вам описатель глобально именованного события. Теперь допустим, что у вас два исполняющихся процесса и в каждом из них есть поток, ожидающий события `MyFooEventName`. Когда один из процессов сигнализирует о событии, этот сигнал будут видеть оба процесса и начнут исполняться. Очевидно, что если вы подразумеваете, что сигнал воспринимается только одним процессом, то вы просто создаете сверхтрудную для отлова ошибку.

Чтобы давать правильные имена описателям, вы должны быть уверены, что генерируете уникальные имена для всех описателей, сигналы которых должны восприниматься единственным процессом. Взгляните, что я делал в WinDBG в главе 4: я добавлял идентификатор процесса или потока к имени для обеспечения уникальности.

Другие интересные команды расширения

Прежде, чем перейти к управлению файлами вывода, хочу отметить несколько команд расширения, которые вы найдете интересными в критических ситуациях, например, когда нужно найти какую-то действительно вызывающую ошибку. Первая — `!imgreloc` — просто просматривает все загруженные модули и сообщает, были ли все модули загружены в предпочитаемые вами адреса. Теперь у вас нет оправдания за то, что вы не проверили. Вывод команды выглядит так (ОС переместила второй модуль TP4UIRES):

```
0:003> !imgreloc
00400000 tp4serv - at preferred address
00c50000 tp4uires - RELOCATED from 00400000
5ad70000 uxtheme - at preferred address
6b800000 S3appd11 - at preferred address
76360000 WINSTA - at preferred address
76f50000 wtsapi32 - at preferred address
77c00000 VERSION - at preferred address
77c10000 msvcrt - at preferred address
77c70000 GDI32 - at preferred address
77cc0000 RPCRT4 - at preferred address
77d40000 USER32 - at preferred address
77dd0000 ADVAPI32 - at preferred address
77e60000 kernel32 - at preferred address
77f50000 ntdll - at preferred address
```

Если вы так ленивы, что не можете вызвать командную строку и вывести команду `NET SEND` для отправки сообщения другим пользователям, вы можете просто ввести `!net_send`. На самом деле это полезно, если вам нужно привлечь чье-то внимание в процессе удаленной отладки. Ввод просто `!net_send` покажет вам необходимые для отправки сообщения параметры.

Поскольку вы располагаете командой `!dreg` для вывода информации о регистрах, вы также располагаете командой `!evlog` для отображения журнала событий. Если каждую из них просто ввести в командной строке, вы получите подсказку, как их использовать. Обе — прекрасные помощники для просмотра регистров или журналов событий. Если вы используете их, особенно при удаленной отладке, сюрпризов не ждите.

Если у вас проблемы с обработкой исключений, команда `!exchain` поможет просмотреть цепочки обработки исключений текущего потока и увидеть, какие функции имеют зарегистрированные обработчики исключений. Вот образец вывода команды при отладке программы `ASSERTTEST.EXE`.

```
0012ffb0: AssertTest!except_handler3+0 (004027a0)
  CRT scope 0, filter: AssertTest!wWinMainCRTStartup+22c (00401e1c)
    func: AssertTest!wWinMainCRTStartup+24d (00401e3d)
0012ffe0: KERNEL32!_except_handler3+0 (77ed136c)
  CRT scope 0, filter: KERNEL32!BaseProcessStart+40 (77ea847f)
    func: KERNEL32!BaseProcessStart+51 (77ea8490)
```

Работу с кучами ОС (т. е. кучами, создаваемыми вызовами функции API `CreateHeap`) облегчит команда `!heap`. Вы можете думать, что вы не используете никакие кучи ОС, но код ОС, исполняющийся внутри вашего процесса, к ним обращается. Вы можете испортить память в одной из этих куч (см. главу 17), а `!heap` покажет ее.

Наконец, я хочу коснуться очень интересной и недокументированной команды `!for_each_frame` из расширения `EXT.DLL`. Как можно понять из ее имени², она исполняет командную строку, переданную в качестве параметра команды, для каждого кадра (фрейма) стека. Прекрасный вариант использования этой команды — `!for_each_frame dv`, в результате чего будут выведены локальные переменные каждого кадра стека.

Работа с файлами дампа

Понимая, какие типы команд может исполнять WinDBG, вы можете перейти к последнему набору команд — командам файлов дампа. Как я уже упоминал в разделе «Основы», сильная сторона WinDBG — управление файлами дампа. Прелесть WinDBG и файлов дампа заключается в том, что почти все информационные команды работают и с файлами дампа, причем почти так же, как и тогда, когда возникли проблемы.

Создание файлов дампа

Выполняя «живую» отладку, вы можете вызвать команду `.DUMP` (Create Dump File — создать файл вывода), чтобы создать файл дампа. Замечу, что при создании файла дампа нужно указывать расширение в имени файла. `.DUMP` производит запись именно в тот файл, какой вы ей указали (полное имя файла и путь к нему) без добавления отсутствующего расширения. Вы всегда должны использовать расширение `.DMP`.

Оставив проблему расширения в стороне, я хочу обсудить некоторые общие возможности, предлагаемые `.DUMP` до того, как перейти к типам файлов дампа. Первый ключ — `/u` — добавляет дату, время и PID (идентификатор процесса) к имени файла, чтобы обеспечить уникальные имена файлов дампа без необходимости бороться с их именами. Так как файлы дампа являются столь замечательным средством выполнения снимков сеанса отладки, позволяющим анализировать поведение программы позже, `/u` заметно упрощает вашу жизнь. Чтобы обеспечить лучшее понимание, что происходило в конкретное время, ключ `/s` позволяет ввести комментарий, который будет отображаться, когда вы загрузите файл вывода. Наконец, если вы отлаживаете несколько процессов сразу, ключ `/a` запишет файлы дампа для всех загруженных процессов. Убедитесь, что вы используете `/u` совместно с `/a`, чтобы дать каждому процессу свое имя.

WinDBG может создавать два типа файлов дампа: полный и краткий. Полный включает все о процессе, от стеков текущих потоков до состояния всей памяти (даже все загруженные процессом двоичные данные). Он указывается с помощью ключа `/f`. Иметь полный файл дампа удобно, так как в нем содержится всего значительно больше, чем вам необходимо, однако он съедает огромный объем дисковой памяти.

² For each frame — для каждого кадра. — *Прим. перев.*

Для создания файла минидампа достаточно указать ключ по умолчанию `/m`, если вы не указываете никаких ключей в `.DUMP`. Записанный таким образом файл минидампа будет таким же, как и файл минидампа по умолчанию, создаваемый Visual Studio .NET, и будет содержать версии загруженных модулей, сведения о стеке для выполнения вызовов стека для всех активных потоков.

Вы также можете указать WinDBG добавить дополнительную информацию к минидампу, задавая флаги в ключе `/m`. Самый полезный — `h (/mh)` — в дополнение к информации по умолчанию для минидампа запишет информацию об активных описателях. Это значит, что вы сможете, используя команду `!handle`, просмотреть состояния всех описателей, созданных при записи дампа. Если понадобится анализировать проблемы с указателями, можно указать `i (/mi)`, чтобы WinDBG включил в файл вывода вторичную память. Этот ключ просматривает указатели на стек или страничную память и выводит состояние памяти, на которую ссылается указатель, в виде небольшого участка памяти около этого места. Таким образом, вы можете узнать, на что ссылаются указатели. Имеется множество других ключей краткого вывода, которые вы можете указать для записи дополнительной информации, но `h` и `i` я использую всегда.

Последний ключ, позволяющий сэкономить много дискового пространства, — `/b` — сожмет файл дампа в файл `.CAB`. Это замечательный ключ, но пропущенное расширение в файле дампа делает его использование проблематичным. Так как `.DUMP` не добавляет автоматически расширение, то вам инстинктивно захочется добавить расширение `.CAB` к файлу дампа. Однако при указании расширения `.CAB` WinDBG создает временный `.DMP`-файл с именем `<name>.CAB.DMP` внутри реального `.CAB`-файла. К счастью, WinDBG прекрасно прочтет такой файл из `.CAB`-файла.

Несмотря на все эти мелкие проблемы с возможностью записи `.CAB`-файлов, мне все же очень нравится использовать ее. В дополнение к сохранению только `.DMP`-файлов в `.CAB`-файлах можно указать ключ `/ba`, если вы хотите также сохранить и таблицу символов в `.CAB`-файле! Чтобы гарантированно сохранить все символы процесса, запустите команду `!d *` (load all symbols — загрузить все символы) перед созданием файла дампа. Таким образом, вы можете быть уверены, что вы располагаете всеми корректными символами, когда переносите `.CAB`-файл на машину, которая может не иметь доступа к вашему хранилищу символов. Используя `/b`, помните, что WinDBG записывает файл дампа и создает соответствующий `.CAB`-файл в каталоге `%TEMP%` машины. Вы, конечно, понимаете, что, имея большой процесс, создавая полный дамп с помощью `/f` и задавая `/ba` для создания `.CAB`-файла с символами, вам понадобится огромный шмат свободного пространства на диске в каталоге `%TEMP%`.

Открытие файлов дампа

Файлы дампа не принесут большой пользы, если вы не умеете открывать их. Проще всего сделать это из нового экземпляра WinDBG. В меню File выберите Open Crash Dump (открыть файл вывода аварийного завершения) или нажмите `Ctrl+D` для вызова диалогового окна Open Crash Dump, а затем найдите каталог, в котором находится файл дампа. Интересно, хотя это и не описано в документации, что WinDBG также откроет `.CAB`-файл, содержащий `.DMP`-файл. После того как файл

вывода будет открыт, WinDBG автоматически получает, что было записано, и вы можете начинать просматривать файл дампа.

Если вы создавали дамп на той же машине, где собирали процесс, ваша жизнь весьма проста, так как WinDBG проделает всю работу по получению символов и информации о номерах строк исходного кода. Однако большинство из нас будут анализировать файлы дампа, созданные на других машинах и других версиях ОС. После открытия файла дампа начинается работа по получению символов, настройке путей к исходному коду и исполняемым модулям.

Во-первых, определите, в каких модулях пропущена информация о символах, запустив команду `LM` с ключом `v`. Если какие-то модули сообщают «no symbols loaded» (символы не загружены), надо настроить путь для поиска символов. Посмотрите на информацию о версиях, ассоциированную с этими модулями, и соответственно обновите Symbol File Path (путь к файлу символов), выбрав Symbol File Path в меню File.

Второй шаг — настройка путей к файлам исполняемых образов. Как я уже говорил в главе 2, WinDBG нужен доступ к двоичным файлам до того, как он сможет загрузить символы для минидампов. Если вы следовали моим рекомендациям и поместили свои программы и необходимые различным ОС двоичные файлы и символы в свой сервер символов, подключить двоичные файлы легко. В диалоговом окне Executable Image Search Path (путь к исполняемому образу), доступном при выборе Image File Path (путь к файлу образа) меню File, можно просто вставить из буфера обмена ту же строку, что вы указали для символьного сервера. WinDBG автоматически найдет ваш сервер символов для соответствующих двоичных файлов.

Если двоичных файлов в хранилище символов нет, нужно указать путь вручную и надеяться, что вы указали его к корректным версиям модулей. Это особенно трудно с двоичными файлами ОС, так как очередное исправление может изменить любое их количество. В действительности, каждый раз, внося «горячие» изменения или устанавливая пакет обновлений, вы должны перезагрузить свое хранилище символов, запустив файл `OSSYMSJS` (см. главу 2).

И наконец, необходимо настроить путь к исходным текстам, выбрав Source File Path (путь к исходному файлу) из меню File. Настроив все три пути, перезагрузите символы командой `.RELOAD /f`, после которой последует команда `LM`, позволяющая увидеть все еще некорректные символы. Если минидамп доставлен от заказчика, вам, возможно, не удастся загрузить все двоичные файлы и символы, так как там может оказаться другой уровень внесенных исправлений или программ третьих поставщиков, напихавших кучу DLL в другие процессы. Однако ваша цель — загрузить все символы ваших программ и как можно больше символов ОС. Как-никак, если вам удалось загрузить все символы, отладка становится простым делом!

Отладка дампа

Если вам удалось корректно загрузить символы и двоичные файлы, то отладка файла дампа почти идентична отладке «живьем». Очевидно, что некоторые команды, такие как `BU`, не будут работать с файлами дампа, но большинство других будет, особен-

но команды расширения. При возникновении проблем с командами, обратитесь к таблице окружения в документации по этой команде и проверьте, что вы можете использовать ее при отладке файлов дампа.

Если вы имеете несколько файлов дампа сразу, вы также можете отлаживать их совместно командой `.OPENDUMP` (Open Dump File — открыть файл дампа). Открыв файл дампа таким способом, необходимо дать команду `G` (Go — запустить), чтобы WinDBG смог все запустить.

Наконец, команда, доступная только при отладке файла дампа, — `.DUMPCAB` (Create Dump File CAB — создать CAB файл дампа) — создаст .CAB-файл из текущего файла дампа. Если вы добавите параметр `-a`, все символы будут записаны в этот файл.

Son of Strike (SOS)

Имеется замечательная поддержка для отладки дампов приложений неуправляемого кода, но не для приложений управляемого кода, и, хотя управляемые приложения меньше подвержены появлению в них ошибок, отлаживать их гораздо труднее. Рассмотрим, например, те проекты, в которые были произведены значительные вложения в COM+ или другие технологии неуправляемого кода. Вы можете и хотите создавать новые внешние интерфейсы в .NET или компоненты, усиливающие ваши COM-компоненты путем использования COM interop. Когда эти приложения завершаются аварийно или зависают, вы тут же получаете головную боль, так как почти невозможно прорваться сквозь ассемблерный код, исследовать стеки вызовов и даже найти исходные тексты и строки для этих .NET-частей приложения.

Чтобы помочь вам увидеть .NET-части дампа или «живого» приложения, некоторые очень умные люди в Microsoft сделали расширение отладчика, названное SOS или Son of Strike («Дитя забастовки»). Основная документация находится в файле SOS.HTM в каталоге <Каталог установки Visual Studio .NET >\SDK\vl1.1\Tool Developers Guide\Samples\SOS. Там вы определенно увидите, что «основная» — это действительно работающий термин. В сущности это список команд расширения SOS.DLL и краткие сведения об их использовании.

Если вы работаете с большими системами .NET, особенно с тяжелыми транзакциями ASP.NET, вам также захочется загрузить 170-страничный PDF-файл «Production Debugging for .NET Framework Applications» (Отладка промышленных приложений для .NET Framework) с <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/DBGrm.asp>. Если вы хотите знать, как управлять зависшими процессами ASNET_WPEXE, работать с потенциальными проблемами управления памятью в .NET и контролировать другие экстремально-пограничные проблемы, это прекрасный документ. Его авторы определенно отладили массу жизненных систем промышленного уровня, и их знание поможет вам преодолеть многие трудности.

Вы кратко познакомитесь с командами SOS, основываясь на этих двух документах и документе свертоткровенных трюков, но вот как начать работать с SOS внутри WinDBG, там не сказано. В этом разделе я хочу помочь вам сделать первые шаги. Надеюсь, вы узнаете здесь достаточно, чтобы понимать документ «Production Debugging for .NET Framework Applications». Я не охвачу всего, например, всех

команд сборщика мусора, так как они рассмотрены в «Production Debugging for .NET Framework Applications».

Прежде всего я хочу показать простой способ загрузить SOS.DLL в WinDBG. SOS.DLL является частью собственно .NET Framework, поэтому фокус в том, чтобы включить нужные каталоги в ваш путь поиска (переменная окружения PATH), чтобы WinDBG справился с загрузкой SOS.DLL. Вам надо открыть командную строку MS-DOS и выполнить VSVAR32.BAT, находящийся в каталоге <Каталог установки Visual Studio .NET >\Common7\Tools. VSVAR32.BAT настраивает вашу среду так, что все соответствующие каталоги .NET будут включены в ваш путь поиска.

После однократного исполнения VSVAR32.BAT вы получаете возможность загрузить SOS.DLL командой `.load sos` из окна Command WinDBG. WinDBG всегда помещает последнее загруженное расширение на верхушку цепочки, поэтому команда `!help` покажет вам краткий список всех команд SOS.DLL.

Использование SOS

Возможно, лучше всего показать, как пользоваться SOS, на примере. Программа ExcerptApp из набора файлов к этой книге покажет вам, как подступиться к важным командам. Чтобы сохранить изложение на приемлемом уровне, я написал этот код, чтобы просто вызвать несколько методов с локальными переменными и в конце вызвать исключительную ситуацию. Я пройду через отладку примера EXCEPTAPP.EXE с помощью SOS, чтобы вы увидели, какие команды помогут узнать, где вы находитесь, когда приложение, использующее управляемый код, валится или зависает. Так вам будет проще применять SOS для решения проблем и понимать «Production Debugging for .NET Framework Applications»

Откомпилировав EXCEPTAPP.EXE и настроив переменные среды, как я описал выше, откройте EXCEPTAPP.EXE в WinDBG и остановитесь на точке прерывания загрузчика. Чтобы заставить WinDBG остановиться, когда приложение .NET вызовет исключительную ситуацию, надо сообщить WinDBG о номере исключения, сгенерированного .NET. Проще всего это можно сделать, щелкнув кнопку Add в диалоговом окне Event Filters и введя в диалоговом окне Exception Filter 0xE0434F4D. Затем выберите Enabled в группе Execution и Not Handled в группе Continue. Щелкнув OK, вы успешно настроите WinDBG так, чтобы он останавливался каждый раз, когда вырабатывается исключение .NET. Если значение 0xE0434F4D кажется чем-то знакомым, узнать, что это такое, поможет команда `.formats`.

Настроив исключения, запустите EXCEPTAPP.EXE, пока она не остановится на исключении .NET. WinDBG сообщит о нем, как о первом исключении, и остановит приложение на реальном вызове Win32 API `RaiseException`. Загрузив SOS командой `.load sos`, выполните `!threads` (ее вы всегда будете хотеть исполнять первой в SOS) и вы увидите, какие потоки в приложении или дампе имеют код .NET. В случае EXCEPTAPP.EXE команда потоков WinDBG ~ указывает, что в приложении исполняются три команды. Однако команда всеобщей важности `!threads` показывает, что только потоки 0 и 2 имеют некоторый код .NET (чтобы все поместилось на странице, я привожу информацию индивидуальных потоков в виде таблицы, в WinDBG вы видите это как длинные горизонтальные строки):


```

at [+0x67] [+0x16] c:\junk\cruft\exceptapp\class1.cs:14
  PARAM: this: 0x04a41b5c (ExceptApp.DoSomething)
  PARAM: value class ExceptApp.Days StrParam
  PARAM: unsigned int8 ValueParam: 0x07
0012f630 06d301e2 [DEFAULT] [hasThis] Void ExceptApp.DoSomething.Reh
                                     (I4,String)
at [+0x6a] [+0x2b] c:\junk\cruft\exceptapp\class1.cs:23
  PARAM: this: 0x04a41b5c (ExceptApp.DoSomething)
  PARAM: class System.String i: 0x00000042
  PARAM: int8 StrParam: 77863812
  LOCAL: class System.String s: 0x04a45670 (System.String)
  LOCAL: value class ExceptApp.Days e: 0x003e5278 0x0012f63c
:

```

Похоже, в отображении параметров есть ошибка, так как команда `!clrstack` не всегда корректно отображает тип параметра. В методе `DoSomething.Doh` вы можете увидеть, что он принимает значения `String (StrParam)` и `Days (ValueParam)`. Однако информация `PARAM:` показывает параметр `StrParam` как `value class ExceptApp.Days` и `ValueParam` как `unsigned int8`. К счастью для параметров размерного типа, даже когда тип их неверен, рядом с именем параметра отображается его корректное значение. В примере с `ValueParam` переданное значение 7 соответствует перечислению `Fri`.

Прежде чем перейти к постижению значений размерных классов и объектов, я хочу отметить одну команду просмотра стека, которую, возможно, вы найдете полезной. Если вы работаете над сложными вызовами между .NET и неуправляемым кодом, вам понадобится увидеть стек вызовов, включающий все, и в этом случае команда `!dumpstack` — ваш лучший друг. В целом она делает отличную работу, но, имея полную PDB-базу символов для .NET Framework, она могла бы делать ее лучше. Временами `!dumpstack` сообщает «Use alternate method which may not work» (воспользуйтесь альтернативным методом, так как этот может не работать), что, кажется, указывает на то, что производится попытка просмотреть стек при отсутствии информации о некоторых символах.

Строки `LOCAL:` показывают, что в `DoSomething.Reh` имеются две локальных переменных: `s` (объект `String`) и `e` (размерный класс `Days`). После каждого выводится шестнадцатеричный адрес, описывающий тип. Для размерного класса `Days` имеются два числа `0x003E5278` и `0x0012F63C`. Первое — таблица методов, второе — расположение значения в памяти. Чтобы увидеть это значение в памяти, просто дайте команду распечатки содержимого памяти WinDBG, такую как `dd 0x0012F63C`.

Просмотр таблицы методов, описывающей данные метода, информацию о модуле и карту интерфейса среди всего прочего осуществляется командой `SOS !dumpmt`. Выполнение `!dumpmt 0x003E5278` для примера `EXCEPTAPPEXE` выводит:

```

0:000> !dumpmt 0x003e5278
EEClass : 06c03b1c
Module : 001521a0
Name: ExceptApp.Days
mdToken: 02000002 (D:\Dev\ExceptApp\bin\Debug\ExceptApp.exe)
MethodTable Flags : 80000
Number of IFaces in IFaceMap : 3

```

Interface Map : 003e5380
Slots in VTable : 55

В таблице методов, отображаемой двумя первыми числами, видно, в каком модуле определен этот метод, а также класс исполняющей системы .NET. Для интерфейсов в документации SOS есть прекрасный пример, как пройтись по картам интерфейсов, и я бы одобрил ваше знакомство с ним. Если у вас есть горячее желание увидеть все методы в виртуальной таблице конкретного класса или объекта вместе с описателями методов, можно задать ключ `md` в команде перед значением таблицы методов. В случае EXCEPTAPP.EXE и ее размерного класса `ExceptApp.Days`, будут перечислены 55 методов. Как указывает документация SOS в разделе «How Do I...?», просмотр дескрипторов методов полезен при установке точек прерывания на конкретных методах.

Так как мы рассматриваем информацию класса и модуля для таблицы методов `ExceptApp.Days`, я сделаю небольшое отступление. Как только вы получите адрес класса исполняющей системы .NET, `!dumpclass` покажет вам все, что вы только могли пожелать узнать о классе, в том числе и информацию обо всех полях данных класса. Чтобы увидеть информацию о модуле, дайте команду `!dumpmodule`. В документации есть пример, как с помощью вывода `!dumpmodule` пройтись по памяти и найти классы и таблицы методов модуля.

Теперь, когда у нас есть основы размерного класса, взглянем осмысленно на локальную переменную `s` класса `String` в `DoSomething.Reh`:

LOCAL: class System.String s: 0x04a45670 (System.String)

Так как `s` — объект, то после имени переменной отображается только одно шестнадцатеричное значение — размещение объекта в памяти. С помощью команды `!dumpobj` вы увидите всю информацию об объекте:

```
0:000> !dumpobj 0x04a45670
Name: System.String
MethodTable 0x79b7daf0
EEClass 0x79b7de3c
Size 92(0x5c) bytes
mdToken: 0200000f (e:\winnt\microsoft.net\framework\v1.1.4322\mscorlib.dll)
String: Tommy can you see me? Can you see me?
FieldDesc*: 79b7dea0
      MT      Field  Offset      Type      Attr      Value Name
79b7daf0 4000013      4 System.Int32 instance    38 _arrayLength
79b7daf0 4000014      8 System.Int32 instance    37 m_stringLength
79b7daf0 4000015      c System.Char  instance    54 m_firstChar
79b7daf0 4000016      0      CLASS      shared    static Empty
      >> Domain:Value 00158298:04a412f8 <<
79b7daf0 4000017      4      CLASS      shared    static WhitespaceChars
      >> Domain:Value 00158298:04a4130c <<
```

Значения некоторых полей `MethodTable`, `EEClass` и `MT` (или `Method Table`) могут использоваться в командах, которые мы только что обсудили. Что касается членов полей, то `!dumpobj` для простых размерных типов показывает их значения прямо в таблице. Для объекта класса `String` из предыдущего примера значение

`m_stringLength`, равное 37, показывает количество символов в строке. Как вы сейчас увидите в полях членов объекта поле `Value` содержит экземпляр объекта, а вы можете дать команду `!dumpobj`, чтобы увидеть это значение.

Строки, ограниченные знаками `>>` и `<<` показывают экземпляр домена и положение в домене статического поля перед знаком `>>`. Если бы в `EXCERTAPP.EXE` содержалось несколько доменов приложения, вы бы увидели сведения о двух доменах и значениях для статического поля `WhitespaceChars`.

Теперь, когда я осветил некоторые основные команды, я хочу связать их вместе и показать, как с их помощью искать полезную информацию. Так как программа `EXCERTAPP.EXE` остановлена WinDBG в результате возникшего исключения, было бы хорошо увидеть, какая исключительная ситуация возникла и что содержат некоторые поля, в результате чего можно было бы узнать, почему `EXCERTAPP.EXE` остановилась.

Из команды `!threads` мы знаем, что первый поток в настоящее время исполняет исключительную ситуацию `System.ArgumentException`. Приглядевшись к выводу команд `!clrstack` или `!dumpstack`, вы заметите, что нет никаких локальных переменных или параметров, для которых был бы указан тип `System.ArgumentException`. Хорошие новости в том, что хорошая команда показывает все объекты, находящиеся в настоящее время в стеке текущего потока:

```
0:000> !dumpstackobjects
ESP/REG Object Name
ebx      04a45670 System.String Tommy can you see me? Can you see me?
0012f50c 04a45f64 System.ArgumentException
0012f524 04a45f64 System.ArgumentException
0012f538 04a45f64 System.ArgumentException
0012f558 04a44bc4 System.String Reh =
0012f55c 04a45f64 System.ArgumentException
0012f560 04a45670 System.String Tommy can you see me? Can you see me?
0012f564 04a4431c System.Byte[]
0012f568 04a43a58 System.IO.__ConsoleStream
0012f5a0 04a45f64 System.ArgumentException
:
```

Так как `!dumpstackobjects` бродит по стеку вверх, вы увидите некоторые элементы много раз, так как они передаются как параметры ко многим функциям. В предыдущей распечатке вы могли увидеть несколько объектов `System.ArgumentException`, но, посмотрев на значение объекта рядом с каждым объектом, вы заметите, что все они ссылаются на один и тот же экземпляр объекта `0x04A45F64`.

Чтобы увидеть объект `System.ArgumentException` я использую команду `!dumpobj`. Я перенес колонку `Name` на следующую строку, чтобы все поместилось на одной странице.

```
0:000> !dumpobj 04a45f64
Name: System.ArgumentException
MethodTable 0x79b87b84
EEClass 0x79b87c0c
Size 68(0x44) bytes
mdToken: 02000038 (e:\winnt\microsoft.net\framework\v1.1.4322\mscorlib.dll)
```

FieldDesc*: 79b87c70						
MT	Field	Offset	Type	Attr	Value	Name
79b7fcd4	400001d	4	CLASS	instance	00000000	_className
79b7fcd4	400001e	8	CLASS	instance	00000000	_exceptionMethod
79b7fcd4	400001f	c	CLASS	instance	00000000	
79b7fcd4	4000020	10	CLASS	instance	04a456cc	_exceptionMethodString
79b7fcd4	4000021	14	CLASS	instance	00000000	_innerException
79b7fcd4	4000022	18	CLASS	instance	00000000	
79b7fcd4	4000023	1c	CLASS	instance	00000000	_helpURL
79b7fcd4	4000024	20	CLASS	instance	00000000	_stackTrace
79b7fcd4	4000025	24	CLASS	instance	00000000	
79b7fcd4	4000026	2c	System.Int32	instance	0	_stackTraceString
79b7fcd4	4000027	30	System.Int32	instance	0	_remoteStackTraceString
79b7fcd4	4000028	28	CLASS	instance	-2147024809	_remoteStackIndex
79b7fcd4	4000029	34	System.Int32	instance	00000000	_HResult
79b7fcd4	400002a	38	System.Int32	instance	0	_source
79b87b84	40000d7	3c	CLASS	instance	-532459699	_xptrs
79b87b84	40000d7	3c	CLASS	instance	04a45708	_xcode
79b87b84	40000d7	3c	CLASS	instance	04a45708	m_paramName

Важным свойством в исключении является `Message`. Так как я не могу вызвать метод прямо из WinDBG, чтобы увидеть это значение, я взгляну на поле `_message`, так как это и есть то место, где свойство `Message` хранит реальную строку. Так как поле `_message` помечено как `CLASS`, то шестнадцатеричное число в столбце `Value` является экземпляром объекта. Чтобы увидеть этот объект, я выполню еще одну команду `!dumpobj`, чтобы просмотреть его. Как мы видим, объект `String` имеет специальное поле, поэтому мы можем видеть его действительное значение, которое выворачивается в безобидное «Throwing an exception».

```
0:000> !dumpobj 04a456cc
Name: System.String
MethodTable 0x79b7daf0
EEClass 0x79b7de3c
Size 60(0x3c) bytes
mdToken: 0200000f (e:\winnt\microsoft.net\framework\v1.1.4322\mscorlib.dll)
String: Thowing an exception
FieldDesc*: 79b7dea0
MT      Field  Offset      Type      Attr      Value Name
79b7daf0 4000013     4 System.Int32 instance    21 m_arrayLength
79b7daf0 4000014     8 System.Int32 instance    20 m_stringLength
79b7daf0 4000015     c System.Char  instance    54 m_firstChar
79b7daf0 4000016     0 CLASS      shared    static Empty
>> Domain:Value 00158298:04a412f8 <<
79b7daf0 4000017     4 CLASS      shared    static WhitespaceChars
>> Domain:Value 00158298:04a4130c <<
```

Резюме

Моя цель была показать мощь WinDBG и помочь вам преодолеть препятствия, которые вы встретите на своем пути. Если я еще недостаточно говорил об этом, позвольте повторить, что вам действительно необходимо читать документацию WinDBG.

Самый большой фокус в WinDBG — настройка символов и пути к исходным текстам, а также при отладке дампа, настройка пути к образу исполняемого файла. WinDBG потрясающе гибок, позволяя управлять поиском символов и источника их загрузки.

В дополнение к возможности отлаживать несколько процессов одновременно WinDBG предлагает средства управления точками прерывания, так что вы можете остановиться точно там и тогда, где и когда пожелаете. Способность исполнять команды как в точках прерывания, так и в исключительных ситуациях, дает исключительную возможность отслеживать самые трудные ошибки. Команды расширения помогают увидеть внутри процессов то, что невозможно было видеть раньше.

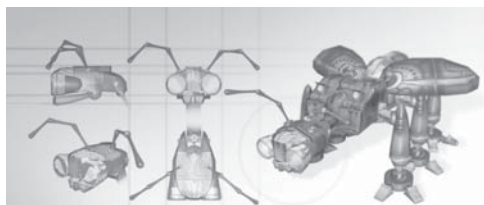
Одна из самых важных причин применения WinDBG — управление дампами. Хотя Visual Studio .NET предлагает читать файлы дампов, но это применимо только для дампов, созданных на машинах разработчиков. WinDBG — инструмент, позволяющий читать реальные файлы дампов, полученных от заказчиков.

Наконец, я надеюсь, что смог предоставить вам стартовые позиции для использования интереснейшей библиотеки расширения SOS, поддерживающей отладку .NET. Как ни крути, это единственный способ отладить приложения, в которых скомбинированы управляемый и неуправляемый код. Вам, конечно, понадобится время на то, чтобы полностью освоить SOS. Кроме того, чтение «Production Debugging for .NET Framework Applications» необходимо для понимания трудностей, которые таят огромные сверхмощные приложения.

ЧАСТЬ III

МОЩНЫЕ СРЕДСТВА И МЕТОДЫ ОТЛАДКИ ПРИЛОЖЕНИЙ .NET





Расширение возможностей интегрированной среды разработки Visual Studio .NET

Эффективные средства разработки должны быть достаточно гибкими, чтобы работать на вас, а не заставлять приспосабливаться к ним. По-моему, до Microsoft Visual Studio .NET предыдущие версии Visual Studio были достаточно хороши только в плане отладки, но не редактирования программ и общего использования. Теперь же я с радостью могу сообщить, что, хотя Visual Studio .NET все еще не лишена недостатков, именно эту среду я использую для редактирования и разработки. В первую очередь это объясняется тем, что в сравнении с предыдущими версиями Visual Studio среда Visual Studio .NET — просто образец расширяемости. Если вы считаете, что какие-то функции в ней реализованы не так или просто упущены, то найдете все средства, необходимые для исправления подобных недостатков.

Когда я работал в NuMega, нас очень часто спрашивали, как встроить собственные окна в Visual Studio. Сейчас я с легкостью могу ответить на этот вопрос, но это уже не важно, поскольку это теперь доступно каждому. Почти каждый разработчик хотел бы встроить в IDE то или иное средство или окно, но до сих пор у нас не было удобного способа сделать это. Новая модель расширяемости Visual Studio .NET позволяет теперь любому программисту добавить в IDE собственные окна. Способы этого описаны и поддерживаются гораздо лучше, чем специальная Package Partner Program, необходимая для внедрения в IDE в предыдущих версиях Visual Studio.

Расширяемость IDE складывается из макросов, надстроек и мастеров. Для создания макросов в Visual Studio .NET служит встроенный редактор макросов (Macros IDE editor). Он выглядит и ведет себя точно так же, как и среда Visual Studio .NET, поэтому ваши усилия по изучению этой среды сполна окупятся при создании макросов. Стоит сказать, что макросы имеют одно ограничение: разрабатывать их можно только на Microsoft Visual Basic .NET. Предполагалось, что .NET будет обеспечивать полную гибкость в использовании языков программирования, поэтому я не могу понять, почему Microsoft пошла на это ограничение, не поддержав C#. По сути это значит, что, даже если вы считаете себя самым преданным поклонником C# (возможно, поскольку вам очень нравятся точки с запятой), для создания макросов все же придется освоить и Visual Basic .NET.

Второй вариант расширения IDE обеспечивают надстройки. Макросы отлично подходят для небольших задач, не связанных с пользовательским интерфейсом, тогда как надстройки — это компоненты COM, позволяющие создавать истинные расширения IDE. Например, надстройки позволяют создавать окна инструментов (свои собственные), добавлять страницы свойств в диалоговое окно Options (свойства) и реагировать из надстроек на команды меню. Любой программист, которому вы предоставите свой макрос, сможет изучить его исходный код, тогда как надстройки распространяются в двоичной форме и для их создания подойдет любой язык, поддерживающий COM.

Наконец, для расширения функциональности IDE служат мастера. Они особенно полезны, когда для решения какой-либо задачи пользователь должен выполнить ряд действий. Отличный пример — мастер Smart Device Application Wizard, помогающий создать приложение для «интеллектуального» устройства. Из всех способов расширения IDE мастера используются реже всего.

В этой главе я хочу показать вам возможности макросов и надстроек, рассмотрев три реальных средства, без которых я уже просто не могу жить. Увидев, на что они способны, вы получите неплохое представление о проблемах, с которыми можно столкнуться, создавая собственное Средство, Без Которого Никто Не Может Жить. Так как очень немногие программисты нуждаются в создании мастеров, я не буду обсуждать эту тему. Что до макросов и надстроек, то в отличие от других авторов я не буду опускаться до подробностей типа «чтобы появилась надстройка, нажмите на эту кнопку мастера». Я полагаю, что вы изучили документацию к Visual Studio .NET, поэтому, чтобы вы могли сэкономить время при создании своих средств, основное внимание я уделю проблемам, с которыми в свое время столкнулся сам.

Первое описываемое мной средство — CommentTater — очень полезный макрос, гарантирующий наличие и актуальность комментариев в программах C#. Первая из двух надстроек, SuperSaver, исправляет один недостаток сохранения файлов в Visual Studio .NET и реализует функции фонового сохранения файлов и добавления страниц свойств в окно Options. Этот отличный пример полной надстройки понравится вам еще и небольшим объемом. Вторая надстройка, SettingsMaster, позволяет автоматизировать конфигурирование параметров сборки программы, чтобы безо всяких усилий вы могли использовать во всех проектах параметры, рекомендованные мной в главе 2. Благодаря SettingsMaster координировать все проекты группы будет проще простого. Вам больше не придется вручную изменять параметры своих проектов! Все эти средства вы найдете на CD.

Расширение IDE при помощи макросов

Сначала я хотел бы рассмотреть некоторые проблемы, связанные с макросами. Прежде всего замечу, что, даже если вам кажется, что вы придумали самую лучшую надстройку в мире и вам не терпится приступить к ее реализации, не спешите. Так как в макросах используются те же объекты и свойства, что и в надстройках, макросы обеспечивают наилучшую возможность изучить подробности объектной модели Visual Studio .NET. Как вы увидите, объектная модель имеет ряд хитростей, поэтому создание надстроек иногда вызывает проблемы. Макросы гораздо проще создавать и отлаживать, поэтому я рекомендую для начала разобраться с ними.

Не торопитесь открывать меню Tools (инструменты) и выбирать пункт Macros (макросы) — почитайте документацию о макросах и объектной модели. Макросы описываются в разделе Visual Studio .NET\Developing With Visual Studio .NET\Manipulating The Development Environment\Automating Repetitive Actions By Using Macros, объектная модель — в разделе Visual Studio .NET\Developing With Visual Studio .NET\Reference\Automation And Extensibility Reference.

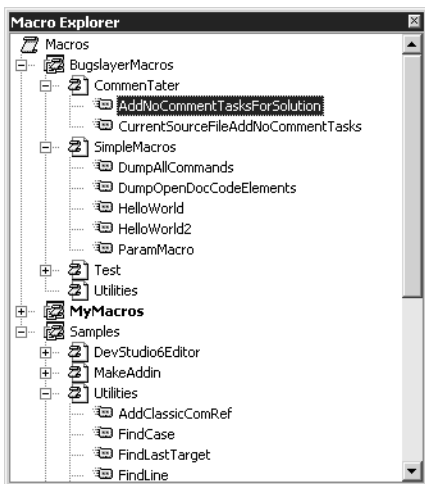


Рис. 9-1. Окно *Macro Explorer*

Изучив объекты, попробуйте записать какие-нибудь макросы, чтобы увидеть объекты в действии. Помните: запись макросов работает преимущественно в редакторах кода [включая диалоговые окна Find/Replace (найти/заменить)], при использовании Solution Explorer (проводник для работы с решением) и при активизации окон. Вы не сможете записать такие действия, как создание формы Web или Windows с элементами управления. Изучите предоставленные Microsoft примеры макросов, которые автоматически загружаются в Macro Explorer (проводник для работы с макросами) (рис. 9-1) при загрузке проекта макросов Samples. Примеры макросов наглядно иллюстрируют использование объектной модели для решения проблем. Больше всего мне нравится макрос MakeAddinFromMacroProj (из проекта MakeAddin), преобразующий макрос в надстройку. Он демонстрирует всю мощь, которую нам дает Visual Studio .NET.

Выполнить макрос можно двумя способами: дважды щелкнуть имя функции макроса в Macro Explorer или вызвать окно Command (команды). Если вы начнете писать в окне Command Window слово «macro», технология IntelliSense отобразит окно подсказки, в котором можно выбрать нужный макрос (рис. 9-2).

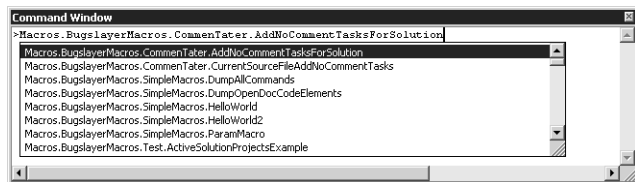


Рис. 9-2. Выполнение макроса из окна Command, поддерживающего технологию IntelliSense

Если вы предпочитаете выполнять макросы или встроенные команды из окна Command, можете назначить им более короткие текстовые имена при помощи встроенной команды `alias`. Это избавит вас от необходимости писать при каждом запуске макроса что-нибудь вроде:

```
Macros.BugslayerMacros.CommentTater.AddNoCommentTasksForSolution
```

Для удаления псевдонима служит команда `alias` с ключом `/d`.

Параметры макросов

Macro Explorer и всплывающая подсказка IntelliSense в окне Command имеют одну не всегда очевидную особенность: в обоих окнах отображаются только макросы, не принимающие параметров. Это имеет смысл в Macro Explorer, так как иначе было бы очень сложно передать макросу параметры при двойном щелчке его имени. Однако при использовании окна Command возможность передачи параметров оказалась бы совсем не лишней. Для этого в объявлении макроса нужно указать, что он принимает единственный необязательный (optional) строковый параметр:

```
Sub ParamMacro(Optional ByVal Param As String = "")
```

В случае нескольких параметров все почти так же просто: для каждого из них нужно добавить по одному необязательному строковому параметру. Вот пример макроса, принимающего три параметра:

```
Sub ParamMacroWithThree(Optional ByVal Param1 As String = "", _  
                        Optional ByVal Param2 As String = "", _  
                        Optional ByVal Param3 As String = "")
```

Использование нескольких необязательных строковых параметров отлично работает в Visual Studio .NET 2003, но если вы хотите, чтобы ваши макросы были обратно совместимы с Visual Studio .NET 2002, то будете разочарованы: такие макросы не запускаются даже в окне Command. В Visual Studio .NET можно указать только один необязательный строковый параметр. Если же вы хотите передавать несколько параметров в окне Command, ситуация становится несколько странной. При наличии пробелов макрос не вызывается. Это значит, что, если вам

нужно передать в макрос три параметра, передавайте их в одной строке, разделяя запятыми. Если ваш строковый параметр содержит пробелы, его нужно передавать отдельной строкой, заключив в кавычки. Странно, но в фактической строке, обрабатываемой вашим макросом, кавычек не будет. Замечу, что передать заключенную в кавычки строку, содержащую запятую, невозможно. Вот два примера правильной передачи строк в макросы с соответствующими значениями строк при обработке параметров макроса:

Вызов макроса из окна Command: `MyMacro x,y,z`

Строка параметров макроса: `x,y,z`

Вызов макроса из окна Command: `MyMacro x,"a string",y`

Строка параметров макроса: `x,a string,y`

Чтобы упростить работу с макросами в обеих версиях Visual Studio, я включил в число файлов к этой книге вспомогательный модуль (Utilities.VB, находящийся в каталоге Macros), который содержит функцию `SplitParams`, разделяющую параметры и заносящую их в массив строк. В этот же модуль я включил некоторые удобные оболочки для объектов окон Command и Output (вывод). Мы используем эти объекты, когда будем работать с макросом SimpleMacros (см. раздел «Элементы кода»).

Проблемы с проектами

Здесь я должен сделать одно важное замечание: при чтении документации не совсем ясно, что в разных языках используются разные объектные модели проектов. В общем обсуждении объектной модели среды для объекта `Project` приводится обширный список полезных методов для манипуляции проектами и их сохранения. Из-за ошибочного впечатления, что общий проект является корнем всех проектов, разрабатываемых с использованием всех языков и технологий, мне пришлось потратить впустую много времени. На самом деле нет ничего, что находилось бы так далеко от истины. Есть только два правильно документированных типа проектов: `VSPProject` для проектов C# и Visual Basic .NET и `VCProject` для проектов C++. Другие типы проектов, такие как `CAB` и `Setup`, не документированы и будут генерировать массу исключений, если вы будете получать к ним доступ через общий объект `Project`. Если методы, которые, по вашему мнению, должны работать, такие как `Save`, генерируют исключения `Not Implemented` (метод не реализован), это очень раздражает. Работая с проектами, убедитесь, что вы реализовали достаточно обработчиков исключений!

При перечислении проектов решения вам предоставляется общий объект `Project`. Для определения типа проекта лучше всего использовать `GUID`, который позволяет получить свойство `Kind`. Строки `GUID` для разных типов проектов см. в табл. 9-1. Если вы разрабатываете проект на C++, работоспособными будут лишь немногие из методов `Project`, поэтому немедленно преобразуйте `Project` в `VCProject` через свойство `Object` и используйте объект `VCProject`. Объекты `VSPProject` относятся к доступу к ним через общий объект `Project` чуть благожелательнее.

Альтернативный метод определения типа проекта — использование свойства `Project.CodeModel`, описывающего элементы кода в файлах проекта. Об элементах кода я расскажу чуть ниже. Объект `CodeModel` содержит свойство `Language`, которое

возвращает строку GUID, определяющую язык. В надстройке SettingsMaster я определил тип проекта с помощью этого метода, так как мне нужно было поддерживать все типы языков. Кстати, в документации допущена ошибка: строки, возвращаемые свойством `Language`, не являются константами `vsCMLanguage` — такой константы нет. На самом деле это свойство возвращает константу `CodeModelLanguageConstants`.

Табл. 9-1. Документированные строки GUID для разных типов проектов

Язык проекта	GUID
C#	{FAE04EC0-301F-11D3-BF4B-00C04F79EFBC}
Visual Basic .NET	{F184B08F-C81C-45F6-A57F-5ABD9991F28F}
C++	{8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942}
J#	{E6FDF86B-F3D1-11D4-8576-0002A516ECE8}

Элементы кода

Одно из самых удивительных свойств Visual Studio .NET в том, что через объектную модель можно легко получить доступ ко всем конструкциям программирования в исходных файлах. То, что теперь мы можем безо всякого синтаксического анализа добавлять/изменять/удалять, скажем, методы в программах на всех языках, поддерживаемых Visual Studio .NET, открывает широкие возможности по созданию уникальных средств, которые могли бы никогда не быть созданы из-за чрезвычайной сложности синтаксического анализа. Каждый раз, когда я использую элементы кода для изменения кода в исходных файлах, я удивляюсь тому, насколько прекрасна эта возможность.

Чтобы показать, насколько легко использовать элементы кода, я написал макрос, который создает дамп элементов кода активного документа (листинг 9-1). Из листинга не видно, что данный код работает для любого языка. В выводе указывается имя элемента кода вместе с его типом. Этот макрос хранится в файле `Macros\SimpleMacros.VB` на CD; макросу нужен также вспомогательный файл `Utilities.VB`. Вот частичный результат обработки этим макросом файла `SETTINGSMaster.VB` из проекта `SettingsMaster`:

```
SettingsMaster.SettingsMaster(vsCMElementClass)
SettingsMaster.SettingsMaster.RegSettings(vsCMElementVariable)
SettingsMaster.SettingsMaster.m_ApplicationObject(vsCMElementVariable)
SettingsMaster.SettingsMaster.New(vsCMElementFunction)
    ApplicationObject(vsCMElementParameter)
    AddInInstance(vsCMElementParameter)
```

При работе с элементами кода возникает небольшая проблема с согласованностью получения дочерних элементов. Например, объект `CodeClass` получает подэлементы классов Visual Basic .NET или C# через свойство `Members`. Однако, чтобы получить подэлементы классов C++, описываемых объектом `CodeClass`, служит свойство `Children`. В заключительной части процедуры `DumpElements` я использовал для нахождения нужных дочерних элементов несколько вложенных блоков `Try...Catch`. Как можно увидеть, для получения дочерних объектов объекты `CodeFunction` используют еще одно свойство — `Parameters`. Важно, чтобы вы знали о разных способах получения дочерних элементов.

Листинг 9-1. Процедура DumpActiveDocCodeElements

```
' Создание дампа всех элементов кода для открытого документа проекта.
Public Sub DumpActiveDocCodeElements()
    ' Создание объекта для вывода информации. Заметьте:
    ' класс OutputPane относится к проекту макросов Utilities.
    Dim ow As OutputPane = New OutputPane("Open Doc Code Elements")
    ' Очистка панели вывода.
    ow.Clear()

    ' Есть ли открытый документ?
    Dim Doc As Document = DTE.ActiveDocument
    If (Doc Is Nothing) Then
        ow.WriteLine("No open document")
        Exit Sub
    End If

    ' Получение модели кода для этого документа. Для работы
    ' с элементами кода надо использовать элемент проекта.
    Dim FileMod As FileCodeModel = Doc.ProjectItem.FileCodeModel

    If (Not (FileMod Is Nothing)) Then
        DumpElements(ow, FileMod.CodeElements, 0)
    Else
        ow.WriteLine("Unable to get the FileCodeModel!")
    End If
End Sub

Private Sub DumpElements(ByVal ow As OutputPane, _
                        ByVal Elems As CodeElements, _
                        ByVal Level As Integer)

    Dim Elem As CodeElement
    For Each Elem In Elems

        Dim i As Integer = 0

        While (i < Level)
            ow.OutPane.OutputString("  ")
            i = i + 1
        End While

        ' Если при получении доступа к свойству FullName генерируется
        ' исключение, вероятно, это объясняется безымянным параметром.
        Dim sName As String
        Try
            sName = Elem.FullName
        Catch e As System.Exception
            sName = "'Empty Name'"
        End Try
        ow.WriteLine(sName + "(" + Elem.Kind.ToString() + ")")
    End For
End Sub
```

```
' Это довольно странно. Некоторые объекты CodeElements используют
' для получения подэлементов свойство Children, тогда как другие –
' свойство Members. Ну, а функции используют свойство Parameters.
Dim SubCodeElems As CodeElements = Nothing

Try
    SubCodeElems = Elem.Children
Catch
    Try
        SubCodeElems = Elem.Members
    Catch
        If (TypeOf Elem Is CodeFunction) Then
            SubCodeElems = Elem.Parameters
        Else
            SubCodeElems = Nothing
        End If
    End Try
End Try

If (Not (SubCodeElems Is Nothing)) Then
    If (SubCodeElems.Count > 0) Then
        DumpElements(ow, SubCodeElems, Level + 1)
    End If
End If
Next
End Sub
```

CommentTater: лекарство от распространенных проблем?

Одним из абсолютно бесценных свойств C# являются документирующие комментарии XML. Так называют теги XML, содержащиеся в комментариях, описывающих свойства или методы в конкретном файле. Фактически IDE помогает вам, автоматически включая такие комментарии для конструкций программы, перед которыми вы пишете `///`. Есть три очень веских причины, почему всегда следует заполнять документирующие комментарии C#. Во-первых, это стандартизирует комментарии между отдельными группами и во всей вселенной C#. Во-вторых, технология IntelliSense среды разработки автоматически отображает информацию, указанную в тэгах `<summary>` и `<param>`, что облегчает использование вашего кода другими программистами, предоставляя им гораздо больше данных об элементах вашей программы. Если код является частью проекта, для получения преимуществ документирующих комментариев ничего делать не нужно. Если вы предоставляете решение только в двоичной форме, документирующие комментарии могут быть собраны в XML-файл при компиляции, поэтому и в такой ситуации вы можете предоставить пользователям отличный набор подсказок. Для этого нужно только разместить итоговый XML-файл в том же каталоге, что и двоичный файл, и Visual Studio .NET будет автоматически отображать комментарии в подсказках IntelliSense.

Наконец, при помощи XSLT из итогового XML-файла можно создать полную систему документации к вашей программе. Заметьте, что я не имею в виду команду Build Comment Web Pages (создать Web-страницы комментариев) из меню Tools. Эта команда не учитывает много важной информации, например, тэги `<exception>`, поэтому она не так уж и полезна. Как я покажу чуть ниже, для генерирования документации можно использовать гораздо лучшие средства.

Чтобы максимально эффективно документировать свой код, изучите в документации к Visual Studio .NET все, что касается тэгов комментариев XML. Для создания файла XML-документа откройте окно Property Pages (страницы свойств), папку Configuration Properties (конфигурационные свойства), страницу Build (сборка программы) и заполните поле XML Documentation File (файл XML-документации) (рис. 9-3). Это поле нужно заполнять отдельно для каждой конфигурации, чтобы файл документации создавался при каждой сборке программы.

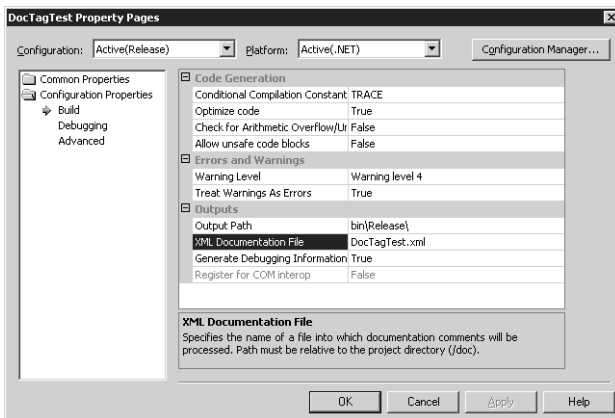


Рис. 9-3. Установка ключа командной строки `/doc` для создания файла документирующих комментариев XML

Чтобы вы могли создать полный вывод из файлов комментариев XML, я разместил в каталоге DocCommentsXSL на CD файл трансформации XSL и каскадную таблицу стилей. Однако гораздо лучше использовать средство NDoc, которое можно загрузить по адресу <http://ndoc.sourceforge.net>. NDoc обрабатывает XML-комментарии и создает файл помощи HTML, который выглядит в точности, как документация MSDN к библиотеке классов .NET Framework. NDoc даже предоставляет ссылки на общие методы вроде `GetHashCode`, так что из него вы можете переходить прямо в документацию MSDN! NDoc — прекрасный способ документирования кода вашей группы, и я настоятельно рекомендую его использовать. Благодаря реализованной в Visual Studio .NET 2003 обработке программы после ее сборки (post build processing) вы можете с легкостью включить NDoc в свой процесс сборки.

Так как документирующие комментарии настолько важны, мне захотелось разработать метод автоматического их добавления в мой код C#. Примерно в то же время, когда я об этом подумал, я обнаружил, что окно Task List (список заданий) автоматически отображает все комментарии, начинающиеся с ключевых фраз вроде «TODO», когда вы нажимаете в нем правую кнопку и выбираете в меню Show Tasks (показать задания) пункт All (все) или Comment (комментарии). Я решил

создать макрос или надстройку, которые добавляли бы все пропущенные документирующие комментарии и обрабатывали в них фразу «TODO», чтобы можно было легко просматривать комментарии и гарантировать их правильное заполнение. Результатом стал CommenTater. Вот пример метода, обработанного CommenTater:

```
/// <summary>
/// TODO - Add Test function summary comment
/// </summary>
/// <remarks>
/// TODO - Add Test function remarks comment
/// </remarks>
/// <param name="x">
/// TODO - Add x parameter comment
/// </param>
/// <param name="y">
/// TODO - Add y parameter comment
/// </param>
/// <returns>
/// TODO - Add return comment
/// </returns>
public static int Test ( Int32 x , Int32 y )
{
    return ( x ) ;
}
```

Visual Studio .NET делает перебор элементов кода в исходном файле тривиальной задачей, поэтому я был очень доволен, так как думал, что мне нужно будет только просмотреть элементы кода, получить строки, предшествующие любому методу или свойству, и вставить комментарии в случае их отсутствия. Когда я обнаружил, что все элементы кода имеют свойство `DocComment`, возвращающее действительный комментарий для данного элемента, я тут же снял шляпу перед разработчиками за то, что они продумали все заранее и сделали элементы кода по-настоящему полезными. Теперь мне нужно было только присвоить свойству `DocComment` нужное значение, и все было бы чудесно.

Возможно, сейчас вам следует открыть файл `CommenTater.VB` из каталога `CommenTater`. Исходный код этого макроса слишком объемен, чтобы воспроизводить его в книге, поэтому следите за моими мыслями по файлу. Моя основная идея заключалась в создании двух процедур, `AddNoCommentTasksForSolution` и `CurrentSourceFileAddNoCommentTasks`. Вы можете по их именам сказать, на каком уровне они работают. Большей частью базовый алгоритм похож на примеры из листинга 9-1: я просто просматриваю все элементы кода и использую их свойства `DocComment`.

Первая проблема, с которой я столкнулся, была связана с тем, что я считаю небольшим недостатком объектной модели элементов кода. Свойство `DocComment` не является общим для класса `CodeElement`, который может быть использован в качестве базового класса для любого общего элемента кода. Поэтому мне пришлось преобразовать общий объект `CodeElement` в действительный тип элемента, опираясь на свойство `Kind`. Вот почему процедура `RecurseCodeElements` содержит большой оператор `Select...Case`.

Вторая проблема была полностью на моей совести. Я почему-то никогда не осознавал, что со свойством `DocComment` конструкции кода нужно обращаться, как с полноценным XML-фрагментом. Я создавал нужную строку комментария, но, когда я пытался назначить ее свойству `DocComment`, генерировалось исключение `ArgumentException`. Я был очень озадачен этим, так как думал, что свойство `DocComment` допускает чтение и запись, но на деле все выглядело так, будто оно поддерживало только чтение. Из-за какого-то помутнения я не понимал, что генерирование исключения объяснялось тем, что я не заключал документирующие комментарии XML в элементы `<doc></doc>`. Вместо этого я решил, что столкнулся с непонятной проблемой, и стал искать альтернативные средства включения текста комментария.

Так как отдельные элементы кода имеют свойство `StartPoint`, мне просто нужно было создать соответствующий объект `EditPoint` и ввести текст. Эксперименты быстро показали, что все работало правильно, и я начал разрабатывать набор процедур для добавления текста. Делать это вручную требуется не так уж и редко, поэтому я закомментировал первоначальные процедуры и оставил в конце файла `CommenTater.VB`.

Первую версию макроса я часто использовал в своих проектах. Иногда макросы могут быть слишком медленными, поэтому я рассматривал возможность преобразования `CommenTater` в полноценную надстройку, но меня его скорость всегда устраивала. Первая версия `CommenTater` только добавляла пропущенные комментарии. Это было прекрасно, но скоро я понял, что мне по-настоящему хочется, чтобы `CommenTater` был умнее и сравнивал имеющиеся комментарии к функциям с тем, что на самом деле присутствует в коде. При изменении прототипов функций, скажем, при добавлении/удалении параметров, я часто забываю обновить соответствующие комментарии. Добавив эту функциональность сравнения, я сделал бы `CommenTater` еще полезнее.

Начав думать о том, что потребуется для обновления существующих комментариев, я слегка загрузил. Если вы помните, в тот момент я думал, что свойство `DocComment` допускает только чтение, поэтому я решил, что для правильного обновления комментариев придется выполнять значительный объем манипуляции с текстом, и это меня не привлекало. Однако, когда я взглянул на `CommenTater` в отладчике макросов, на меня снизошло радостное озарение, и я понял, что для записи в свойство `DocComment` нужно просто размещать вокруг каждого комментария элементы `<doc></doc>`. Когда я преодолел собственную глупость, написать процедуру `ProcessFunctionComment` оказалось гораздо проще (листинг 9-2).

В этот момент в игру вступила мощь библиотеки классов `Microsoft .NET Framework`. Чтобы выполнить всю трудную работу, нужную для получения информации из существующих строк документирующих комментариев и их преобразования, я использовал прекрасный класс `XmlDocument`. Процедура `ProcessFunctionComment` должна была поддерживать переупорядочение комментариев, поэтому я должен был подобрать порядок размещения отдельных узлов в файле. Хочу отметить, что я форматирую комментарии так, как мне нравится, поэтому `CommenTater` может изменить тщательное форматирование ваших комментариев, но никакой информации он не выбросит.

Листинг 9-2. Процедура ProcessFunctionComment из файла Commentater.VB

```
' Эта процедура получает имеющиеся комментарии к функциям
' и гарантирует, что все в порядке. Она может преобразовывать
' ваши комментарии, поэтому вы можете захотеть изменить ее.
Private Sub ProcessFunctionComment(ByVal Func As CodeFunction)

    Debug.Assert("'" <> Func.DocComment, "'''''''" <> Func.DocComment")

    ' Объект, содержащий исходный документирующий комментарий.
    Dim XmlDocOrig As New XmlDocument()
    ' ЭТО ЗДОРОВО! После присвоения свойству PreserveWhitespace
    ' значения True класс XmlDocument будет отвечать почти за все,
    ' что касается форматирования...

    XmlDocOrig.PreserveWhitespace = True
    XmlDocOrig.LoadXml(Func.DocComment)

    Dim RawXML As New StringBuilder()

    ' Получение узла "summary".
    Dim Node As XmlNode
    Dim Nodes As XmlNodeList = XmlDocOrig.GetElementsByTagName("summary")
    If (0 = Nodes.Count) Then
        RawXML.Append(SimpleSummaryComment(Func.Name, "function"))
    Else
        RawXML.AppendFormat("<summary>{0}", vbCrLf)
        For Each Node In Nodes
            RawXML.AppendFormat("{0}{1}", Node.InnerXml, vbCrLf)
        Next
        RawXML.AppendFormat("</summary>{0}", vbCrLf)
    End If

    ' Получение узла "remarks".
    Nodes = XmlDocOrig.GetElementsByTagName("remarks")
    If (Nodes.Count > 0) Then
        RawXML.AppendFormat("<remarks>{0}", vbCrLf)
        For Each Node In Nodes
            RawXML.AppendFormat("{0}{1}", Node.InnerXml, vbCrLf)
        Next
        RawXML.AppendFormat("</remarks>{0}", vbCrLf)
    ElseIf (True = m_FuncShowsRemarks) Then
        RawXML.AppendFormat("<remarks>{0}TODO - Add {1} function " + _
            "remarks comment{0}</remarks>", _
            vbCrLf, Func.Name)
    End If

    ' Получение всех параметров, описанных в документирующих комментариях.
    Nodes = XmlDocOrig.GetElementsByTagName("param")
```

см. след. стр.

```

' Имеет ли функция параметры?
If (0 <> Func.Parameters.Count) Then

    ' Занесение всех существующих параметров комментариев
    ' в хэш-таблицу с именем параметра в качестве ключа.
    Dim ExistHash As New Hashtable()
    For Each Node In Nodes
        Dim ParamName As String
        Dim ParamText As String
        ParamName = Node.Attributes("name").InnerText
        ParamText = Node.InnerText
        ExistHash.Add(ParamName, ParamText)
    Next

    ' Просмотр параметров.
    Dim Elem As CodeElement
    For Each Elem In Func.Parameters
        ' Есть ли этот элемент в хэше заполненных параметров?
        If (True = ExistHash.ContainsKey(Elem.Name)) Then
            RawXML.AppendFormat("<param name=""{0}"">{1}{2}{1}" + _
                                "</param>{1}", _
                                Elem.Name, _
                                vbCrLf, _
                                ExistHash(Elem.Name))
            ' Удаление этого ключа.
            ExistHash.Remove(Elem.Name)
        Else
            ' Был добавлен новый параметр.
            RawXML.AppendFormat("<param name=""{0}"">{1}TODO - Add " + _
                                "{0} parameter comment{1}</param>{1}", _
                                Elem.Name, vbCrLf)
        End If
    Next

    ' Если в хэш-таблице что-то осталось, параметр был или удален,
    ' или переименован. Я добавлю описания оставшихся параметров
    ' с пометками TODO, чтобы пользователь мог удалить их вручную.
    If (ExistHash.Count > 0) Then
        Dim KeyStr As String
        For Each KeyStr In ExistHash.Keys
            Dim Desc = ExistHash(KeyStr)
            RawXML.AppendFormat("<param name=""{0}"">{1}{2}{1}{3}" + _
                                "{1}</param>{1}", _
                                KeyStr, _
                                vbCrLf, _
                                Desc, _
                                "TODO - Remove param tag")
        Next
    End If
End If

```

```
' Обработка возвращаемых значений, если таковые имеются.
If (" " <> Func.Type.AsFullName) Then
    Nodes = XmlDocOrig.GetElementsByTagName("returns")
    ' Обработка узлов "returns".
    If (0 = Nodes.Count) Then
        RawXML.AppendFormat("<returns>{0}TODO - Add return comment" + _
                               "{0}</returns>{0}", _
                               vbCrLf)
    Else
        RawXML.AppendFormat("<returns>{0}", vbCrLf)

        For Each Node In Nodes
            RawXML.AppendFormat("{0}{1}", Node.InnerXml, vbCrLf)
        Next

        RawXML.AppendFormat("</returns>{0}", vbCrLf)
    End If
End If

' Обработка узлов "example".
Nodes = XmlDocOrig.GetElementsByTagName("example")
If (Nodes.Count > 0) Then
    RawXML.AppendFormat("<example>{0}", vbCrLf)
    For Each Node In Nodes
        RawXML.AppendFormat("{0}{1}", Node.InnerXml, vbCrLf)
    Next
    RawXML.AppendFormat("</example>{0}", vbCrLf)
End If

' Обработка узлов "permission".
Nodes = XmlDocOrig.GetElementsByTagName("permission")
If (Nodes.Count > 0) Then
    For Each Node In Nodes
        RawXML.AppendFormat("<permission cref=""{0}"">{1}", _
                               Node.Attributes("cref").InnerText, _
                               vbCrLf)
        RawXML.AppendFormat("{0}{1}", Node.InnerXml, vbCrLf)
        RawXML.AppendFormat("</permission>{0}", vbCrLf)
    Next
End If

' Наконец, узлы "exception".
Nodes = XmlDocOrig.GetElementsByTagName("exception")

If (Nodes.Count > 0) Then
    For Each Node In Nodes
        RawXML.AppendFormat("<exception cref=""{0}"">{1}", _
                               Node.Attributes("cref").InnerText, _
                               vbCrLf)
        RawXML.AppendFormat("{0}{1}", Node.InnerXml, vbCrLf)
```

см. след. стр.

```
RawXML.AppendFormat("</exception>{0}", vbCrLf)
Next
End If
Func.DocComment = FinishOffDocComment(RawXML.ToString())
End Sub
```

Разработав обновление документирующих комментариев, я подумал, что неплохо было бы реализовать обработку контекста отмены. Благодаря этому вы могли бы в случае ошибки нажать Ctrl+Z и восстановить все изменения, сделанные CommentTater. Увы, контекст отмены представляет реальную проблему. Когда у меня нет открытого контекста отмены, все изменения вносятся в документирующие комментарии прекрасно. Однако при открытии перед внесением изменений контекста отмены все путается и выглядит так, будто контекст отмены и элементы кода мешают друг другу. Когда CommentTater выполняет запись в свойство DocComment, стартовые точки элементов кода не обновляются, в результате чего обновление происходит по старым позициям, повреждая файл. Я обнаружил, что, если вместо использования контекста отмены для глобального внесения всех изменений применять его для обновления каждого комментария к методу или свойству, все работает. Это хуже, чем глобальная отмена всех изменений, но хоть какая-то форма отмены. Надеюсь, Microsoft решит проблему с контекстом отмены, чтобы вы могли использовать его глобально для отмены крупномасштабных изменений.

Одна интересная проблема была связана с зарезервированными символами XML, которые вполне могут содержаться в именах функций. Ваша функция может называться `operator &`, но вторая попытка использовать символ `&` в документирующем комментарии XML приведет к исключению, указывающему на некорректный символ. Конечно, это же справедливо для символов `>` и `<`, поэтому операторы `operator <`, `operator >`, `operator <<` и `operator >>` также вызовут проблемы. Чтобы синтаксический анализатор XML не жаловался, функция `BuildFunctionComment` в `CommentTater` производит все нужные замены (например, подставляет `&` вместо `&`).

`CommentTater` — очень полезный макрос, но вы могли бы внести в него одно прекрасное дополнение, работая над которым вы к тому же очень многое узнали бы об объектной модели IDE. Учитывая наличие тэга `<exception>`, вы могли бы документировать генерируемые функцией исключения. Попробуйте сделать так, чтобы ваш код искал функцию каждого оператора `throw` и автоматически добавлял новый элемент для этого конкретного типа исключения. Конечно, когда метод больше не генерирует исключение, вам следует отмечать соответствующие тэги `<exception>` как требующие удаления.

Стандартный вопрос отладки

Есть ли какие-нибудь хитрости отладки макросов и надстроек, написанных на управляемом коде?

Приступив к созданию макросов и надстроек, вы очень быстро заметите, что IDE Visual Studio .NET безумно любит поглощать исключения. Конечно, IDE хочет, чтобы никакие исключения не нарушали ее работу, но она с таким усердием пережевывает все необработанные исключения, что вы мо-

жете даже не подозревать, что ваша программа генерирует исключения. Когда я разрабатывал свои первые макросы, я минут 20 сидел, удивляясь, почему я не могу достигнуть установленной точки прерывания.

В конце концов, чтобы исключить сюрпризы, я открыл окно Exceptions, выбрал в дереве исключений узел Common Language Runtime Exceptions (исключения общезыковой исполняющей среды) и отметил в блоке When The Exception Is Thrown (что делать при генерировании исключения) пункт Break Into The Debugger (выходить в отладчик) (рис. 9-4). Вероятно, после этого вы гораздо чаще будете прерываться в отладчике, но зато это избавит вас от любых сюрпризов.

На рис. 9-4 вы можете увидеть, что я не выбрал узел JScript Exceptions, потому что при разработке надстроек я не использую JScript .NET. Если вы достаточно храбры для создания надстроек на JScript .NET, выберите и этот узел, чтобы прерываться на всех исключениях.

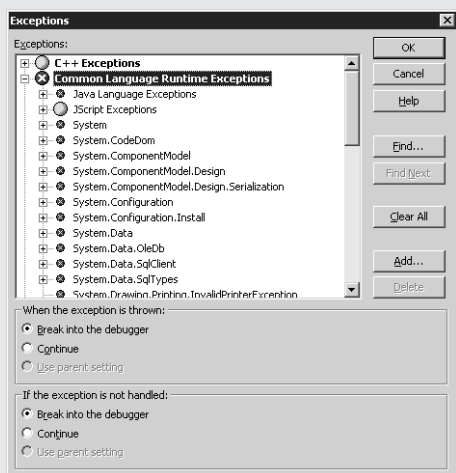


Рис. 9-4. Параметры окна Exceptions, приказывающие выходить в отладчик при всех исключениях

А вот при отладке макросов я заметил, что, даже если в окне Exceptions указано выходить в отладчик при всех исключениях, это может быть не так. Чтобы отладчик макросов начал работать правильно, после настройки окна Exceptions установите где-нибудь в своем коде точку прерывания.

Введение в надстройки

Макросы прекрасно подходят для решения небольших изолированных задач, но, если вам нужен более развитый пользовательский интерфейс, расширенные возможности ввода или вы хотите защитить свой исходный код, надо написать надстройку. Разрабатывать макросы гораздо проще, однако надстройки позволяют решать некоторые задачи, перед которыми макросы бессильны, а именно:

- добавление в IDE собственных инструментальных и диалоговых окон;
- добавление в IDE собственных командных панелей (т. е. меню и панелей инструментов);
- добавление собственных страниц свойств в диалоговое окно Options.

Как вы скоро увидите, разрабатывать и отлаживать надстройки намного труднее, чем макросы, поэтому я рекомендую пытаться сделать все, что можно, используя макросы, и только в случае неудачи приступать к сражению с надстройками.

По своей сути надстройки — это объекты COM, подключающиеся к IDE. Если вы волновались о том, как бы не забыть все, что вы изучили о COM за последние годы, не беспокойтесь: кое-что вам понадобится в мире надстроек. Интересно, что вы можете писать свои надстройки на Visual Basic .NET или C#, потому что управляемые языки тоже поддерживают COM. Как и многим из вас, мне очень нравится C++, но повышение производительности, обеспечиваемое .NET, мне нравится еще больше, поэтому в этой главе я буду основное внимание уделять вопросам, связанным с созданием надстроек на управляемых языках.

Как обычно, путешествие в мир надстроек следует начать с документации. Посетите также страницу <http://msdn.microsoft.com/vstudio/downloads/samples/automation.asp>, которая содержит все примеры надстроек и мастеров от Microsoft. Вам непременно захочется провести за чтением кода этих примеров немало времени, так как лучшего способа обучения еще никто не придумал.

Многие надстройки реализованы на нескольких языках, поэтому никаких проблем в данном смысле возникнуть не должно. Некоторые из более сложных примеров, таких как RegExplore, доступны только на C++. Замечу, что код C++ в примерах Microsoft — прекрасный пример плохого программирования. Значительная часть кода изобилует магическими макросами, которые обрабатывают ошибки при помощи `goto` и основаны на предполагаемых именах. Печально, но похожий код генерирует и мастер создания надстроек (Add-In wizard). Если вы решите писать надстройки на C++, не следуйте примеру Microsoft!

Не знаю, как насчет всего остального, но вам обязательно захочется использовать пакет Unsupported Tools (неподдерживаемые средства). Вы можете или загрузить его с указанного сайта, или найти его текущую версию на CD, в каталоге UnsupportedAddInTools. Этот пакет включает программу Generate ICO Data for Extensibility (генерирование данных ICO для системы расширяемости) (Generate-IcoData.exe), которая генерирует шестнадцатеричные данные для значка, нужные для его вывода в окне About. Как это сделать, я покажу ниже. В Unsupported Tools входит также отличная надстройка Extensibility Browser (браузер расширяемости) (ExtBrws.dll), которая отображает все свойства с поздним связыванием для объекта DTE (Development Tools Environment, среда инструментов разработки), корневого объекта в модели расширяемости Visual Studio .NET. Так как некоторые из этих свойств не очень хорошо описаны в документации, отображение их при помощи ExtBrws.dll может оказаться полезным. Если вы считаете себя опытным COM-программистом, можете просмотреть эти свойства, используя программу OLE/COM Object Viewer.

Исправление кода, сгенерированного мастером Add-In Wizard

Если надстройку C# или Visual Basic .NET вы создаете, используя мастер Add-In Wizard среды Visual Studio .NET, который можно найти в диалоговом окне New Project (новый проект) в каталоге Other Projects\Extensibility Projects (другие проекты\проекты расширяемости), сгенерированный им код может требовать некоторых исправлений. В этом разделе я хочу рассказать о том, что сделать сразу после создания скелета надстройки, чтобы облегчить процесс разработки и не обезуметь, решая проблемы в созданном коде. По ходу дела я укажу на ряд важных подробностей работы надстроек.

Нажав кнопку Finish в мастере надстроек, в самую первую очередь надо открыть редактор реестра. Мастер надстроек создает некоторые параметры реестра, которые нужно экспортировать в REG-файл. Путь к нужному разделу реестра начинается или на HKEY_LOCAL_MACHINE, или на HKEY_CURRENT_USER в зависимости от того, указали ли вы, чтобы надстройка была доступна всем пользователям. Оставшаяся часть пути одинакова: \Software\Microsoft\VisualStudio\7.1\AddIns\<имя надстройки>. Сохраните все параметры, относящиеся к этому разделу, который далее я буду называть разделом надстройки.

Изучив содержание раздела, созданного мастером надстроек, вы заметите, что роль некоторых параметров, например, AboutBoxDetails, AboutBoxIcon, FriendlyName и Description, в пояснениях не нуждается. Пара других параметров требует более подробного рассмотрения, так как они очень важны для отладки и разработки надстройки. Первый — CommandPreload — определяет, приказать ли надстройке зарегистрировать команды, которые она, возможно, хочет зарегистрировать. Многие из моих проблем при отладке надстроек были связаны с некорректной регистрацией команд.

Описание CommandPreload в документации, похоже, ошибочно: это не булево поле. Когда CommandPreload имеет значение 1, Visual Studio .NET загружает надстройку для регистрации ее команд; если 2 — полагает, что надстройка уже зарегистрировала свои команды. Если у вас возникли проблемы с выполнением команд надстройки, присвойте CommandPreload значение 1 и перезагрузите IDE, чтобы гарантировать их регистрацию.

Параметр LoadBehavior характеризует загрузку надстройки. Если данное битовое поле равно 0, значит, ваша надстройка не загружается. Значение 1 указывает, что надстройка должна быть загружена при запуске IDE; 4 — что надстройка должна быть загружена при компоновке программы из командной строки. В Visual Studio .NET 2002 была одна проблема: при компоновке из командной строки надстройки загружались всегда, даже если вы указывали не использовать их в таких случаях. К счастью, в Visual Studio .NET 2003 эта ошибка исправлена.

Есть два параметра реестра, которые не создаются мастером надстроек по умолчанию, но их нужно добавить, если вы хотите использовать собственные растровые изображения на панели команд или другие ресурсы Win32. Это параметры SatelliteDllName и SatelliteDllPath. Работать с собственными управляемыми растровыми изображениями и ресурсами в управляемых надстройках было бы весьма удобно, но Visual Studio .NET требует только COM, поэтому вы должны поместить свои ресурсы в DLL ресурсов Microsoft Win32. Как можно догадаться

по названиям, `SatelliteDllName` содержит только имя DLL, а `SatelliteDllPath` содержит путь к сателлитной DLL. В документации к `SatelliteDllPath` говорится, что IDE в конечном счете будет искать DLL по указанному пути (во время предыдущих попыток поиска к указанному пути прибавляются региональные идентификаторы), но не загрузит ее оттуда, и вы не получите никаких ресурсов. Например, если `SatelliteDllPath` содержит путь `C:\FOO\` и вы работаете на компьютере, настроенном на американский вариант английского языка, ваша сателлитная DLL должна находиться в каталоге `C:\FOO\1033`.

Задав сателлитную DLL, вы можете локализовать значения, указываемые в разделе реестра вашей надстройки. Если указанное вами строковое значение состоит из символа `#`, за которым следует число, IDE будет искать это значение в строковой таблице вашей сателлитной DLL. Сателлитные DLL используются обеими надстройками из этой главы: и `SuperSaver`, и `SettingsMaster`.

Нам осталось рассмотреть один странный параметр реестра — `AboutBoxIcon`. Он содержит код значка надстройки, который вы хотите вывести в окне `About`. Как я говорил выше, этот шестнадцатеричный код может быть сгенерирован программой `GenerateIcoData` из состава `Unsupported Tools`. Его нужно скопировать в поле параметра `AboutBoxIcon`, имеющее тип `REG_BINARY`.

Оба проекта — и `SuperSaver`, и `SettingsMaster` — включают файлы `<имя проекта>.ADDIN.REG`, присваивающие нужные значения всем параметрам обеих надстроек. Эти `REG`-файлы позволяют удалять и быстро восстанавливать нужные значения реестра, облегчая установку. Единственный их недостаток в том, что вы должны жестко закодировать значение `SatelliteDllPath`.

Наведя порядок со значениями реестра, нужно заняться исправлением сгенерированного мастером кода. Вероятно, сначала вам захочется изменить атрибут `ProgId`, ассоциированный с созданным классом `Connect`. Мастеру нравится добавлять к имени надстройки слово «`Connect`», что излишне. К сожалению, мастер надстроек во многих местах жестко кодирует имя команды, поэтому, если вы удалите из атрибута `ProgId` слово «`Connect`», измените еще несколько мест:

- раздел надстройки в реестре;
- использование команды в методе `QueryStatus` (файл `CONNECT.CS/VB`);
- использование команды в методе `Exec` (файл `CONNECT.CS/VB`).

Я настоятельно рекомендую создавать для имен команд константы и использовать их везде, где требуются имена. Для своих надстроек я создал файл `RESCONSTANTS.CS/VB`, содержащий все константы для всех команд. Так я исключаю проблемы с опечатками, и, если мне хочется изменить имя команды, сделать это очень просто.

Наверное, самый большой недостаток кода, сгенерированного мастером, в том, что он глотает исключения при регистрации команд и добавлении элементов на панели инструментов. Когда я только начал разрабатывать надстройки, я недоумевал, почему некоторые из моих команд недоступны. Оказалось, они не регистрировались, потому что регистрация генерировала исключение, вызывавшее пропуск оставшейся части функции. Сгенерированный мастером код похож на следующий фрагмент, и это довольно опасно. Выполняя обзоры кода, вы обязательно должны убеждаться, что пустые выражения `catch` представляют собой что-то действительно безопасное.

```
try
{
    Command command = commands.AddNamedCommand (...);
    CommandBar commandBar = (CommandBar)commandBars["Tools"];
    CommandBarControl commandBarControl =
        command.AddControl ( commandBar ,
                             1           );
}
catch(System.Exception /*e*/)
{
}
```

Все действия по созданию команд и панелей инструментов выполняются в надстройках по умолчанию в методе `OnConnection`, когда параметр режима подключения содержит значение `ext_cm_UISetup`. Я всегда выношу создание команд и панелей инструментов в отдельный метод, за пределы `OnConnection`. Между прочим, при режиме подключения `ext_cm_UISetup` ваша надстройка выгружается сразу после возврата из метода `OnConnection`. При режиме подключения `ext_cm_Startup` или `ext_cm_AfterStartup` надстройка перезагружается.

Перед регистрацией своих команд и добавлением командных панелей удаляйте все команды и панели инструментов, которые вы, возможно, уже добавили. Так вы гарантируете, что любые регистрируемые вами для надстройки команды и панели команд создаются «свежими». Удаление добавленных команд и панелей инструментов позволит также безопасно изменять параметры команд или командных панелей и избегать проблем с исключениями, возможных при наличии предыдущих элементов с тем же именем.

Для облегчения разработки надстроек я всегда создаю макрос, удаляющий команды и командные панели, создаваемые моими надстройками. Такой макрос я могу использовать и для уничтожения следов надстройки. Перед запуском макроса, удаляющего команды, ваша надстройка должна быть выгружена. Это значит, что вы должны отключить надстройку в диалоговом окне `Add-In Manager` (диспетчер надстроек), закрыть запущенные копии IDE и удалить раздел надстройки в реестре.

Если методам создания команд и панелей инструментов что-то не нравится, они генерируют исключения. Обязательно помещайте все, что можно, в блоки `try...catch` и сообщайте о причинах исключений, чтобы знать, что происходит. Примеры удаления и установки команд и командных панелей вы увидите в коде надстроек `SuperSaver` и `SettingsMaster`.

Решение проблем с кнопками панелей инструментов

Исправив код, сгенерированный мастером надстроек, вы, вероятно, столкнетесь с проблемой правильного показа растровых изображений на панелях инструментов. Решить ее нетрудно, но это не описано в документации. Поиск заклинаний потребовал от меня некоторых усилий, поэтому я надеюсь, что это обсуждение поможет вам сэкономить время и сберечь нервы.

Для загрузки на панель инструментов собственные растровые изображения нужно разместить в сателлитной DLL Win32; встроенные управляемые растровые изображения на панелях инструментов использовать нельзя. Создавая команду

методом `Commands.AddNamedCommand`, вы должны передать ему значение `false` в параметре `MSOButton` и идентификатор ресурса растрового изображения в сателлитной DLL в параметре `Bitmap`.

Самые неприятные проблемы с размещением собственных растровых изображений на панелях инструментов — сами изображения! Во-первых, поддерживаются только 16-цветные изображения. Если вы находите свое изображение странным, значит, в нем используется больше цветов. Вторая проблема состоит в получении правильной маски.

При взгляде на растровые изображения в надстройке `RegExplorer` мне показалось, что в них в качестве цвета маски применяется зеленый. Получить правильную маску очень важно, так как именно благодаря этому ваше изображение может казаться трехмерным при наведении на него курсора. При создании изображений для своих кнопок я также использовал в качестве маски зеленый цвет. Но, когда я загрузил свою надстройку, маска определенно не работала, и все места, которые я хотел сделать прозрачными, имели безобразный ярко-зеленый цвет. В ходе расследования я выяснил, что на самом деле в качестве маски нужно было использовать не истинный зеленый, а цвет с RGB-значением 0, 254, 0 (зеленому соответствует 0, 255, 0).

Однако даже после изменения в палитре значения зеленого цвета на 0, 254, 0 маска осталась зеленой. Оказалось, что я использовал устаревший графический редактор, который так хотел мне «помочь», что «исправлял» палитру, отображая зеленый как 0, 255, 0, а не так, как я хотел. Тогда я с помощью редактора растровых изображений `Visual Studio .NET` переназначил один из цветов палитры (я всегда устанавливаю вместо розового, принятого по умолчанию, цвет 0, 254, 0), и все получилось. Помните: когда вы откроете растровое изображение в редакторе `Visual Studio .NET`, он изменит зеленый цвет в палитре на 0, 254, 0, потому что это ближайший цвет к зеленому. Это значит, что, если вы хотите использовать в своем изображении истинный зеленый цвет, вам придется изменить значение другого элемента на 0, 255, 0.

Исправив цвет маски, следует обновить и свой код, чтобы гарантировать, что ваши панели инструментов не будут отличаться от других. Кнопки панелей инструментов, добавляемые к объекту `CommandBar`, по умолчанию отображаются как кнопки с текстом. Чтобы они отображались как стандартные, нужно вручную перебрать все элементы `CommandBarControl` объекта `CommandBar` и присвоить им стиль `MsoButtonStyle.msoButtonIcon`. Вот как я сделал это в `SuperSaver`:

```
foreach ( CommandBarControl ctl in
                                SuperSaverCmdBar.Controls )
{
    if ( ctl is CommandBarButton )
    {
        CommandBarButton btn = (CommandBarButton)ctl ;
        btn.Style = MsoButtonStyle.msoButtonIcon ;
    }
}
```

Создание окон инструментов

Почти все надстройки, добавляющие команды, имеют и панель инструментов с растровыми изображениями, но иногда желательно добавить в Visual Studio .NET полноценный пользовательский интерфейс. Отобразить из надстройки управляемое диалоговое окно не сложнее, чем из приложения Windows Forms. Для отображения полного окна, называемого окном инструментов, требуется чуть больше работы.

В IDE Visual Studio .NET два типа окон: документов и инструментов: в окнах документов вы редактируете код. Все остальные — это окна инструментов (например, приведу окна Task List, Solution Explorer и Toolbox). Окно инструментов может быть стыкуемым или, если вы щелкнете правой кнопкой его заголовок и измените параметр Dockable, оно может отображаться как полное окно в основной области редактирования.

Так как все окна инструментов являются COM-объектами, их можно создавать на C++ и бороться со всеми неприятностями, которые за этим следуют. Создание окон инструментов на управляемом коде в документации не описано, но зато в число предоставляемых Microsoft примеров входит подобный проект, грамотно названный ToolWindow.

Суть создания управляемого окна инструментов в том, чтобы ваша управляемая надстройка создала компонент ActiveX, который в свою очередь обеспечивает работу CLR. Как только это сделано, можно приказать компоненту ActiveX загрузить и отобразить в окне ActiveX нужный элемент управления. Этот компонент ActiveX иногда называют элементом управления «хост-прослойка» (host shim control), так как он просто внедряется в выполнение управляемого кода, чтобы вы могли сделать то, что вам нужно.

Все звучит так, как будто этот «хост-прослойку» написать очень сложно, но есть хорошая новость: он содержится в примере ToolWindow, и вы можете его использовать. Увы, он почти не проверяет ошибок, поэтому, если что-то потерпит крах, вам останется только чесать голову. А теперь самое приятное: я проработал весь код этого элемента, реализовал проверку ряда ошибок и добавил диагностические выражения, чтобы вы знали, что происходит при его использовании.

Я переименовал свой «хост-прослойку» в VSNetToolHostShim и включил в пример SimpleToolWindow на CD. Все, что делает SimpleToolWindow, заключается в добавлении в IDE Visual Studio .NET окна редактирования а la окно WinDBG. Так как создание элемента управления «хост-прослойка» контролируется вашей управляемой надстройкой, вы можете использовать VSNetToolHostShim из любого своего проекта окна инструментов.

Объяснить необходимые действия проще всего на примере обработчика `OnConnection` из проекта SimpleToolWindow. Вы должны создать окно инструментов с элементом управления VSNetToolWinShim, который возвращает ссылку на элемент управления VSNetToolHostShim. Используя возвращенный объект VSNetToolHostShim, вызовите метод `HostUserControl12`, чтобы загрузить свой управляемый элемент и создать кнопку для открытия окна инструментов. Все это в действии можно увидеть в листинге 9-3.

Листинг 9-3. Использование VSNetToolHostShim

```

public void OnConnection ( object          application ,
                           ext_ConnectMode connectMode ,
                           object          addInInst ,
                           ref System.Array custom      )
{
    try
    {
        ApplicationObject = (_DTE)application;
        AddInInstance = (AddIn)addInInst;

        // Ваше окно инструментов должно иметь уникальный GUID.
        String guid = "{E16579A4-5E96-4d84-8905-566988322B37}";

        // Объект для получения элемента VSNetToolHostShim.
        Object RefObj = null ;

        // Создание основного окна инструментов
        // путем загрузки "хоста-прослойки".
        TheToolWindow = ApplicationObject.Windows.
            CreateToolWindow ( AddInInstance
                              ,
                              "VSNetToolHostShim.VSNetToolWinShim",
                              "Scratch Pad Window"
                              ,
                              guid
                              ,
                              ref RefObj
                              );

        // До вызова метода HostUserControl нужно сделать
        // окно видимым, иначе все пойдет не по плану.
        TheToolWindow.Visible = true ;

        // Получение "прослойки"(это переменная уровня класса):
        // private VSNetToolHostShimLib.IVSNetToolWinShim ShimObj ;
        ShimObj = (VSNetToolHostShimLib.VSNetToolWinShimClass)
            RefObj ;

        // Получение данной сборки. Это нужно, чтобы я мог
        // передать "прослойке" расположение надстройки.
        System.Reflection.Assembly CurrAsm =
            System.Reflection.Assembly.GetExecutingAssembly ( ) ;

        // Получение каталога данной надстройки и присоединение к пути
        // имени DLL ресурсов. Это нужно для загрузки кнопки-ярлычка.
        StringBuilder StrSatDll = new StringBuilder ( ) ;

        String StrTemp = CurrAsm.Location.ToLower ( ) ;
        int iPos = StrTemp.IndexOf ("simpletoolwindow.dll" ) ;
        StrSatDll.Append ( CurrAsm.Location.Substring ( 0 , iPos ));
        StrSatDll.Append ("SimpleToolWindowResources.DLL" ) ;
    }
}

```

```
// Этот метод загружает управляемый элемент в элемент управления
// ActiveX и приказывает ему загрузить растровое изображение.
ShimObj.HostUserControl2 ( TheToolWindow
                           CurrAsm.Location
                           ,
                           "SimpleToolWindow.ScratchPadControl" ,
                           StrSatDll.ToString ( )
                           ,
                           1
                           );
}
catch ( System.Exception eEx )
{
    MessageBox.Show ( eEx.Message + "\r\n" +
                      eEx.StackTrace.ToString ( ) ,
                      "ExceptBion in OnConnection" );
}
}
```

Создание на управляемом коде страниц свойств окна Options

Создать управляемое окно инструментов относительно легко. Разработать управляемую страницу свойств, отображаемую в диалоговом окне Options (рис. 9-5), немного сложнее. Это важно потому, что именно в окне Options пользователи обычно будут искать страницу изменения параметров вашей надстройки, и так вы сможете улучшить свою репутацию.

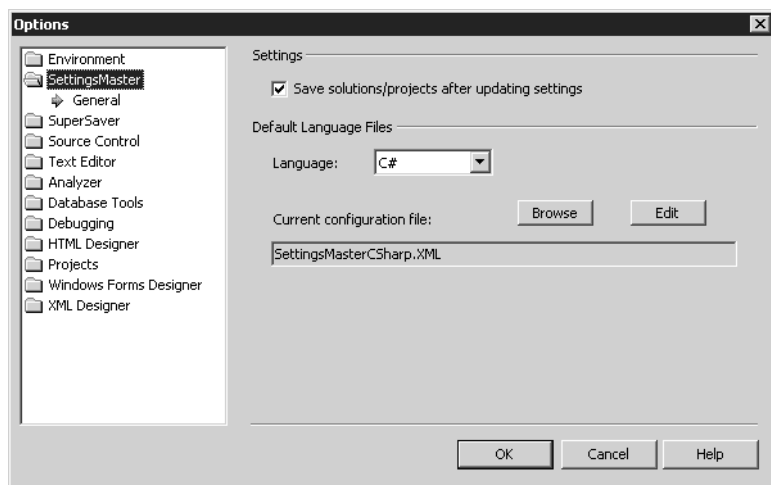


Рис. 9-5. Страница свойств надстройки SettingsMaster в окне Options

Как вы, наверное, уже догадались, страница свойств в окне Options представляет собой элемент управления ActiveX, реализующий интерфейс `IDTToolsOptionsPage`. Чтобы узнать, есть ли у вас такая страница свойств, Visual Studio .NET изучает раздел надстройки в реестре. В основном разделе надстройки она ищет раздел `Options`. В разделе `Options` будут находиться один или больше разделов, которые будут добавлены в дерево окна Options как узлы верхнего уровня. У вас будет один раздел

этого уровня, названный так же, как и надстройка. В этом разделе будет находиться очередной набор разделов, формирующих подузлы верхнего узла дерева. По умолчанию первый раздел называется `General`. В каждом заключительном разделе будет находиться строковый параметр `Control`, содержащий `ProgID` элемента управления ActiveX, создаваемого для отображения страницы свойств.

Наверное, лучше всего проиллюстрировать сказанное на примере. Разделы реестра для страницы свойств `SettingsMaster` (рис. 9-5) выглядят так:

```
HKEY_CURRENT_USER\  
  Software\  
    Microsoft\  
      VisualStudio\  
        7.1\  
          AddIns\  
            SettingsMaster\      <- Раздел надстройки  
              Options\          <- Раздел Options  
                SettingsMaster\  
                  General      <- Корневой узел в окне Options  
                                <- Подузел узла SettingsMaster
```

Параметр в разделе `General`:

```
Control REG_SZ SettingsMasterShim.SettingsMasterOption
```

Создание страниц свойств и управление ими контролируется диалоговым окном `Options`, а не вашей надстройкой; когда дело касается разработки собственных страниц свойств на управляемом коде, это представляет небольшую проблему. Дело в том, что в каждом конкретном случае запускается элемент управления ActiveX, указанный в строке `Control`. Это значит, что созданный элемент управления ActiveX должен иметь априорные знания об управляемом элементе, который вы хотите отобразить. Я прямо ломал голову над этим, когда на мой стол попал февральский номер «MSDN Magazine» за 2002 год, в котором Лео Нотенбум (Leo Notenboom) описал великолепное решение проблемы.

Лео предлагает, чтобы всю работу выполнял «элемент-прослойка» ActiveX, написанный на C++. Так как в отличие от окон инструментов создать общий элемент управления ActiveX для страниц свойств в окне `Options` нельзя, вы должны будете создавать новый элемент управления для каждого проекта. К счастью, для этого нужно только взять код Лео, изменить GUID и имя элемента управления в файлах `.RGS` и изменить GUID загружаемого элемента управления в коде C++. Если хотите полностью разобраться в решении Лео, прочитайте его прекрасную статью.

Чтобы облегчить поиск проблем, я добавил в код Лео несколько диагностических выражений и обработчиков ошибок. Если вы решите использовать проекты `SuperSaverOptionsShim` или `SettingsMasterShim` с CD, найдите в главных CPP-файлах строки `k_HOSTCLSID` и замените в них значение GUID на GUID вашей конкретной страницы свойств. Естественно, нужно изменить и имя элемента управления и его GUID в файлах `.RGS`.

Когда я отобразил свои страницы свойств, мне показалось, что все отлично. Но, загрузив надстройку на ноутбуке и взглянув на страницу свойств в окне `Options`, я понял, что что-то не так: моя страница не была похожа на остальные страницы свойств. Чтобы диалоговые окна и окна инструментов выглядели лучше на экране моего ноутбука (который имеет совершенно безумное разрешение), я прибег-

нул к одной небольшой хитрости и, открыв папку Environment (среда) и выбрав узел Fonts And Colors (шрифты и цвета), изменил шрифт диалоговых окон и окон инструментов (Dialogs And Tool Windows). Элементы управления, разработанные на управляемом коде, используют по умолчанию фиксированный шрифт Microsoft Sans Serif размером 8,25 точки, поэтому я не мог узнать корректный шрифт у хостов этих элементов, а должен был искать определенный для хоста шрифт сам.

Открыв проект SuperSaver, вы увидите элемент управления OptionPropPageBase.CS. Это базовый класс, который определяет текущий тип и размер шрифта диалогового окна и применяет эти параметры для всех элементов управления на странице. Вам может показаться, что получение типа и размера шрифта — тривиальная задача, но подобные свойства позднего связывания большей частью не документированы. Вот почему упомянутый выше Extensibility Browser из пакета Unsupported Tools иногда может оказаться крайне полезным. Как только я узнал нужные заклинания, все оказалось просто. Вот код метода OptionPropPageBase.OnAfterCreated, который получает нужный шрифт, создает его и настраивает диалоговое окно и все элементы управления (листинг 9-4):

Листинг 9-4. Получение и установка шрифтов для страницы свойств в окне Options

```
public virtual void OnAfterCreated ( DTE DTEObject )
{
    // Чтобы гарантировать правильное отображение этой страницы свойств,
    // я должен выбрать в качестве всех шрифтов тот шрифт, который выбран
    // пользователем в пункте Dialog and Tool Windows. Чтобы получить
    // значения из свойств DTE, я использую позднее связывание.
    Properties Props = DTEObject.getProperties ( "FontsAndColors",
                                                "Dialogs and Tool Windows" );

    String FntName = (String)Props.Item ( "FontFamily" ).Value ;

    Object ObjTemp = Props.Item ( "FontSize" ).Value ;
    Int32 FntSize = Convert.ToInt32 ( ObjTemp ) ;

    // Создание шрифта.
    Font DlgFont = new Font ( FntName
                             , FntSize
                             , GraphicsUnit.Point ) ;

    // Установка шрифта для диалогового окна.
    this.Font = DlgFont ;

    // Перебор всех элементов управления в диалоговом окне и установка
    // их шрифтов. Некоторые элементы будут использовать шрифт, заданный
    // чуть выше, но не все, поэтому мне нужно сделать это вручную.
    foreach ( Control Ctl in this.Controls )
    {
        Ctl.Font = DlgFont ;
    }
}
```


Конечно, задать шрифт для завершения работы недостаточно. Хотя элементы управления типа «метка» (label control) могут настраивать свои размеры автоматически, большинство элементов управления на это неспособно, поэтому вам нужно будет просмотреть их и увеличить размеры. Чтобы узнать, как я реализовал настройку размеров для конкретного диалогового окна, изучите мой метод `SuperSaverOptions.OnAfterCreated`.

Возможно, вас интересует, почему я не реализовал изменение шрифтов диалоговых окон «на лету» после того, как кто-нибудь это запросит. Хочу вас обрадовать: шрифты диалоговых окон можно изменить путем простого перезапуска IDE. Интересно, что Visual Studio .NET позволяет изменять «на лету» все шрифты, кроме шрифтов диалоговых окон.

Мой крутой класс из файла `OptionPropPageBase.CS` позаботится за вас кое о чем, но из-за ошибки в Visual Studio .NET его очень сложно использовать. Если вы унаследуете свой элемент управления от `OptionPropPageBase`, IDE не будет открывать его в режиме разработки (design mode), а будет рассматривать его как обычный текстовый файл. Чтобы Visual Studio .NET загружала элемент управления в режиме разработки, позволяя редактировать его при помощи дизайнера, в качестве базового для него нужно временно указывать класс `System.Windows.Forms.UserControl`. Надеюсь, Microsoft исправит эту проблему в пакете обновлений или в будущей версии Visual Studio.

Стандартный вопрос отладки

Моя сборка загружается только из моей надстройки.

Должен ли я устанавливать ее в глобальный кэш сборки (global assembly cache, GAC)?

GAC — специальное место, и вам не следует устанавливать в него что-либо без крайней нужды. К счастью, разработчики IDE Visual Studio .NET предусмотрели в каталоге <каталог установки VS.NET>\Common7\IDE два подкаталога для сборок только надстроек или макросов: `PublicAssemblies` и `PrivateAssemblies`. Если вы хотите, чтобы код в вашей сборке мог вызывать другие надстройки или макросы, поместите сборку в каталог `PublicAssemblies`. А чтобы сборка вызывалась только вашей надстройкой, сохраните ее в каталоге `PrivateAssemblies`.

Стандартный вопрос отладки

Есть ли более простые способы отладки надстроек, если уж они могут загружаться в IDE, используемую для отладки?

Отладка надстройки может быть очень сложной, так как для правильного конфигурирования экземпляров IDE, которые вы хотите использовать для отладки, нужно удалить раздел надстройки в реестре, открыть проект надстройки в целевой IDE и восстановить раздел надстройки. Это не очень трудно, но тут легко запутаться. Кроме того, если нужно скомпилировать надстройку после исправления ошибки и надстройка загружается IDE, ваша компоновка никогда не будет работать.

На помощь приходит недокументированный ключ командной строки `/rootsuffix`. Он приказывает Visual Studio .NET добавить суффикс к нормальному имени раздела реестра и загрузить все пакеты, надстройки и параметры из этого раздела реестра, а не из раздела по умолчанию. Это похоже на наличие двух отдельных Visual Studio .NET на одном компьютере.

Для этого вы должны прежде всего запустить REGEDIT.EXE и найти раздел `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1`. Выбрав этот раздел, выберите Export в меню File и сохраните разделы реестра в файл. Сохранив файл, откройте его в NOTEPAD.EXE и замените все выражения `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1` на `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\VisualStudio\7.1NoAddIns`. Заметьте, что к названию раздела надо добавить суффикс `NoAddIns`. Заменяв все выражения, сохраните файл. Снова запустите REGEDIT.EXE, выберите Import в меню File и импортируйте измененный файл в реестр. Если хотите сделать то же самое со своими пользовательскими параметрами (кроме, конечно, любых надстроек!), выполните указанные действия с разделом `HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\7.1`.

Чтобы Visual Studio .NET использовала раздел с суффиксом `NoAddIns`, просто запустите ее:

```
devenv /rootsuffix NoAddIns
```

Так вы получите загруженную копию Visual Studio .NET, в которой сможете делать с кодом своей надстройки что угодно без проблем.

Надстройка SuperSaver

Вы получили некоторое представление о надстройках, и я думаю, что сейчас лучше всего обсудить реальные надстройки. Первую созданную мной надстройку — SuperSaver — я представил в колонке Bugslayer в «MSDN Magazine». Однако ее версия в этой книге радикально отличается от того варианта — прекрасный пример всех проб и ошибок, связанных с созданием надстроек!

SuperSaver решает две проблемы: добавляет в IDE Visual Studio .NET функцию автоматического фоновое сохранения, чтобы вы не лишились измененного документа, а при сохранении файлов удаляет пустые места в концах строк. В случае файлов C++ и C# IDE Visual Studio .NET, противореча компьютерным богам, оставляет в концах строк пустое пространство. Чтобы вернуть в мир порядок, я просто обязан был исправить это недоразумение.

Реализация фоновое сохранения файлов оказалась чуть ли не слишком легкой. Я устанавливаю фоновый таймер и при его срабатывании выполняю команду, ассоциированную с пунктом Save All (сохранить все) меню File, или `File.SaveAll`. При удалении пустых мест возникли проблемы.

Работая над сохранением активного элемента, я заметил, что на самом деле выполняется команда `File.SaveSelectedItems`. Интересно, что вместо сохранения только редактируемого в данный момент документа она сохраняет также проект и все элементы, выбранные в Solution Explorer (для выбора нескольких элементов нужно щелкнуть их, удерживая клавишу Ctrl). Мне хотелось, чтобы моя

команда была полной заменой `File.SaveSelectedItems`, поэтому я должен был симитировать это поведение. Сначала я последовал такому алгоритму:

```
получить набор DTE.SelectedItems
для каждого элемента из числа выбранных элементов
    если это документ и он не сохранен,
        получить текстовый документ
        вызвать для текстового документа метод ReplacePattern
    конец блока если
    вызвать для данного элемента метод Save
следующая итерация цикла
```

Хотя алгоритм казался совсем простым, при его реализации я столкнулся с массой проблем. Пытаясь их решить, я разработал в итоге семь абсолютно разных версий SuperSaver. Первая проблема заключалась в самих проектах. Не все типы проектов поддерживают метод `Save`. Наглядный пример — проекты Setup. Это значит, что у меня не было гарантированной возможности сохранения всех проектов. Я попытался разработать специальные случаи вызова метода `Save`, но это не позволило бы правильно обрабатывать все будущие типы проектов, которые также могут не поддерживать `Save`.

Вторая проблема была связана с ошибкой в объекте `TextDocument`. Я вызывал метод `ReplacePattern` с регулярным выражением «`[\t]+$`» (без кавычек) в качестве шаблона поиска и пустой строкой в качестве текста замены. Все бы прекрасно, но это выражение также очищало все пустые строки, оставленные просто ради пропуска. Поэтому, если я сохранял активный файл, содержащий 20-символьный отступ, сделанный при помощи символов табуляции или пробелов, автоматические отступы удалялись, и мне пришлось бы восстанавливать их. Клавиша табуляции мне нравится не меньше, чем вам, но я решил, что гораздо лучше исправить это, чтобы SuperSaver удалял пустые места только в тех строках, которые на самом деле содержат текст.

После многих часов борьбы с регулярными выражениями я обнаружил нужное выражение поиска — «`[^ \t]+{[\t]*$}`», а также выражение замены — «`\1`». Замена с применением подвыражений — одно из самых больших достоинств регулярных выражений. Если вы не считаете себя гуру в регулярных выражениях, то это выражение поиска приказывает искать любые выражения, соответствующие первой группе (ограниченной первой парой фигурных скобок), т. е. содержащие любые символы, кроме пробела или табуляции. Это гарантирует, что соответствие будет найдено только в строках, содержащих символы. Вторая группа (во вторых фигурных скобках) ищет один или более пробелов или символов табуляции в конце строки. Соответствие этим двум группам будет найдено только в строках, содержащих после себя пустое пространство. Строка замены «`\1`» говорит анализатору регулярных выражений заменить найденное выражение текстом, соответствующим первой группе. В данном случае это символы в конце строки без пустого пространства. Так что пустое место из строки удаляется.

Я изменил параметр метода `TextDocument.ReplacePattern` на мое новое выражение поиска и испытал сохранение в действии. Результатом стало то, что любые строки, завершавшиеся лишними пробелами, теперь заканчивались на завершающий текст плюс символы «`\1`», заменяющие лишние пробелы! Это было очень

интересно, но искажение кода, вероятно, не сделало бы SuperSaver очень полезным. Оказалось, что в методе `TextDocument.ReplacePattern` допущена ошибка, нарушающая подстановку подвыражений.

Если я не мог заставить работать подстановку подвыражений регулярных выражений, надстройка SuperSaver была бы менее полезной. Рассматривая обходные пути решения этой проблемы, я записал макрос поиска и замены, который использует глобальный объект `Find` и работает с подстановкой подвыражений. Этот способ не так чист, как применение метода `ReplacePattern`, но он хоть стал началом решения.

Объект `Find` помог мне быстро понять, что, так как это глобальный объект, любые его изменения во время подстановок подвыражений регулярных выражений в коде SuperSaver приводят к изменению текущих значений, оставленных пользователем в диалоговом окне `Find`. Поэтому я создал для `Find` оболочку `SafeFindObject`, позволяющую сохранять и восстанавливать все значения, исключая потерю параметров пользователей.

Дополнительная проблема с глобальным объектом `Find` состояла в том, что, когда активным документом был дизайнер Windows Forms, я мог получить для документа объект `TextDocument`. Однако метод `Find.Execute`, выполняющий фактический поиск и замену, генерировал исключение. Разрабатывая `SafeFindObject`, я решил, что ничего не могу с этим поделать, кроме как «глотать» все исключения, генерируемые при вызове `Find.Execute`.

Все шло своим чередом, когда я по-настоящему захотел, чтобы пустые места удалялись и при автосохранении. Проведя основную работу по очистке и сохранению активного документа, я думал, что все будет ОК. Увы, если я приказывал `Find` продемонстрировать свои чудесные способности на всех файлах и при этом были открыты какие-нибудь файлы, допускавшие только чтение, надстройка не работала. Поэтому я просто реализовал перебор всех открытых файлов, и, если некоторые были отмечены как только для чтения, я не вызывал `SafeFindObject` для их сохранения с удалением пустых мест.

Кроме трудностей с автосохранением, я заметил проблему с «новыми» файлами, т. е. несохраненными файлами, созданными командой `New` меню `File`. Вызов метода `Save` на этих файлах приводил к появлению диалогового окна `Save File As` (сохранить как). Я думал, что смогу найти внутри метода `Save` блокирующий вызов, но его не оказалось, поэтому, поработав с IDE, я в итоге получал кучу окон `Save File As`, настраивая не улучшавших.

Тут я стал немного одержим идеей доведения автосохранения файлов с удалением пустых мест до логического завершения. Чтобы следить за ходом обсуждения, откройте файл `TrimAndSave.CS` и перейдите к методу `TrimAndSave.SaveAll`. Посмотрите закомментированный блок — «Original Attempt» (первая попытка). Сейчас я расскажу, почему этот код не работает.

Так как я работал с текущими файлами при помощи объекта `Find`, я подумал, что могу перевести все требующие сохранения файлы в активное состояние и быстро их сохранить. Это казалось разумным, пока я не столкнулся с огромной проблемой. Если я работал с документом Windows Forms, открытым в окнах проектирования (design view) и кода (code view), вызов метода `Active` всегда активизировал окно проектирования, даже когда активным было окно кода. Это означа-

ло, что, когда я набирал код для формы Windows, автосохранение иногда предъявляло свои права и передо мной появлялось окно проектирования. Удивительно, но в модели автоматизации нет способа активизации окна кода из документа.

Мне нужен был способ поиска заголовка активного документа, т. е. окна с активной вкладкой. Для этого можно вызвать `DTE.ActiveWindow`, но так вы получите окно, которое в данный момент имеет фокус во всех окнах IDE, а не окно, в котором вы выполняете редактирование или проектирование. После многих проб я увидел, что командная панель Window всегда содержит активный заголовок документа в пункте меню, начинающегося с «&1» (амперсанд показывает, что данный элемент меню должен быть подчеркнут). Использовать меню для получения заголовка действительного активного и впрямь глупо, но сделать это в модели автоматизации иначе нельзя. Вооруженный заголовком активного документа и значением, возвращенным `DTE.ActiveWindow`, я наконец мог обдумать реализацию сохранения, так как теперь я хоть мог восстанавливать текущее активное окно документа и окно с действительным фокусом.

Просматривая в цикле документы, я должен был сделать пару вещей и только потом приступить к сохранению файла. Прежде всего я по первому символу файла должен был определить, является ли он новым. Если бы первый символ был тильдой (~), это значило бы, что файл был создан, но никогда не сохранялся. Так как я хотел, чтобы метод `TrimAndSave.SaveAll` выполнял реальное автосохранение (которое, например, прекрасно реализовано в редакторе Visual SlickEdit компании SlickEdit), я мог сделать так, чтобы `TrimAndSave.SaveAll` не сохранял новые файлы или файлы только для чтения. Это позволило бы мне избежать затопления экрана окнами Save File As при каждом автосохранении. Я мог указать в диалоговом окне параметров SuperSaver, что хочу пропускать новые файлы и файлы только для чтения. Если бы файл был новым или допускал только чтение, метод `TrimAndSave.SaveAll` пропускал бы его и переходил к следующему файлу.

После определения, что документ нуждается в сохранении, мне следовало перевести документ в активное состояние, чтобы над ним мог поработать объект `DTE.Find`. Так как я должен был гарантировать, что окно редактирования файла активно, мне нужно было найти заголовок окна, соответствующий имени документа. Если я находил такое окно, я мог наконец удалить пустые места из строк. Если я не находил текстового окна, я просто переходил к сохранению файла.

Для новых файлов я создал собственное окно Save File As, что совсем нетрудно. Если файл уже имеет имя, я могу просто вызвать метод `Document.Save`. Интересно, что метод `Document.Save` — классический пример того, как не надо выполнять обработку исключений. Если файл допускает только чтение, `Document.Save` выведет диалоговое окно, спрашивающее, хотите ли вы перезаписать файл или сохранить его с новым именем. Если вы нажмете Cancel для отмены сохранения файла, `Document.Save` сгенерирует исключение класса `COMException`, которое сообщит «User pressed escape out of save dialog» (пользователь вышел из диалога сохранения). Это вполне нормальное взаимодействие с пользователем, поэтому об этом следовало бы сообщать через возвращаемое значение.

После обработки всех документов я мог заняться восстановлением исходного окна активного документа, а также самого активного окна. После восстановления окон я мог перейти к сохранению проектов и решения.

Что касается проектов, требующих сохранения, то тут в первую очередь надо определить, допускает ли проект только чтение. Объект `Project` не имеет свойства, которое может сообщить вам это. Поэтому я должен получить и проверить атрибуты файла проекта. Если проект допускает только чтение и пользователь не хочет, чтобы ему предлагали сохранить проект при автосохранении, я не сохраняю проект.

Если проект должен быть сохранен, дело становится интереснее. Как я упоминал в разделе «Проблемы с проектами», объектная модель проектов в Visual Studio .NET не очень хорошо продумана и плохо документирована. Так как объект `Project` далеко не во всем соответствует другим типам проектов, особенно `VCProject`, я пытаюсь получить из проекта объект `VCProject`, проверяя сначала язык проекта. Если проект разрабатывается на C++, я вызываю `Project.Object` и привожу возвращаемое значение к типу `VCProject`. Вооружившись `VCProject`, я могу уверенно вызывать `VCProject.Save`. Если проект имеет какой-то другой тип, а не `VCProject`, я пытаюсь сначала вызвать `Save` и, если вызов `Save` генерирует исключение, вызываю `SaveAs`, передавая в каждом случае полное имя проекта. Так как Microsoft не включила в документацию полное описание разных типов проектов, это лучшее, что я могу сделать для их сохранения.

Разобравшись с проектами, я могу сохранить решение, если это требуется. Как и в случае с проектами и документами, если решение допускает только чтение и пользователь не хочет, чтобы его беспокоили окнами `Save File As`, я не сохраняю решение.

Я думал, что после этого получу рабочую надстройку, но тестирование быстро меня в этом разубедило. Протестировав автосохранение с включенной функцией удаления пустых мест, я был огорчен масштабами моргания, связанного с переводом текстовых окон в активное состояние. Я вспомнил, что объект `DTE` поддерживает свойство `SuppressUI`, которое при установке в `true` блокирует пользовательский интерфейс во время выполнения кода автоматизации. Поняв, что `SuppressUI` решит проблемы с морганием панели задач, я присвоил ему `true` в начале метода `TrimAndSave.SaveAll`. Увы, безрезультатно: мерцание продолжалось как ни в чем не бывало.

Конечно, я мог бы с этим смириться, но была еще одна проблема, связанная с методом `Window.Active`: он пытался перевести в активное состояние всю IDE, а не просто конкретное окно документа. Кроме того, если IDE была минимизирована, `Window.Active` восстанавливал окно.

Дальше — больше: применение объекта `Find` при фоновом сохранении, выполняемом в другом потоке из-за применения таймера, по-видимому, изменяло его состояние. Вызов `SuperSaver.SuperSaverSave`, которому я назначил комбинацию `Ctrl+S`, работал прекрасно. Однако после фонового сохранения при каждом вызове `SuperSaver.SuperSaverSave` начало появляться информационное окно `Find/Replace`, которое иногда выводится при работе с окном `Find`. Я не хотел заставлять вас отключать информационные окна `Find/Replace` для использования `SuperSaver`, но мне пришлось пойти на это ограничение. Окно `Find/Replace` можно отключить, открыв окно `Options`, папку `Environment`, страницу свойств `Documents` и убрав флажок возле пункта `Show message boxes` (показывать информационные окна). После отключения окон `Find/Replace` я при каждом выполнении `SuperSaver.SuperSaverSave` стал слышать стандартный сигнал, который слышите вы, работая с окном `Find`.

К этому моменту я испытал сильное разочарование, но решил не сдаваться. К счастью, в последней попытке, хотя и далекой от совершенства, я получил большую часть того, что хотел. Так как я должен был использовать объект `Find` с параметром `vsFindTarget.vsFindTargetOpenDocuments`, который ограничивает поиск и замену только открытыми документами, мне нужно было быть очень внимательным. Я мог безопасно удалить пустые места при фоновой обработке, только если в числе активных документов не было файлов только для чтения и новых файлов. Я по-настоящему хотел реализовать фоновое сохранение с удалением пустых мест из всех файлов, но это лучшее, что я мог сделать. Для обработки самого сохранения у меня был единственный вариант: вызывать реальный `File.SaveAll`. Так как я все еще хотел реализовать отмену появления диалоговых окон `Save File As` и информационных окон, предупреждающих о перезаписи файлов, я не вызываю `File.SaveAll`, если пользователь отменил в `SuperSaver` пункт `Save New And Read-only Files When Auto Saving` (сохранять при автосохранении новые файлы и файлы только для чтения); в активных документах нет файлов только для чтения, нуждающихся в сохранении, или новых файлов.

Конечно, несмотря на относительную простоту только что описанного алгоритма, я не мог не столкнуться с еще одной ошибкой. Свойство `Document.ReadOnly`, которое будто должно возвращать `true`, если файл допускает только чтение, не работает. Мне пришлось вручную проверять права доступа к файлам методом `File.GetAttributes`.

Наконец, я реализовал в надстройке команды `SuperSaver.SuperSaverSaveAll` и `SuperSaver.SuperSaverSave`, которые, по-моему, работали вполне хорошо. Я приступил к созданию для них командной панели инструментов и столкнулся с проблемами маски растровых изображений, рассмотренными выше. После их решения передо мной возникла последняя проблема с `SuperSaver`.

Так как я собирался написать команды для замены методов `File.SaveSelectedItems` и `File.SaveAll`, я хотел убедиться, что кнопки моей панели инструментов ведут себя, как кнопки реальных панелей. После серии экспериментов я заметил, что только кнопка `File.SaveSelectedItems` становилась серой в отключенном состоянии. Я перепробовал все способы заставить кнопку `SuperSaver.SuperSaverSave` вести себя аналогично. Активное состояние определялось элементами, выбранными в текущем решении и проекте, поэтому я никак не мог найти заклинания для проверки выполнения метода `File.SaveSelectedItems`, нужной для блокирования/разблокирования его кнопки.

Я уже собрался сдаться, но тут меня осенило, что незачем все так усложнять. Мне нужно было только получить командный объект `File.SaveSelectedItems` и проверить, истинно ли свойство `IsAvailable`; если да, кнопка панели инструментов активна. Соответственно, если команда `File.SaveSelectedItems` неактивна, я возвращаю в методе `IDTCommandTarget.QueryStatus` значение `vsCommandStatus.vsCommandStatusUnsupported`, и с моими кнопками и целым миром все в норме.

Разрабатывать `SuperSaver` было очень сложно, но мне это удалось, и я доволен. Я не только узнал многие слабости надстроек и модели автоматизации IDE Visual Studio .NET, но и осчастливил богов программирования, уничтожив пробелы в концах строк. В комментариях к `SuperSaver` я оставил все работоспособные алгоритмы, чтобы вы могли реализовать команды заново, используя будущие версии Visual Studio .NET с исправленными проблемами автоматизации.

Настройка SettingsMaster

После всех удовольствий с SuperSaver мне совсем не хотелось возиться с другой надстройкой для этой главы. Однако SettingsMaster оказалось не только легко разработать — эта надстройка стала одним из самых полезных средств из всех, когда-либо мной созданных. Надеюсь, вам она тоже пригодится.

Как говорит название, SettingsMaster должна правильно настраивать параметры, т. е. все ваши параметры сборки проектов. Многие из ошибок, с которыми я сталкиваюсь годами, связаны с проблемами сборки проектов, поэтому я хотел создать единственное средство, гарантирующее правильную настройку всего проекта. Кроме того, Visual Studio .NET довольно плохо поддерживает создание приложений группами разработчиков; единственный способ настройки параметров нескольких проектов нескольких программистов — установить их вручную. Это огромный недостаток Visual Studio .NET. Я хотел решить эти две проблемы и для проектов .NET, и для проектов на неуправляемом C++.

SettingsMaster добавляет в IDE две команды: SettingsMaster.CorrectCurrentSolution использует для автоматического применения желаемых вами параметров конфигурационный файл по умолчанию (я еще расскажу о нем), а SettingsMaster.CustomProjectUpdate выводит диалоговое окно Open File (открыть файл), предлагая выбрать конфигурационный файл для обновления проекта, выбранного в окне Solution Explorer.

Идея SettingsMaster заключается в хранении общих параметров группы в общих файлах, благодаря чему при создании нового проекта программист может нажать кнопку SettingsMaster.CorrectCurrentSolution и сразу начать проект с корректными параметрами группы. Команда SettingsMaster.CustomProjectUpdate обновляет проект, предлагая указать файл ввода, содержащий изменения, которые вы хотите сделать. Так, если вы решите определить новое значение для своих проектов C#, вы легко сможете добавить его во все проекты.

Страница свойств SettingsMaster в диалоговом окне Options (рис. 9-5) позволяет задать файлы по умолчанию для каждого поддерживаемого языка. Первая версия SettingsMaster поддерживает C#, Visual Basic .NET и неуправляемый C++. В качестве отличного упражнения предлагаю вам добавить в этот набор и J#. Вы также можете указать, чтобы команды SettingsMaster автоматически сохраняли обновленные проекты после исправления параметров.

Конфигурационные файлы представляют собой относительно простые XML-файлы, что продиктовано простотой синтаксического анализа и данью моде. Так как природа систем проектов в языках .NET и неуправляемом C++ сильно различна, им соответствуют разные схемы. Основная идея в том, что конфигурационный файл специфичен для конкретного языка и определяет индивидуальные параметры каждой конфигурации проектов на данном языке.

Для проектов .NET базовая схема такова (список полей см. в табл. 9-2):

```
<Configurations>
  <ProgLanguage></ProgLanguage>
  <Configuration>
    <ConfigName></ConfigName>
    <Properties>
      <Property>
```



```
        <PropertyName></PropertyName>
        <PropertyType></PropertyType>
        <PropertyValue></PropertyValue>
    </Property>
</Properties>
</Configuration>
</Configurations>
```

Табл. 9-2. Схема конфигурации проекта .NET

Узел	Описание
<Configurations>	Основной элемент, содержащий одну или более конфигураций.
<ProgLanguage>	Содержит строку GUID языка программирования, поддерживаемого данным файлом. Пример: <ProgLanguage> {B5E9BD34-6D3E-4B5D-925E-8A43B79820B4} </ProgLanguage>
<Configuration>	Набор свойств одной конфигурации сборки проекта.
<ConfigName>	Имя конфигурации. Соответствует целевой конфигурации в диспетчере конфигураций IDE Visual Studio .NET. Пример: <ConfigName>Debug</ConfigName>
<Properties>	Набор свойств данной конфигурации.
<Property>	Описание отдельного свойства.
<PropertyName>	Имя свойства объекта Project. Это свойство должно существовать в объекте автоматизации Project конкретного языка. Пример: <PropertyName>CheckForOverflowUnderflow</PropertyName>
<PropertyType>	Тип свойства. Тип может иметь только значения Boolean, String или Enum. Для свойства типа String вы должны включить атрибут типа Optype, Overwrite или Append, определяющий, как будет изменено строковое значение. Для свойства типа Enum вы должны включить атрибут типа Name, который представляет собой имя перечислимого типа, используемого корректным свойством объекта Project. Пример: <PropertyType>Boolean</PropertyType> Пример: <PropertyType Name="prjWarningLevel"> Enum</PropertyType>
<PropertyValue>	Значение, которое вы хотите присвоить свойству. В случае типов Boolean оно может быть равно или 1, или 0. Для типов String это строка, которую вы хотите присоединить или записать вместо предыдущей. Для типов Enum это численное значение перечисления. Пример: <PropertyValue>1</PropertyValue>

Наверное, лучше всего показать, как выглядит конфигурация .NET, на двух простых примерах. В листинге 9-5 приведен минимальный конфигурационный файл, необходимый для включения компоновки с приращением в отладочных компоновках и для ее отключения в завершающих компоновках проекта Visual Basic .NET, а в листинге 9-6 — присвоение уровню диагностики значения `prjWarningLevel14` только в заключительных компоновках проектов C#.

Листинг 9-5. Проект `SettingsMaster` для включения компоновки с приращением в отладочных компоновках и ее отключения в завершающих компоновках проекта Visual Basic .NET

```
<Configurations>
  <ProgLanguage>{B5E9BD33-6D3E-4B5D-925E-8A43B79820B4}</ProgLanguage>
  <Configuration>
    <ConfigName>Debug</ConfigName>
    <Properties>
      <Property>
        <!--Turn on (/incremental+)-->
        <PropertyName>IncrementalBuild</PropertyName>
        <PropertyType>Boolean</PropertyType>
        <PropertyValue>1</PropertyValue>
      </Property>
    </Properties>
  </Configuration>
  <Configuration>
    <ConfigName>Release</ConfigName>
    <Properties>
      <Property>
        <!--Turn off (/incremental-)-->
        <PropertyName>IncrementalBuild</PropertyName>
        <PropertyType>Boolean</PropertyType>
        <PropertyValue>0</PropertyValue>
      </Property>
    </Properties>
  </Configuration>
</Configurations>
```

Листинг 9-6. Проект `SettingsMaster` для включения 4-го уровня диагностики в заключительных компоновках проектов C#

```
<Configurations>
  <ProgLanguage>{B5E9BD34-6D3E-4B5D-925E-8A43B79820B4}</ProgLanguage>
  <Configuration>
    <ConfigName>Release</ConfigName>
    <Properties>
      <Property>
        <!--Turn on to level 4-->
        <PropertyName>WarningLevel</PropertyName>
        <PropertyType Name="prjWarningLevel">Enum</PropertyType>
        <PropertyValue>4</PropertyValue>
      </Property>
    </Properties>
  </Configuration>
</Configurations>
```

```
        </Property>
    </Properties>
</Configuration>
</Configurations>
```

Схема конфигурационного файла для неуправляемых приложений C++ похожа, однако она должна учитывать, что неуправляемые приложения определяют инструмент, с которым вы хотите работать. Базовая схема представлена в следующем фрагменте (список полей см. в табл. 9-3):

```
<Configurations>
  <ProgLanguage></ProgLanguage>
  <Configuration>
    <ConfigName></ConfigName>
    <Tools>
      <Tool>
        <ToolName></ToolName>
        <Properties>
          <Property>
            <PropertyName></PropertyName>
            <PropertyType></PropertyType>
            <PropertyValue></PropertyValue>
          </Property>
        </Properties>
      </Tool>
    </Tools>
  </Configuration>
</Configurations>
```

Табл. 9-3. Схема конфигурации проекта на неуправляемом C++

Узел	Описание
<Configurations>	Основной элемент, содержащий одну или более конфигураций.
<ProgLanguage>	Содержит строку GUID для неуправляемого C++. Она всегда будет иметь значение, указанное в примере. Пример: <ProgLanguage> {B5E9BD32-6D3E-4B5D-925E-8A43B79820B4} </ProgLanguage>
<Configuration>	Набор свойств одной конфигурации сборки проекта.
<ConfigName>	Имя конфигурации. Соответствует целевой конфигурации в диспетчере конфигураций IDE Visual Studio .NET. Пример: <ConfigName>Debug</ConfigName>
<Tools>	Набор инструментов данной конфигурации.
<Tool>	Свойства отдельного инструмента.

Табл. 9-3. Схема конфигурации проекта ... *(продолжение)*

Узел	Описание
<ToolName>	<p>Имя конкретного инструмента. Им может быть любой из объектов инструментов, поддерживаемых объектом VCPProject, а именно: VCAlinkTool, VCAuxiliaryManagedWrapperGeneratorTool, VCCLCompilerTool, VCCustomBuildTool, VCLibrarianTool, VCLinkerTool, VCManagedResourceCompilerTool, VCManagedWrapperGeneratorTool, VCMidlTool, VCNMakeTool, VCPPostBuildEventTool, VCPreBuildEventTool, VCPreLinkEventTool, VCPPrimaryInteropTool, VCResourceCompilerTool или VCXMLDataGeneratorTool. Вы также можете указать специальный объект VCConfiguration проекта VCPProject для доступа к общим параметрам проекта.</p> <p>Пример:</p> <p><ToolName>VCCLCompilerTool</ToolName></p>
<Properties>	<p>Набор свойств данной конфигурации инструментов.</p>
<Property>	<p>Описание отдельного свойства.</p>
<PropertyName>	<p>Имя свойства проекта. Это свойство должно существовать в объекте автоматизации VCPProject. Если свойство инструмента используется только для DLL, добавьте атрибут Type и присвойте ему значение "DLL", а если оно используется только для EXE-файлов — "EXE". Если свойство используется и для EXE-файлов, и для DLL, не включайте атрибут Type.</p> <p>Пример:</p> <p><PropertyName>BasicRuntimeChecks</PropertyName></p> <p>Пример:</p> <p><PropertyName Type="EXE"> OptimizeForWindowsApplication</PropertyName></p>
<PropertyType>	<p>Тип свойства. Тип может иметь только значения Boolean, String или Enum. Для свойства типа String вы должны включить атрибут типа OrType, Overwrite или Append, определяющий, как будет изменено строковое значение. Для свойства типа Enum вы должны включить атрибут типа Name, который представляет собой имя перечислимого типа, используемого конкретным свойством объекта Project.</p> <p>Пример:</p> <p><PropertyType>Boolean</PropertyType></p> <p>Пример:</p> <p><PropertyType Name="basicRuntimeCheckOption"> Enum</PropertyType></p>
<PropertyValue>	<p>Значение, которое вы хотите присвоить свойству. В случае типов Boolean оно может быть равно или 1, или 0. Для типов String это строка, которую вы хотите присоединить или записать вместо предыдущей. Для типов Enum это численное значение перечисления.</p> <p>Пример:</p> <p><PropertyValue>4</PropertyValue></p>

Как и в случае конфигурации .NET, я хочу привести пару простых примеров конфигураций проектов на неуправляемом C++. В листинге 9-7 показано, как включить оптимизацию для всей программы в заключительных компоновках. Код листинга 9-8 задает DEF-файл для отладочных и заключительных компоновок.

Листинг 9-7. Проект SettingsMaster для включения оптимизации всей программы в заключительных компоновках проекта на неуправляемом C++

```
<Configurations>
  <ProgLanguage>{B5E9BD32-6D3E-4B5D-925E-8A43B79820B4}</ProgLanguage>
  <Configuration>
    <ConfigName>Release</ConfigName>
    <Tools>
      <Tool>
        <ToolName>VCConfiguration</ToolName>
        <Properties>
          <Property>
            <!--Turns on /GL and /LTCG.-->
            <!--(Whole program optimization!)-->
            <PropertyName>WholeProgramOptimization</PropertyName>
            <PropertyType>Boolean</PropertyType>
            <PropertyValue>1</PropertyValue>
          </Property>
        </Properties>
      </Tool>
    </Tools>
  </Configuration>
</Configurations>
```

Листинг 9-8. Проект SettingsMaster, определяющий DEF-файл для отладочных и заключительных компоновок проекта на неуправляемом C++

```
<Configurations>
  <ProgLanguage>{B5E9BD32-6D3E-4B5D-925E-8A43B79820B4}</ProgLanguage>
  <Configuration>
    <ConfigName>Debug</ConfigName>
    <Tools>
      <Tool>
        <ToolName>VCLinkerTool</ToolName>
        <Properties>
          <Property>
            <!--Sets the .DEF file for the project.-->
            <PropertyName Type="DLL">ModuleDefinitionFile</PropertyName>
            <PropertyType OpType="Overwrite">String</PropertyType>
            <PropertyValue>.\$(ProjectName).DEF</PropertyValue>
          </Property>
        </Properties>
      </Tool>
    </Tools>
  </Configuration>
```

```
<Configuration>
  <ConfigName>Release</ConfigName>
  <Tools>
    <Tool>
      <ToolName>VCLinkerTool</ToolName>
      <Properties>
        <Property>
          <!--Sets the .DEF file for the project.-->
          <PropertyName Type="DLL">ModuleDefinitionFile</PropertyName>
          <PropertyType OpType="Overwrite">String</PropertyType>
          <PropertyValue>.\$(ProjectName).DEF</PropertyValue>
        </Property>
      </Properties>
    </Tool>
  </Tools>
</Configuration>
</Configurations>
```

Полные примеры для любого языка см. в файлах, которые я включил на CD вместе с проектом SettingsMaster; они настраивают ваши проекты с учетом всех рекомендаций, данных в главе 2. Для проектов .NET их можно использовать как есть, а вот некоторые указанные по умолчанию параметры проектов на неуправляемом C++ вам, возможно, захочется изменить. В проектах C++ я включаю строки Unicode и другие параметры, которые нравятся лично мне, но в ваших проектах они могут вызвать проблемы. Все узлы, на которые вам стоит обратить внимание, я прокомментировал в XML-файлах.

Вопросы реализации SettingsMaster

Многие из вас могут быть счастливы просто от использования надстройки SettingsMaster, но ее реализация также представляет некоторый интерес. Когда я только подумал о SettingsMaster, я начал разрабатывать макрос, потому что это гораздо проще, чем создавать полную надстройку. Исходный макрос вы найдете в каталоге SettingsMaster\OriginalMacro. Когда макрос заработал, я не захотел переводить весь код на C#, поэтому я реализовал надстройку на Visual Basic .NET. Так как Visual Basic .NET — это то же самое, что и C#, только без точек с запятой, переключаться между языками очень легко.

Самая сложная часть работы над SettingsMaster состояла в определении схемы XML. Благодаря относительно небольшому размеру файлов SettingsMaster я смог использовать удивительный класс XmlDocument, что сделало навигацию по документу тривиальной. Если б мне понадобилось создать эту надстройку еще раз, я попробовал бы разработать схему XML, позволяющую объединить всю информацию в один файл. Изучая код, вы увидите, что в нескольких местах я продублировал обработку двух типов проектов.

Чудо SettingsMaster основано на удивительном механизме отражения .NET. Возможность создания класса на лету и вызова его методов или установки и получения свойств — одна из самых лучших в .NET. Так как у меня были конфигурации и проекты, я применил отражение для создания отдельных инструментов

и свойств. Я включил в код массу комментариев, так что вы легко поймете, как все работает.

Наибольшая проблема при создании SettingsMaster была связана с созданием значения перечислимого типа. Так как .NET строго типизирована, если бы я не смог создать специфическое значение Enum, мне было бы сложно задать множественные параметры через объекты Project и VCPProject. После ряда безуспешных проб я обратился за помощью. Франческо Балена (Francesco Balena) напомнил мне, что с этим успешно справляется метод System.Enum.Parse. Все остальное оказалось простой нудной работой с XML-файлами.

Будущие усовершенствования SettingsMaster

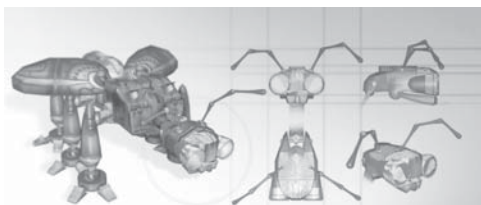
SettingsMaster — очень полезная надстройка, но если вы ищете проект, то можете внести в SettingsMaster ряд усовершенствований, чтобы сделать ее еще лучше.

- В SettingsMaster отсутствует редактор конфигурационных XML-файлов. С решетками свойств (property grid) работать довольно легко, поэтому вы могли бы создать такой редактор, чтобы избавиться от редактирования конфигурационных файлов вручную. Этот редактор конфигураций следует сделать доступным с командной панели SettingsMaster, а также из страницы свойств окна Options.
- Есть одна функция, добавить которую будет относительно легко: это обработчик событий, отслеживающий загрузку проектов и автоматически обновляющий их параметры.
- Некоторые глобальные параметры проектов на неуправляемом C++ устанавливаются при помощи объекта VCPlatform. Было бы неплохо реализовать поддержку этого объекта, чтобы пользователи могли задавать каталоги включаемых файлов и другие свойства, полезные при работе в группе.
- Отличной функцией была бы команда записи текущих параметров проекта в конфигурационный файл SettingsMaster, чтобы вы могли применить их к другим проектам.
- Чтобы предоставить пользователям дополнительной обратной связи, вы могли бы выводить изменения, сделанные SettingsMaster, в окно Output.

Резюме

Новые возможности создания макросов и надстроек, реализованные в IDE Visual Studio .NET, дали разработчикам могучую силу, позволяющую сделать среду именно такой, какая нужна для быстрого решения проблем. В этой главе я рассмотрел ряд вопросов, связанных с созданием реальных, жизнеспособных макросов и надстроек. Хотя в IDE все еще есть недостатки, общая картина более чем привлекательна.

Программисты давно желали получить мощь Visual Studio .NET. Теперь мы ее получили, и мне хотелось бы побудить вас реализовать средства, о которых вы всегда мечтали. Ими могли бы пользоваться все мы!



Мониторинг управляемых исключений

Наверное, вы уже поняли, что в разработке под Microsoft Visual Studio .NET гораздо больше исключений, чем в традиционной разработке под Microsoft Win32. Прелесть .NET в том, что обработка исключений была встроена изначально. Она не заимствовала прикрученные и привитые исключения, с которыми мы бились долгие годы, работая с приложениями Microsoft Windows и C++. Теперь исключения поддерживаются естественно и полноценно.

Но, как и всегда, исключения — для исключительных условий. Не стоит применять исключения вместо таких конструкций, как операторы `switch` и `case`, если не хотите получить истинно медленный код. В этой главе я представлю утилиту `ExceptionMon`, которая служит для наблюдения за исключениями, возникающими в приложении. Хотя через диалоговое окно исключений в отладчике можно установить все исключения CLR на остановку при инициации, вам потребовалась бы вечность, потому что пришлось бы постоянно нажимать кнопку `Continue`. `ExceptionMon` позволяет наблюдать за исключениями почти без хлопот.

`ExceptionMon` использует одно из великолепнейших средств .NET — `Profiling API`. Я писал средства профилирования (`profilers`) и инструменты обнаружения ошибок (`error detection tools`) без поддержки ОС и, когда в .NET увидел `Profiling API`, сразу возблагодарил богов разработки. `Profiling API` прекрасно продуман и работает точно, как заявлено. Его сила позволяет видеть то, что практически не увидеть иными способами. Интересно, что название `Profiling API` несколько обманчиво, так как `Profiling API` позволяет гораздо больше простого хронометрирования операций. К концу главы в вашей голове будут роиться идеи других «продвинутых» инструментов, которые можно создать, используя `Profiling API`. Вообще я буду применять `Profiling API` в следующих главах как основу для других прекрасных инструментов.

Наш путь к ExceptionMon начнется с обсуждения средств и целей Profiling API. Разобравшись с этим, я объясню как работать с ExceptionMon и как она реализована. И в заключение я расскажу о своем видении применения исключений в мире .NET.

Введение в Profiling API

Документация и примеры для Profiling API из .NET отсутствуют в MSDN, но они есть у вас на компьютере, если вы установили Visual Studio .NET. Волшебное место — <Каталог Visual Studio .NET>\SDK\v1.1\Tools Developers Guide. Там вы найдете каталог Docs с Word-документами, описывающими все: от Profiling API до Debugging API и Metadata API, а также полными спецификациями ECMA для общезыковой инфраструктуры (Common Language Infrastructure, CLI). Каталог Samples содержит примеры .NET-компиляторов, примеры Profile API и средство обхода зависимостей сборок (assembly dependency walker). В документах и примерах масса полезного, и, если вам любопытно, как в .NET все работает, каталог Samples — прекрасное место для начала исследований. Документ, описывающий Profiling API, — вполне очевидно — Profiling.DOC.

Есть два способа профилирования. В первом применяется процесс выборки (sampling). В нем средство профилирования через определенные интервалы в миллисекундах «заглядывает» в профилируемое приложение (profilee) и проверяет, что выполняется в данный момент, — это средство профилирования с выборкой имен (name sampling profiler). Второй метод — безвыборочный (nonsampling), где средство профилирования синхронно контролирует каждый вызов и возврат, отслеживая все, что происходит в профилируемом приложении. .NET Profiling API легко работает с обоими типами профилирования. Как я уже говорил, Profiling API позволяет делать гораздо больше, чем просто профилировать. Вот полный список элементов, о которых вы можете получать уведомления, создавая программу с помощью Profiling API (табл. 10-1). Получение этих уведомлений относительно тривиально, так что в будущем вы наверняка увидите массу изящных инструментов.

Табл. 10-1. Обеспечение Profiling API

Элемент	Типы уведомлений
Исполняющая среда	Приостановка (suspend) и возобновление (resume) управляемого выполнения (всех потоков), приостановка и возобновление отдельного управляемого потока
AppDomain	Старт (startup), завершение (shutdown)
Сборка	Загрузка (load), выгрузка (unload)
Модуль	Загрузка, выгрузка, присоединение (attach)
Класс	Загрузка, выгрузка
Функция	JIT-компиляция, поиск в кэше, изъятие (удаление из памяти), подстановка (inlined), выгрузка
Поток	Создание (create), уничтожение (destroy), присвоение потоку ОС
Remoting	Активизация клиента, отправка клиентом сообщения, получение клиентом ответа, получение сервером сообщения, активизация, отправка сервером ответа

Табл. 10-1. Обеспечение Profiling API *(продолжение)*

Элемент	Типы уведомлений
Переключения	Управляемое в неуправляемое, неуправляемое в управляемое, создание COM VTable, уничтожение COM Vtable
Приостановка исполняющей среды	Приостановка, отмена приостановки, возобновление, приостановка потока, возобновление потока
Сбор мусора	Выделение объекта, выделения по классам, перемещение ссылки, объектные ссылки, корневые ссылки
Исключение	Инициация, поиск, фильтрация, вход в перехватчик (catcher), перехватчик обнаружен, вызов ОС-обработчика, раскрутка функции, раскрутка finally, обнаружен перехватчик CLR, запущен перехватчик CLR

Для написании средства профилирования реализуется интерфейс `ICorProfilerCallback`. Хотя было бы прекрасно писать средство профилирования в управляемом коде, из-за архитектуры, поддерживаемой Profiling API, этого делать нельзя. Средство профилирования выполняется в адресном пространстве профилируемого управляемого приложения. Возможность использования управляемого кода вызвала бы чрезвычайно опасные ситуации разного рода. Так, если бы вы получили уведомление о проводимой операции сбора мусора и вам потребовалось бы выделить управляемую память для хранения собираемых элементов, это инициировало бы рекурсивный сбор мусора. Не нужно говорить, что архитекторы Microsoft выбрали более разумный подход, минимизирующий взаимное влияние. Для поддержки управляемых средств профилирования все уведомления должны быть межпроцессовыми (cross-process), что серьезно замедлило бы профилируемое приложение.

Поскольку средства профилирования — это всего лишь COM DLL, концепции должны быть знакомы всем, кто занимался разработкой под Windows с 2000 года. Всю нудную работу я инкапсулировал в библиотеке, что позволит вам сосредоточиться на важных моментах, не увязая в COM. Ниже я расскажу о ProfilerLib подробнее. Хочу отметить ключевой COM-аспект: ваш COM-код средства профилирования будет вызываться в модели со свободными потоками (free-threaded model), так что вам придется защищать структуры данных от повреждения в многопоточной среде (multithreaded corruption).

В интерфейсе `ICorProfilerCallback` лишь два метода нужны всегда: `Initialize` и `Shutdown`. `Initialize` вызывается самым первым. Вам передается интерфейс `IUnknown`, через который надо будет сразу запросить интерфейс `ICorProfilerInfo` и сохранить возвращенный интерфейс, чтобы запрашивать информацию о профилируемом приложении.

Многие методы `ICorProfilerCallback` получают идентификаторы. С помощью сохраненного интерфейса `ICorProfilerInfo` идентификатор преобразуется в удобное значение. Так, метод `ICorProfilerCallback::ModuleLoadFinished` получает значение `ModuleID`, представляющее идентификатор загруженного метода. Чтобы определить имя модуля и другую полезную информацию, такую как адрес загрузки (load address) и идентификатор сборки, вызовите метод `ICorProfilerInfo::GetModuleInfo`. Дополнительные задачи, выполняемые с помощью методов интерфейса, включают получение интерфейсов метаданных, инициацию сбора мусора и запуск отладки

процесса. Не буду описывать интерфейс `ICorProfilerInfo` полностью — подробную информацию см. в файле `Profiling.DOC`.

Сохранив интерфейс `ICorProfilerInfo` в методе `ICorProfilerCallback::Initialize`, следует сообщить CLR, какие уведомления вы хотели бы видеть. Красота системы `ICorProfilerCallback` в том, что вы будете получать уведомления только для нужных вам элементов, так что CLR сможет минимизировать использование ресурсов и выполнять профилируемое приложение как можно быстрее. Элементы, для которых требуются уведомления, позволяет указать метод `ICorProfilerInfo::SetEventMask`, принимающий битовое поле, которое указывает нужные элементы.

Устанавливаемые битовые флаги описаны в табл. 10-2. Большинство не требует объяснений. Некоторые значения — для которых в колонке Неизменяемый указано «Да» — могут быть установлены только во время вызова метода `ICorProfilerCallback::Initialize`. Если флаг уведомления не неизменяемый, его можно переключать в любой момент работы профилирующего средства. Чтобы увидеть включенные флаги уведомлений, вызовите метод `ICorProfilerInfo::GetEventMask`. Флаг `COR_PRF_ENABLE_OBJECT_ALLOCATED` устанавливается в методе `ICorProfilerCallback::Initialize`, указывая на необходимость установки CLR на отслеживание выделения объектов, а `COR_PRF_MONITOR_OBJECT_ALLOCATED` включает и выключает уведомления.

Табл. 10-2. Флаги уведомлений `SetMethod`

Флаг ¹	Неизменяемый	Описание
ALL	Да	Включает все флаги уведомлений.
APPDMAIN_LOADS	Нет	Уведомление о каждой загрузке или выгрузке <code>AppDomain</code> .
ASSEMBLY_LOADS	Нет	Уведомление о каждой загрузке или выгрузке сборки.
CACHE_SEARCHES	Нет	Уведомление, когда код периода инсталляции находит функции, запущенные через <code>Native Image Generator (NGEN)</code> .
CCW	Нет	Уведомление о каждой COM-оболочке.
CLASS_LOADS	Нет	Уведомление о каждой загрузке или выгрузке класса.
CLR_EXCEPTIONS	Нет	Уведомление о каждой внутренней обработке исключений в CLR.
CODE_TRANSITIONS	Да	Уведомление о каждом переключении между управляемым и неуправляемым кодом.
DISABLE_INLINING	Да	Отключает подстановку методов во всем процессе. Если не установлен, уведомления о подстановках проходят через уведомление <code>ICorProfilerCallback.JITInlining</code> .
DISABLE_OPTIMIZATIONS	Да	Предписывает JIT-компилятору отключить оптимизации.
ENABLE_IN_PROC_DEBUGGING	Да	Разрешает использование внутрипроцессной отладки (in-process debugging) вместе с <code>Profiling API</code> .
ENABLE_JIT_MAPS	Да	Разрешает отслеживание JIT-сопоставлений.

¹ Для ясности из имен флагов удалены префиксы `COR_PRF_` и `COR_PRF_MONITOR_`.

Табл. 10-2. Флаги уведомлений SetMethod (продолжение)

Флаг	Неизменяемый	Описание
ENABLE_OBJECT_ALLOCATED	Да	Уведомление о каждом объекте, выделенном из кучи собранного мусора.
ENABLE_REJIT	Да	Вызывает повторную JIT-компиляцию кода периода инсталляции (NGEN), чтобы включить для этих функций JIT-уведомления.
ENTERLEAVE	Нет	Ловушки (hooks) на входе и выходе функции вызова (call function).
EXCEPTIONS	Нет	Уведомление о каждом не-CLR-исключении (т. е. обо всех общих исключениях).
FUNCTION_UNLOADS	Нет	Уведомление о выгрузке функций.
GC	Да	Уведомление о готовящемся сборе мусора.
JIT_COMPILATION	Нет	Уведомление о каждой функции непосредственно до и сразу после ее JIT-компиляции.
MODULE_LOADS	Нет	Уведомление о каждой загрузке и выгрузке модуля.
NONE	Нет	Не посылать уведомления.
OBJECT_ALLOCATED	Нет	Уведомление о каждом объекте, выделяемом в кучу собранного мусора.
REMOTING	Да	Уведомление о пересечении каждого контекста удаленного взаимодействия (remoting).
REMOTING_ASYNC	Да	Уведомление о каждом асинхронном событии удаленного взаимодействия.
REMOTING_COOKIE	Да	Создание файлов «cookie», чтобы средство профилирования могло спаривать обратные вызовы удаленного взаимодействия.
SUSPENDS	Нет	Уведомление о приостановке CLR.
THREADS	Нет	Уведомление о каждом создании и уничтожении потока.

После возврата `S_OK` из метода `ICorProfilerCallback::Initialize` вы будете получать запрошенные уведомления через соответствующий метод `ICorProfilerCallback`. Я расскажу, что с этим делать чуть позже, так как сначала хочу упомянуть последний необходимый метод — `ICorProfilerCallback::Shutdown`.

Если профилируемый процесс начинает выполнение в виде управляемого приложения, метод `Shutdown` будет обязательно вызван. Однако, если приложение начинает выполнение как неуправляемое приложение, загружающее CLR, как Visual Studio .NET, то ваш метод `Shutdown` вызван не будет. Чтобы обеспечить остановку средства профилирования, в `DllMain` средства профилирования надо обрабатывать флаг `DLL_PROCESS_DETACH` и проверять, вызван ли метод `Shutdown`. Если нет, следует провести очистку вручную, помня, что, поскольку приложение завершается, надо быть осведомленным о выполняемых операциях. Пример действий в такой ситуации см. в коде `ExceptionMon`.

Кроме специальных алгоритмов, необходимых для реализации вашего конкретного профиля, основная работа будет заключаться в том, чтобы следить за

значениями, получаемыми методами уведомления `ICorProfilerCallback`. Многие методы уведомления получают значения идентификаторов, которые можно применять для получения определенной информации об объекте. Эти уникальные для Profiling API идентификаторы являются просто адресами элементов в памяти, благо интерфейс `ICorProfilerInfo` предлагает методы, помогающие преобразовать эти идентификаторы в реальные значения. Обычно для этого нужно вызвать соответствующий метод `ICorProfilerInfo`, получить интерфейс метаданных, напрямую связанный с идентификатором, и задействовать этот интерфейс в работе.

Метаданные ссылаются на данные, описывающие каждый объект в .NET. Самоописание объектов с помощью метаданных — загвоздка .NET. При разработке управляемых приложений метаданные доступны через механизм отражения. При разработке неуправляемых приложений, которым нужен доступ к метаданным, используется интерфейс чтения (reader interface) `IMetaDataImport` и интерфейс записи (writer interface) `IMetaDataEmit`. В основном работа, выполняемая в средствах профилирования, сопряжена с чтением данных через `IMetaDataImport`. `IMetaDataEmit` используется компиляторами для создания метаданных в скомпилированных в .NET двоичных файлах. Интерфейсы метаданных подробно описываются в файле `Metadata Unmanaged API.DOC`, так что я отправлю вас туда, так как по большей части работа с метаданными — чистая морока.

Наверное, лучший способ продемонстрировать работу с идентификаторами и метаданными — показать, как получить имя класса и метода из идентификатора функции (function ID). Значения идентификаторов функций получают многими методами `ICorProfilerCallback`, такими как `ExceptionUnwindFunctionEnter` (чтобы показать, какая функция раскручена), `JITCompilationFinished` (чтобы показать, какая функция прошла JIT-компиляцию) и `ManagedToUnmanagedTransition` (чтобы показать, какая функция переключается на неуправляемый код). В листинге 10-1 показан метод `GetClassAndMethodFromFunctionId` из `ProfilerLib`, который получает имя класса и метода из идентификатора функции. Как видите, для этого надо лишь прорваться через интерфейс метаданных.

Листинг 10-1. `GetClassAndMethodFromFunctionId`

```

BOOL CBaseProfilerCallback ::
    GetClassAndMethodFromFunctionId ( FunctionID uiFunctionId ,
                                     LPWSTR      szClass      ,
                                     UINT         uiClassLen   ,
                                     LPWSTR      szMethod      ,
                                     UINT         uiMethodLen  )
{
    // Магия метаданных в том, как найти эту информацию.

    // Возвращаемое значение.
    BOOL bRet = FALSE ;

    // Маркер для идентификатора функции.
    mdToken MethodMetaToken = 0 ;
    // Интерфейс метаданных.
    IMetaDataImport * pIMetaDataImport = NULL ;

```

```

// Запрашиваем через ICorProfilerInfo интерфейс
// метаданных для этого идентификатора функции.
HRESULT hr = m_pICorProfilerInfo->
    GetTokenAndMetaDataFromFunction ( uiFunctionId      ,
                                       IID_IMetaDataImport ,
                                       (IUnknown**) &pIMetaDataImport ,
                                       &MethodMetaToken    );

ASSERT ( SUCCEEDED ( hr ) );
if ( SUCCEEDED ( hr ) )
{
    // Маркер для класса.
    mdTypeDef ClassMetaToken ;
    // Суммарные копии символов.
    ULONG ulCopiedChars ;

    // Получаем из метаданных информацию о методе.
    hr = pIMetaDataImport->GetMethodProps ( MethodMetaToken ,
                                             &ClassMetaToken ,
                                             szMethod        ,
                                             uiMethodLen     ,
                                             &ulCopiedChars   ,
                                             NULL             ,
                                             NULL             ,
                                             NULL             ,
                                             NULL             ,
                                             NULL             );

    ASSERT ( SUCCEEDED ( hr ) );
    ASSERT ( ulCopiedChars < uiMethodLen );
    if ( ( SUCCEEDED ( hr ) ) &&
          ( ulCopiedChars < uiMethodLen ) )
    {
        // Имея маркер метаданных для класса, я могу найти класс.
        hr = pIMetaDataImport->GetTypeDefProps ( ClassMetaToken ,
                                                 szClass        ,
                                                 uiClassLen     ,
                                                 &ulCopiedChars ,
                                                 NULL           ,
                                                 NULL           );

        ASSERT ( SUCCEEDED ( hr ) );
        ASSERT ( ulCopiedChars < uiClassLen );
        if ( ( SUCCEEDED ( hr ) ) &&
              ( ulCopiedChars < uiClassLen ) )
        {
            bRet = TRUE ;
        }
        else
        {
            bRet = FALSE ;
        }
    }
}

```

```
        else
        {
            bRet = FALSE ;
        }
        pIMetaDataImport->Release ( ) ;
    }
    else
    {
        bRet = FALSE ;
    }

    return ( bRet ) ;
}
```

Запуск средства профилирования

До сих пор я рассказывал, как работают средства профилирования, но так и не упомянул, как их запускать. Увы, это слабое звено системы профилирования.

Загружаемое средство профилирования определяется двумя переменными окружения. Первая, которой следует установить ненулевое значение, — `Cor_Enable_Profiling`; она сообщает CLR, что следует включить профилирование. Второй — `Cor_Profiler` — следует задать CLSID или ProgID средства профилирования. Вот как установить средство профилирования ExceptionMon из командной строки.

```
set Cor_Enable_Profiling=0x1
set COR_PROFILER={F6F3B5B7-4EEC-48f6-82F3-A9CA97311A1D}
```

Установка переменных окружения прекрасно работает для Windows Forms и консольных приложений .NET, но как профилировать приложения Microsoft ASP.NET? Ох, это непросто. Поскольку надо устанавливать переменные окружения, придется установить две переменные в системном окружении (рис. 10-1), так как отсюда Microsoft Internet Information Services (IIS) и ASPNET_W3P.EXE/W3P.EXE будут считывать переменные окружения.

С помощью Visual Studio .NET 2003 и .NET Framework 1.1 можно перезапустить IIS, чтобы новый экземпляр ASPNET_W3P.EXE принял новые глобальные переменные окружения. Чтобы перезапустить IIS, вызовите консоль Internet Information Services, щелкните правой кнопкой имя машины, укажите на All Tasks и выберите из контекстного меню команду Restart IIS. В диалоговом окне Start/Stop/Reboot выберите Restart Internet Services On <имя компьютера> из раскрывающегося списка и щелкните OK. ASPNET_W3P.EXE/W3P.EXE не запустится, пока вы не запросите IIS об ASP.NET-приложении. Проверить, установлены ли переменные окружения, позволяет Process Explorer (см. главу 3): дважды щелкните ASPNET_W3P.EXE/W3P.EXE в верхнем окне и в диалоговом окне Properties перейдите на вкладку Environment.

Устанавливая переменные системного окружения, вы сталкиваетесь с еще одной проблемой. Будучи общесистемным (system-wide), любой процесс, загружающий CLR, автоматически профилируется. Это может соответствовать вашим намерениям, но с ростом количества процессов, загружающих CLR, могут быстро

возникнуть проблемы. Так, если в вашем средстве профилирования есть ошибка (я знаю, что этого не может быть, но просто в порядке шутки) и вы собираетесь отладить его с помощью Visual Studio .NET, ваше средство профилирования также будет загружено и в Visual Studio .NET, что может сорвать всю отладку. Можно использовать удаленную отладку, но я предпочитаю устанавливать еще одну переменную окружения, указывающую, в каком процессе или процессах запускать средство профилирования. Тогда при старте вы сможете проверить определенную переменную окружения и указать, запускаться ли в данном процессе. Если запускаться в процессе не нужно, просто вызовите `ICorProfilerInfo::SetEventMask`, передав `COR_PRF_MONITOR_NONE` как маску в методе `ICorProfilerCallback::Initialize`.

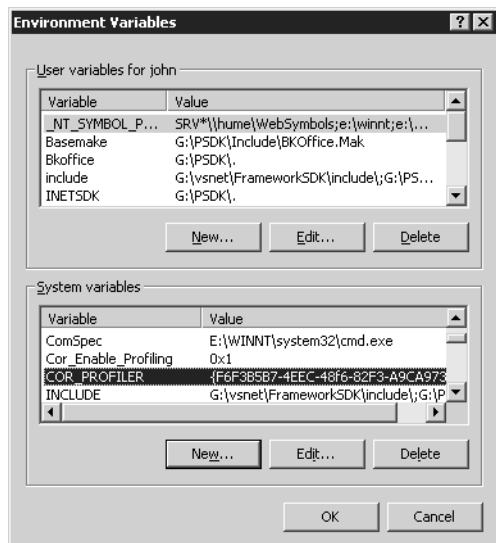


Рис. 10-1. Установка переменных системного окружения

Поскольку проверка того, запускать ли средство профилирования в определенном процессе, — операция стандартная, я реализовал в ProfilerLib метод `CBaseProfilerCallback::SupposedToProfileThisProcess`, выполняющий для вас эту проверку. Передайте как параметр проверяемую переменную окружения, и функция вернет `TRUE` в следующих случаях.

1. Переменная окружения не установлена, что предполагает желание профилировать все процессы.
2. Значение переменной окружения полностью соответствует диску, пути и имени текущего процесса.
3. Значение переменной окружения совпадает только с именем файла текущего процесса.

Здесь я хочу закончить введение в Profiling API. Он позволяет делать гораздо больше того, что было рассказано. Но, вместо того чтобы затягивать ваши глаза поволокой все новых подробностей и оставить вас в раздумьях о том, как применить Profiling API, думаю, лучше всего объяснить это, показав вам применение некоторых из самых «продвинутых» его функций. К концу этих двух глав, касаю-

щихся Profiling API, у вас сформируется гораздо более глубокое понимание, чем то, что можно составить с одним лишь Profiling.DOC.

ProfilerLib

Прежде чем мы погрузимся в глубины средства профилирования ExceptionMon, хочу посвятить немного времени разговору о ProfilerLib. Как вы, наверное, догадались из предшествующих разговоров о COM Profiling API, здесь много стереотипного кода. Поскольку я не в восторге от того, чтобы при разработке ПО снова и снова набирать одно и то же, я быстро понял, что мне нужна библиотека для выполнения черной работы, особенно с учетом большого числа методов, поддерживаемых интерфейсом `ICorProfilerCallback`.

Два примера средств профилирования, поставляемых с Visual Studio .NET, также придерживаются курса повторного использования COM-кода, но их способ написания кода непригляден тем, что смешивает всю инфраструктуру во вспомогательные структуры данных. Я работал над тем, чтобы исправить это, когда в «MSDN Magazine» за декабрь 2001 года появилась колонка Мэтта Питрека (Matt Pietrek) «Under the Hood». Мэтт взял пример кода средства профилирования и устранил путаницу. Я решил, что это хорошая основа, поэтому взял код Мэтта и улучшил его еще больше, упростив написание средств профилирования.

Процесс настройки для использования ProfilerLib довольно прост. Сначала надо создать DLL-проект и настроить его на подключение ProfilerLib.LIB, а в файл STDAFX.H проекта включить ProfileLib.H. В одном из ваших CPP-файлов определите следующие переменные и присвойте им значения, нужные для вашего средства профилирования:

```
// Строка GUID средства профилирования
wchar_t * G_szProfilerGUID
// CLSID средства профилирования
GUID G_CLSID_PROFILER
// Префикс ProgID средства профилирования
wchar_t * G_szProgIDPrefix
// Имя средства профилирования
wchar_t * G_szProfilerName

// Для примера вот значения для ExceptionMon
wchar_t * G_szProfilerGUID =
    L"{F6F3B5B7-4EEC-48f6-82F3-A9CA97311A1D}";

GUID G_CLSID_PROFILER =
    { 0xf6f3b5b7, 0x4eec, 0x48f6,
      { 0x82, 0xf3, 0xa9, 0xca, 0x97, 0x31, 0x1a, 0x1d } };

wchar_t * G_szProgIDPrefix = L"ExceptionMonProfiler";

wchar_t * G_szProfilerName = L"ExceptionMon";
```

Объявив уникальные COM-значения, добавьте `DllMain` в ваш CPP-файл:

```
HINSTANCE G_hInst = NULL ;
extern "C" BOOL WINAPI DllMain ( HINSTANCE hInstance ,
                                DWORD      dwReason ,
                                LPVOID
                                )
{
    switch ( dwReason )
    {
        case DLL_PROCESS_ATTACH:
            DisableThreadLibraryCalls ( hInstance ) ;
            G_hInst = hInstance ;
            break ;
        default :
            break ;
    }
    return( TRUE ) ;
}
```

ProfilerLib содержит базовый класс `CBaseProfilerCallback`, который реализует методы, необходимые интерфейсу `ICorProfilerCallback`. В своем средстве профилирования наследуйте классы обратного вызова от `CBaseProfilerCallback` и переопределяйте конкретные методы для получения нужных вам уведомлений. Так вы сможете сосредоточиться только на важных элементах, а не на остальных методах, которые просто путаются под ногами.

Присвоив имя наследующему классу, реализуйте функцию `AllocateProfilerCallback` со следующим прототипом. В этой функции создайте наследующий от `CBaseProfilerCallback` класс и верните его. Код в `ProfilerLib.h` позаботится об остальном.

```
ICorProfilerCallback * AllocateProfilerCallback ( ) ;
```

Наконец, возьмите из `ProfilerLib` файл `EXAMPLE.DEF`, скопируйте его в свой проект, переименуйте и замените в нем оператор `LIBRARY` для правильного выполнения всех экспортов, необходимых в создании COM DLL.

Кроме выполнения всей рутины COM, `ProfilerLib` вносит в `CBaseProfilerCallback` дополнительные методы, которые облегчат вам жизнь. Некоторые из них я уже упоминал, но есть и другие. Встречая что-либо, что, по моему мнению, может быть использовано повторно, я добавляю это в `ProfilerLib`, так что не забудьте проверить файлы проектов — вы увидите, что для вас написаны и другие экономящие время подпрограммы.

Как вы могли догадаться из кода, в подкаталоге `Tests` каталога `ProfilerLib` есть программа-пример `DoNothing`. Это простейшее средство профилирования, которое можно сделать, и оно демонстрирует применение `ProfilerLib`. Оно обрабатывает все уведомления, но лишь подает звуковой сигнал при инициализации и выгрузке. Это мой запатентованный метод разработки «Отладка ушами». Кроме того, все другие написанные мною утилиты, которые используют `Profiling API`, применяют `ProfilerLib` в качестве базовых классов, так что вы сможете увидеть более сложные примеры использования. `ProfilerLib` спасла мне огромное количество времени, и, надеюсь, сэкономит массу времени и вам.

ExceptionMon

Установив и запустив ProfilerLib, я смог приступить к ExceptionMon. Взглянув на интерфейс `ICorProfilerCallback`, вы увидите, что к вашим услугам все виды потрясающих обратных вызовов, позволяющие точно узнать, что делается при возникновении исключения. Как будто кто-то в Microsoft читал мои мысли! Могло показаться, что реализация ExceptionMon была совершенно тривиальной. Как всегда, на самом деле оказалось не так.

В ExceptionMon я хотел записывать инициированные исключения, вызванные обработчики `finally` и где обрабатывалось исключение. Методы уведомлений обработки исключений, содержащиеся интерфейсе `ICorProfilerCallback` и приведенные ниже, подходили точь-в-точь. Дополняет картину то, что вы также получаете идентификаторы функции и объекта инициированного и перехваченного исключения.

```
STDMETHOD ( ExceptionThrown ) ( ObjectID thrownObjectId );
STDMETHOD ( ExceptionUnwindFinallyEnter ) ( FunctionID functionId );
STDMETHOD ( ExceptionCatcherEnter ) ( FunctionID functionId ,
                                     ObjectID objectId );
```

С помощью ProfilerLib я быстро набросал первоначальный вариант ExceptionMon, записывая вывод в файл журнала в том же каталоге, откуда загружен процесс ExceptionMon. Первая маленькая проблема, с которой я столкнулся, — как лучше отладить ExceptionMon. Поскольку CLR выполняло всю работу по внесению указанного средства профилирования в адресное пространство, я хотел обеспечить возможность начать отладку с самого начала. Раз уж мы используем переменные окружения для запуска средства профилирования, я решил пойти дальше и добавить еще одну — `EXPMONBREAK`, установка которой заставляла ExceptionMon вызвать `DebugBreak`, чтобы я смог подключить отладчик.

Хотя средство профилирования можно отлаживать как любую неуправляемую DLL, загруженную в процесс, я предпочитаю вызов `DebugBreak`, так как средство профилирования будет загружено в Visual Studio .NET, поскольку здесь размещается CLR. Можно ограничить загрузку процессов, установив переменную окружения `EXPMONPROC`, чтобы отлаживать только один процесс. Однако для нужд разработки и отладки я предпочитаю запускать для тестирования несколько программ. Используя схему `EXPMONBREAK`, я легко могу подключить несколько отладчиков к нескольким процессам.

Раз я говорю о переменных окружения, следует упомянуть две важнейшие для мониторинга исключений — `ASPNET_WPEXE/W3WPEXE`. По умолчанию ExceptionMon не сбрасывает файл вывода на диск, поэтому, чтобы увидеть отчет об исключениях, надо остановить `ASPNET_WPEXE/W3WPEXE`. Однако, если установить переменную окружения `EXPMONFLUSH`, все записи сбрасываются на диск немедленно.

Еще одна проблема с записью файлов в том, что ExceptionMon поместит файл журнала в каталог процесса, тогда как стандартная учетная запись `ASPNET`, вероятно, не имеет разрешения на создание файлов в `%SYSTEMROOT%\Microsoft.NET\Framework\%FRAMEWORKVERSION%`, где располагается `ASPNET_WPEXE`. В Windows Server 2003 применяется учетная запись `NETWORK SERVICE`, а `W3WPEXE` располагается в `%SYSTEMROOT%\System32\inetsrv`. Полный путь и имя для файла вывода для ExceptionMon можно указать в переменной окружения `EXPMONFILENAME`. Оче-

видно, вам придется выполнять двойную проверку наличия у учетной записи ASP.NET прав на создание и запись файла в данном каталоге.

Первоначальная версия ExceptionMon работала прекрасно, так как получала идентификаторы функции и объекта и могла просто вызывать соответствующие методы в интерфейсе `ICorProfilerInfo` для получения маркеров (tokens) класса и функции, чтобы найти имена в метаданных. Код из листинга 10-1 — `CBaseProfiler-Callback::GetClassAndMethodFromFunctionId` — демонстрирует все, что нужно сделать для получения имен классов и методов по идентификатору функции.

Внутрипроцессная отладка и ExceptionMon

Доведя базовую версию до рабочего состояния, я подумал, что полезно было бы добавить обзор стека на момент инициации исключения. Тогда вы смогли бы понять, как сложилась такая ситуация, и взглянуть на условия. В документации для Profiling API я заметил, что `ICorProfilerInfo::SetEventMask` можно передать битовый параметр `COR_PRF_ENABLE_INPROC_DEBUGGING`, чтобы включить внутрипроцессную отладку.

При внутрипроцессной отладке Profiling API передает уведомления о событиях, но вам потребуется способ получения более подробной информации, чем та, что можно получить через интерфейс `ICorProfilerInfo`. Поскольку в Microsoft уже разработали прекрасный «продвинутый» отладочный API, работающий рука об руку с CLR, идея состояла в том, чтобы предоставить ограниченную версию отладочного API, способного выполнять такие задачи, как контроль значений переменных в реальном времени и просмотр стека.

Полностью отладочный API описывается в `DebugRef.DOC` из того же каталога, что и документы по Profiling API и API метаданных. Как и все документы в каталоге `Tools Developers Guide`, `DebugRef.DOC` пространен в описании интерфейсов, методов и значений параметров и вполне конкретен в применении. Каталог `Samples` содержит рабочий отладчик, составляющий примерно 98% исходного кода реального `CORDBG`, но сам код иногда трудно проследить, хотя в конечном счете он раскрывает свои секреты.

Читая документацию по отладочному API, уделите особое внимание тому, какие методы вызываются из внутрипроцессной отладки. Если под именем метода есть зеленый текст «Not Implemented In-Process», использовать его нельзя. Вы увидите, что большинство методов, которые нельзя использовать, относится к установке точек прерывания и изменению значений. Поскольку главная причина проведения внутрипроцессной отладки — простой сбор информации, все важные элементы полностью доступны.

Первый этап в применении внутрипроцессной отладки с Profiling API — установка флага `COR_PRF_ENABLE_INPROC_DEBUGGING` при вызове `ICorProfilerInfo::SetEventMask`. Интересно, что его простая установка вызывает два побочных эффекта. Первый состоит в том, что, раз вы потребовали внутрипроцессную отладку, профилируемое приложение будет выполняться медленнее. Это потому, что CLR не будет использовать прекомпилированный (precompiled) код, скомпилированный с помощью `NGEN.EXE`, заставляя этот код пройти JIT-компиляцию, как в обычных условиях. Возможно, вы не используете `NGEN.EXE`, но его весьма интенсивно применяет .NET Framework, так что здесь будут потери.

Если вы запускали NGEN.EXE, то могли заметить параметр командной строки /PROF, добавляющий к создаваемому коду информацию профилирования. Хотя это и может показаться хорошим решением, пока Profiling API не поддерживает его, так что использовать его нельзя. Я все-таки считаю, что замедление кода окупается преимуществами.

Вторая проблема, с которой вам придется столкнуться, не документирована и в первый раз совершенно сбита меня с толку. Метод `ICorProfilerCallback::ExceptionThrown` получает идентификатор объекта, который описывает сгенерированный класс. В моей первой реализации, не использующей внутрипроцессную отладку, я всегда получал идентификатор, способный передать `CBaseProfilerCallback::GetClassAndMethodFromFunctionId`. Простое добавление флага `COR_PRF_ENABLE_INPROC_DEBUGGING` к `ICorProfilerInfo::SetEventMask` даже без действительного применения API внутрипроцессной отладки меняет что-то изнутри, так что в качестве идентификатора объекта передается только 0. Хотя API внутрипроцессной отладки и содержит методы для получения информации, было весьма неприятно выяснять, что случилось с идентификатором объекта просто из-за установки флага!

Чтобы задействовать отладочные интерфейсы, надо вызвать метод `ICorProfilerInfo::BeginInProcDebugging` для запуска процесса получения соответствующего интерфейса. Как часть этого вызова передается указатель `DWORD` на файл «cookie» контекста. Этот файл следует сохранить, чтобы передать его методу `ICorProfilerInfo::EndInProcDebugging`, который надо вызвать, чтобы указать на окончание внутрипроцессной отладки. Второй шаг — получение соответствующего отладочного интерфейса. Если вас интересует только текущий поток, вызовите метод `ICorProfilerInfo::GetInProcInspectionIThisThread`, чтобы получить интерфейс `IUnknown`, через который можно запросить интерфейс `ICorDebugThread`. Чтобы провести общепроцессную отладку, вызовите `ICorProfilerInfo::GetInProcInspectionInterface` и запросите через возвращенный `IUnknown` интерфейс `ICorDebug`. Лично я не понимаю, почему два метода `ICorProfilerInfo` не могут просто возвращать соответствующие интерфейсы.

Получив отладочный интерфейс, вы готовы к обращению к отладочному API за необходимой информацией. В моем случае я хотел получить последнее исключение в потоке, так что все, что мне надо было сделать, это вызвать метод `ICorDebugThread::GetCurrentException`, чтобы получить интерфейс `ICorDebugValue`, описывающий последнее инициированное исключение. Странно, что каждый раз при вызове метода `ICorDebugThread::GetCurrentException` происходил сбой, так что я и вправду начал волноваться удастся ли заставить `ExceptionMon` работать!

Простудировав документацию на профилирующий и отладочный API, я обнаружил предложение, говорящее, что для выполнения любых операций со стеком во внутрипроцессной отладке надо вызвать `ICorDebugThread::EnumerateChains`. Отладочный API использует концепцию стековых цепочек (stack chains) чтобы связать информацию об управляемом и неуправляемом стеках, составляющую полную информацию о стеке. Я не видел, чтобы вызов `ICorDebugThread::GetCurrentException` был как-то связан со стеком, но решил, что стоит попробовать вызвать `ICorDebugThread::EnumerateChains`, прежде чем делать что-то еще. Хотя это не документировано (по крайней мере, неявно), я выяснил, что для работы с отладочным API, надо сначала вызвать `ICorDebugThread::EnumerateChains`, иначе большинство методов не

сработает. В листинге 10-2 показан метод-оболочка, который я использую в ExserptionMon для запуска внутрипроцессной отладки.

Листинг 10-2. BeginInprocDebugging

```
HRESULT CExceptionMon ::
    BeginInprocDebugging ( LPDWORD                pdwProfContext    ,
                          ICorDebugThread **      pICorDebugThread  ,
                          ICorDebugChainEnum **    pICorDebugChainEnum )
{
    // Сообщаем Profiling API о необходимости внутрипроцессной отладки.
    HRESULT hr = m_pICorProfilerInfo->
        BeginInprocDebugging ( TRUE                ,
                              pdwProfContext );

    ASSERT ( SUCCEEDED ( hr ) );
    if ( SUCCEEDED ( hr ) )
    {
        IUnknown * pIUnknown = NULL ;

        // Запрашиваем у Profiling API интерфейс IUnknown,
        // от которого можно получить ICorDebugThread.
        hr = m_pICorProfilerInfo->
            GetInprocInspectionIThisThread ( &pIUnknown ) ;
        ASSERT ( SUCCEEDED ( hr ) );
        if ( SUCCEEDED ( hr ) )
        {
            hr = pIUnknown->
                QueryInterface ( __uuidof ( ICorDebugThread ) ,
                                (void**)pICorDebugThread );
            ASSERT ( SUCCEEDED ( hr ) );

            // В любом случае IUnknown мне больше не нужен.
            pIUnknown->Release ( ) ;

            // Я делаю это в ходе обычной работы потому,
            // что, если прежде всего из ICorDebugThread
            // не вызвать ICorDebugThread::EnumerateChains,
            // многие другие методы не сработают.
            if ( SUCCEEDED ( hr ) )
            {
                hr = (*pICorDebugThread)->
                    EnumerateChains ( pICorDebugChainEnum ) ;
                ASSERT ( SUCCEEDED ( hr ) );
                if ( FAILED ( hr ) )
                {
                    (*pICorDebugThread)->Release ( ) ;
                }
            }
        }
    }
    return ( hr ) ;
}
```

Добившись от `ICorDebugThread::GetCurrentException` возврата правильного значения, я решил что я уже у цели, поскольку оставалось лишь получить имя класса из `ICorDebugValue`. Увы, просматривая соответствующие интерфейсы — `ICorDebugGenericValue`, `ICorDebugHeapValue`, `ICorDebugObjectValue`, `ICorDebugReferenceValue` и `ICorDebugValue`, я понял, что придется еще многое сделать, так как только `ICorDebugObjectValue` содержал метод `GetClass`, необходимый для получения интерфейса класса, который предоставил бы имя. Это означало, что мне придется поработать, чтобы преобразовать исходный `ICorDebugValue` от `ICorDebugThread::GetCurrentException` в `ICorDebugObjectValue`. Проще всего мне показать вам код, выполняющий всю работу (листинг 10-3). Как видите, нужно разыменовать (dereference) объект и запросить интерфейс `ICorDebugObjectValue`.

Листинг 10-3. `GetClassNameFromValueInterface`

```
HRESULT CExceptionMon ::
    GetClassNameFromValueInterface ( ICorDebugValue * pICorDebugValue ,
                                    LPTSTR          szBuffer          ,
                                    UINT             uiBuffLen        )
{
    HRESULT hr = S_FALSE ;

    ICorDebugObjectValue * pObjVal = NULL ;

    ICorDebugReferenceValue * pRefVal = NULL ;

    // Получаем ссылку на это значение. Так должны поступать исключения.
    // Если получить ICorDebugReferenceValue не удалось, значит,
    // это тип ICorDebugGenericValue. Я ничего не могу сделать
    // с ICorDebugGenericValue, так как мне нужно имя класса.
    hr = pICorDebugValue->

        QueryInterface ( __uuidof ( ICorDebugReferenceValue ),
                        (void**)&pRefVal
                        );

    if ( SUCCEEDED ( hr ) )
    {
        // Разыменовываем значение.
        ICorDebugValue * pDeRef ;
        hr = pRefVal->Dereference ( &pDeRef ) ;

        if ( SUCCEEDED ( hr ) )
        {
            // Разыменовав, я могу запросить объектное значение.
            hr = pDeRef->

                QueryInterface ( __uuidof ( ICorDebugObjectValue ),
                                (void**)&pObjVal
                                );

            // Разыменование мне больше не нужно.
            pDeRef->Release ( ) ;
        }
        // Ссылка мне больше не нужна.
```

```

    pRefVal->Release ( ) ;
}

ASSERT ( SUCCEEDED ( hr ) ) ;
if ( SUCCEEDED ( hr ) )
{
    // Получаем интерфейс класса для этого объекта.
    ICorDebugClass * pClass ;

    hr = pObjVal->GetClass ( &pClass ) ;

    // Объектная ссылка мне больше не нужна.
    pObjVal->Release ( ) ;

    ASSERT ( SUCCEEDED ( hr ) ) ;
    if ( ( SUCCEEDED ( hr ) ) )
    {
        // Получаем маркер синонима типа для класса.
        mdTypeDef ClassDef ;
        hr = pClass->GetToken ( &ClassDef ) ;

        ASSERT ( SUCCEEDED ( hr ) ) ;
        if ( SUCCEEDED ( hr ) )
        {
            // Для просмотра маркера класса мне нужен модуль,
            // чтобы запросить интерфейс метаданных.
            ICorDebugModule * pMod ;
            hr = pClass->GetModule ( &pMod ) ;

            ASSERT ( SUCCEEDED ( hr ) ) ;
            if ( SUCCEEDED ( hr ) )
            {
                // Получаем метаданные.
                IMetaDataImport * pIMetaDataImport = NULL ;

                hr = pMod->
                    GetMetaDataInterface ( IID_IMetaDataImport ,
                                            (IUnknown**)&pIMetaDataImport ) ;

                ASSERT ( SUCCEEDED ( hr ) ) ;
                if ( SUCCEEDED ( hr ) )
                {
                    // Наконец, получаем имя класса.
                    ULONG ulCopiedChars ;

                    hr = pIMetaDataImport->
                        GetTypeDefProps ( ClassDef
                                          ,
                                          szBuffer
                                          ,
                                          uiBuffLen
                                          ,
                                          &ulCopiedChars ,

```

см. след. стр.


```

                                NULL
                                NULL
                                ) ;

    ASSERT ( ulCopiedChars < uiBuffLen ) ;
    if ( ulCopiedChars == uiBuffLen )
    {
        hr = S_FALSE ;
    }

    pIMetaDataImport->Release ( ) ;
}
pMod->Release ( ) ;
}
}
pClass->Release ( ) ;
}
}
return ( hr ) ;
}

```

Имя класса для исключения получено — оставалось посмотреть стек. Я уже получил интерфейс `ICorDebugChainEnum`, так что просмотр стека сводился к следованию алгоритму, описанному в файле `DebugRef.DOC`. Единственное, что интересно в просмотре стека: нельзя посмотреть неуправляемый стек с помощью отладочного API. Чтобы проверить, является ли цепочка управляемой, вызовите `ICorDebugChain::IsManaged`.

Для меня `ExceptionMon` оказался бесценным помощником в слежении за исключениями, которые генерируют мои приложения. Я совершенно доволен выводом в текстовый файл, но вам может прийти мысль добавить возможность вывода информации через GUI, чтобы видеть исключения почти в реальном времени. Это несложно, и это прекрасный способ изучить программирование `Windows Forms`!

Использование исключений в .NET

Теперь, когда `ExceptionMon` следит за вашими исключениями, хочу поговорить о применении исключений в .NET. То, что в .NET исключения встроены внутрь, — определенно повод для ликования. Для тех, кто перешел из C++ Win32, исключения C++ казались прекрасной идеей, но их реализация оставляла желать много лучшего. Поскольку .NET обладает ясной и целостной манерой обработки исключений библиотеки классов .NET Framework (FCL), разработка в .NET становится гораздо проще.

Я готов был написать отдельную главу по обработке исключений, но мой коллега Джеффри Рихтер (Jeffrey Richter) уже проделал замечательную работу в своей книге «Applied Microsoft .NET Framework Programming» (издательство Microsoft Press, 2002 год) и «Applied Microsoft .NET Framework Programming in Microsoft Visual Basic .NET» (Microsoft Press, 2003). Его главы по обработке исключений (глава 18 в обеих книгах) следует прочесть всем, кто занимается разработкой в .NET. Но я хочу особо подчеркнуть некоторые аспекты использования и создания собственных исключений в программах.

Первое: исключения для исключительных событий. Мы все слышали это, но я обнаружил, что у многих разработчиков проблемы с определением. Мое определение состоит в том, что исключение следует инициировать, только когда встречается ошибка или непредвиденные условия. Одной из виденных мною у разработчиков ошибок было использование исключений вместо оператора `switch...case`. (Я правда это видел!) Иницилируйте исключения, только когда что-то не так. Не возвращайте общие коды состояния с помощью исключений.

Аргумент в поддержку постоянного использования исключений состоит в том, что разработчики никогда не проверяют возвращаемые значения. Для меня это ложный аргумент, так как, если разработчики не проверяют возвращаемые значения, значит, они не выполняют свои обязанности и их следует уволить. Я имею в виду, что мне встречались люди, которые злоупотребляют исключениями, тогда как код был бы намного четче и быстрее, если бы они просто возвращали значение. В своем коде я применяю такое общее правило: всегда инициировать исключения в открытых методах и свойствах при ошибке. Таким образом, для тех, кто использует мой код, формируется единый подход к обработке ошибок. Внутри своего класса вместо инициации внутренних вспомогательных функций я применяю возвращаемые значения, оставляя инициацию исключений для главных методов. Разумеется, если один из этих внутренних методов действительно попадает в ошибочные условия, я тут же инициирую исключение. Все это вполне логично.

Я говорил об ущербе производительности, потому что, несмотря на свободу исключений в .NET, внутри они реализуются через SEH. Если хотите это проверить, отладьте приложение .NET, используя отладку в неуправляемом режиме (native mode-only debugging), — вы увидите те же первые случаи исключения при инициации вашего исключения. Это подразумевает переход в режим ядра при каждом исключении. Идея, повторяю, в том, чтобы инициировать исключения при ошибках, а не в нормальном ходе выполнения программы.

Серьезнейшая проблема с исключениями состоит в том, что трудно узнать, что перехватывать при использовании FCL. Как говорит Джеффри в главе об исключениях (правило, которое вы, вероятно, затвердили), перехватывайте только те исключения, что подходят используемым объектам. Каждый метод и свойство в документации FCL содержит раздел Exceptions. Когда я применяю каждое свойство или метод, то всегда дважды сверяюсь со справкой (к счастью, справка по F1 достаточно «поумнела», чтобы открывать правильный раздел) и проверяю все инициируемые исключения, дабы убедиться, что я перехватываю лишь те, что инициируются согласно документации. Следите за перехватом исключений, чтобы избежать неожиданностей.

Microsoft в C# использует те же документирующие комментарии, что и для создания справочной документации MSDN, и, как я говорил в главе 9, почти такую же документацию можно создавать с помощью прекрасного инструмента Ndoc (его можно скачать по адресу <http://ndoc.sourceforge.net>). Чтобы облегчить жизнь тем, кто использует ваши объекты, заполняйте теги `<exception>` и указывайте все исключения, инициируемые в вашем коде. Кроме того, неплохо бы дважды проверить все выполняемые вами FCL-вызовы и указать, какие исключения могут быть инициированы в этих методах, чтобы предоставить полный от-

чет. При проверке кода контроль того, что исключения полностью документированы, — одна из моих «горячих клавиш», и я всегда стараюсь убедиться, что это так.

Раз я упомянул о проверке кода и исключениях, укажу еще три цели, к которым всегда стремлюсь. Первая: блоки `Finally` в любых методах или свойствах, открывающие что-то, что может быть истолковано как ресурс с описателем (`handle-based resource`), что обеспечивает очистку этих элементов. Я также ищу любые блоки `catch (Exception) {...}` или `catch {...}` и убеждаюсь, что они выполняют инициацию. Наконец, я всегда перепроверяю, чтобы повторные инициации не содержали после себя параметра, как здесь:

```
try
{
    // Что-то выполняем.
}
catch (DivideByZeroException e )
{
    // НЕ ДЕЛАЙТЕ ЭТОГО!!
    throw e ;
}
```

Повторно иницируя исключение, вы теряете информацию о его происхождении.

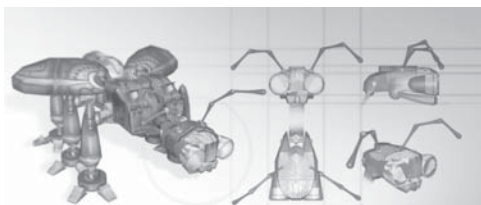
Последнее, что хочется отметить об исключениях, касается оператора `using` в C#. Оператор `using` разворачивается в тот же IL-код, что и блок `try...finally`, и вызывает метод `Dispose` для единственного объекта, указанного в операторе `using`. Применение оператора `using` абсолютно оправданно, но я предпочитаю этого не делать, так как происходящее за кадром не вполне очевидно.

Резюме

Концепция исключений в .NET радикально отличается от исключений в Win32. Благодаря `ExceptionMon`, у вас теперь есть способ мониторинга исключений, а значит, теперь вы сможете применять их более эффективно. Советую поэкспериментировать с `ExceptionMon` — вы удивитесь тому, что происходит в ваших приложениях.

Волшебная сила `ExceptionMon` в невероятном `Profiling API`. Поскольку `Profiling API` позволяет видеть все интересное, что происходит внутри CLR, в ваших руках огромная сила для написания таких инструментов, о которых в прошлом можно было только мечтать. Как вы увидите дальше, `Profiling API` позволяет делать еще больше.

Надеюсь, я дал вам пищу для размышлений о применении и реализации исключений в ваших собственных проектах .NET. Но все же библией в изучении исключений остаются главы книг Джеффри Рихтера, и я рекомендую вам прочесть их. Ключ в том, чтобы продумывать и планировать использование исключений с самого начала. Теперь в нашем распоряжении есть этот прекрасный инструмент, но если мы будем использовать его неправильно, то можем спровоцировать проблемы по мере развития продукта.



Трассировка программы

В главе 10 я вкратце затронул описание возможностей Profiling API. В этой главе я расскажу про Profiling API подробнее и рассмотрю программу, которую мне всегда хотелось иметь в своем распоряжении. В главе 6 я упоминал очень полезную команду `wt` (Watch and Trace — наблюдение и трассировка) консольного отладчика управляемого кода CORDBG.EXE. Как вы помните, она позволяет увидеть поток вызовов методов, а значит, и поток выполнения всей программы. Команда `wt` обеспечивает фантастический способ обнаружения «узких мест», которые просто невозможно найти путем простого изучения исходного кода.

К сожалению, CORDBG — консольное приложение, что не способствует его пониманию. Кроме того, CORDBG работает медленно, так как использует для трассировки пошаговый механизм отладки. Я хочу, чтобы трассировка была быстрой, а выводимая в результате информация — простой в использовании. Вот для этого и нужна моя любимая утилита FlowTrace. Она дает вам силу `wt` без всякого горького привкуса!

Сначала я покажу, насколько легко и эффективно устанавливать ловушки для вызовов методов при помощи Profiling API. Объяснив, как использовать программу FlowTrace, я опишу некоторые вопросы ее реализации, чтобы ее работа стала понятнее. Наконец, функциональность FlowTrace легко расширить, поэтому в конце главы я поделюсь кое-какими идеями, которые помогут вам сделать эту программу еще полезнее.

Установка ловушек при помощи Profiling API

Одна из самых сложных проблем при написании реальной программы профилирования для Microsoft Win32 заключалась в том, что установить ловушки для потока вызовов функций было почти невозможно без значительной помощи со стороны компилятора или без изменения двоичного файла на диске. Поэтому о

получении правильных временных интервалов, связанных с элементами пользовательских приложений, оставалось только мечтать. Теперь этот механизм уведомления о вызовах функций встроен в Profiling API — еще одно доброе дело со стороны Microsoft, заслуживающее искренней благодарности. Благодаря этому разработчики инструментов могут сосредоточиться на решении важных проблем профилирования, не тратя длительного времени на создание инфраструктуры своих утилит.

Запрос уведомлений входа и выхода

Profiling API позволяет получать уведомления обо всех вызовах методов и обо всех возвратах из них. Ключи `/Gh` и `/Gh` (включающие функции-ловушки `_penter` и `_pexit` соответственно) неуправляемого компилятора C++ играют по сути ту же роль, что и Profiling API, однако Profiling API делает уведомления еще проще, предоставляя также `FunctionID` выполняемой функции.

Как и в случае всех остальных уведомлений, исполняющей среде сначала нужно сообщить, что вы хотите получать уведомления входа и выхода; для этого надо установить в битовой маске при помощи операции ИЛИ флаг `COR_PRf_MONITOR_ENTER_LEAVE` и передать маску методу `ICorProfilerInfo::SetEventMask`. Готов спорить, что, как только вы запросите уведомления входа и выхода, вам не понадобится изменять их на протяжении всего существования процесса. И все же славные парни из Microsoft позволяют вам включать/отключать уведомления входа и выхода сколько душе угодно. Не забывайте про эту возможность, так как на ее основе можно создать очень интересные инструменты, измеряющие, например, только время обработки исключений.

После этого исполняющей среде нужно сообщить, какие функции-ловушки вам хотелось бы вызывать; это делается при помощи метода `ICorProfilerInfo::SetEnterLeaveFunctionHooks`, который принимает три указателя на вызываемые функции-ловушки: функцию входа, функцию выхода и функцию выхода типа `tailcall`. Назначение первых двух функций очевидно, а вот третья нуждается в пояснении. В настоящей версии CLR функции `tailcall` никогда не вызываются. Вызов типа `tailcall` имеет место, когда кадр стека текущего метода удаляется до выполнения команды `call`. Иначе говоря, если при вызове метода кадр стека вызывающего метода уже не нужен, он очищается. В будущих версиях CLR компилятор будет поддерживать `tailcall`-оптимизацию, тогда она вам и понадобится. Так как для большинства пользователей Profile API различия между функцией выхода типа `tailcall` и обычной функцией выхода не имеют значения, можете с чистой совестью применять обычную функцию выхода.

Реализация функций-ловушек

Особенность установки ловушек заключается в определении функций-ловушек. Для обеспечения максимально высокой производительности Profiling API требует, чтобы они использовали соглашение вызова `naked`. По сути ваши функции встраиваются в код JIT-компилятором, так что вы должны сами написать для них пролог и эпилог.

Объявления `typedef` для всех функций-ловушек выглядят так:

```
typedef void FunctionEnter ( FunctionID funcID ) ;
```

Из документации не совсем ясно (к счастью, это становится понятным при изучении примеров профилирования), что в одном аспекте функции-ловушки похожи на стандартные вызовы: они сами отвечают за извлечение параметра `FunctionID` из стека. В комментариях в файле `CorProf.IDL`, которому всегда нужно доверять больше, чем `Profiling.DOC`, указано, что функции-ловушки должны сохранять и все изменяемые регистры, в том числе регистры для работы с числами с плавающей точкой.

Пример функции-ловушки — в листинге 11-1. Функции-ловушки используют соглашение вызова `naked`, поэтому вы сами должны написать пролог и эпилог. Вся действительная работа выполняется в методе `CFlowTrace::FuncEnter`, таким образом, функция-ловушка — всего лишь оболочка для его вызова. Пролог (первые три команды `PUSH`) сохраняет изменяемые регистры в стеке. Последние четыре команды — эпилог, который восстанавливает сохраненные регистры и выполняет возврат из функции. Команда `RET 4` возвращает и удаляет из стека переданный функции-ловушке идентификатор функции, сохраняя мне одну команду `POP`.

Четыре команды, расположенные в середине функции, вызывают метод `CFlowTrace::FuncEnter`, передавая ему указатель на экземпляр класса и идентификатор функции. Идентификатор функции был передан функции-ловушке входа. Теперь он находится в стеке на 16 (0x10) байт выше: до трех сохраненных регистров и адреса возврата. Команда `PUSH [ESP + 10h]` помещает в стек его копию для передачи функции `CFlowTrace::FuncEnter`. Внимательные читатели заметили, что в объявлении функции `CFlowTrace::FuncEnter` указано, что она принимает только один параметр. Это объясняется тем, что в методы классов C++ всегда сначала передается указатель на экземпляр класса (или указатель `this`); это скрытый параметр. Я пытался написать на встроенном ассемблере функцию-ловушку меньшего объема, но мне кажется, что функцию, представленную в листинге 11-1, уменьшить уже невозможно.

Листинг 11-1. Пример функции-ловушки

```
void __declspec ( naked ) NakedEnter ( FunctionID /*funcID*/ )
{
    __asm
    {
        PUSH EAX                // Сохранение изменяемых регистров.
        PUSH ECX
        PUSH EDX

        PUSH [ESP + 10h]        // В стек в качестве параметра
                                // помещается идентификатор функции.
        MOV ECX , g_pFlowTrace  // В стек помещается указатель
        PUSH ECX                // на экземпляр класса.
        CALL CFlowTrace::FuncEnter // Вызов метода FuncEnter.

        POP EDX                 // Восстановление сохраненных
                                // регистров.

        POP ECX
        POP EAX
```

см. след. стр.

```
RET 4                                // Возврат и удаление из стека
                                    // полученного идентификатора
                                    // функции.
    }
}
```

Встраивание

При обсуждении уведомлений от функций-ловушек нельзя не рассмотреть вопрос встраивания (inlining). Ядро исполняющей подсистемы CLR оптимизировано, поэтому, чтобы сэкономить пару тактов процессора, оно очень часто будет встраивать методы прямо в код. Это значит, что вы увидите сообщения о вызовах и возвратах не для всех методов, а только для тех, которые не были встроены.

Если вы хотите получить полный список всех вызовов программы, есть два способа отключения встраивания. Однако, как вы можете представить, запрещение встраивания может заметно ухудшить быстроту выполнения управляемого кода. Проще всего отключить встраивание — установив при помощи операции ИЛИ флаг `OR_PRF_DISABLE_INLINING` в битовой маске, передаваемой методу `ICorProfilerInfo::SetEventMask` при обработке уведомления `ICorProfilerCallback::Initialize`. Недостаток этого метода в том, что флаг `COR_PRF_DISABLE_INLINING` нельзя изменить, поэтому вы выключите его на все время жизни процесса независимо от того, где выполняется ваш код.

Второй способ предоставляет более точный контроль над встраиванием, но требует гораздо больше работы. В число получаемых вами уведомлений JIT входит уведомление `JITInlining`, которое, как можно догадаться по его названию, указывает, что функция встраивается в другую функцию (для получения уведомлений JIT нужно операцией ИЛИ установить в маске событий (event mask) флаг `COR_PRF_MONITOR_JIT_COMPILATION`). `JITInlining` имеет такие параметры: `FunctionID` вызывающей функции, `FunctionID` вызываемой (встраиваемой функции) и указатель на `BOOL`, который при установке в `FALSE` предотвращает встраивание.

Уведомление `JITInlining` позволяет сделать очень интересные вещи. Например, вы можете оставить встраивание включенным для классов библиотеки классов .NET Framework (.NET Framework class library, FCL), отключив его для другого кода. Однако при этом нужно быть внимательным, так как CLR вызывает `JITInlining` огромное количество раз и, если ваш код каждый раз будет просматривать значения `FunctionID` вызывающей и вызываемой функции, это приведет к куда более серьезному снижению быстродействия, чем выключение встраивания для всего процесса. Вы можете рассмотреть вариант сохранения интересующих вас значений `FunctionID`, но помните, что из-за сборки мусора CLR они могут измениться, поэтому для поддержания таблиц данных в правильном состоянии вам придется обрабатывать уведомления сборки мусора.

Преобразователь идентификаторов функций

В дополнение к очень полезным функциям-ловушкам мне нужно рассказать вам про еще одну специальную функцию — `FunctionIDMapper`. Ее предназначение заключается в изменении значений `FunctionID`, передаваемых трем названным выше

функциям-ловушкам. CLR вызывает ее перед любой из функций-ловушек. Вы не обязаны применять `FunctionIDMapper`, однако это может открыть перед вами очень интересные возможности.

Изменение значений `FunctionID` при помощи `FunctionIDMapper` можно выполнить только один раз; это следует делать в методе `ICorProfilerCallback::Initialize` путем передачи указателя на функцию методу `ICorProfilerInfo::SetFunctionIDMapper`. В свое время у меня возникли проблемы из-за того, что прототип этой функции в файле `Profiling.DOC` описан неверно. `FunctionIDMapper` возвращает тип `UINT_PTR`, соответствующий `FunctionID`, а не указанный в документе `void`. Вот правильный ее прототип:

```
UINT_PTR __stdcall FunctionIDMapper ( FunctionID functionId ,  
                                     BOOL *pbHookFunction ) ;
```

Интересно, что `FunctionIDMapper` использует стандартное соглашение вызова, а не соглашение `naked`, как функции, требуемые другими функциями-ловушками. Параметр `FunctionID` — это функция, для которой CLR вызывает одну из функций-ловушек. Указатель на `Boolean` позволяет указать CLR, следует ли ей на самом деле вызывать функцию-ловушку. Если вы хотите разрешить вызов ловушки, присвойте `*pbHookFunction` значение `TRUE`. Если вы установите его в `FALSE`, функция-ловушка вызываться не будет. Чтобы изменить значение, передаваемое в качестве параметра функции-ловушке, нужно возвратить это значение из `FunctionIDMapper`.

Мне кажется, что `FunctionIDMapper` будет интересной прежде всего тому, кто работает с `Profiling API` в рамках крупных проектов. Так, почти при всех вызовах функций-ловушек вы должны просматривать имена функций и методов. Вместо этого можно задействовать `FunctionIDMapper`, которая будет просматривать функции и передавать нужные значения функции-ловушке. При этом просмотр функций будет выполняться в одном месте.

Благодаря контролю над действительными вызовами функций-ловушек вы получаете в свое распоряжение еще больше возможностей. Так, если вам нужно протоколировать или анализировать выполнение только одного потока, то при помощи `FunctionIDMapper` вы можете определить идентификатор потока и, если он вас не интересует, пропустить функцию-ловушку. Возможность пропуска функции-ловушки облегчит реализацию программы профилирования. Я сам воспользовался этим преимуществом при написании программы `FlowTrace`.

Использование FlowTrace

Я ознакомил вас с основами установки ловушек при помощи `Profiling API` и теперь хочу перейти к описанию работы с `FlowTrace`. В результате этого вы лучше поймете некоторые вопросы реализации этой утилиты. Прежде всего хочу отметить, что для настройки и запуска любых программ, работающих с `Profiling API`, нужно задать много переменных среды. Как и утилита `ExceptionMon` из главы 10, `FlowTrace` позволяет определить, хотите ли вы выполнить вызов `DebugBreak` в начале программы (`FLOWTRACEBREAK`), а также указать, какой именно процесс вы желаете профилировать (`FLOWTRACEPROC`). Кроме того, при помощи переменной среды `FLOWTRACEFILEDIR` можно указать конкретный каталог, в котором будут создаваться файлы вывода. Файл настройки `FlowTrace`, имеющий расширение `.FLS`, я опишу чуть

ниже, а пока скажу, что, если вы зададите переменную `FLOWTRACEFILEDIR`, `FlowTrace` будет искать файл `.FLS` в указанном вами каталоге, а не там, где находится ваш исполняемый файл.

Чтобы облегчить работу с многопоточными приложениями, `FlowTrace` записывает сведения о потоках в разные файлы. Имена файлов формируются так:

```
<имя процесса>_<ID процесса Win32>_<ID управляемого потока>.FLOW
```

Отдельные части имени файла говорят сами за себя. Интерес представляет только то, что вместо идентификатора потока `Win32` я использовал идентификатор управляемого потока. Как вы увидите в разделе, посвященном реализации `FlowTrace`, взаимосвязь управляемых потоков и потоков `Win32` отсутствует.

Наконец, перед использованием `FlowTrace` можно сконфигурировать необязательный файл параметров. Это простой файл инициализации, расположенный в том же каталоге, что и исполняемый файл, или в каталоге, указанном в переменной среды `FLOWTRACEFILEDIR`. Он называется так же, как и программа, только имеет расширение `.FLS`. Я мог бы задействовать для этого ультрасовременный файл XML, но мне не хотелось проводить несколько месяцев за написанием и тестированием кода C++, работающего с `MSXML3.DLL`, и раздувать рабочий набор `FlowTrace` десятками мегабайт. В усложнении вещей нет никакого смысла, если их можно сделать простыми.

Первый из трех необязательных параметров `FlowTrace` позволяет включить/отключить встраивание. По умолчанию встраивание включено, благодаря чему все процессы, выполняемые под `FlowTrace`, работают быстрее. Второй параметр предназначен для запрещения протоколирования потока финализации (`finalizer thread`). Все процессы .NET имеют сборщик мусора, работающий в отдельном потоке. Я обнаружил, что большинство вызовов потока финализации связано с очисткой объектов, созданных CLR. Чтобы минимизировать объем выводимой информации, я решил регистрировать только реальные действия процесса, выполняемые в других потоках, потому что мне кажется, что это дает более полезные сведения. Итак, по умолчанию `FlowTrace` не регистрирует поток финализации, но позволяет с легкостью включить эту функцию. Третий параметр определяет, протолировать ли стартовый код для первоначальной `AppDomain`, создаваемой основным потоком, так как метод `System.AppDomain.SetupDomain` создает большой объем вывода. По умолчанию я не регистрирую стартовый код. В листинге 11-2 приведен файл `.FLS` по умолчанию. Чтобы включить конкретный параметр, присвойте ему 1; чтобы отключить — 0.

Листинг 11-2. Файл `.FLS` по умолчанию

```
; Пример конфигурационного файла .FLS. Назовите этот файл
; так же, как и исполняемый, и поместите его в тот же каталог.

; Здесь указаны все общие параметры. Все они имеют
; значения по умолчанию, принимаемые, если исполняемый
; файл не имеет соответствующего файла .FLS.
[General Options]
```

```
; Значение 1 отключает встраивание. Это приведет к получению  
; гораздо большего объема информации о выполнении программы.  
TurnOffInlining=0  
  
; Отключить обработку потока финализации.  
IgnoreFinalizerThread=1  
  
; Пропускать все вызовы при создании AppDomain в основном потоке.  
SkipStartupCodeOnMainThread=1
```

Некоторые сведения о реализации FlowTrace

Теперь я хочу обсудить некоторые вопросы реализации FlowTrace. Первая проблема заключалась в том, что в будущем между управляемыми потоками и потоками Win32 однозначного соответствия не будет. В первых версиях Microsoft .NET Framework такое соответствие имеется. Сначала я реализовал FlowTrace при помощи надежного варианта локальной памяти потока, гарантирующего, что каждый поток имеет специфические данные. Однако, изучая Profiling.DOC, я заметил специальное уведомление о потоке `ICorProfilerCallback::ThreadAssignedToOSThread`. В описании говорится: «Во время выполнения конкретный поток исполняющей среды может переключаться между различными потоками, что зависит от исполняющей среды и внешних компонентов, выполняющихся в процессе». Конечно, это не могло не привлечь моего внимания, и после консультации с программистами Microsoft я понял, что простое решение с локальной памятью потока в будущем работать не будет.

К счастью, уведомления интерфейса `ICorProfilerCallback` о создании и уничтожении потока предоставляют идентификатор управляемого потока в качестве параметра; кроме того, узнать идентификатор управляемого потока в любое время позволяет `ICorProfilerInfo::GetCurrentThreadID`, так что идентификация управляемого потока проблем не представляет. Обратная сторона такого подхода заключалась в том, что мне нужно было создать собственную «локальную память управляемого потока» при помощи глобального класса отображения (map) библиотеки стандартных шаблонов (STL). Конечно, для предотвращения проблем с несколькими потоками я должен был сделать ее критической секцией. Многие методы обратного вызова интерфейса `ICorProfilerCallback` в Profiling API очень негативно относятся к блокировке при их обработке, поэтому я был немного смущен. Однако длительное тестирование позволяет мне утверждать, что это не оказывает заметного эффекта.

Вторая проблема была связана с тем, как пропускать стартовые вызовы, выполняемые методом `System.AppDomain.SetupDomain` в основном потоке. Поэкспериментировав с многочисленными управляемыми приложениями, я заметил, что в начале приложение включает три потока. В документации к Profiling API упоминается, что при внедрении программы профилирования в управляемый процесс для ее старта выделяется специальный поток, который, к счастью, не выполняет управляемого кода. Я обнаружил, что первое уведомление о создании потока всегда относилось к основному потоку приложения, а второе — к потоку финализации.

Узнав, как идентифицировать потоки, я смог составить план пропуска стартового кода. Для этого мне нужно было не регистрировать действий основного потока, пока функция-ловушка выхода не увидит вызова `System.AppDomain.ResetBindingRedirects`.

Я мог видеть, какой поток выполняет функцию финализации, но я хотел также, чтобы его можно было игнорировать. Сначала я хотел установить ловушку `FunctionIDMapper`, чтобы можно было проверять идентификатор управляемого потока. Если бы этот идентификатор соответствовал потоку финализации, я присваивал бы параметру `pbHookFunction` значение `FALSE`, чтобы CLR не вызывала функцию-ловушку. При тестировании этого способа на простейшей программе, написанной на промежуточном языке, все работало великолепно.

Однако, тестируя `FlowTrace` с простым приложением `Microsoft Windows Forms`, я получил сообщения об ошибке, утверждавшие, что специфичные для управляемого потока данные имеют значение `NULL`. Я игнорировал поток финализации, поэтому я не добавлял его в отображение управляемых потоков: я хотел, чтобы оно имело минимальный размер. Изучая эти ошибки, я заметил, что для потока финализации всегда вызывалась функция-ловушка входа. Я решил, что допустил в алгоритме какую-то ошибку, но в результате тщательной проверки так ничего и не обнаружил.

Зарегистрировав только эти специфические проблемы с потоком финализации и проштудировав документацию, я наконец выяснил, в чем дело. В приложениях `Windows Forms` все еще присутствуют некоторые странности COM, в том числе недостатки моделей разделенных потоков. То, что я видел, было кросс-поточными вызовами с маршалингом из основного потока в поток финализации. Интересно, что CLR никогда не вызывала ловушку `FunctionIDMapper`. Значит, у меня не было способа заблокировать вызовы ловушек. Я надеялся, что мне не придется проверять поток финализации в функциях-ловушках, чтобы не снижать быстродействия программы, но ничего не оставалось делать. Поэтому для избежания протоколирования вызовов финализации я должен был проверять поток финализации.

Все получилось, и я смог при необходимости отключать протоколирование финализации. Где-то через день я понял, что вызывать `FunctionIDMapper` нужно, только когда пользователь специально запрашивает, чтобы я не отслеживал поток финализации. Первоначально я делал это во всех случаях.

Последняя проблема, с которой я должен был справиться, состояла в том, чтобы гарантировать правильность выводимой информации в любых обстоятельствах. Это означало, что я должен был регистрировать для функций все «разворачиваемые» исключения, так как я никогда не встречал для них функцию-ловушку выхода. Задача оказалась достаточно простой. Для этого нужно было только вести для потока счетчик развертывания, когда CLR вызывала `ICorProfilerCallback::ExceptionUnwindFunctionLeave`. Как только исключение достигало `ICorProfilerCallback::ExceptionCatcherLeave`, я просто вычитал число развернутых функций для текущего уровня программы.

Что после FlowTrace

FlowTrace — очень полезное средство обучения. Но, как и все утилиты, ее можно сделать еще лучше. Если вам нужен интересный проект, вот список некоторых отличных функций, которые вы могли бы попытаться добавить в FlowTrace.

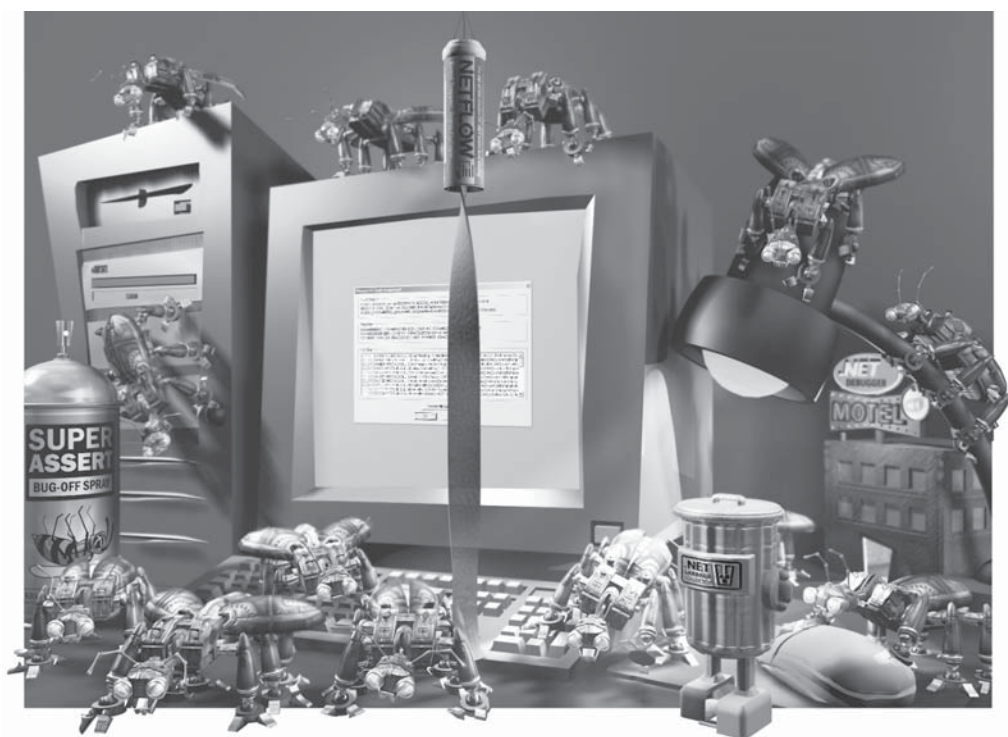
- Добавьте в файл .FLS функцию, позволяющую начинать и останавливать протоколирование для конкретных классов и методов. Благодаря этому вы сможете точно указывать, что вы желаете регистрировать, не путаясь среди всех остальных сообщений. В идеале эта функция должна поддерживать протоколирование как для отдельных, так и для всех потоков. Еще лучше реализовать указание интересующих вас классов и методов при помощи регулярных выражений; так вы сможете еще точнее определять, что нужно и не нужно регистрировать.
- Было бы неплохо начинать работу вообще без протоколирования и запускать его в конкретной точке при помощи внешнего события. Конечно, нужно реализовать и остановку протоколирования!
- Вы можете добавить в FlowTrace возможность вывода времени выполнения функций. Вся необходимая для этого инфраструктура в FlowTrace уже реализована.
- Вместо сохранения вывода в текстовом файле вы могли бы записывать и отображать информацию в псевдореальном времени при помощи приложения с графическим интерфейсом.
- Наконец, файлы вывода FlowTrace могут быть очень объемными. Неплохо было бы иметь возможность фильтрации частых базовых вызовов, таких как `Object..ctor`.

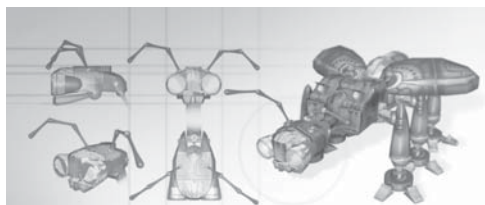
Резюме

Определить последовательность вызовов по исходному коду иногда очень трудно. Надеюсь, FlowTrace сделает слежение за потоком выполнения ваших управляемых приложений более простой задачей, и вы сможете оптимизировать отладку и настройку производительности своих программ. Подумайте, какие другие утилиты можно разработать на основе Profiling API, настолько облегчающего установку ловушек для входа и выхода из функций. Конечно, у Profiling API есть и недостатки, но я все равно считаю, что это одно из самых удивительных средств платформ .NET.

ЧАСТЬ IV

МОЩНЫЕ СРЕДСТВА И МЕТОДЫ ОТЛАДКИ НЕУПРАВЛЯЕМОГО КОДА





Нахождение файла и строки ошибки по ее адресу

Ваша программа потерпела крах. ОС оказалась достаточно любезной и предоставила вам адрес ошибки. Что дальше? Мой друг Крис Селлз (Chris Sells) называет такой сценарий проблемой «моя программа ушла в отпуск, оставив мне только этот никчемный адрес». Конечно, иметь адрес ошибки лучше, чем не иметь ничего, но знать исходный файл и номер строки ошибки было бы еще полезнее. Вы можете предоставить исходные коды своим клиентам, чтобы они отладили проблему сами, но я не думаю, что это станет реальностью в ближайшем будущем.

Адрес ошибки — обычно все, что вы получаете, и то, если вы очень удачливы. Microsoft прилагает немалые усилия, чтобы облегчить своим сотрудникам поиск проблем с Microsoft Windows 2000/XP/Server 2003. Поиск ошибок усложняется, если ваши пользователи просто сообщают вам, что ваша программа не работает. Так, обработчик необработанных исключений, используемый по умолчанию, выводит «чудесное» диалоговое окно «Приносим извинения за неудобства» (рис. 12-1). К великому сожалению, в этом окне больше не выводится адрес ошибки! Классический пример изменения, дружественного к пользователям, но неприятного для нас, программистов.

Для получения адреса ошибки щелкните ссылку To See What Data This Error Report Contains, Click Here (для просмотра данных, содержащихся в отчете, щелкните здесь), а в следующем диалоговом окне — ссылку To View Technical Information About The Error Report, Click Here (для просмотра технических сведений об ошибке щелкните здесь). В диалоговом окне Error Report Contents (содержание отчета об ошибке) (рис. 12-2) вы увидите адрес ошибки и все загруженные модули. Кроме того, вы получите по-настоящему полезный дамп стека, достаточно большой, чтобы можно было изучить историю неправильного потока почти до начала времен. Удивительно, но кто-то принял очень странное решение и поместил всю эту замеча-

тельную информацию в статический элемент управления, поэтому скопировать ее нельзя. Надеюсь, в следующем пакете обновлений для Windows XP эта досадная оплошность будет исправлена. Есть и хорошие новости: Windows 2000/Server 2003 всегда выводят в диалоговом окне адрес ошибки.

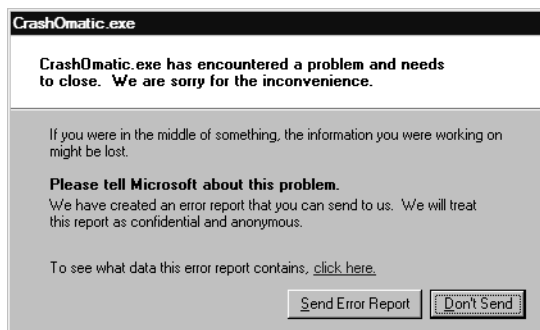


Рис. 12-1. Стандартное диалоговое окно сообщения об ошибке Windows XP

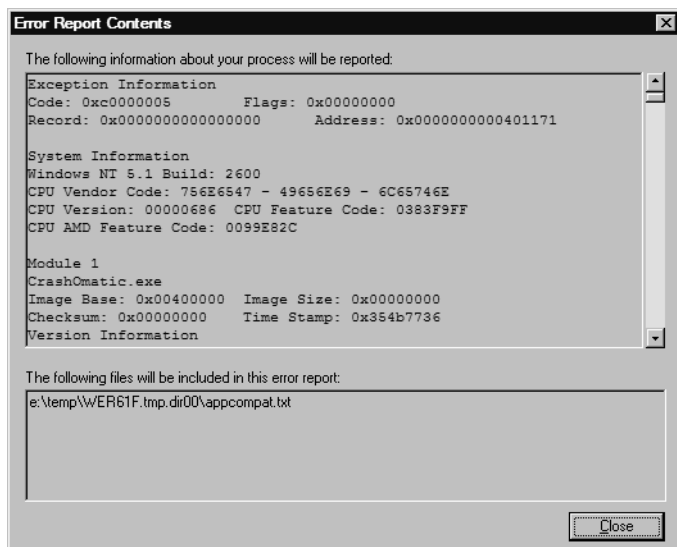


Рис. 12-2. Диалоговое окно «Содержание отчета об ошибке» Windows XP

Вас порадует и то, что независимо от нажатой в диалоговом окне кнопки будет запускаться Dr. Watson, если, конечно, он назначен стандартным отладчиком, что имеет место по умолчанию. Если это не так, пусть ваши пользователи сделают Dr. Watson отладчиком по умолчанию, выполнив команду `DRWTSN32.EXE -i`. Чтобы получать информацию Dr. Watson, вы должны объяснить пользователям, как запускать Dr. Watson и копировать аварийный дамп памяти при помощи пользовательского интерфейса Dr. Watson (подробнее о журнале регистрации Dr. Watson и его интерпретации см. приложение А). Вы будете очень довольны, если пользователи смогут прислать вам минидампы, поэтому попросите их установить в окне Dr. Watson флажок Create Crash Dump File (создание файла аварийной копии па-

мией) и запомнить место записи файлов. Конечно, большинство из нас не боится получить адрес ошибки. Если вы обо всем позаботились заранее, то при ошибке ваши приложения автоматически будут высылать вам минидампы, о которых я расскажу в главе 13. Очевидно, что адрес ошибки нужно как-то преобразовать в имя исходного файла и номер некорректной строки. Этой теме и посвящена данная глава. Я расскажу про два основных способа преобразования адреса в нужную информацию: при помощи MAP-файлов и при помощи утилиты CrashFinder, прилагаемой к книге.

Чтобы получить максимум от рассматриваемых в этой главе методов, настройте компилятор, как указано в главе 2. Все заключительные компоновки нужно создавать с полным набором отладочных символов и MAP-файлами. Кроме того, вам нужно разрешать все конфликты с адресами загрузки DLL. Если эти условия соблюдены не будут, то описанные ниже методы окажутся бесполезными, и определить исходный файл и строку ошибки вы сможете только путем гадания.

Создание и чтение MAP-файла

Меня часто спрашивают, почему я так настойчиво рекомендую создавать MAP-файлы для заключительных компоновок. Если коротко, то MAP-файлы — единственный способ представления данных о глобальных символах, исходных файлах и номерах строк вашей программы в текстовом виде. Использовать CrashFinder проще, зато MAP-файлы можно читать всегда и везде, и они не требуют никакой поддерживающей программы и двоичных файлов вашего приложения, предоставляя ту же информацию. Поверьте, рано или поздно вам понадобится найти расположение ошибки в более старой версии вашей программы, и у вас будет только один способ сделать это — изучить MAP-файл.

MAP-файлы полезны только в случае заключительных компоновок, потому что при создании MAP-файла компоновщик вынужден отключать компоновку с приращением. Включить генерирование MAP-файлов в Microsoft Visual C++ .NET гораздо проще, чем в предыдущих версиях Visual Studio. Откройте диалоговое окно Property Pages, папку Linker, страницу Debugging и просто выберите значение Yes в полях Generate Map File (генерировать MAP-файл), Map Exports (включать в MAP-файл информацию об экспортируемых функциях) и Map Lines (включать в MAP-файл информацию о номерах строк). Это установит ключи компоновщика /MAP, /MAPINFO:EXPORTS и /MAPINFO:LINEs. Как вы, наверное, догадались, программа Settings-Master из главы 9 с файлами проекта по умолчанию сделает это автоматически.

Если вы работаете над реальным проектом, то, вероятно, сохраняете двоичные файлы в отдельном каталоге. По умолчанию компоновщик записывает MAP-файл в тот же каталог, что и промежуточные файлы, поэтому вам нужно явно указать, чтобы он хранился вместе с двоичными файлами. Для этого можно ввести в поле Map File Name (имя MAP-файла) выражение `$(OutDir)/$(ProjectName).map`. `$(OutDir)` — это встроенный макрос, который система сборки программы заместит именем реального каталога вывода, а вместо `$(ProjectName)` будет использовано название проекта. На рис. 12-3 показаны все значения параметров MAP-файлов для заключительной компоновки проекта MapDLL, прилагаемого к книге.

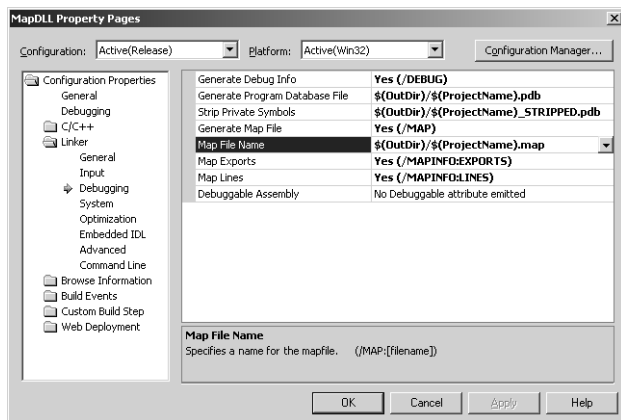


Рис. 12-3. Значения параметров MAP-файлов в диалоговом окне *Property Pages*

Вполне возможно, что в повседневной работе MAP-файлы вам не понадобятся, но почти наверняка они потребуются вам в будущем. Работа CrashFinder и вашего отладчика основана на таблице символов, которую они читают при помощи сервера символов. Если формат таблицы символов изменится или вы забудете сохранить файлы базы данных программы (Program Database, PDB), у вас возникнут крупные неприятности. Ответственность за сохранение PDB-файлов лежит на вас, но формат таблиц символов вам неподвластен. Он часто изменяется. Например, многие люди, перешедшие с Microsoft Visual Studio 6 на Microsoft Visual Studio .NET, заметили, что такие средства, как CrashFinder, отказываются работать с программами, откомпилированными при помощи Visual Studio .NET. Microsoft изменила формат таблицы символов и делает это регулярно. В таких случаях единственное ваше спасение — MAP-файлы.

Даже если лет через пять вы будете писать программы с помощью Visual Studio .NET 2007 Service Pack 6 для Windows Server 2008, я абсолютно уверен, что некоторые ваши клиенты еще будут работать с вашими приложениями, созданными в 2003 году. Когда они обратятся к вам за помощью и предоставят адрес ошибки, несколько дней вам понадобится только на то, чтобы найти компакт-диски со старой версией Visual Studio .NET, нужной для чтения сохраненных PDB-файлов. Если у вас будут MAP-файлы, вы найдете место проблемы за пять минут.

Содержание MAP-файла

Пример MAP-файла показан в листинге 12-1. В начале MAP-файла указывается имя модуля, время компоновки и предпочтительный адрес загрузки. После заголовка располагается информация о разделах, показывающая, какие разделы созданы компоновщиком из файлов OBJ и LIB.

После информации о разделах находятся действительно полезные данные: информация об открытых (public) функциях. Обратите внимание на слово «открытых». Если у вас есть статические функции, сведения о них сохраняются в аналогичной таблице после таблицы об открытых функциях. К счастью, их номера строк не разделяются и выводятся вместе.

Важные сведения заключены в именах открытых функций и данных столбца Rva+Base, содержащего стартовые адреса функций. Буква `f` после некоторых адресов Rva+Base означает, что адрес соответствует действительной функции, а не глобальной переменной или какой-либо импортируемой функции. За разделом открытых функций следует информация о строках в формате:

24 0001:00000006

Первое число — номер строки, а второе — смещение от начала раздела кода, к которому эта строка относится. Согласен, это звучит несколько запутанно, поэтому позднее я объясню, что нужно сделать для преобразования адреса в исходный файл и номер строки.

Если модуль содержит экспортируемые функции, их список будет приведен в заключительном разделе MAP-файла. Такую же информацию вы получите, запустив программу `DUMPBIN` с параметрами `/EXPORTS <имя модуля>`.

Листинг 12-1. Пример MAP-файла

```
MapDLL

Timestamp is 3e2b44a3 (Sun Jan 19 19:36:51 2003)

Preferred load address is 03900000

Start          Length      Name                                Class
0001:00000000  00000304H  .text                              CODE
0002:00000000  00000028H  .idata$5                            DATA
0002:00000030  000000f8H  .rdata                              DATA
0002:00000128  00000063H  .rdata$debug                        DATA
0002:00000190  00000004H  .rdata$sxdata                      DATA
0002:00000194  00000004H  .rtc$IAA                            DATA
0002:00000198  00000004H  .rtc$IZZ                            DATA
0002:0000019c  00000004H  .rtc$TAA                            DATA
0002:000001a0  00000004H  .rtc$TZZ                            DATA
0002:000001a4  00000014H  .idata$2                            DATA
0002:000001b8  00000014H  .idata$3                            DATA
0002:000001cc  00000028H  .idata$4                            DATA
0002:000001f4  00000082H  .idata$6                            DATA
0002:00000280  0000007bH  .edata                              DATA
0003:00000000  00000004H  .CRT$XCA                            DATA
0003:00000004  00000004H  .CRT$XCZ                            DATA
0003:00000008  00000004H  .CRT$XIA                            DATA
0003:0000000c  00000004H  .CRT$XIZ                            DATA
0003:00000010  00000004H  .data                              DATA
0003:00000014  00000014H  .bss                               DATA


Address          Publics by Value                    Rva+Base    Lib:Object
0000:00000001  ___safe_se_handler_count          00000001    <absolute>
0001:00000000  _DllMain@12                       03901000 f    MapDLL.obj
0001:00000006  ?MapDLLFunction@@YAHXZ           03901006 f    MapDLL.obj
```

```

0001:00000023 ?MapDLLHappyFunc@@YAPADPAD@Z 03901023 f MapDLL.obj
0001:0000003c __CRT_INIT@12 0390103c f MSVCRT:crtldll.obj
0001:000000fa __DllMainCRTStartup@12 039010fa f MSVCRT:crtldll.obj
0001:000001de __initterm 039011de f MSVCRT:MSVCR71.dll
0001:000001e4 __onexit 039011e4 f MSVCRT:atonexi t.obj
0001:0000020a __atexit 0390120a f MSVCRT:atonexi t.obj
0001:0000021c __RTC_Initialize 0390121c f MSVCRT:initsec t.obj
0001:00000260 __RTC_Terminate 03901260 f MSVCRT:initsec t.obj
0001:000002a4 ___CxxXcptFilter 039012a4 f MSVCRT:MSVCR71.dll
0001:000002ac __SEH_prolog 039012ac f MSVCRT:sehprol g.obj
0001:000002e7 __SEH_epilog 039012e7 f MSVCRT:sehprol g.obj
0001:000002f8 __except_handler3 039012f8 f MSVCRT:MSVCR71.dll
0001:000002fe ___dllonexit 039012fe f MSVCRT:MSVCR71.dll
0002:00000000 __imp_printf 03902000 MSVCRT:MSVCR71.dll
0002:00000004 __imp_free 03902004 MSVCRT:MSVCR71.dll
0002:00000008 __imp__initterm 03902008 MSVCRT:MSVCR71.dll
0002:0000000c __imp_malloc 0390200c MSVCRT:MSVCR71.dll
0002:00000010 __imp__adjust_fdiv 03902010 MSVCRT:MSVCR71.dll
0002:00000014 __imp___CxxXcptFilter 03902014 MSVCRT:MSVCR71.dll
0002:00000018 __imp__except_handler3 03902018 MSVCRT:MSVCR71.dll
0002:0000001c __imp___dllonexit 0390201c MSVCRT:MSVCR71.dll
0002:00000020 __imp__onexit 03902020 MSVCRT:MSVCR71.dll
0002:00000024 \177MSVCR71_NULL_THUNK_DATA 03902024 MSVCRT:MSVCR7 1.dll
0002:0000007c ??_C@_OCE@EBHAKCA@Whoops?0?5a?5crash?5is?5about?5to?5 occu@
0390207c MapDLL.obj
0002:000000a0 ??_C@_OCD@OILENIK0@Hello?5from?5InternalStaticFunction@
039020a0 MapDLL.obj
0002:000000c4 ??_C@_OBM@DFMPKPOD@Hello?5from?5MapDLLFunction?$CB?6?$ AA@
039020c4 MapDLL.obj
0002:000000e0 __load_config_used 039020e0 MSVCRT:loadcfg .obj
0002:00000190 ___safe_se_handler_table 03902190 <linker-defined>
0002:00000194 ___rtc_iaa 03902194 MSVCRT:initsec t.obj
0002:00000198 ___rtc_izz 03902198 MSVCRT:initsec t.obj
0002:0000019c ___rtc_taa 0390219c MSVCRT:initsec t.obj
0002:000001a0 ___rtc_tzz 039021a0 MSVCRT:initsec t.obj
0002:000001a4 __IMPORT_DESCRIPTOR_MSVCRT71 039021a4 MSVCRT:MSVCR7 1.dll
0002:000001b8 __NULL_IMPORT_DESCRIPTOR 039021b8 MSVCRT:MSVCR71.dll
0003:00000000 ___xc_a 03903000 MSVCRT:cinitex e.obj
0003:00000004 ___xc_z 03903004 MSVCRT:cinitex e.obj
0003:00000008 ___xi_a 03903008 MSVCRT:cinitex e.obj
0003:0000000c ___xi_z 0390300c MSVCRT:cinitex e.obj
0003:00000010 ___security_cookie 03903010 MSVCRT:seccook .obj
0003:00000018 __adjust_fdiv 03903018 <common>
0003:0000001c ___onexitend 0390301c <common>
0003:00000020 ___onexitbegin 03903020 <common>
0003:00000024 __pRawDllMain 03903024 <common>

```

entry point at 0001:000000fa

Static symbols

см. след. стр.

```
0001:00000016      ?InternalStaticFunction@@YAXXZ 03901016 f   MapDLL .obj
```

```
Line numbers for .\Release\MapDLL.obj(d:\dev\booktwo\disk\chapter
examples\chapter 12\mapfile\mapdll\mapdll.cpp) segment .text
```

```
11 0001:00000000  20 0001:00000000  21 0001:00000003  26 0001:00000006
25 0001:00000006  27 0001:00000012  28 0001:00000015  31 0001:00000016
32 0001:00000016  33 0001:00000022  37 0001:00000023  36 0001:00000023
38 0001:00000028  39 0001:00000033  41 0001:0000003b
```

```
Line numbers for R:\VSNET2003\VC7\lib\MSVCRT.lib(f:\vs70builds\2292\vc
\crtbld\crt\src\atoneexit.c) segment .text
```

```
81 0001:000001e4  76 0001:000001e4  90 0001:00000209  96 0001:0000020a
95 0001:0000020a  97 0001:0000021b
```

```
Line numbers for R:\VSNET2003\VC7\lib\MSVCRT.lib(f:\vs70builds\2292\vc
\crtbld\crt\src\crt.dll.c) segment .text
```

```
134 0001:0000003c  129 0001:0000003c  135 0001:00000044  136 0001:0000 004c
158 0001:00000052  163 0001:00000065  168 0001:0000007a  170 0001:0000 007e
172 0001:00000081  178 0001:0000008b  179 0001:00000090  184 0001:0000 009a
189 0001:000000ab  192 0001:000000b2  219 0001:000000b8  220 0001:0000 00c1
225 0001:000000c3  226 0001:000000cf  234 0001:000000e5  236 0001:0000 00ec
234 0001:000000f3  240 0001:000000f4  241 0001:000000f7  249 0001:0000 00fa
250 0001:00000106  252 0001:0000010c  257 0001:00000111  258 0001:0000 011e
260 0001:00000124  262 0001:0000012d  263 0001:00000136  265 0001:0000 0142
266 0001:0000014b  268 0001:0000015a  269 0001:0000015c  272 0001:0000 015e
275 0001:0000016e  283 0001:00000177  286 0001:00000181  288 0001:0000 018a
289 0001:00000198  291 0001:0000019b  292 0001:000001a9  298 0001:0000 01b7
294 0001:000001bc  295 0001:000001d4  299 0001:000001d6
```

Exports

```
ordinal    name
```

```
1      ?MapDLLFunction@@YAHXZ (int __cdecl MapDLLFunction(void))
2      ?MapDLLHappyFunc@@YAPADPAD@Z (char * __cdecl MapDLLHappyFunc(char *))
```

Получение информации об исходном файле, имени функции и номере строки

Алгоритм извлечения данных об исходном файле, имени функции и номере строки из MAP-файла прост, но для этого нужно выполнить несколько действий в шестнадцатеричной системе счисления. Допустим, ошибка произошла по адресу 0x03901038 в модуле MAPDLL.DLL из листинга 12-1.

Прежде всего из MAP-файлов проекта нужно выбрать файл, содержащий адрес ошибки. Для этого достаточно взглянуть на предпочтительный адрес загруз-

ки и последний адрес в разделе открытых функций. Если адрес ошибки попадает в этот диапазон, значит, это и есть нужный вам MAP-файл.

Чтобы определить нужную функцию, найдите в столбце Rva+Base первую функцию с адресом, превышающим адрес ошибки. Предыдущий элемент в MAP-файле и будет функцией, в которой случилась ошибка. В нашем случае это функция `?MapDLLHappyFunc@@YAPADPAD@Z` с адресом `0x03901023`. Любое имя функции, начинающееся с вопросительного знака, — это расширенное имя C++.

Вы, возможно, удивляетесь, почему я не упоминал про расширение имен C++ в главе 6, когда рассказывал про расширения имен для различных соглашений вызова. Хотя оба типа расширений играют сходную роль, их источники различны. Расширения имен соглашений вызова просто указывают генератору кода, по каким правилам создавать код занесения параметров в стек и очистки стека, и основаны на определениях ОС. Расширение имен C++ обусловлено языком. C++ допускает перегрузку методов, поэтому компилятор должен как-то их различать. Для этого он «расширяет» имя метода при помощи информации о типе возвращаемого значения, соглашении вызова и параметрах. Так компилятор сможет точно узнать, какую функцию вы хотите вызвать. Для преобразования имени передайте его в командной строке программе `UNDNAME.EXE` из Visual Studio .NET. В нашем примере `?MapDLLHappyFunc@@YAPADPAD@Z` преобразуется в `char * __cdecl MapDLLHappyFunc(char *)`. Некоторые из вас, вероятно, смогли определить, что первоначальным именем функции было `MapDLLHappyFunc`, только по ее расширенному имени. Другие расширенные имена C++ расшифровать труднее, особенно в случае перегруженных функций.

Для определения смещения строки нужно выполнить простое шестнадцатеричное вычитание по формуле:

(адрес ошибки) – (предпочтительный адрес загрузки) – `0x1000`

Помните: адреса представляют собой смещения от начала первого раздела кода, поэтому нужное преобразование выполняется именно по такой формуле. Вы скорее всего догадались, зачем вычитается предпочтительный адрес загрузки, но знаете ли вы, почему нужно вычесть еще и `0x1000`? Адрес ошибки — это смещение от начала раздела кода, но раздел кода не является первой частью двоичного файла. Первая часть двоичного файла состоит из заголовка PE и связанной с ним загрузки DOS, которые занимают `0x1000` байт. Да, все двоичные файлы Win32 по-прежнему вынуждены бороться с тяжелым наследием MS-DOS.

Мне неизвестно, почему компоновщик генерирует MAP-файлы, требующие этого вычисления. Разработчики компоновщика включили столбец Rva+Base в MAP-файл недавно, поэтому я не понимаю, что им помешало подкорректировать номера строк.

Как только вы подсчитали смещение, ищите в разделе строк наибольшее число, не превышающее найденное значение. Помните, что при генерировании кода компилятор может перемешивать его, так что номера строк не всегда располагаются в восходящем порядке. Вот что получается в моем примере:

`0x03901038 – 0x03900000 – 0x1000 = 0x38`

Просмотрев листинг 12-1, вы увидите, что самое большое смещение, не превышающее `0x38`, входит в выражение `39 0001:00000033` (строка 39), которое относится к файлу `MAPDLL.CPP`.

PDB2MAP: создание MAP-файлов постфактум

При обсуждении поиска адресов ошибок меня постоянно спрашивают, что делать, если значительная часть цикла разработки была пройдена без создания MAP-файлов. Другие внимательные разработчики также обращают внимание, что для получения отличных MAP-файлов нужно задавать базовые адреса всех DLL при сборке программы. Работая над проектом, приближающимся к завершению, вы, вероятно, не захотите дестабилизировать сборку приложения, изменяя некоторые параметры. Кроме того, без программы SettingsMaster из главы 9 вносить эти глобальные изменения проекта в Visual Studio неудобно, поэтому разработчики обычно предпочитают задавать базовые адреса своих DLL через REBASE.EXE.

Не оставляя без внимания ни одной интересной задачи, я рассмотрел эту проблему внимательней. Мне нужен был способ нумерации функций, файлов и строк исходного кода. Учитывая, что сервер символов DBGHELP.DLL уже включает эти возможности, следующий шаг на пути к генерированию MAP-файла из PDB-файла оказался простым.

Первая проблема, с которой я столкнулся, заключалась в том, что функции `SymGetSymNext` и `SymGetSymPrev` возвращают не то, что вы ожидаете. Я думал, что могу получить адрес в исходном файле, после чего вызывать `SymGetSymPrev` до тех пор, пока не окажусь в начале исходного файла и `SymGetSymNext`, пока не доберусь до его конца. Я забыл принять во внимание один небольшой аспект — встраиваемые функции. Эти функции и соответствующие им строки исходного кода располагаются в теле других функций, поэтому информация о номерах строк на самом деле хранится в виде диапазонов. Это означало, что для концентрации данных об исходном коде и номерах строк мне нужно было разработать схему отслеживания всех диапазонов. Как только я справился с этим препятствием, написать программу оказалось довольно просто.

Еще одна проблема с серверами символов была связана с библиотекой стандартных шаблонов. Сначала я стал разрабатывать структуры данных при помощи STL, но вскоре обнаружил, что даже частичная реализация PDB2MAP.EXE работала мучительно медленно. Эта ошибка была на моей совести, потому что я использовал класс вектора для линейного поиска, а это просто глупо. Попробовав несколько похожих вариантов, я понял, что STL всегда будет работать гораздо медленней, так как она выполняет много неявных операций выделения памяти и копирования. Поскрипев зубами и попытавшись понять некоторые подробности реализации STL, я осознал, что напрасно усложнял проблему. В конце концов я сам написал простую систему из нескольких массивов, очень быструю и крайне легкую для понимания. Она также оказалась гораздо проще для сопровождения, чем то, что я мог сделать при помощи STL.

Файлы, генерируемые PDB2MAP, очень похожи на MAP-файлы. Так как сервер символов DBGHELP.DLL не предоставляет сведений о статических функциях, мне пришлось от этого отказаться. Увидев файл .P2M, вы поймете, что проблем с его пониманием у вас не будет. Я счел ненормальную систему нумерации строк MAP-файлов пережитком прошлого, поэтому реализовал в PDB2MAP более простой и современный подход. Моя информация о строках генерируется на основании действительных адресов памяти.

Наконец, возможно, вас заинтересует содержание файла .P2M. Как я уже говорил в главе 2, компактный код — хороший код. Однако узнать влияние различных ключей компилятора на размер отдельных функций можно только по общему размеру двоичного файла. Кроме того, мы никак не можем узнать, как на конкретную функцию влияет встраивание функций. Работая над PDB2MAP, я понял, что могу также сообщать размеры символов, потому что эту возможность поддерживает сервер символов DBGHELP.DLL. Поэтому, как показано в листинге 12-2, представляющем собой сокращенный файл .P2M, после заголовочной информации между адресом и именем функции выводится также ее размер. Вы увидите размеры почти всех функций, однако DBGHELP.DLL не гарантирует, что эта информация будет возвращена, поэтому в файлах .P2M возможно появление размеров, равных 0.

Листинг 12-2. Сокращенный файл .P2M

PDB2MAP Generated Map File

Image: AssertTest

Timestamp is 3E0E7E2A -> Sat Dec 28 23:46:34 2002

Preferred load address is 00400000

Address	Size	Function
0x00401050	36	??2@YAPAXI@Z
0x00401080	260	?MyThread@@YGKPAX@Z
0x00401190	38	?SleepThread@@YGKPAX@Z
0x004011C0	535	?TestThree@@YAXPAD@Z
0x004013E0	258	?TestTwo@@YAXXZ
0x004014F0	421	?TestOne@@YAXPAG@Z
0x004016A0	453	_wWinMain@16
0x00401A5E	6	_InitCommonControls@0
0x00401A64	6	_SuperAssertionW

. . .

Line numbers for

d:\dev\booktwo\disk\bugslayerutil\tests\asserttest\asserttest.cpp

16 : 0x00401080	18 : 0x0040109F	19 : 0x004010CB	20 : 0x004 010DE
21 : 0x004010E5	22 : 0x0040113C	23 : 0x0040113E	26 : 0x004 01190
27 : 0x00401194	28 : 0x004011A8	29 : 0x004011AA	32 : 0x004 011C0
33 : 0x004011D7	39 : 0x004011DE	40 : 0x00401201	41 : 0x004 01207
43 : 0x00401223	44 : 0x0040127A	45 : 0x0040129D	46 : 0x004 012B2
47 : 0x0040131A	48 : 0x0040131F	49 : 0x00401334	50 : 0x004 0139C
53 : 0x004013E0	55 : 0x004013F7	57 : 0x0040140F	59 : 0x004 01427
60 : 0x0040143A	61 : 0x0040143C	62 : 0x0040143E	63 : 0x004 01498
64 : 0x004014A5	67 : 0x004014F0	68 : 0x00401515	70 : 0x004 01527
74 : 0x0040153E	76 : 0x00401548	78 : 0x004015CF	80 : 0x004 01632
81 : 0x00401638	82 : 0x0040163D	90 : 0x004016A0	91 : 0x004 016C4
92 : 0x0040171B	93 : 0x00401772	94 : 0x004017D2	96 : 0x004 01829
97 : 0x00401838	98 : 0x00401845	99 : 0x00401852	100 : 0x004 01854

см. след. стр.

```
Line numbers for f:\vs70builds\2292\vc\crtbld\crt\src\atonexit.c
```

```
76 : 0x00402810      81 : 0x00402814      90 : 0x0040284B      95 : 0x004 02850
96 : 0x00402853      97 : 0x00402866
:
```

Использование CrashFinder

Как вы только что увидели, читать MAP- и P2M-файлы не так уж и сложно. Однако это довольно утомительное и определенно немасштабируемое решение для других членов вашей группы, таких как сотрудники отдела контроля качества, техподдержки и даже руководители. Для улучшения масштабируемости CrashFinder я решил включить в отчеты об ошибках максимально подробную информацию, чтобы сделать его полезным для всех членов группы разработки: от программистов до тестировщиков и инженеров по сопровождению программы. Если вы следуете моим советам и правильно создаете символы отладки (см. главу 2), все члены вашей группы смогут работать с CrashFinder без всяких проблем.

При использовании CrashFinder в группе вам нужно быть особенно внимательным к поддержанию доступа к двоичным образам и связанным с ними PDB-файлам, поскольку CrashFinder не хранит информации о вашем приложении, кроме путей к двоичным файлам. Это позволяет вам использовать один проект CrashFinder на всем протяжении цикла разработки. Если бы CrashFinder хранил более подробные сведения о вашем приложении, такие как таблицы символов, то вам, вероятно, нужно было бы создавать отдельный проект CrashFinder для каждой компоновки. Если вы примете мой совет и обеспечите легкий доступ к двоичным файлам и PDB-файлам, то при возникновении ошибки тестировщики и члены группы сопровождения должны будут только запустить CrashFinder и добавить важную информацию в отчет об ошибке. Все мы знаем, что чем больше мы знаем о конкретной проблеме, тем проще будет ее решение.

Для некоторых приложений вам, наверное, захочется создать несколько проектов CrashFinder. Скажем, вы могли бы иметь один проект, указывающий на место ежедневной компоновки, а также проекты для каждой версии программы, соответствующей контрольной точке. Если вы решите включить в проект CrashFinder системные DLL, вам нужно будет создать отдельные проекты для каждой поддерживаемой вами ОС. Вам также понадобятся отдельные проекты CrashFinder для всех версий программы, отсылаемых тестировщикам за пределы группы разработки, поэтому для каждой такой версии нужно будет отдельно хранить двоичные образы и PDB-файлы.

Работа над CrashFinder оказалась весьма интересной. Говорят, он значительно облегчает труд, и я очень горжусь этим. Более того, целым рядом усовершенствований пользовательского интерфейса и функциональности CrashFinder обязан не мне, а другим талантливым программистам. К версии CrashFinder, которую вы найдете на CD, приложили руку Скотт Блум (Scott Bloom), Чинг Минг Квок (Ching Ming Kwok), Джефф Шанхольтц (Jeff Shanholtz), Рич Петерс (Rich Peters), Пабло Преседо (Pablo Presedo), Джулиан Онионс (Julian Onions) и Кен Глэдстоун (Ken Gladstone). Всем им я очень признателен.

На рис. 12-4 показан пользовательский интерфейс CrashFinder с одним из моих личных проектов. В верхней части дочернего окна — древовидный список, отображающий исполняемый файл и его DLL. Зеленые галочки говорят о том, что символы для каждого двоичного образа были загружены правильно. Если бы CrashFinder не смог загрузить символы, он указал бы на это при помощи красной буквы X и развернул проблемный элемент списка, поясняя причину проблемы.

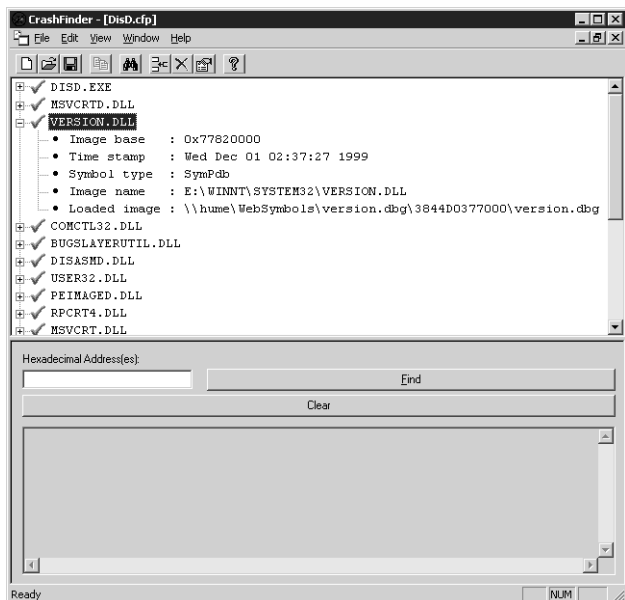


Рис. 12-4. Пользовательский интерфейс CrashFinder

Красный X появляется по трем причинам. Во-первых, это может означать, что CrashFinder не нашел PDB-файла, соответствующего двоичному файлу. Лучше всего хранить двоичные и PDB-файлы в одном месте — тогда проблем быть не должно. Вторая причина в том, что при открытии сохраненного проекта CrashFinder больше не может найти двоичный файл. Наконец, проблема может быть обусловлена конфликтом адреса загрузки файла с какой-либо из DLL проекта. ОС не допускает возможность конфликта DLL, поэтому CrashFinder также этого не позволяет. При конфликте загрузки вы можете изменить адрес конфликтующей DLL только для текущего проекта CrashFinder. Как я уже говорил, модификация базовых адресов DLL очень важна для успешного поиска ошибок.

Вся магия преобразования мистического адреса в информацию об исходном файле, функции и номере строки отображается в нижней части дочернего окна. Однако сначала я должен рассказать, как загрузить двоичные файлы в проект CrashFinder. Нажав на панели инструментов кнопку New, вы увидите стандартное диалоговое окно работы с файлами, предлагающее добавить в проект двоичный образ. Если у вас есть EXE-файл, его нужно добавить в проект в первую очередь. В диалоговом окне Add Binary Image (добавить двоичный образ) вы можете выбрать несколько двоичных файлов, включив тем самым в CrashFinder сразу весь свой проект.

После выбора пункта Open (открыть) происходит много операций. В новую версию CrashFinder внесено одно отличное усовершенствование: теперь он автоматически ищет все неявно загружаемые DLL и добавляет их в проект. Если вы явно загрузили DLL как объекты COM, добавить их можно, выбрав в меню Edit (изменить) пункт Add Image (добавить образ). CrashFinder также добавит в проект все дополнительные неявно скомпонованные модули.

Добавляя в проект двоичные образы, помните: проект CrashFinder может включать только один EXE-файл. Если приложение состоит из нескольких EXE-файлов, создайте для каждого отдельный проект CrashFinder. CrashFinder — приложение с многодокументным интерфейсом (multiple-document interface, MDI), поэтому вы легко сможете открыть все проекты для всех EXE-файлов. Когда вы добавляете в проект DLL, CrashFinder проверяет, не конфликтует ли ее адрес загрузки с прочими DLL проекта. Обнаружив конфликт, CrashFinder позволит изменить адрес загрузки конфликтующей DLL только для текущего проекта. Это очень удобно, когда у вас есть проект CrashFinder для отладочной компоновки и вы случайно забыли модифицировать базовый адрес своих DLL.

По мере развития приложения вы можете удалять двоичные образы из проекта командой Remove Image (удалить образ) из меню Edit. Кроме того, вы всегда можете изменить адрес загрузки двоичного образа, выбрав пункт Image Properties (свойства образа) из меню Edit. Так как CrashFinder автоматически добавляет в проект системные DLL, нужные вашим двоичным файлам, вы сможете облегчить отладку, если используете сервер символов (см. главу 2). Теперь у вас есть еще более веская причина установки сервера символов: благодаря его поддержке CrashFinder'ом вы сможете изучать ошибки даже в системных модулях. Просмотрев информацию, изображенную для VERSION.DLL, вы увидите, что она загружает символы из моего сервера символов.

Назначение CrashFinder заключается в преобразовании адреса ошибки в информацию об исходном файле, имени функции и номере строки. Интересующие вас адреса нужно вводить в поле Hexadecimal Address(es) (шестнадцатеричные адреса) — в нижней половине дочернего окна. Когда вы нажмете на кнопку Find (найти), исходный файл, имя функции и номер строки появятся в поле вывода в низу окна. При желании вы можете ввести несколько адресов, разделив их пробелами или запятыми. Так, вы можете ввести полный список адресов стека вызовов из журнала Dr. Watson, получив сразу все нужные сведения.

По умолчанию CrashFinder не показывает смещение адреса от начала функции или от строки исходного кода. Чтобы увидеть эти сведения, укажите это в диалоговом окне Options (свойства). Смещение от начала функции показывает, на сколько байт адрес отстоит от начала функции. Смещение от строки говорит, на сколько байт отстоит адрес от начала ближайшей строки исходного кода. Помните, что одной строке исходного кода могут соответствовать много ассемблерных команд, особенно если вы используете вызовы функций как параметры. Работая с CrashFinder, вы не сможете просмотреть адрес, не являющийся корректным адресом команды. Скажем, при неправильном обращении с указателем this вы можете столкнуться с ошибкой по такому адресу, как 0x00000001. К счастью, эти типы ошибок не так часты, как обычные ошибки нарушения доступа к памяти, которые CrashFinder находит с легкостью.

Некоторые сведения о реализации

CrashFinder — простое приложение, написанное при помощи библиотеки MFC, поэтому большая его часть должна быть вам понятной. Тем не менее, чтобы вам было легче реализовать функции, которые я предлагаю в разделе «Что после CrashFinder?», мне хотелось бы пояснить три главных аспекта этой программы: работу сервера символов, функцию, в которой выполняется основной объем работы, и архитектуру данных CrashFinder.

CrashFinder использует сервер символов DBGHELP.DLL (см. главу 4). Я только хочу обратить ваше внимание на то, что загрузка всей информации об исходных файлах и номерах строк выполняется явно, путем передачи флага `SYMOPT_LOAD_LINES` в функцию `SymSetOptions`. По умолчанию сервер символов DBGHELP.DLL не загружает информацию об исходных файлах и номерах строк.

Почти все операции CrashFinder обеспечивает класс документа, `CCrashFinderDoc`. Он содержит класс `CSymbolEngine`, выполняет весь просмотр символов и контролирует представление данных. Ключевая функция `CrashFinder, CCrashFinderDoc::LoadAndShowImage`, показана в листинге 12-3. Именно в ней осуществляется проверка корректности двоичного образа, его сравнение с существующими элементами проекта для предотвращения конфликтов адресов загрузки, загрузка символов и включение образа в конец дерева. Эта функция вызывается и при добавлении образа в проект, и при открытии проекта. Выполнение всех этих задач в функции `CCrashFinderDoc::LoadAndShowImage` позволило мне сосредоточить базовую логику `CrashFinder` в одном месте и хранить в проекте только имена двоичных образов вместо копий таблицы символов.

Листинг 12-3. Функция `CCrashFinderDoc::LoadAndShowImage`

```
BOOL CCrashFinderDoc :: LoadAndShowImage ( CBinaryImage * pImage      ,
                                           BOOL          bModifiesDoc  ,
                                           BOOL          bIgnoreDups   )
{
    // Проверка данных, внешних по отношению к этой функции.
    ASSERT ( this ) ;
    ASSERT ( NULL != m_pcTreeControl ) ;

    // Строка для вывода информационных сообщений.
    CString sMsg ;
    // Состояние отображения дерева.
    int iState = STATE_NOTVALID ;
    // Возвращаемое значение типа BOOL
    BOOL bRet ;

    // Проверка правильности параметра.
    ASSERT ( NULL != pImage ) ;
    if ( NULL == pImage )
    {
        // Недопустимое значение указателя.
        return ( FALSE ) ;
    }
}
```

см. след. стр.

```

// Проверка корректности образа. Если он корректен, я проверяю,
// не включен ли он уже в список и не конфликтует ли он с другими
// адресами загрузки. Если образ некорректен, я все равно добавляю
// его в список, потому что игнорировать данные пользователя нельзя.
// Если образ плохой, я просто отображаю его с соответствующим значком
// и не загружаю его в символьную машину.
if ( TRUE == pImage->IsValidImage ( ) )
{

    // В этом блоке выполняется просмотр элементов массива данных,
    // при этом осуществляется поиск следующих проблем.
    // 1. Двоичный образ уже находится в списке. Если это так,
    // я могу только прервать операцию.
    // 2. Двоичный файл хочет загрузиться по адресу, который уже
    // имеется в списке. В этом случае я вывожу для двоичного
    // образа окно Properties, чтобы его адрес загрузки можно
    // было изменить.
    // 3. Проект уже включает образ EXE, и pImage также
    // указывает на исполняемый файл.

    // Я всегда исхожу из предположения, что данные, на которые
    // указывает pImage, корректны. Я - оптимист!
    BOOL bValid = TRUE ;
    INT_PTR iCount = m_cDataArray.GetSize ( ) ;
    for ( INT_PTR i = 0 ; i < iCount ; i++ )
    {
        CBinaryImage * pTemp = (CBinaryImage *)m_cDataArray[ i ] ;

        ASSERT ( NULL != pTemp ) ;
        if ( NULL == pTemp )
        {
            // Недопустимое значение указателя!
            return ( FALSE ) ;
        }

        // Совпадают ли имена образов (два значения CString)?
        if ( pImage->GetFullName ( ) == pTemp->GetFullName ( ) )
        {
            if ( FALSE == bIgnoreDups )
            {
                // Сообщить об этом пользователю!!
                sMsg.FormatMessage ( IDS_DUPLICATEFILE ,
                                     pTemp->GetFullName ( ) ) ;
                AfxMessageBox ( sMsg ) ;
            }
            return ( FALSE ) ;
        }
    }

    // Если текущий образ из структуры данных некорректен,
    // у меня неприятности. Выше я смог проверить совпадение

```

```
// имен образов, однако конфликт адресов загрузки и попытку
// добавления двух EXE-файлов проверить труднее. Если pTemp
// некорректен, я должен пропустить эти проверки. Это может
// привести к проблемам, но pTemp будет отмечен в списке как
// некорректный, поэтому пользователь сможет сам переназначить
// соответствующие свойства.
if ( TRUE == pTemp->IsValidImage ( FALSE ) )
{

    // Проверка, позволяющая исключить
    // добавление в проект двух EXE-файлов.
    if ( 0 == ( IMAGE_FILE_DLL &
                pTemp->GetCharacteristics ( ) ) )
    {
        if ( 0 == ( IMAGE_FILE_DLL &
                    pImage->GetCharacteristics ( ) ) )
        {
            // Сообщить пользователю!!
            sMsg.FormatMessage ( IDS_EXEALREADYINPROJECT ,
                                pImage->GetFullName ( ) ,
                                pTemp->GetFullName ( ) );
            AfxMessageBox ( sMsg );
            // Попытка загрузки двух EXE-образов приводит
            // к автоматическому отбрасыванию данных
            // для соответствующего pImage.
            return ( FALSE );
        }
    }

    // Проверка конфликтов адресов загрузки.
    if ( pImage->GetLoadAddress ( ) ==
        pTemp->GetLoadAddress ( ) )
    {
        sMsg.FormatMessage ( IDS_DUPLICATELOADADDR ,
                            pImage->GetFullName ( ) ,
                            pTemp->GetFullName ( ) );

        if ( IDYES == AfxMessageBox ( sMsg , MB_YESNO ) )
        {
            // Пользователь желает изменить
            // свойства вручную.
            pImage->SetProperties ( );

            // Проверка того, что адрес загрузки на самом
            // деле был изменен и уже не конфликтует
            // с другими двоичными образами.
            int iIndex ;
            if ( TRUE ==
                IsConflictingLoadAddress (
```

[illegible]

```

                                0
                                );
// Внимание! SymLoadModule возвращает
// адрес загрузки образа, а не TRUE.
ASSERT ( FALSE != bRet );
if ( FALSE == bRet )
{
    TRACE ( "m_cSymEng.SymLoadModule failed!!\n" );
    iState = STATE_NOTVALID ;
}
else
{
    CImageHlp_Module cModInfo ;
    BOOL bRet =
        m_cSymEng.SymGetModuleInfo64(pImage->GetLoadAddress(),
                                     &cModInfo
                                     );
    ASSERT ( TRUE == bRet );
    if ( TRUE == bRet )
    {
        // Проверка того, что тип символа не равен SymNone.
        if ( SymNone != cModInfo.SymType )
        {
            iState = STATE_VALIDATED ;
            // Задание информации о символах образа.
            pImage->SetSymbolInformation ( cModInfo );
        }
        else
        {
            iState = STATE_NOTVALID ;
            // Выгрузка модуля. Символьная машина загружает
            // модуль даже без символов, поэтому я должен
            // выгрузить его. Я хочу, чтобы загрузились
            // только достойные этого модули.
            m_cSymEng.SymUnloadModule64(
                pImage->GetLoadAddress());
            pImage->SetBinaryError ( eNoSymbolsAtAll );
        }
    }
    else
    {
        iState = STATE_NOTVALID ;
    }
}
}

// Для данного pImage устанавливается признак
// состояния загрузки символов.
if ( STATE_VALIDATED == iState )
{
    pImage->SetExtraData ( TRUE );
}

```

```
else
{
    pImage->SetExtraData ( FALSE );
}

// Элемент помещается в массив.
m_cdataArray.Add ( pImage );

// Изменился ли документ в результате добавления элемента?
if ( TRUE == bModifiesDoc )
{
    SetModifiedFlag ( );
}

// Образ помещается в дерево.
bRet = m_cTreeDisplay.InsertImageInTree ( pImage ,
                                           iState );

ASSERT ( bRet );

// Все OK!!
return ( bRet );
}
```

Наконец, два слова об архитектуре данных CrashFinder. Основная структура данных CrashFinder — простой массив объектов класса `CBinaryImage`. Объект `CBinaryImage` соответствует одному двоичному образу и содержит базовую информацию о нем, такую как адрес загрузки, свойства и имя. При добавлении в проект двоичного образа документ помещает объект `CBinaryImage` в основной массив данных и сохраняет соответствующий указатель в слоте данных узла дерева. При выборе элемента дерева оно возвращает его указатель документу, который получает `CBinaryImage` и просматривает его символическую информацию.

Что после CrashFinder?

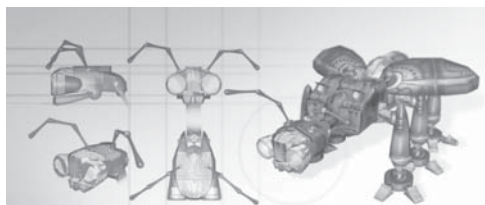
Первая версия CrashFinder поддерживала все описанные функции, но была недостаточно удобной в использовании, что было исправлено мной и упомянутыми выше людьми с учетом многих пожеланий. Тем не менее CrashFinder всегда можно усовершенствовать. Если вы хотите лучше разобраться в образах двоичных файлов, попробуйте добавить в CrashFinder какие-нибудь из следующих функций.

- Реализуйте поддержку двоичных файлов ОС и сделайте так, чтобы CrashFinder автоматически переключался между ними. Это повысит эффективность поиска ошибок в коде ОС. В настоящий момент CrashFinder работает только с системными DLL, установленными на компьютере пользователя.
- Выводите в древовидном списке более подробную информацию для каждого двоичного файла. Класс `CBinaryImage` позволяет сделать это при помощи метода `GetAdditionalInfo`. Вы можете добавить функцию вывода информации о заголовке, импортируемых и экспортируемых функциях.

- Реализуйте возможность вставки (pasting) списков DLL, чтобы их можно было добавлять в проект автоматически. В окне отладчика Output выводится список всех загружаемых приложением DLL, поэтому было бы удобно, если бы CrashFinder позволял вставлять этот текст и умел отыскивать в нем имена DLL.
- Добавьте в CrashFinder поддержку аварийных дампов памяти. Тогда CrashFinder мог бы сопоставлять ошибку с аварийным дампом и определять все ее обстоятельства.

Резюме

В этой главе я рассказал, как найти расположение ошибки в исходном коде, имея только ее адрес. Первый метод нахождения исходного файла и номера строки конкретной ошибки — изучение MAP-файла. MAP-файлы — это единственное средство представления информации о символах в текстовом виде, и вам следует создавать их для каждой заключительной компоновки вашего приложения. Второй метод преобразования адреса ошибки в информацию об исходном файле, имени функции и номере строки обеспечивает утилита CrashFinder. CrashFinder выполняет это преобразование самостоятельно, позволяя получить максимально подробную информацию об ошибке. CrashFinder проще в использовании, чем MAP-файлы, но это не избавляет вас от необходимости их создания, потому что форматы файлов символов иногда изменяются. Когда в вашу дверь постучит призрак одного из ваших старых приложений, ничто не спасет вашу душу — только MAP-файлы.



Обработчики ошибок

И для кого не секрет, что пользователи ненавидят диалоговое окно сообщения об ошибке, появляющееся при аварийном завершении приложения. Если вы читаете эту книгу, это означает, что вы делаете все возможное для профилактики ошибок. Однако все мы знаем, что ошибки происходят даже в самых лучших программах, и к ним нужно быть готовым.

Как было бы хорошо, если бы вместо раздражающего пользователей сообщения об ошибке появлялось дружелюбное к ним окно, которое описывало бы проблему и спрашивало его о том, что он делал в момент ошибки! А еще лучше, если б это любезное и вежливое окно записывало не только обычную информацию об адресе ошибки и стеке вызовов, предоставляемую такими утилитами, как Dr. Watson, но и внутреннее состояние приложения, позволяющее узнать состояние выполнения программы и ее данных в момент ошибки! И разве не было бы уж совсем великолепно, если б диалоговое окно автоматически отсылало вам информацию об ошибке по электронной почте и сохраняло отчет об ошибке в вашей системе отслеживания ошибок?

Обработчики ошибок могут сделать эти мечты реальностью, обеспечив вас всей этой полезной информацией. Обработчиками ошибок я называю и обработчики исключений, и фильтры необработанных исключений. Если вы работали с языком C++, обработчики исключений должны быть вам известны. Вероятно, вам хуже знакомы фильтры необработанных исключений, которые представляют собой интересные функции, позволяющие получить контроль прямо перед появлением отвратительного окна с сообщением об ошибке. Обработчики исключений имеются только в C++, в то время как фильтры необработанных исключений можно использовать и в C, и в C++.

В этой главе я приведу код, который вы можете включать в свои приложения для получения такой информации, как значения регистров и стеки вызовов. Кро-

ме того, он скроет значительную часть грязной работы по сбору этих данных, благодаря чему вы сможете сосредоточиться на информации, специфичной для вашего приложения, и смягчении отрицательных эмоций своих пользователей. Я также расскажу о том, как извлечь максимальную выгоду из прекрасной API-функции `MiniDumpWriteDump`, чтобы получать минидампы всегда, когда они нужны. Однако, прежде чем перейти к рассмотрению этого кода, я опишу типы обработки исключений в ОС Microsoft Win32.

Структурная обработка исключений против обработки исключений C++

Изучение обработки исключений осложняется тем, что C++ поддерживает два основных ее типа: структурную обработку исключений (structured exception handling, SEH), предоставляемую ОС, и обработку исключений C++, входящую в состав языка. Многие считают оба типа обработки исключений одинаковыми. Могут вас заверить, что в основе этих двух типов обработки исключений лежат совершенно разные подходы. Общая их черта в том, что исключения обоих типов предназначены для применения в исключительных ситуациях, а не при нормальном выполнении программы. По-моему, многих людей путают слухи, что оба типа исключений можно комбинировать. Ниже я коснусь различий и сходств между этими типами обработки исключений, а также расскажу про то, как избежать крупнейшего источника связанных с ними ошибок.

Структурная обработка исключений

SEH обеспечивается ОС и предназначена для непосредственной обработки таких ошибок, как нарушение доступа. SEH не привязана к конкретному языку; в программах C и C++ она обычно реализуется в виде пар `__try/__except` и `__try/__finally`. Использование пары `__try/__except` заключается в размещении некоторого кода внутри блока `__try` и описании обработки возникающих в нем исключений в блоке `__except` (также называемом обработчиком исключений). Входящий в состав пары `__try/__finally` блок `__finally` (его еще называют обработчиком завершения) выполняется при выходе из функции всегда, даже при преждевременном завершении блока `__try`, что позволяет гарантировать освобождение ресурсов.

Типичная функция с SEH представлена в листинге 13-1. Блок `__except` выглядит почти так же, как вызов функции, только в его скобках указано значение специального выражения, называемого фильтром исключений. Фильтр исключений имеет значение `EXCEPTION_EXECUTE_HANDLER`, которое указывает на то, что блок `__except` должен выполняться каждый раз при возникновении любого исключения в блоке `__try`. Возможны еще два значения фильтра исключений: `EXCEPTION_CONTINUE_EXECUTION` позволяет проигнорировать исключение, а `EXCEPTION_CONTINUE_SEARCH` передает исключение по цепи вызовов следующему блоку `__except`. Вы можете делать фильтр исключений сколь угодно простым или сложным, обрабатывая только те исключения, которые желаете.

Листинг 13-1. Пример обработчика SEH

```
void Foo ( void )
{
    __try
    {
        __try
        {
            // Какие-нибудь действия.
        }
        __except ( EXCEPTION_EXECUTE_HANDLER )
        {
            // Этот блок будет выполнен, если код в блоке __try
            // вызовет нарушение доступа или другую ошибку.
            // Блок __except называют также обработчиком исключений.
        }
    }
    __finally
    {
        // Этот блок будет выполнен независимо от того,
        // вызвала ли функция ошибку. Выполняйте здесь
        // все действия, необходимые для освобождения ресурсов.
    }
}
```

Процесс поиска и выполнения обработчика исключения иногда называют разворачиванием (unwinding) исключения. Обработчики исключений хранятся во внутреннем стеке; по мере роста цепи вызовов функций в этот внутренний стек помещаются обработчики исключений (если они есть) всех новых функций. При возникновении исключения ОС находит стек обработчиков исключений данного потока и начинает вызывать их, пока какой-нибудь из них не согласится обработать данное исключение. По мере прохождения исключения через стек обработчиков ОС очищает стек вызовов и выполняет все обработчики завершений, которые ей при этом встречаются. Если разворачивание достигает конца стека обработчиков исключений, появляется окно сообщения об ошибке или загружается JIT-отладчик.

Обработчик исключений может определить значение исключения посредством специальной функции `GetExceptionCode`, которая может вызываться только в фильтрах исключений. При работе над математической программой вы, например, могли бы установить обработчик исключений, который обрабатывал бы попытки деления на 0 и возвращал значение `NaN` (not a number — не число). Пример такого обработчика см. в листинге 13-2. Фильтр исключений вызывает `GetExceptionCode`, и, если исключение вызвано делением на 0, выполняется обработчик. Любому другому исключению соответствует значение `EXCEPTION_CONTINUE_SEARCH`, приказывающее ОС выполнить следующий блок `__except` в цепи вызовов.

Листинг 13-2. Пример обработчика SEH, включающего обработку фильтра исключений

```
""long IntegerDivide ( long x , long y )
{
    long lRet ;

    __try
    {
        lRet = x / y ;
    }
    __except ( EXCEPTION_INT_DIVIDE_BY_ZERO ==
               GetExceptionCode ( )
               ? EXCEPTION_EXECUTE_HANDLER
               : EXCEPTION_CONTINUE_SEARCH
             )
    {
        lRet = NaN ;
    }
    return ( lRet ) ;
}
```

Фильтром исключений может быть даже вызов вашей собственной функции, если только она определяет обработку исключения, возвращая одно из корректных значений фильтра исключений. Кроме специальной функции `GetExceptionCode`, в выражении фильтра исключений можно вызывать функцию `GetExceptionInformation`. Она возвращает указатель на структуру `EXCEPTION_POINTERS`, которая полностью описывает причину ошибки и состояние процессора в момент ее возникновения. Как вы уже догадались, структуру `EXCEPTION_POINTERS` я использую ниже.

Возможности SEH не ограничиваются обработкой ошибок. Вы можете определять собственные исключения, используя API-функцию `RaiseException`. Большинство программистов ее не применяет, хотя она обеспечивает один из способов быстрого выхода из глубоко вложенных условных выражений. Этот способ более корректен, чем старый метод, основанный на использовании функций стандартной библиотеки `setjmp` и `longjmp`.

Чтобы грамотно работать с SEH, вам нужно знать о двух ее ограничениях. Первое не очень существенно: число кодов ваших ошибок ограничено одним беззнаковым целым. Вторая проблема серьезнее: SEH плохо сочетается с C++, потому что исключения C++ на самом деле реализованы при помощи SEH, и беспорядочное комбинирование обоих типов исключений вызывает недовольство компилятора. Причина конфликта в том, что при разворачивании SEH деструкторы объектов C++, созданных в стеке, не вызываются. В конструкторах объектов C++ часто выполняется самая разнообразная инициализация, например, выделение памяти для внутренних структур данных, поэтому пропуск деструкторов может приводить к утечкам памяти и другим проблемам.

Если вы хотите лучше изучить основы SEH, то, кроме Microsoft Developer Network (MSDN), я рекомендую еще два источника. Самый лучший обзор SEH можно найти в книге Джеффри Рихтера (Jeffrey Richter. *Programming Applications for Microsoft*

Windows. — Microsoft Press, 1999). Если вас интересует действительная реализация SEH, прочитайте статью Мэтта Питрека (Matt Pietrek) «A Crash Course on the Depths of Win32 Structured Exception Handling» («Тонкости структурной обработки исключений Win32») в «Microsoft Systems Journal» в январе 1997 г.

Я хочу также упомянуть про одно усовершенствование SEH, появившееся в Microsoft Windows XP/Server 2003, — векторную обработку исключений. При обычной SEH способа глобального уведомления об исключениях нет. Обычно в повседневной работе векторная обработка исключений не нужна, однако она предоставляет одну очень полезную возможность — получение первого и последнего уведомления обо всех SEH-исключениях приложения. Как только я узнал, что Microsoft добавила в ОС векторную обработку исключений, у меня сразу же созрел план программы мониторинга SEH-исключений, позволяющей следить за генерируемыми приложением исключениями, не запуская его под управлением отладчика.

Для получения векторных исключений просто вызовите функцию `AddVectoredExceptionHandler`, вторым параметром которой является указатель на функцию, которая будет вызываться при возникновении в вашей программе любого первого случая исключения (first-chance exception). Первый параметр — булево значение, показывающее, как вы хотите получать уведомления: до или после нормального развертывания цепи исключений. При исключении ваша функция обратного вызова получит указатель на структуру `EXCEPTION_POINTERS`, описывающую исключение. Как вы уже поняли, эта информация делает получение исключений тривиальным.

На CD вы найдете проект `XPExcerptMon`, иллюстрирующий использование векторных исключений, записывая все исключения, возникающие в вашей программе. Вся работа по установке и удалению ловушки векторных исключений выполняется в функции `DllMain` библиотеки `XPExcerptMon.DLL`, поэтому задействовать ее в своих приложениях вам будет очень просто. Я хотел просто продемонстрировать работу с векторными исключениями, поэтому все, что делает `XPExcerptMon`, заключается в записи типа и адреса исключения в текстовый файл. Чтобы попрактиковаться с сервером символов `DBGHELP.DLL`, можете добавить в `XPExcerptMon` просмотр функций и анализ стека.

Если вам хотелось бы получать уведомления об исключениях для более ранних версий Windows, вам повезло. Юджин Гершник (Eugene Gershnik) написал отличную статью «Visual C++ Exception-Handling Instrumentation» («Обработка исключений в Visual C++»), опубликованную в декабрьском номере «Windows Developer Magazine» за 2002 год. Юджин не только показывает, как устанавливать ловушки для исключений, но и отлично описывает саму обработку исключений.

Обработка исключений C++

Обработка исключений C++ входит в состав языка C++, поэтому большинству программистов она скорее всего известна лучше, чем SEH. Для обработки исключений C++ используются ключевые слова `try` и `catch`. Ключевое слово `throw` позволяет инициировать развертывание исключений. В то время как число кодов ошибок SEH ограничено одним беззнаковым целым, обработчик `catch` языка C++ может

перехватывать любые типы переменных, включая классы. Если вы выполните наследование классов обработки ошибок от общего базового класса, вы сможете обрабатывать почти любые ошибки. Именно такой иерархический подход к обработке ошибок реализован в библиотеке Microsoft Foundation Class (MFC) с базовым классом `CException`. Обработка исключений C++ показана в листинге 13-3, где выполняется чтение объекта класса `CFile` библиотеки MFC.

Листинг 13-3. Пример обработчика исключений C++

```
BOOL ReadFileHeader ( CFile * pFile , LPHEADERINFO pHeader )
{
    ASSERT ( FALSE == IsBadReadPtr ( pFile , sizeof ( CFile * ) ) ) ;
    ASSERT ( FALSE == IsBadReadPtr ( pHeader ,
                                     sizeof ( LPHEADERINFO ) ) ) ;
    if ( ( TRUE == IsBadReadPtr ( pFile , sizeof ( CFile * ) ) ) ||
        ( TRUE == IsBadReadPtr ( pHeader ,
                                 sizeof ( LPHEADERINFO ) ) ) )
    {
        return ( FALSE ) ;
    }

    BOOL bRet ;
    try
    {
        pFile->Read ( pHeader , sizeof ( HEADERINFO ) ) ;
        bRet = TRUE ;
    }
    catch ( CFileException * e )
    {
        // Если заголовочный файл не был прочитан из-за обнаружения
        // преждевременного конца файла, исключение обрабатывается,
        // в противном случае разворачивание продолжается.
        if ( CFileException::endOfFile == e->m_cause )
        {
            e->Delete();
            bRet = false;
        }
        else
        {
            // Само по себе ключевое слово throw генерирует то же
            // исключение, что было передано в этот блок catch.
            throw ;
        }
    }
    return ( bRet ) ;
}
```

Обработка исключений C++ имеет некоторые недостатки, о которых следует помнить. Во-первых, ошибки вашей программы не обрабатываются автоматически. Во-вторых, обработка исключений C++ не так уж и грациозна. Даже если ваша программа никогда не генерирует исключений, компилятор проделает много

работы, устанавливая и удаляя блоки `try` и `catch`, что может оказаться слишком накладно при высоких требованиях к быстродействию программы. Если вы новичок в обработке исключений C++, для начала их изучения прекрасно подойдет MSDN.

Избегайте использования обработки исключений C++

Обработка исключений C++ — один из самых частых вопросов, с которыми я сталкиваюсь как консультант. При разработке Windows-приложений программисты тратят на решение проблем с исключениями C++ больше времени, чем на все прочие ошибки (за исключением искажений памяти). Опираясь на большой опыт разрешения многих ужасных ситуаций, я рекомендую избегать обработки исключений C++, потому что это бесконечно облегчит вашу жизнь и упростит отладку программ.

Первая проблема обработки исключений C++ в том, что ее нельзя считать чистым свойством языка. Очень часто она кажется чужеродной и незавершенной. Отсутствие стандартного класса ANSI, содержащего информацию об исключении, означает, что у нас нет согласованного способа обработки общих ошибок. Кое-кто из вас, возможно, считает, что стандартным механизмом для обработки всех исключений является конструкция `catch (...)`, но ниже вы увидите, насколько она небезопасна.

Обработка исключений C++ принадлежит к числу технологий, великолепных с теоретической точки зрения, но приводящих к проблемам, если вы пишете что-нибудь посерьезнее, чем «Hello World!». Я часто сталкиваюсь с абсолютно безумными ситуациями, когда один из членов группы так влюбляется в исключения C++, что начинает ими свой код в невообразимом количестве. Это заставляет работать с исключениями C++ и всех остальных членов группы, даже если далеко не все из них до конца понимают тонкости проектирования и использования исключений. При этом кто-либо из программистов неизбежно забывает про перехват какого-нибудь случайного неожиданного исключения, и приложение терпит крах. Кроме того, программы, в которых для сообщения об ошибках используются и возвращаемые значения, и исключения C++, практически не поддаются модернизации и сопровождению, вследствие чего многие компании предпочитают отказаться от существующего кода и начать разработку программы с самого начала, значительно увеличивая расходы.

Многим программистам исключения C++ нравятся тем, что они позволяют отказаться от проверки возвращаемых из функций значений. Однако этот аргумент не только ошибочен, но и служит оправданием плохого стиля программирования. Если у одного из членов вашей группы постоянно возникают проблемы с проверкой возвращаемых значений, проведите соответствующую консультацию. Если и после этого он не будет выполнять проверку возвращаемых значений, смело увольняйте его — он просто отлынивает от работы.

До этого момента я обсуждал вопросы проектирования исключений C++ и управления ими. Но не стоит также забывать, что с ними связано много дополнительных затрат. Для создания блоков `try` и `catch` нужно проделать большую работу, что заметно ухудшает быстродействие программы, даже если вы редко (если вообще) генерируете исключения.

Кроме того, в компиляторах Microsoft исключения C++ реализованы при помощи SEH, а это означает, что каждый оператор `throw` вызывает функцию `Raise-Exception`. В этом нет ничего плохого, но каждый `throw` вызывает всеми любимое переключение в режим ядра. Само по себе переключение выполняется очень быстро, но манипуляции с вашим исключением в режиме ядра требуют огромных затрат. В разделе «Советы и уловки» главы 7 я рассказал о способе контроля исключений C++, который сможет точнее указать вам на эти затраты.

Похоже, иногда разработчики забывают о цене исключений C++. Однажды меня наняла одна компания, чтобы я решил проблему с производительностью программы. Приступив к работе, я никак не мог понять, почему функция `_except_handler3`, выполняющаяся при обработке исключений, вызывается столько раз. При проверке кода я понял, что вместо испытанной конструкции `switch...case` кто-то использовал обработку исключений C++. Чтобы ускорить приложение, компании пришлось переписать значительную часть кода этого программиста просто для возврата значений перечисления. На мой вопрос, почему он решил прибегнуть к обработке исключений C++, он ответил, что думал, что оператор `throw` просто изменяет указатель команд. Итак, исключения C++ допустимы только в тех программах, быстроедействие которых не имеет никакого значения.

Никогда, ни за что, НИ В КОЕМ СЛУЧАЕ не используйте `catch (...)`

Мне очень нравится конструкция `catch (...)`, потому что она весьма благоприятно сказывается на моем банковском счете, вызывая больше ошибок, чем вы можете себе представить. С ней связаны две огромных проблемы. Первая заключается в ее спецификации согласно стандарту ANSI. Многозначие означает, что блок `catch` перехватывает любой тип исключений.

Однако из-за отсутствия в `catch` какой-либо переменной вы не можете узнать, как вы очутились в этом блоке и почему. Это, например, означает, что исключение может быть вызвано присвоением случайного значения указателю команд. Вследствие невозможности узнать тип и причину исключения единственное безопасное и благоразумное действие, которое вы можете предпринять, — завершить приложение. Некоторым из вас, возможно, покажется, что это слишком радикальное решение, но лучшего я предложить не могу.

Вторая проблема связана с реализацией `catch (...)`. Многие программисты не осознают, что в стандартной библиотеке C для Windows блок `catch (...)` перехватывает не только исключения C++, но и исключения SEH! Вы не только не будете знать, как вы оказались в блоке `catch`, но и сможете очутиться в нем из-за нарушения доступа к памяти или какой-нибудь другой аппаратной ошибки. В этом случае вы можете не только заблудиться, но и столкнуться с совершенно нестабильным поведением программы в блоке `catch`, поэтому самым лучшим решением будет немедленное завершение процесса.

Меня просто поражает, как часто программисты, и довольно опытные в том числе, пишут что-нибудь вроде:

```
BOOL DoSomeWork ( void )
{
    BOOL bRet = TRUE ;
    try
    {
```

```
// ...Какие-нибудь действия...
}
catch ( ... )
{
    // ВНИМАНИЕ! ЭТОТ БЛОК ПУСТ!
}
return ( bRet );
}
```

Если в такой ситуации произойдет нарушение доступа, оно будет перехвачено блоком `catch (...)`, но вы об этом даже не узнаете. Минут через двадцать ваша программа потерпит крах, однако о его причине вы не будете иметь абсолютно никакого представления, потому что стек вызовов не будет иметь причинно-следственного отношения. Вам останется только удивляться проблеме. Опираясь на свой богатый опыт, я утверждаю, что главной причиной непонятных ошибок является `catch (...)`. Гораздо лучше допустить аварийное завершение приложения, потому что тогда у вас хотя бы будет неплохой шанс обнаружения ошибки. При использовании `catch (...)` вероятность ее обнаружения снижается до 5% и ниже.

Если вы еще не догадались, я сам скажу, что советую вам удалить все блоки `catch (...)` из ваших программ. Фактически я ожидаю, что вы сделаете это сразу, иначе вы скоро обратитесь ко мне за услугами и поможете мне выплатить очередной взнос за автомобиль.

Отладка: фронтовые очерки

О вреде catch (...)

Боевые действия

Однажды я ехал в аэропорт, собираясь вылететь к клиенту. Тут мне позволил руководитель отделения и сказал, что мы получили отчаянный — просто безумный — запрос о проведении консультации. Помогать обезумевшим людям — наша работа, поэтому я решил узнать, что случилось. Поднявший трубку менеджер не просто обезумел — он был на грани инфаркта! Он сказал, что сотрудники его компании бьются над решением абсолютно непонятной ошибки, задерживающей выпуск программы. Он также сказал, что это еще не самое страшное, и добавил, что если ошибка не будет решена и программа не будет закончена, его компания обанкротится. Более 10 программистов уже работали над этой ошибкой три недели подряд, но безрезультатно. В то время моя жизнь была не особо увлекательной по сравнению с работой на предыдущих должностях, поэтому возможность спасти чью-то компанию определенно привлекла мой интерес. Этот человек спросил меня, насколько быстро я смогу добраться к ним. Я ответил, что вылетаю в Сиэтл на неделю, поэтому не смогу заняться их проблемой до окончания этого срока. Мы недавно основали компанию Wintellect, и у нас не было свободных сотрудников.

В этот момент менеджер начал говорить гораздо громче (ладно, что греха таить, он начал орать), утверждая, что не может столько ждать. Он спросил, буду ли я занят в Сиэтле постоянно. Это было не так, и он предложил

мне работать над их проблемой по вечерам. Меня это вполне устраивало, и я сказал, что смогу работать над их ошибкой каждый день до полуночи, а он тут же отодвинул трубку от уха и прокричал кому-то: «Он будет в Сиэтле! Немедленно вылетайте!» Менеджер сказал мне, что отправил в аэропорт двух программистов с необходимым оборудованием. Когда я спросил его, что случится, если мы не решим проблему до того, как мне придется уехать из Сиэтла, он ответил: «Мы последуем за вами в Нью-Хэмпшир». Ошибка автоматически перешла в категорию суперсерьезных.

Следующим вечером я приехал в отель на встречу с этими программистами и увидел двух человек, едва стоявших на ногах. Они работали над этой ошибкой примерно три недели подряд почти без перерыва. Увидев приложение, я сразу покрылся холодным потом! Они разрабатывали модуль GINA (Graphical Identification and Authentication, — графическая идентификация и аутентификация), предназначенный для регистрации на терминальном сервере при помощи смарт-карты! Да уж, трудно представить более неприятного приложения для отладки! Так как значительная часть программы выполнялась внутри LSASS.EXE, вы могли начать отладку, но любой щелчок вне отладчика блокировал компьютер. Чтобы сделать мою жизнь еще интересней, программисты повсюду использовали STL, поэтому программа не только содержала в себе ошибку, но и была очень тяжелой в понимании. Всем нам стоит помнить, что основное достоинство STL заключается не в повторно используемых структурах данных, а в гарантии занятости. Понимание и сопровождение кода, написанного при помощи STL, доступно только его автору, что обеспечивает ему уверенность в завтрашнем дне и постоянный источник дохода.

Я спросил их, могут ли они показать мне что-нибудь, напоминающее воспроизводимую ошибку или искажение данных. Они дали мне листинг и указали 10 или 12 мест, где они сталкивались с ошибками. Сначала я предположил, что ошибка объясняется записью данных в неинициализированную память. Потратив несколько часов на изучение работы системы и привыкание к отладке их приложения, я попытался найти эти неинициализированные указатели. Пробираясь через дебри исходного кода, я обнаружил огромное число блоков `catch (...)`. К концу первого вечера я сказал им удалить все операторы `catch (...)`, чтобы мы могли видеть искажение данных немедленно и начали сужать диапазон причин проблемы.

Исход

Когда я вернулся к ним на следующий день, физические возможности программистов приближались к концу. Они сообщили мне, что, когда они закомментировали все операторы `catch (...)`, приложение перестало инициализироваться. Отправив их спать, я начал просматривать код инициализации и быстро обнаружил следующее:

```
//catch ( ... )  
//{  
    return ( FALSE ) ;  
//}
```

Полусонные программисты просто забыли закомментировать один оператор `return`. Я закомментировал его и запустил программу. Она потерпела крах почти сразу. При втором запуске она завершилась там же, и это был первый раз, когда разработчики увидели согласованную ошибку. Третья ошибка в том же месте показалась всем благословением, и я начал исследовать все, что происходит со стеком.

Тщательно изучив код, мы обнаружили ошибку всего за пару часов. В документации требовалось, чтобы один буфер, передаваемый другой функции, имел размер 250 символов. Кто-то из программистов передавал в качестве буфера локальную переменную и написал 25 вместо 250. Как только мы исправили опечатку, приложение заработало совершенно нормально!

Полученный опыт

Урок прост: не используйте `catch (...)`! В данном конкретном случае компании пришлось потратить недели труда (и огромные деньги) в поисках легкой ошибки, которую нельзя было воспроизвести из-за `catch (...)`.

Не используйте `_set_se_translator`

В первом издании этой книги я описал интересную API-функцию `_set_se_translator`, которая волшебным образом преобразует ваши ошибки SEH в исключения C++. Для этого она вызывает определенную вами функцию, вызывающую `throw` для каждого типа, который вы хотите использовать для преобразования. Сейчас я могу признаться, что тот мой совет оказался ошибочным, хотя в его основе и лежали добрые намерения. При использовании `_set_se_translator` вы быстро обнаружите, что в заключительных компоновках она не работает.

Первая проблема с `_set_se_translator` в том, что она не является глобальной; область ее действия ограничена конкретным потоком. Это значит, что вам, вероятно, придется переработать свою программу, чтобы гарантировать вызов `_set_se_translator` в начале каждого потока. Увы, это не всегда просто. Кроме того, если вы разрабатываете компонент, используемый другими, не контролируемые вами процессами, `_set_se_translator` полностью запутает обработку исключений этих процессов, если они ожидают исключения SEH, а вместо этого получают исключения C++.

Более серьезная проблема касается скрытых деталей реализации обработки исключений C++. Обработка исключений C++ может быть реализована двумя способами: асинхронным и синхронным. При асинхронном режиме генератор кода предполагает, что исключение может быть сгенерировано любой командой, при синхронном исключения генерируются явно только оператором `throw`. Различия между асинхронной и синхронной обработкой исключений не кажутся такими уж большими, но на самом деле это именно так.

Асинхронные исключения имеют один недостаток: компилятор должен сгенерировать для каждой функции то, что называется кодом слежения за временем жизни объекта (`object lifetime tracking code`). Компилятор полагает, что исключение может быть сгенерировано любой командой, поэтому каждая функция, помещающая объект C++ в стек, должна включать код, гарантирующий при возник-

новении исключения вызов деструкторов каждого объекта. А так как предполагается, что исключения — редкие или почти невозможные события, неиспользуемый код слежения за временем жизни объектов приводит к значительному снижению быстродействия программы.

Синхронная обработка исключений решает эту проблему, генерируя код слежения за временем жизни объекта, только когда метод в дереве вызовов включает явный `throw`. Фактически синхронные исключения настолько хороши, что именно этот тип исключений используется компилятором. Однако компилятор предполагает, что исключения происходят только в результате явного `throw` в стеке вызовов, в то время как функция преобразования выполняет `throw`, который находится вне нормального потока выполнения программы и, таким образом, является асинхронным. Из-за этого ваш тщательно разработанный класс оболочки исключений C++ никогда не будет обработан, и ваше приложение все равно потерпит крах. Чтобы лучше изучить различия между асинхронной и синхронной обработкой исключений, включите асинхронный режим, добавив в командную строку компилятора ключ `/EHa` и удалив ключи `/GX` и `/EHs`.

Ситуация ухудшается еще и тем, что в отладочных компоновках `_set_se_translator` работает правильно. Проблемы возникают только в заключительных компоновках. Это объясняется тем, что в отладочных компоновках применяется синхронная обработка исключений вместо асинхронной, используемой по умолчанию в заключительных компоновках. Из-за описанных мной проблем просмотрите свой код и убедитесь в том, что эта функция нигде не вызывается.

API-функция `SetUnhandledExceptionFilter`

Ошибки имеют привычку никогда не происходить там, где вы их ожидаете. Увы, когда пользователи сталкиваются с ошибкой вашей программы, они просто видят диалоговое окно сообщения об ошибке; возможно, Dr. Watson предоставит им некоторую информацию, которую они смогут послать вам, чтобы облегчить поиск проблемы. Как я уже говорил, для получения информации, действительно нужной для исправления ошибок, вы можете разработать собственные диалоговые окна и обработчики. Я всегда называю эти обработчики исключений вместе с соответствующими им фильтрами исключений обработчиками ошибок.

Судя по моему опыту, обработчики ошибок значительно облегчают отладку. Во многих проектах мы получали контроль сразу же после краха приложения, записывали всю информацию об ошибке (включая состояние системы пользователя) в файл, и, если проект был клиентским приложением, выводили диалоговое окно с телефонным номером службы поддержки. В некоторых случаях мы реализовывали возможность циклического изучения основных объектов программы, что позволяло нам опускаться до уровня классов и регистрировать активные объекты и состояние их данных. Можно сказать, что записываемая нами информация о состоянии программы была чуть ли не избыточной. Такие отчеты об ошибках предоставляли нам 90%-ый шанс воспроизведения проблемы пользователя. Если это не проактивная отладка, то я не знаю, что это такое!

Создание обработчиков ошибок обеспечивает API-функция `SetUnhandledExceptionFilter`. Удивительно, но эта возможность присутствует в Win32 со времен

Microsoft Windows NT 3.5, однако почти не упоминается в документации. На момент написания этой главы данная функция встречается в MSDN Online только восемь раз.

Надо сказать, что я нашел эту функцию очень эффективной. Просто взглянув на ее имя — `SetUnhandledExceptionFilter`, вы можете догадаться, что она делает. Она позволяет указать функцию-фильтр необработанных исключений, которая будет вызываться при возникновении в процессе необработанного исключения. Единственным параметром `SetUnhandledExceptionFilter` является указатель на функцию-фильтр исключений, которая вызывается в заключительном блоке `__except` приложения. Этот фильтр исключений может возвращать те же значения, что и любой другой фильтр исключений: `EXCEPTION_EXECUTE_HANDLER`, `EXCEPTION_CONTINUE_EXECUTION` или `EXCEPTION_CONTINUE_SEARCH`. Вы можете выполнять в фильтре исключений почти любые действия по их обработке, но при этом нужно помнить о переполнении стека. Чтобы обезопасить себя, вам, вероятно, следует избегать вызовов функций стандартной библиотеки C, а также MFC. Я должен был предупредить вас об этих неприятностях, однако я могу гарантировать, что большинство ваших ошибок будет ошибками нарушения доступа, поэтому, реализуя в фильтре и обработчике исключений полную систему обработки ошибок, вы, скорее всего, не столкнетесь с какими-либо проблемами, если будете проверять причину исключения и избегать вызовов функций при переполнении стека.

Ваш фильтр исключений получает указатель на структуру `EXCEPTION_POINTERS`. В листинге 13-4 я привожу несколько функций, которые помогут вам выполнять преобразование этой структуры. Благодаря этому вы сможете писать собственные обработчики ошибок, так как в каждой компании к ним предъявляются различные требования.

Используя `SetUnhandledExceptionFilter`, нужно кое о чем помнить. Во-первых, вы не сможете отлаживать установленный фильтр необработанных исключений, применяя стандартные отладчики пользовательского режима. Это ограничение определенно имеет смысл, так как при выполнении программы под отладчиком ОС должна взять на себя управление заключительным фильтром исключений, чтобы сообщить отладчику правильную информацию о последней ошибке. Это затрудняет отладку заключительного обработчика ошибок. Одно из возможных решений данной проблемы — вызвать фильтр необработанных исключений из обычного фильтра исключений SEH. Пример такого подхода вы найдете в функции `Baz` из файла `BugslayerUtil\Tests\CrashHandler\CrashHandler.CPP`, который находится на CD.

Другая проблема в том, что фильтр исключений, указываемый вами при вызове `SetUnhandledExceptionFilter`, является для вашего процесса глобальным. Если вы создаете самый лучший обработчик ошибок в мире для своего элемента управления ActiveX и ошибка возникает в контейнере, то даже несмотря на то, что она не ваша, будет выполнен ваш обработчик ошибок. Однако не позволяйте этой проблеме воспрепятствовать использованию `SetUnhandledExceptionFilter`. Ниже я приведу некоторый код, который поможет справиться с этой неприятностью.

Стандартный вопрос отладки

Что можно сделать с переполнением стека при бесконечной рекурсии?

Нам повезло: бесконечная рекурсия встречается не так часто, но такую ошибку почти невозможно отладить. Если вы когда-нибудь видели приложение, которое приостанавливается на секунду-другую, после чего полностью исчезает безо всякого сообщения об ошибке, почти наверняка это было вызвано бесконечной рекурсией. Если приложение не оставляет даже шанса на свою отладку, обнаружить ошибку чрезвычайно сложно.

К счастью, при помощи новой функции `resetstkoflw` стандартной библиотеки вы можете попробовать получить некоторое пространство в стеке, чтобы хотя бы сообщить об ошибке. Если вы хотите увидеть, как `_resetstkoflw` делает свою магию, посмотрите ее реализацию в файле `RESETSTK.C`.

Использование API CrashHandler

Мой модуль `BUGSLAYERUTIL.DLL` включает API `CrashHandler`, который вы можете использовать для ограничения своего обработчика ошибок конкретным модулем или модулями. Это достигается благодаря передаче всех необработанных исключений моему фильтру. При вызове моего фильтра необработанных исключений я проверяю модуль, из которого пришло исключение. Если исключение возникло в одном из указанных вами модулей, я вызываю ваш обработчик ошибок; если оно пришло из других модулей, я вызываю первоначальный фильтр необработанных исключений. Вызов первоначального фильтра исключений означает, что мой API `CrashHandler` могут использовать несколько модулей, не мешая друг другу. Если никакие модули не указаны, ваш обработчик ошибок будет вызываться всегда. Все функции API `CrashHandler` приведены в листинге 13-4. Я советую вам тщательно изучить этот код, потому что, если вы его поймете, вы неплохо разберетесь в обработке исключений, использовании символьной машины `DBGHELP.DLL` и анализе стека.

Листинг 13-4. CRASHHANDLER.CPP

```
/*-----
Отладка приложений для Microsoft .NET и Microsoft Windows
Copyright © 1997-2003 John Robbins - All rights reserved.
-----*/

#include "pch.h"
#include "BugslayerUtil.h"
#include "CrashHandler.h"

// Внутренний заголовочный файл проекта
#include "Internal.h"

/*//////////////////////////////////////
// Определения с областью видимости файла
```

см. след. стр.


```

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Максимальный размер символов для модуля
#define MAX_SYM_SIZE 512
#define BUFF_SIZE 2048
#define SYM_BUFFER_SIZE 1024

// Константы формата строк. Чтобы избежать частого
// преобразования строк ANSI в формат UNICODE вручную,
// я делаю это с помощью функции wsprintf. Работая с ANSI,
// в строках формата нужно использовать %s, а не %S.
#ifdef UNICODE
#define k_NAMEDISPFMT _T ( " %S()+%04d byte(s)" )
#define k_NAMEFMT _T ( " %S " )
#define k_FILELINEDISPFMT _T ( " %S, line %04d+%04d byte(s)" )
#define k_FILELINEFMT _T ( " %S, line %04d" )
#else
#define k_NAMEDISPFMT _T ( " %s()+%04d byte(s)" )
#define k_NAMEFMT _T ( " %s " )
#define k_FILELINEDISPFMT _T ( " %s, line %04d+%04d byte(s)" )
#define k_FILELINEFMT _T ( " %s, line %04d" )
#endif

#ifdef _WIN64
#define k_PARAMFMTSTRING _T ( " (0x%016X 0x%016X 0x%016X 0x%016X)" )
#else
#define k_PARAMFMTSTRING _T ( " (0x%08X 0x%08X 0x%08X 0x%08X)" )
#endif

// Определение типа компьютера.
#ifdef _X86_
#define CH_MACHINE IMAGE_FILE_MACHINE_I386
#elif _AMD64_
#define CH_MACHINE IMAGE_FILE_MACHINE_AMD64
#elif _IA64_
#define CH_MACHINE IMAGE_FILE_MACHINE_IA64
#else
#pragma FORCE COMPILE ABORT!
#endif

/*////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Глобальные переменные с областью видимости файла
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
// Новый фильтр необработанных исключений (обработчик ошибок)
static PFNCHFLT g_pfnCallback = NULL ;

// Первоначальный фильтр необработанных исключений
static LPTOP_LEVEL_EXCEPTION_FILTER g_pfnOrigFilt = NULL ;

// Массив модулей, ограничивающих применение обработчика ошибок
static HMODULE * g_ahMod = NULL ;

```



```

// Размер массива g_ahMod в элементах
static UINT g_uiModCount = 0 ;

// Статический буфер, возвращаемый различными функциями,
// обеспечивает передачу данных без использования стека.
static TCHAR g_szBuff [ BUFF_SIZE ] ;

// Статический буфер для просмотра символов
static BYTE g_stSymbol [ SYM_BUFF_SIZE ] ;

// Статическая структура, содержащая сведения
// об исходном файле и номере строки
static IMAGEHLP_LINE64 g_stLine ;

// Структура кадра стека, используемая при анализе стека
static STACKFRAME64 g_stFrame ;

// Флаг, указывающий на состояние инициализации символьной машины
static BOOL g_bSymEngInit = FALSE ;

// Первоначальный вариант этого кода изменял при анализе стека
// структуру CONTEXT. Поэтому, если пользователь применял для
// записи минидампа структуру EXCEPTION_POINTERS, включающую
// указатель на CONTEXT, дамп получался некорректным. Теперь
// я сохраняю CONTEXT глобально, во многом как и кадр стека.
static CONTEXT g_stContext ;

/*//////////////////////////////////////
// Объявления функций с областью видимости файла
//////////////////////////////////////*/
// Обработчик исключений
LONG __stdcall CrashHandlerExceptionFilter ( EXCEPTION_POINTERS *
                                             pExPtrs
                                             ) ;

// Функция преобразования идентификатора
// простого исключения в строковое значение
LPCTSTR ConvertSimpleException ( DWORD dwExcept ) ;

// Внутренняя функция, отвечающая за весь анализ стека
LPCTSTR __stdcall InternalGetStackTraceString ( DWORD dwOpts ) ;

// Функция, инициализирующая в случае надобности символьную машину
void InitSymEng ( void ) ;

// Функция, очищающая в случае надобности символьную машину
void CleanupSymEng ( void ) ;

/*//////////////////////////////////////
// Класс деструктора
//////////////////////////////////////*/

```

см. след. стр.

```

// См. примечание об автоматических классах в MEMDUMPERVALIDATOR.CPP.
// Отключение предупреждения "инициализаторы в области инициализации
// библиотеки" (initializers put in library initialization area)
#pragma warning (disable : 4073)
#pragma init_seg(lib)
class CleanupCrashHandler
{
public :
    CleanupCrashHandler ( void )
    {
    }
    ~CleanupCrashHandler ( void )
    {
        // Имеются ли неосвобожденные блоки выделенной памяти?
        if ( NULL != g_ahMod )
        {
            VERIFY ( HeapFree ( GetProcessHeap ( ) ,
                                0 ,
                                g_ahMod ) ) ;

            g_ahMod = NULL ;
            // ИСПРАВЛЕННАЯ ОШИБКА. Спасибо Геннадию Майко (Gennady Mayko).
            g_uiModCount = 0 ;
        }
        if ( NULL != g_pfnOrigFilt )
        {
            // Восстановление первоначального фильтра необработанных исключений.
            SetUnhandledExceptionFilter ( g_pfnOrigFilt ) ;
            g_pfnOrigFilt = NULL ;
        }
    }
} ;

// Статический класс
static CleanupCrashHandler g_cBeforeAndAfter ;

/*////////////////////////////////////*/
// Реализация функций обработчика ошибок.
/*////////////////////////////////////*/

BOOL __stdcall SetCrashHandlerFilter ( PFNCHFILTERFN pFn )
{
    // Если pFn равен NULL, новый фильтр необработанных исключений удаляется.
    if ( NULL == pFn )
    {
        if ( NULL != g_pfnOrigFilt )
        {
            // Восстановление первоначального фильтра необработанных исключений.
            SetUnhandledExceptionFilter ( g_pfnOrigFilt ) ;
            g_pfnOrigFilt = NULL ;
            if ( NULL != g_ahMod )

```

```

    {
        // ИСПРАВЛЕННАЯ ОШИБКА:
        // Раньше я вызвал функцию "free" вместо "HeapFree."
        VERIFY ( HeapFree ( GetProcessHeap ( ),
                           0
                           , g_ahMod ) );
        g_ahMod = NULL ;
        // ИСПРАВЛЕННАЯ ОШИБКА. Спасибо Геннадию Майко.
        g_uiModCount = 0 ;
    }
    g_pfnCallBack = NULL ;
}
}
else
{
    ASSERT ( FALSE == IsBadCodePtr ( (FARPROC)pFn ) );
    if ( TRUE == IsBadCodePtr ( (FARPROC)pFn ) )
    {
        return ( FALSE ) ;
    }
    g_pfnCallBack = pFn ;

    // Если новый обработчик ошибок еще не установлен, выполняется
    // установка CrashHandlerExceptionFilter; первоначальный
    // фильтр необработанных исключений при этом сохраняется.
    if ( NULL == g_pfnOrigFilt )
    {
        g_pfnOrigFilt =
            SetUnhandledExceptionFilter(CrashHandlerExceptionFilter);
    }
}
return ( TRUE ) ;
}

BOOL __stdcall AddCrashHandlerLimitModule ( HMODULE hMod )
{
    // Проверка тривиального случая.
    ASSERT ( NULL != hMod ) ;
    if ( NULL == hMod )
    {
        return ( FALSE ) ;
    }

    // Создание временного массива. Он должен создаваться в той памяти,
    // которая точно будет в нашем распоряжении даже при плохом
    // самочувствии процесса. Куча стандартной библиотеки этому условию
    // не удовлетворяет, поэтому я создаю временный массив в куче процесса.
    HMODULE * phTemp = (HMODULE*)
        HeapAlloc ( GetProcessHeap ( )
                   ,
                   HEAP_ZERO_MEMORY |

```

см. след. стр.

```

                                HEAP_GENERATE_EXCEPTIONS
                                (sizeof(HMODULE)*(g_uiModCount+1)) ) ;
ASSERT ( NULL != phTemp ) ;
if ( NULL == phTemp )
{
    TRACE ( "Serious trouble in the house! - "
            "HeapAlloc failed!!!\n"
            );
    return ( FALSE ) ;
}

if ( NULL == g_ahMod )
{
    g_ahMod = phTemp ;
    g_ahMod[ 0 ] = hMod ;
    g_uiModCount++ ;
}
else
{
    // Копирование старых значений.
    CopyMemory ( phTemp
                ,
                g_ahMod
                ,
                sizeof ( HMODULE ) * g_uiModCount ) ;
    // Освобождение старой памяти.
    VERIFY ( HeapFree ( GetProcessHeap ( ) , 0 , g_ahMod ) ) ;
    g_ahMod = phTemp ;
    g_ahMod[ g_uiModCount ] = hMod ;
    g_uiModCount++ ;
}
return ( TRUE ) ;
}

UINT __stdcall GetLimitModuleCount ( void )
{
    return ( g_uiModCount ) ;
}

int __stdcall GetLimitModulesArray ( HMODULE * pahMod , UINT uiSize )
{
    int iRet ;

    __try
    {
        ASSERT ( FALSE == IsBadWritePtr ( pahMod ,
                                           uiSize * sizeof ( HMODULE ) ) ) ;
        if ( TRUE == IsBadWritePtr ( pahMod ,
                                     uiSize * sizeof ( HMODULE ) ) )
        {
            iRet = GLMA_BADPARAM ;
            __leave ;
        }
    }
}

```

```

    if ( uiSize < g_uiModCount )
    {
        iRet = GLMA_BUFFTOOSMALL ;
        __leave ;
    }

    CopyMemory ( pahMod      ,
                 g_ahMod      ,
                 sizeof ( HMODULE ) * g_uiModCount ) ;

    iRet = GLMA_SUCCESS ;
}
__except ( EXCEPTION_EXECUTE_HANDLER )
{
    iRet = GLMA_FAILURE ;
}
return ( iRet ) ;
}

LONG __stdcall CrashHandlerExceptionFilter (EXCEPTION_POINTERS* pExPtrs)
{
    LONG lRet = EXCEPTION_CONTINUE_SEARCH ;

    // Если исключение имеет тип EXCEPTION_STACK_OVERFLOW, с ним почти
    // ничего нельзя сделать из-за плачевного состояния стека. Любые
    // действия скорее всего приведут к еще одной ошибке сразу же около
    // вашего фильтра исключений. Я не рекомендую этого делать, однако
    // вы можете попытаться как-нибудь изменить указатель стека, чтобы
    // освободить место для вызова этих функций. Конечно, если вы измените
    // регистр указателя стека, у вас возникнут проблемы с анализом стека.
    // Я использую безопасный подход и выполняю несколько вызовов функции
    // OutputDebugString. Они также могут привести к повторной ошибке,
    // но попробовать это стоит, так как OutputDebugString требует совсем
    // небольшого пространства в стеке (8-16 байт). Чтобы ваши пользователи
    // могли хотя бы рассказать вам, что они видят, попросите их загрузить
    // с сайта www.sysinternals.com программу DebugView, написанную
    // Марком Руссиновичем (Mark Russinovich). Единственная проблема в том,
    // что в стеке может не быть места даже для размещения указателя команд.
    // К счастью, исключение EXCEPTION_STACK_OVERFLOW случается довольно
    // редко. Вас, возможно, интересует, почему я не вызываю здесь новую
    // функцию _resetstkoflw. Она вызывается только при критических
    // исключениях, поэтому, если приложение "умирает", попытка восстановления
    // стека ни к чему не приведет. Функция _resetstkoflw полезна,
    // только если ее вызвать до этого момента.
    __try
    {

        // Обратите внимание: я все же вызываю ваш обработчик ошибок.
        // Если же переполнение стека нарушит работу вашего
        // обработчика ошибок, я вывожу информацию о типе исключения.

```

см. след. стр.

```

if ( EXCEPTION_STACK_OVERFLOW ==
        pExPtrs->ExceptionRecord->ExceptionCode )
{
    OutputDebugString(_T("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n"));
    OutputDebugString(_T("EXCEPTION_STACK_OVERFLOW occurred\n"));
    OutputDebugString(_T("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n"));
}

if ( NULL != g_pfnCallBack )
{
    // Сейчас нужно инициализировать символьную машину,
    // чтобы подготовить ее к работе и иметь возможность
    // получения базового адреса модуля по адресу ошибки.
    InitSymEng ( ) ;

    // Проверка списка g_ahMod.
    BOOL bCallIt = FALSE ;
    if ( 0 == g_uiModCount )
    {
        bCallIt = TRUE ;
    }
    else
    {
        HINSTANCE hBaseAddr = (HINSTANCE)
            SymGetModuleBase64( GetCurrentProcess ( )
                               ,
                               (DWORD64)pExPtrs->
                               ExceptionRecord->
                               ExceptionAddress);

        if ( NULL != hBaseAddr )
        {
            for ( UINT i = 0 ; i < g_uiModCount ; i ++ )
            {
                if ( hBaseAddr == g_ahMod[ i ] )
                {
                    bCallIt = TRUE ;
                    break ;
                }
            }
        }
    }
}

if ( TRUE == bCallIt )
{
    // Прежде чем вызвать обработчик ошибок, я проверяю его
    // наличие в памяти. Пользователь может забыть отменить
    // регистрацию, и обработчик ошибок может быть
    // некорректным, если он был выгружен. Однако, если
    // по тому же адресу будет загружена какая-нибудь другая
    // функция, я ничего не смогу сделать.
    ASSERT ( FALSE == IsBadCodePtr((FARPROC)g_pfnCallBack));
}

```

```

        if ( FALSE == IsBadCodePtr ( (FARPROC)g_pfnCallBack ) )
        {
            lRet = g_pfnCallBack ( pExPtrs ) ;
        }
    }
    else
    {
        // Вызов предыдущего фильтра, но только после
        // проверки. Я становлюсь немного подозрительным.
        ASSERT ( FALSE == IsBadCodePtr((FARPROC)g_pfnOrigFilt));
        if ( FALSE == IsBadCodePtr ( (FARPROC)g_pfnOrigFilt ) )
        {
            lRet = g_pfnOrigFilt ( pExPtrs ) ;
        }
    }
    CleanupSymEng ( ) ;
}

__except ( EXCEPTION_EXECUTE_HANDLER )
{
    lRet = EXCEPTION_CONTINUE_SEARCH ;
}
return ( lRet ) ;
}

/*//////////////////////////////////////
// Реализация функций преобразования структуры EXCEPTION_POINTERS.
////////////////////////////////////*/

LPCTSTR __stdcall GetFaultReason ( EXCEPTION_POINTERS * pExPtrs )
{
    ASSERT ( FALSE == IsBadReadPtr ( pExPtrs ,
                                     sizeof ( EXCEPTION_POINTERS ) ) ) ;
    if ( TRUE == IsBadReadPtr ( pExPtrs ,
                               sizeof ( EXCEPTION_POINTERS ) ) )
    {
        TRACE0 ( "Bad parameter to GetFaultReason\n" ) ;
        return ( NULL ) ;
    }

    // Переменная для хранения возвращаемого значения
    LPCTSTR szRet ;

    __try
    {

        // Инициализация символьной машины, если она не инициализирована.
        InitSymEng ( ) ;

        // Текущая позиция в буфере

```

см. след. стр.

```

int iCurr = 0 ;
// Переменная для хранения временного значения. Это
// позволяет свести использование стека к минимуму.
DWORD64 dwTemp ;

iCurr += BSUGetModuleBaseName ( GetCurrentProcess ( ) ,
                                NULL ,
                                g_szBuff ,
                                BUFF_SIZE ) ;

iCurr += wsprintf ( g_szBuff + iCurr , _T ( " caused an " ) ) ;

dwTemp = (DWORD_PTR)
    ConvertSimpleException(pExPtrs->ExceptionRecord->
                           ExceptionCode);

if ( NULL != dwTemp )
{
    iCurr += wsprintf ( g_szBuff + iCurr ,
                        _T ( "%s" ) ,
                        dwTemp ) ;
}
else
{
    iCurr += FormatMessage( FORMAT_MESSAGE_IGNORE_INSERTS |
                           FORMAT_MESSAGE_FROM_HMODULE,
                           GetModuleHandle (_T("NTDLL.DLL")) ,
                           pExPtrs->ExceptionRecord->
                               ExceptionCode ,
                           0 ,
                           g_szBuff + iCurr ,
                           BUFF_SIZE ,
                           0 ) ;
}

ASSERT ( iCurr < ( BUFF_SIZE - MAX_PATH ) ) ;

iCurr += wsprintf ( g_szBuff + iCurr , _T ( " in module " ) ) ;

dwTemp =
    SymGetModuleBase64( GetCurrentProcess ( ) ,
                       (DWORD64)pExPtrs->ExceptionRecord->
                           ExceptionAddress ) ;

ASSERT ( NULL != dwTemp ) ;

if ( NULL == dwTemp )
{
    iCurr += wsprintf ( g_szBuff + iCurr , _T ( "<UNKNOWN>" ) ) ;
}
else

```



```

    {
        iCurr += BSUGetModuleBaseName ( GetCurrentProcess ( ),
                                         (HINSTANCE)dwTemp
                                         ,
                                         g_szBuff + iCurr
                                         ,
                                         BUFF_SIZE - iCurr
                                         ) ;
    }

#ifdef _WIN64
    iCurr += wsprintf ( g_szBuff + iCurr
                        ,
                        _T ( " at %016X" )
                        ,
                        pExPtrs->ExceptionRecord->ExceptionAddress);
#else
    iCurr += wsprintf ( g_szBuff + iCurr
                        ,
                        _T ( " at %04X:%08X" )
                        ,
                        pExPtrs->ContextRecord->SegCs
                        ,
                        pExPtrs->ExceptionRecord->ExceptionAddress);
#endif

    ASSERT ( iCurr < ( BUFF_SIZE - 200 ) ) ;

    // Начало поиска адреса исключения.
    PIMAGEHLP_SYMBOL64 pSym = (PIMAGEHLP_SYMBOL64)&g_stSymbol ;
    ZeroMemory ( pSym , SYM_BUFF_SIZE ) ;
    pSym->SizeOfStruct = sizeof ( IMAGEHLP_SYMBOL64 ) ;
    pSym->MaxNameLength = SYM_BUFF_SIZE -
                        sizeof ( IMAGEHLP_SYMBOL64 ) ;

    DWORD64 dwDisp ;
    if ( TRUE ==
        SymGetSymFromAddr64 ( GetCurrentProcess ( )
                             ,
                             (DWORD64)pExPtrs->ExceptionRecord->
                                 ExceptionAddress ,
                             &dwDisp
                             ,
                             pSym
                             ))
    {
        iCurr += wsprintf ( g_szBuff + iCurr , _T ( "," ) ) ;

        // Копируемая в буфер информация о символах
        // не должна превышать объем свободного места.
        // Помните: имена символов имеют формат ANSI!
        int iLen = lstrlenA ( pSym->Name ) ;
        // Проверка того, что у нас хватает пространства
        // для самого длинного имени символа и смещения.
        if ( iLen > ( ( BUFF_SIZE - iCurr) -
                      ( MAX_SYM_SIZE + 50 ) ) )
        {
#ifdef UNICODE
            // Получение места в стеке для преобразования строки.
            TCHAR * pWideName = (TCHAR*)_alloca ( iLen + 1 ) ;

```

```

        BSUAnsi2Wide ( pSym->Name , pWideName , iLen + 1 ) ;

        lstrcpyn ( g_szBuff + iCurr ,
                  pWideName ,
                  BUFF_SIZE - iCurr - 1 ) ;
    #else
        lstrcpyn ( g_szBuff + iCurr ,
                  pSym->Name ,
                  BUFF_SIZE - iCurr - 1 ) ;
    #endif // UNICODE
        // Выход
        szRet = g_szBuff ;
        __leave ;
    }
    else
    {
        if ( dwDisp > 0 )
        {
            iCurr += wsprintf ( g_szBuff + iCurr ,
                               k_NAMEDISPFMT ,
                               pSym->Name ,
                               dwDisp ) ;
        }
        else
        {
            iCurr += wsprintf ( g_szBuff + iCurr ,
                               k_NAMEFMT ,
                               pSym->Name ) ;
        }
    }
}
else
{
    // Если символ не был найден, информация об исходном файле
    // и номере строки также не будет получена, поэтому выходим.
    szRet = g_szBuff ;
    __leave ;
}

ASSERT ( iCurr < ( BUFF_SIZE - 200 ) ) ;

// Поиск информации об исходном файле и номере строки.
ZeroMemory ( &g_stLine , sizeof ( IMAGEHLP_LINE64 ) ) ;
g_stLine.SizeOfStruct = sizeof ( IMAGEHLP_LINE64 ) ;

DWORD dwLineDisp ;
if ( TRUE ==
     SymGetLineFromAddr64 ( GetCurrentProcess ( ) ,
                           (DWORD64)pExPtrs->
                           ExceptionRecord->

```

```

                                ExceptionAddress ,
                                &dwLineDisp      ,
                                &g_stLine         ) )
{
    iCurr += wsprintf ( g_szBuff + iCurr , _T ( "," ) ) ;

    // Копируемая в буфер информация об исходном файле и номере
    // строки не должна превышать объем свободного места.
    int iLen = lstrlenA ( g_stLine.FileName ) ;
    if ( iLen > ( BUFF_SIZE - iCurr -
                  MAX_PATH - 50 ) )
    {
#ifdef UNICODE
        // Получение места в стеке для преобразования строки.
        TCHAR * pWideName = (TCHAR*)_alloca ( iLen + 1 ) ;

        BSUAnsi2Wide(g_stLine.FileName , pWideName , iLen + 1);

        lstrcpy ( g_szBuff + iCurr ,
                  pWideName ,
                  BUFF_SIZE - iCurr - 1 ) ;
#else
        lstrcpy ( g_szBuff + iCurr ,
                  g_stLine.FileName ,
                  BUFF_SIZE - iCurr - 1 ) ;
#endif
    }

#ifdef UNICODE
    // Выход
    szRet = g_szBuff ;
    __leave ;
}
else
{
    if ( dwLineDisp > 0 )
    {
        iCurr += wsprintf ( g_szBuff + iCurr ,
                            k_FILELINEDISPFMT ,
                            g_stLine.FileName ,
                            g_stLine.LineNumber ,
                            dwLineDisp ) ;
    }
    else
    {
        iCurr += wsprintf ( g_szBuff + iCurr ,
                            k_FILELINEFMT ,
                            g_stLine.FileName ,
                            g_stLine.LineNumber ) ;
    }
}
}
}

```

см. след. стр.

```

        szRet = g_szBuff ;
    }
    __except ( EXCEPTION_EXECUTE_HANDLER )
    {
        ASSERT ( !"Crashed in GetFaultReason" );
        szRet = NULL ;
    }
    return ( szRet ) ;
}

// Вспомогательная функция, позволяющая изолировать заполнение
// структуры кадра стека, которое зависит от процессора.
void FillInStackFrame ( PCONTEXT pCtx )
{
    // Инициализация структуры STACKFRAME.
    ZeroMemory ( &g_stFrame , sizeof ( STACKFRAME64 ) ) ;

#ifdef _X86_
    g_stFrame.AddrPC.Offset      = pCtx->Eip    ;
    g_stFrame.AddrPC.Mode        = AddrModeFlat ;
    g_stFrame.AddrStack.Offset   = pCtx->Esp     ;
    g_stFrame.AddrStack.Mode     = AddrModeFlat ;
    g_stFrame.AddrFrame.Offset   = pCtx->Ebp     ;
    g_stFrame.AddrFrame.Mode     = AddrModeFlat ;
#elif _AMD64_
    g_stFrame.AddrPC.Offset      = pCtx->Rip     ;
    g_stFrame.AddrPC.Mode        = AddrModeFlat ;
    g_stFrame.AddrStack.Offset   = pCtx->Rsp     ;
    g_stFrame.AddrStack.Mode     = AddrModeFlat ;
    g_stFrame.AddrFrame.Offset   = pCtx->Rbp     ;
    g_stFrame.AddrFrame.Mode     = AddrModeFlat ;
#elif _IA64_
    #pragma message ( "IA64 NOT DEFINED!!" )
    #pragma FORCE COMPILATION ABORT!
#else
    #pragma message ( "CPU NOT DEFINED!!" )
    #pragma FORCE COMPILATION ABORT!
#endif
}

LPCTSTR BUGSUTIL_DLLINTERFACE __stdcall
GetFirstStackTraceString ( DWORD dwOpts ,
                           EXCEPTION_POINTERS * pExPtrs )
{
    ASSERT ( FALSE == IsBadReadPtr ( pExPtrs
                                     ,
                                     sizeof ( EXCEPTION_POINTERS * )) ) ;
    if ( TRUE == IsBadReadPtr ( pExPtrs
                                ,
                                sizeof ( EXCEPTION_POINTERS * ) ) )
    {
        TRACE0 ( "GetFirstStackTraceString - invalid pExPtrs!\n" ) ;
    }
}

```

```

        return ( NULL ) ;
    }

    // Заполнение структуры кадра стека.
    FillInStackFrame ( pExPtrs->ContextRecord ) ;

    // Чтобы не повредить поля структуры EXCEPTION_POINTERS,
    // я выполняю их копирование.
    g_stContext = *(pExPtrs->ContextRecord) ;

    return ( InternalGetStackTraceString ( dwOpts ) ) ;
}

LPCTSTR BUGSUTIL_DLLINTERFACE __stdcall
    GetNextStackTraceString ( DWORD dwOpts ,
                             EXCEPTION_POINTERS * /*pExPtrs*/)
{
    // Вся проверка ошибок выполняется в InternalGetStackTraceString.
    // Предполагается, что GetFirstStackTraceString уже инициализировала
    // информацию о кадре стека.
    return ( InternalGetStackTraceString ( dwOpts ) ) ;
}

BOOL __stdcall CH_ReadProcessMemory ( HANDLE hProcess ,
                                      DWORD64 qwBaseAddress ,
                                      PVOID lpBuffer ,
                                      DWORD nSize ,
                                      LPDWORD lpNumberOfBytesRead )
{
    return ( ReadProcessMemory ( GetCurrentProcess ( ) ,
                                (LPCVOID)qwBaseAddress ,
                                lpBuffer ,
                                nSize ,
                                lpNumberOfBytesRead ) ) ;
}

// Внутренняя функция, отвечающая за весь анализ стека
LPCTSTR __stdcall InternalGetStackTraceString ( DWORD dwOpts )
{
    // Возвращаемое значение
    LPCTSTR szRet ;
    // Базовый адрес модуля. Я проверяю его сразу же после вызова
    // функции StackWalk, чтобы гарантировать корректность модуля.
    DWORD64 dwModBase ;

    __try
    {
        // Инициализация символьной машины, если она не инициализирована.
        InitSymEng ( ) ;
    }

```

см. след. стр.

```

// Примечание: При использовании функций получения информации
// об исходном файле и номере строки StackWalk
// может вызвать нарушение доступа.
BOOL bSWRet = StackWalk64 ( CH_MACHINE
                            GetCurrentProcess ( )
                            GetCurrentThread ( )
                            &g_stFrame
                            &g_stContext
                            CH_ReadProcessMemory
                            SymFunctionTableAccess64
                            SymGetModuleBase64
                            NULL
                            );
if ( ( FALSE == bSWRet ) || ( 0 == g_stFrame.AddrFrame.Offset ) )
{
    szRet = NULL ;
    __leave ;
}

// Прежде чем я начну все вычислять, мне нужно удостовериться
// в том, что адрес, возвращенный из StackWalk, действительно
// существует. Мне известны случаи, когда StackWalk возвращала
// TRUE, но адрес не относился к модулю данного процесса.
dwModBase = SymGetModuleBase64 ( GetCurrentProcess ( )
                                g_stFrame.AddrPC.Offset ) ;

if ( 0 == dwModBase )
{
    szRet = NULL ;
    __leave ;
}

int iCurr = 0 ;

// Как минимум помещаем в буфер адрес.
#ifdef _WIN64
    iCurr += wsprintf ( g_szBuff + iCurr
                        ,
                        _T ( "0x%016X" )
                        ,
                        g_stFrame.AddrPC.Offset ) ;
#else
    iCurr += wsprintf ( g_szBuff + iCurr
                        ,
                        _T ( "%04X:%08X" )
                        ,
                        g_stContext.SegCs
                        ,
                        g_stFrame.AddrPC.Offset ) ;
#endif

// Выводить параметры?
if ( GSTSO_PARAMS == ( dwOpts & GSTSO_PARAMS ) )
{
    iCurr += wsprintf ( g_szBuff + iCurr
                        ,
                        k_PARAMFMTSTRING
                        ,
                        g_stFrame.Params[ 0 ]
                        ,

```

```

        g_stFrame.Params[ 1 ]      ,
        g_stFrame.Params[ 2 ]      ,
        g_stFrame.Params[ 3 ]      ) ;
    }
    // Выводить имя модуля?
    if ( GSTSO_MODULE == ( dwOpts & GSTSO_MODULE ) )
    {
        iCurr += wsprintf ( g_szBuff + iCurr , _T ( " " ) ) ;

        ASSERT ( iCurr < ( BUFF_SIZE - MAX_PATH ) ) ;
        iCurr += BSUGetModuleBaseName ( GetCurrentProcess ( ) ,
                                         (HINSTANCE)dwModBase ,
                                         g_szBuff + iCurr ,
                                         BUFF_SIZE - iCurr      ) ;
    }

    ASSERT ( iCurr < ( BUFF_SIZE - MAX_PATH ) ) ;
    DWORD64 dwDisp ;

    // Выводить имя символа?
    if ( GSTSO_SYMBOL == ( dwOpts & GSTSO_SYMBOL ) )
    {
        // Начало поиска адреса исключения.
        PIMAGEHLP_SYMBOL64 pSym = (PIMAGEHLP_SYMBOL64)&g_stSymbol ;
        ZeroMemory ( pSym , SYM_BUFF_SIZE ) ;
        pSym->SizeOfStruct = sizeof ( IMAGEHLP_SYMBOL64 ) ;
        pSym->MaxNameLength = SYM_BUFF_SIZE -
                               sizeof ( IMAGEHLP_SYMBOL64 ) ;
        pSym->Address = g_stFrame.AddrPC.Offset ;

        if ( TRUE ==
              SymGetSymFromAddr64 ( GetCurrentProcess ( ) ,
                                   g_stFrame.AddrPC.Offset ,
                                   &dwDisp ,
                                   pSym ) )
        {
            if ( dwOpts & ~GSTSO_SYMBOL )
            {
                iCurr += wsprintf ( g_szBuff + iCurr , _T ( ", " ) ) ;
            }

            // Копируемая в буфер информация о символах
            // не должна превышать объем свободного места.
            // Имена символов имеют формат ANSI
            int iLen = ( lstrlenA ( pSym->Name ) * sizeof ( TCHAR ) ) ;
            if ( iLen > ( BUFF_SIZE - iCurr -
                          ( MAX_SYM_SIZE + 50 ) ) )
            {

```

```

#ifdef UNICODE

```

см. след. стр.

```

        // Получение места в стеке для преобразования строки.
        TCHAR * pWideName = (TCHAR*)_alloca ( iLen + 1 );

        BSUAnsi2Wide ( pSym->Name , pWideName , iLen + 1 );

        lstrcpy ( g_szBuff + iCurr ,
                  pWideName ,
                  BUFF_SIZE - iCurr - 1 );
    #else

        lstrcpy ( g_szBuff + iCurr ,
                  pSym->Name ,
                  BUFF_SIZE - iCurr - 1 );

    #endif // UNICODE

    // Выход
    szRet = g_szBuff ;
    __leave ;
}
else
{
    if ( dwDisp > 0 )
    {
        iCurr += wsprintf ( g_szBuff + iCurr ,
                            k_NAMEDISPFMT ,
                            pSym->Name ,
                            dwDisp );
    }
    else
    {
        iCurr += wsprintf ( g_szBuff + iCurr ,
                            k_NAMEFMT ,
                            pSym->Name );
    }
}
}
else
{
    // Если символ не был найден, информация об исходном файле
    // и номере строки также не будет получена, поэтому выходим.
    szRet = g_szBuff ;
    __leave ;
}

}

ASSERT ( iCurr < ( BUFF_SIZE - MAX_PATH ) );

// Выводить информацию об исходном файле и номере строки?
if ( GSTSO_SRCLINE == ( dwOpts & GSTSO_SRCLINE ) )
{

```



```

ZeroMemory ( &g_stLine , sizeof ( IMAGEHLP_LINE64 ) );
g_stLine.SizeOfStruct = sizeof ( IMAGEHLP_LINE64 );

DWORD dwLineDisp ;
if ( TRUE == SymGetLineFromAddr64 ( GetCurrentProcess ( ) ,
                                     g_stFrame.AddrPC.Offset,
                                     &dwLineDisp
                                     ,
                                     &g_stLine
                                     ))
{
    if ( dwOpts & ~GSTSO_SRCLINE )
    {
        iCurr += wsprintf ( g_szBuff + iCurr , _T ( ", " ));
    }

    // Копируемая информация об исходном файле и номере
    // строки не должна превышать объем свободного места.
    int iLen = lstrlenA ( g_stLine.FileName );
    if ( iLen > ( BUFF_SIZE - iCurr -
                  ( MAX_PATH + 50 ) ) )
    {
#ifdef UNICODE
        // Получение места в стеке для преобразования строки.
        TCHAR * pWideName = (TCHAR*)_alloca ( iLen + 1 );

        BSUAnsi2Wide ( g_stLine.FileName ,
                       pWideName
                       ,
                       iLen + 1
                       ) ;

        lstrcpy ( g_szBuff + iCurr
                  ,
                  pWideName
                  ,
                  BUFF_SIZE - iCurr - 1 ) ;
#else
        lstrcpy ( g_szBuff + iCurr
                  ,
                  g_stLine.FileName
                  ,
                  BUFF_SIZE - iCurr - 1 ) ;
#endif
    }

    // Выход
    szRet = g_szBuff ;
    __leave ;
}
else
{
    if ( dwLineDisp > 0 )
    {
        iCurr += wsprintf( g_szBuff + iCurr
                           ,
                           k_FILELINEDISPFMT
                           ,
                           g_stLine.FileName
                           ,
                           g_stLine.LineNumber
                           ,
                           dwLineDisp
                           ) ;
    }
}

```

см. след. стр.

```

        }
        else
        {
            iCurr += wsprintf ( g_szBuff + iCurr ,
                                k_FILELINEFMT ,
                                g_stLine.FileName ,
                                g_stLine.LineNumber ) ;
        }
    }
}

szRet = g_szBuff ;
}
__except ( EXCEPTION_EXECUTE_HANDLER )
{
    ASSERT ( !"Crashed in InternalGetStackTraceString" ) ;
    szRet = NULL ;
}
return ( szRet ) ;
}

LPCTSTR __stdcall GetRegisterString ( EXCEPTION_POINTERS * pExPtrs )
{
    // Проверка параметра.
    ASSERT ( FALSE == IsBadReadPtr ( pExPtrs ,
                                     sizeof ( EXCEPTION_POINTERS ) ) ) ;
    if ( TRUE == IsBadReadPtr ( pExPtrs ,
                                sizeof ( EXCEPTION_POINTERS ) ) )
    {
        TRACE0 ( "GetRegisterString - invalid pExPtrs!\n" ) ;
        return ( NULL ) ;
    }

#ifdef _X86_
    // Этот вызов помещает в стек 48 байт, что может
    // представлять проблему, если стек близок к переполнению.
    wsprintf(g_szBuff ,
        _T ("EAX=%08X EBX=%08X ECX=%08X EDX=%08X ESI=%08X\n")\
        _T ("EDI=%08X EBP=%08X ESP=%08X EIP=%08X FLG=%08X\n")\
        _T ("CS=%04X DS=%04X SS=%04X ES=%04X  ")\
        _T ("FS=%04X GS=%04X" ) ,
        pExPtrs->ContextRecord->Eax ,
        pExPtrs->ContextRecord->Ebx ,
        pExPtrs->ContextRecord->Ecx ,
        pExPtrs->ContextRecord->Edx ,
        pExPtrs->ContextRecord->Esi ,
        pExPtrs->ContextRecord->Edi ,
        pExPtrs->ContextRecord->Ebp ,
        pExPtrs->ContextRecord->Esp ,

```

```

        pExPtrs->ContextRecord->Eip      ,
        pExPtrs->ContextRecord->EFlags   ,
        pExPtrs->ContextRecord->SegCs     ,
        pExPtrs->ContextRecord->SegDs     ,
        pExPtrs->ContextRecord->SegSs     ,
        pExPtrs->ContextRecord->SegEs     ,
        pExPtrs->ContextRecord->SegFs     ,
        pExPtrs->ContextRecord->SegGs     ) ;

#elif _AMD64_
    wsprintf ( g_szBuff ,
        _T ("RAX=%016X RBX=%016X RCX=%016X RDX=%016X RSI=%016X\n")\
        _T ("RDI=%016X RBP=%016X RSP=%016X RIP=%016X FLG=%016X\n")\
        _T (" R8=%016X R9=%016X R10=%016X R11=%016X R12=%016X\n")\
        _T ("R13=%016X R14=%016X R15=%016X" ) ,
        pExPtrs->ContextRecord->Rax      ,
        pExPtrs->ContextRecord->Rbx      ,
        pExPtrs->ContextRecord->Rcx      ,
        pExPtrs->ContextRecord->Rdx      ,
        pExPtrs->ContextRecord->Rsi      ,
        pExPtrs->ContextRecord->Rdi      ,
        pExPtrs->ContextRecord->Rbp      ,
        pExPtrs->ContextRecord->Rsp      ,
        pExPtrs->ContextRecord->Rip      ,
        pExPtrs->ContextRecord->EFlags   ,
        pExPtrs->ContextRecord->R8       ,
        pExPtrs->ContextRecord->R9       ,
        pExPtrs->ContextRecord->R10      ,
        pExPtrs->ContextRecord->R11      ,
        pExPtrs->ContextRecord->R12      ,
        pExPtrs->ContextRecord->R13      ,
        pExPtrs->ContextRecord->R14      ,
        pExPtrs->ContextRecord->R15      ) ;

#elif _IA64_
    #pragma message ( "IA64 NOT DEFINED!!" )
    #pragma FORCE COMPILATION ABORT!
#else
    #pragma message ( "CPU NOT DEFINED!!" )
    #pragma FORCE COMPILATION ABORT!
#endif

    return ( g_szBuff ) ;

}

LPCTSTR ConvertSimpleException ( DWORD dwExcept )
{
    switch ( dwExcept )
    {
        case EXCEPTION_ACCESS_VIOLATION :
            return ( _T ( "EXCEPTION_ACCESS_VIOLATION" ) ) ;
    }
}

```

```
break ;

case EXCEPTION_DATATYPE_MISALIGNMENT :
    return ( _T ( "EXCEPTION_DATATYPE_MISALIGNMENT" ) );
break ;

case EXCEPTION_BREAKPOINT :
    return ( _T ( "EXCEPTION_BREAKPOINT" ) );
break ;

case EXCEPTION_SINGLE_STEP :
    return ( _T ( "EXCEPTION_SINGLE_STEP" ) );
break ;

case EXCEPTION_ARRAY_BOUNDS_EXCEEDED :
    return ( _T ( "EXCEPTION_ARRAY_BOUNDS_EXCEEDED" ) );
break ;

case EXCEPTION_FLT_DENORMAL_OPERAND :
    return ( _T ( "EXCEPTION_FLT_DENORMAL_OPERAND" ) );
break ;

case EXCEPTION_FLT_DIVIDE_BY_ZERO :
    return ( _T ( "EXCEPTION_FLT_DIVIDE_BY_ZERO" ) );
break ;

case EXCEPTION_FLT_INEXACT_RESULT :
    return ( _T ( "EXCEPTION_FLT_INEXACT_RESULT" ) );
break ;

case EXCEPTION_FLT_INVALID_OPERATION :
    return ( _T ( "EXCEPTION_FLT_INVALID_OPERATION" ) );
break ;

case EXCEPTION_FLT_OVERFLOW :
    return ( _T ( "EXCEPTION_FLT_OVERFLOW" ) );
break ;

case EXCEPTION_FLT_STACK_CHECK :
    return ( _T ( "EXCEPTION_FLT_STACK_CHECK" ) );
break ;

case EXCEPTION_FLT_UNDERFLOW :
    return ( _T ( "EXCEPTION_FLT_UNDERFLOW" ) );
break ;

case EXCEPTION_INT_DIVIDE_BY_ZERO :
    return ( _T ( "EXCEPTION_INT_DIVIDE_BY_ZERO" ) );
break ;
```

```
case EXCEPTION_INT_OVERFLOW          :
    return ( _T ( "EXCEPTION_INT_OVERFLOW" ) );
break ;

case EXCEPTION_PRIV_INSTRUCTION      :
    return ( _T ( "EXCEPTION_PRIV_INSTRUCTION" ) );
break ;

case EXCEPTION_IN_PAGE_ERROR         :
    return ( _T ( "EXCEPTION_IN_PAGE_ERROR" ) );
break ;

case EXCEPTION_ILLEGAL_INSTRUCTION   :
    return ( _T ( "EXCEPTION_ILLEGAL_INSTRUCTION" ) );
break ;

case EXCEPTION_NONCONTINUABLE_EXCEPTION :
    return ( _T ( "EXCEPTION_NONCONTINUABLE_EXCEPTION" ) );
break ;

case EXCEPTION_STACK_OVERFLOW        :
    return ( _T ( "EXCEPTION_STACK_OVERFLOW" ) );
break ;

case EXCEPTION_INVALID_DISPOSITION    :
    return ( _T ( "EXCEPTION_INVALID_DISPOSITION" ) );
break ;

case EXCEPTION_GUARD_PAGE            :
    return ( _T ( "EXCEPTION_GUARD_PAGE" ) );
break ;

case EXCEPTION_INVALID_HANDLE        :
    return ( _T ( "EXCEPTION_INVALID_HANDLE" ) );
break ;

case 0xE06D7363                      :
    return ( _T ( "Microsoft C++ Exception" ) );
break ;

default :
    return ( NULL );
break ;
}
}

// Инициализация символьной машины в случае надобности.
void InitSymEng ( void )
{
    if ( FALSE == g_bSymEngInit )
```

см. след. стр.

```

{
    // Получение параметров символьной машины.
    DWORD dwOpts = SymGetOptions ( ) ;

    // Включение загрузки информации о номерах строк.
    SymSetOptions ( dwOpts |
                    SYMOPT_LOAD_LINES ) ;

    // Установка флага fInvalidateProcess.
    BOOL bRet = SymInitialize ( GetCurrentProcess ( ) ,
                                NULL ,
                                TRUE ) ;

    ASSERT ( TRUE == bRet ) ;
    g_bSymEngInit = bRet ;
}

// Очистка символьной машины в случае надобности
void CleanupSymEng ( void )
{
    if ( TRUE == g_bSymEngInit )
    {
        VERIFY ( SymCleanup ( GetCurrentProcess ( ) ) ) ;
        g_bSymEngInit = FALSE ;
    }
}

```

Для установки собственной функции-фильтра исключений просто вызовите `SetCrashHandlerFilter`, которая сохраняет указатель на вашу функцию-фильтр исключений в статической переменной и вызывает `SetUnhandledExceptionFilter` для установки действительного фильтра исключений — `CrashHandlerExceptionFilter`. Если вы не укажете модулей, ограничивающих фильтрацию исключений, `CrashHandlerExceptionFilter` будет всегда вызывать ваш обработчик исключений независимо от того, в каком модуле произошла ошибка. Это было сделано намеренно, чтобы вы могли устанавливать собственный заключительный обработчик исключений единственным вызовом API. Лучше всего вызывать `SetCrashHandlerFilter` пораньше и обязательно вызывать ее еще раз с параметром `NULL` прямо перед выгрузкой фильтра, чтобы мой обработчик мог удалить вашу функцию-фильтр. Диаграмма обработчика ошибок показана на рис. 13-1.

Добавление модуля, ограничивающего обработку ошибок, выполняет, функция `AddCrashHandlerLimitModule`. Для этого нужно только передать в нее `HMODULE` нужного модуля. Если вы хотите ограничить обработку ошибок несколькими модулями, просто вызовите `AddCrashHandlerLimitModule` для каждого из них. Массив описателей модулей создается в куче основного процесса.

Изучая листинг 13-4, вы увидите, что я не вызываю функций стандартной библиотеки C. Поскольку функции обработчика ошибок вызываются только в экстраординарных ситуациях, я не могу быть уверен в стабильной работе библиотечных функций. Для освобождения выделенной памяти я применяю автоматический

статический класс, деструктор которого вызывается при выгрузке BUGSLAYER-UTIL.DLL. Я также предоставляю две функции, обеспечивающие получение размера массива ограничивающих модулей в элементах и копирование массива: `GetLimitModuleCount` и `GetLimitModulesArray`. Реализацию функции `RemoveCrashHandlerLimitModule` (удаление модуля, ограничивающего обработку ошибок) я оставил вам.

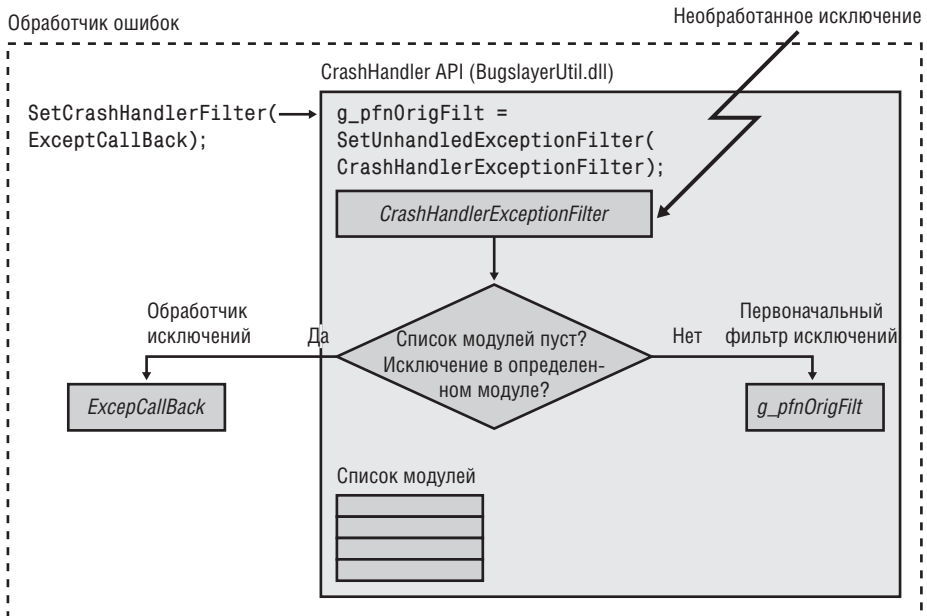


Рис. 13-1. Диаграмма обработчика ошибок

Некоторый интерес представляет то, как я инициализирую сервер символов DBGHELP.DLL. Код обработчика ошибок может быть вызван в любое время, поэтому мне был нужен способ загрузки всех модулей процесса в момент ошибки. Это выполняется автоматически функцией `SymInitialize`, которая получает третий параметр, `fInvalidateProcess`, имеющий значение `TRUE`.

Еще один интересный момент — как я работаю с символами ANSI в мире Unicode. Так как код `CrashHandler`, представленный мной в первом издании, оказался очень популярным и применяется в бесчисленном множестве программ, я должен был учесть потребности программистов, желающих работать с новым `CRASHHANDLER.CPP` в существующих проектах. Я не хотел использовать для преобразования символов свою крупную оболочку `SYMBOLENGINE.LIB` для символьной машины, потому что это препятствовало бы непосредственной модернизации кода. В конце концов я решил выполнять большинство преобразований при помощи функции `wsprintf` с форматом `%S`, который при компиляции Unicode указывает, что параметр является строкой ANSI.

В тех фрагментах, где я должен был выполнять преобразование сам, я решил выделять память в стеке при помощи функции `_alloca`, а не в куче стандартной библиотеки C или ОС, потому что кучи могут быть повреждены и стать причиной ошибки. Если ошибка будет вызвана переполнением стека, любой выполняе-

мый мной код, вероятно, приведет к повторной ошибке задолго до того, как программа достигнет одного из вызовов `_alloca`.

Преобразование структур EXCEPTION_POINTERS

Наверняка вы уже написали свои обработчики исключений и ошибок, поэтому пора поговорить о структурах `EXCEPTION_POINTERS`, указатели на которые передаются в оба обработчика. В этих структурах хранится вся интересная информация об ошибке, поэтому я хотел разработать набор функций, которые вы могли бы вызывать для преобразования сведений в удобную для понимания форму. Благодаря этим функциям вы можете сосредоточиться на отображении пользователям информации в том виде, который уместен для конкретного приложения. Все эти функции можно найти в листинге 13-4.

Я пытался сделать эти функции как можно проще. Все, что вам нужно сделать, — передать в них указатель на структуру `EXCEPTION_POINTERS`. Каждая функция возвращает указатель на глобальную текстовую строку, поэтому я могу не выделять память в куче и всегда уверен в наличии буферов достаточного объема. Кое-кого из вас, наверное, смущает то, что я работаю с глобальными переменными и часто использую буферы, но мне кажется, что это самый безопасный код, который я мог написать.

Функция `GetRegisterString` просто возвращает указатель на отформатированную строку, содержащую значения регистров. Функция `GetFaultReason` чуть интереснее: она возвращает полное описание проблемы. Возвращаемая строка содержит информацию о процессе, причине исключения, модуле, вызвавшем исключение, адресе исключения и — если доступны символы — информацию о функции, исходном файле и номере строки ошибки.

```
CrashHandlerTest.exe caused an EXCEPTION_ACCESS_VIOLATION in module
CrashHandlerTest.exe at 001B:004011D1, Baz()+0088 byte(s),
d:\dev\booktwo\disk\bugslayerutil\tests\crashhandler\crashhandler.cpp,
line 0061+0003 byte(s)
```

Наибольший интерес представляют функции `GetFirstStackTraceString` и `GetNextStackTraceString`. Как показывают имена, эти функции позволяют вам анализировать стек. Как и в случае API-функций `FindFirstFile` и `FindNextFile`, для анализа всего стека вы можете вызвать `GetFirstStackTraceString` и затем продолжить вызывать `GetNextStackTraceString`, пока она не вернет `FALSE`. Второй параметр этих функций является указателем на структуру `EXCEPTION_POINTERS`, передаваемым вашей функции обработчика ошибок. Код обработчика ошибок поступает правильно: он кэширует значение, переданное в `GetFirstStackTraceString`, так что структура `EXCEPTION_POINTERS` в вашем обработчике ошибок остается нетронутой на тот случай, если вы позднее захотите передать указатель на нее в функции записи минидампов. `GetNextStackTraceString` на самом деле не использует переданную ей структуру `EXCEPTION_POINTERS`, но я не хотел нарушать совместимость `CRASHHANDLER.CPP` с программами, которые уже работают с ним.

Первый параметр функций `GetFirstStackTraceString` и `GetNextStackTraceString` — это параметр флагов, позволяющий контролировать объем информации, которую

вы желаете видеть в итоговой строке. Если включены все флаги, будет выведено что-то вроде:

```
001B:004017FD (0x00000001 0x00000000 0x00894D00 0x00000000)
CrashHandlerTest.exe, wmain()+1407 byte(s), d:\dev\booktwo\disk\bugslayerutil\tes
ts\crashhandler\crashhandler.cpp,
line 0226+0007 byte(s)
```

В скобках выводятся первые четыре возможных параметра функции. Список флагов приведен в табл. 13-1. Некоторые из вас, возможно, удивляются, почему в качестве одного из вариантов я не включил вывод информации о локальных переменных. Это объясняется двумя причинами. Во-первых, CrashHandler предназначен прежде всего для использования клиентами. Если вы не желаете выдавать секреты, вы, вероятно, не предоставляете своим клиентам PDB-файлы с частной информацией. Во-вторых, локальные переменные, особенно в расширенном виде, занимают довольно большой объем памяти. Я и так чувствовал, что приближаюсь к пределам возможностей из-за статических буферов, поэтому решил, что описание локальных переменных будет чрезмерным.

Табл. 13-1. Флаги GetFirstStackTraceString и GetNextStackTraceString

Флаг	Выводимая информация
0	Только адрес стека
GSTSO_PARAMS	Первые четыре возможных параметра
GSTSO_MODULE	Имя модуля
GSTSO_SYMBOL	Имя символа для адреса стека
GSTSO_SRCLINE	Информация об исходном файле и номере строки для адреса стека

Чтобы показать функции `GetFirstStackTraceString` и `GetNextStackTraceString` в действии, я прилагаю к этой книге две тестовых программы: `BugslayerUtil\Tests\CrashHandler` выполняет методы `CrashHandler`, а `CrashTest` отображает пример диалогового окна, которое вы можете вывести при необработанной ошибке. Благодаря этим двум программам вы должны получить достаточно хорошее представление о том, как использовать представленные мной функции. На рис. 13-2 показано окно сообщения об ошибке, выводимое программой `CrashTest`.

Минидампы

Возможно, вы удивляетесь, зачем я продолжаю разработку и сопровождение кода для манипулирования структурами `EXCEPTION_POINTERS` в библиотеке `CrashHandler`, потому что вы много слышали о минидампах. Я делаю это главным образом потому, что не хочу нарушать совместимость моего кода со многими программами, в которых он уже используется. Однако возможности минидампов настолько удивительны, что я уверен, что многие программисты просто заменят код своих обработчиков ошибок вызовами функций создания минидампов, причем сделают это так быстро, как только смогут.

Я уже объяснял, как читать файлы минидампов при помощи `Microsoft Visual Studio .NET` и `WinDBG` в главах 7 и 8 соответственно. Теперь я хочу рассказать, как создавать собственные минидампы прямо из своей программы. Я считаю, что API

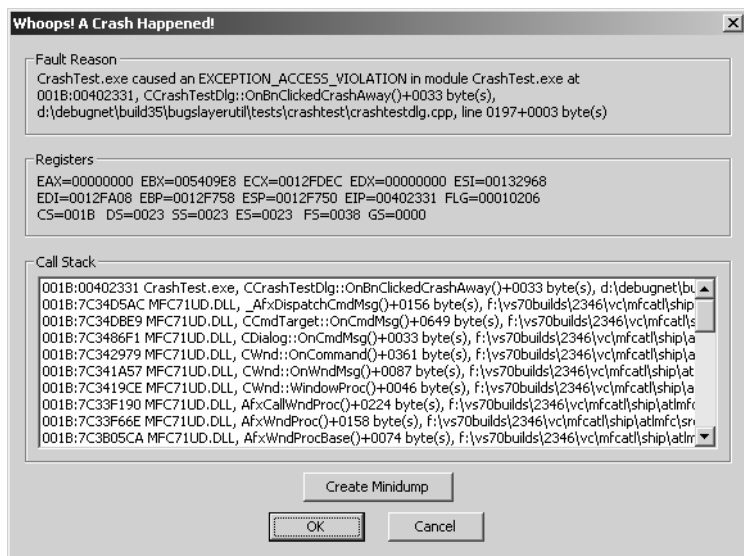


Рис. 13-2. Диалоговое окно программы *CrashTest*

минидампов — самое полезное средство, предоставленное Microsoft разработчикам неуправляемого кода за последние несколько лет после технологии серверов символов и самой Visual Studio .NET! Однако создание собственных минидампов имеет свои тонкости, поэтому я покажу вам, как получать самые лучшие минидампы, что позволит вам значительно ускорить исправление ошибок.

API-функция MiniDumpWriteDump

Всю работу по созданию минидампов выполняет функция `MiniDumpWriteDump`, содержащаяся в `DBGHELP.DLL` версии 5.1 или более поздней. Это значит, что все версии этой библиотеки из Microsoft Windows 2000 (до Service Pack 3 включительно) эту функцию не экспортируют. Кроме того, в `MiniDumpWriteDump` из библиотек `DBGHELP.DLL` до версии 6.0 есть ошибка, приводящая к взаимной блокировке при записи минидампа из текущего процесса. К счастью, эти версии библиотек распространялись только с Debugging Tools for Windows, поэтому их не должно быть на машинах пользователей. `DBGHELP.DLL` уже не имеет ограничений на распространение, поэтому, чтобы гарантировать своим приложениям благополучную жизнь, включайте в их состав `DBGHELP.DLL` 6.1.17.1 или ее более позднюю версию и устанавливайте ее в каталог своей программы, но не в каталог `%SYSTEM-ROOT%\System32`. `DBGHELP.DLL` входит в пакет Debugging Tools for Windows (т. е. WinDBG), который вы можете найти на CD. Чтобы получить последнюю версию `DBGHELP.DLL`, зайдите на сайт <http://www.microsoft.com/ddk/debugging/> и загрузите Debugging Tools for Windows. После установки всех компонентов вы сможете скопировать `DBGHELP.DLL` из каталога Debugging Tools for Windows.

В следующем фрагменте представлен прототип `MiniDumpWriteDump`. Имена параметров говорят сами за себя, однако я хотел бы кое-что отметить. Первый параметр, описатель процесса, должен иметь права на чтение и запрос. Так как мно-

гие наши программы выполняются без прав администратора, вам, возможно, придется поработать с параметрами безопасности, чтобы гарантировать права на вызов `MiniDumpWriteDump`, если вы заимствуете атрибуты безопасности (`impersonation`) или иным образом манипулируете атрибутами пользователей и прав. Однако в самых разнообразных типах приложений, в которых я применял `MiniDumpWriteDump`, мне ни разу не приходилось изменять атрибуты безопасности описателя процесса.

```

BOOL MiniDumpWriteDump ( HANDLE          hProcess      ,
                        DWORD            ProcessId      ,
                        HANDLE           hFile           ,
                        MINIDUMP_TYPE    DumpType       ,
                        PMINIDUMP_EXCEPTION_INFORMATION ExceptionParam ,
                        PMINIDUMP_USER_STREAM_INFORMATION UserStreamParam,
                        PMINIDUMP_CALLBACK_INFORMATION CallbackParam );

```

Четвертый параметр — тип дампа, который вы хотите записать. Похоже, это перечисление изменяется с каждой версией Debugging Tools for Windows, поэтому убедитесь, что у вас установлена соответствующая версия Debugging Tools for Windows и установите компоненты SDK для получения последнего заголовочного файла `DBGHELP.H`. Из документации не совсем ясно, что флаги перечисления `MINIDUMP_TYPE` можно объединять, запрашивая запись в дамп дополнительной информации. Что бы вы ни делали, всегда устанавливайте флаг `MiniDumpWithHandleData` для получения информации об описателях.

Если вы работаете над приложениями, к безопасности которых предъявляются повышенные требования, или ваши клиенты очень обеспокоены защитой данных, знайте, что `MiniDumpWriteDump` может вывести информацию, которую выводить не следовало бы. Для защиты пользователей Microsoft включила в `DBGHELP.DLL` 6.1.17.1 и более поздние ее версии два флага: `MiniDumpFilterMemory` и `MiniDumpFilterModulePaths`. Первый удаляет из дампа частные данные, которые не требуются для анализа стека, а второй исключает из путей к модулям имена пользователей и важные имена каталогов. Флаг `MiniDumpFilterModulePaths` очень полезен, однако он может затруднить нахождение модулей в минидампе.

Для нас также представляет интерес пятый параметр — `ExceptionParam`. Для добавления в минидамп информации об ошибке ему нужно присвоить значение указателя на структуру `EXCEPTION_POINTERS`. В качестве двух последних параметров вы почти всегда будете передавать `NULL`. Тем не менее параметр `UserStreamParam` может пригодиться, если вам захочется записать в минидамп собственную информацию: состояние программы, предпочтения пользователей, список объектов и все, что ваша душа пожелает. Чтение пользовательских данных при помощи функции `MiniDumpReadDumpStream` вам придется выполнять самому, но я могу вас обрадовать: содержание минидампа при этом будет ограничено только вашим воображением.

Укрощение `MiniDumpWriteDump`

Увидев `MiniDumpWriteDump`, я сразу же понял, что я должен написать для нее функцию-оболочку. Это было нужно, во-первых, чтобы скрыть функцию `GetProcAddress`, так как я хотел гарантировать работу моего кода под управлением имевшейся версии Windows 2000, и во-вторых, чтобы избежать необходимости открытия файла

перед каждым вызовом `MiniDumpWriteDump`. Создав первую версию простой оболочки, я понял, что мне никогда больше не придется ничего делать, кроме установки параметра `ExceptionParam` для указания на обрабатываемую при ошибке структуру `EXCEPTION_POINTERS`. Моя функция записи минидампа из вашей функции обработчика ошибок называется `CreateCurrentProcessCrashDump`. Я также написал функцию `IsMiniDumpFunctionAvailable`, которая возвращает `TRUE`, если `MiniDumpWriteDump` присутствует в адресном пространстве. Вы можете найти обе функции в файле `MINIDUMP.CPP` утилиты `BugslayerUtil`.

Все шло отлично, пока однажды я не решил написать функцию, которая записывала бы минидамп в любой момент выполнения программы, а не только при ошибке. Мы работали над серверным приложением, и нам хотелось иметь возможность записи минидампа при конкретном внешнем событии. Благодаря этому мы могли бы изучать состояние приложения постфактум, не подключая к компьютеру отладчик. Увы, минидампы, создаваемые `MiniDumpWriteDump`, не всегда были удобочитаемы.

Отображая эти минидампы, `WinDBG` всегда выводил нечто, выглядевшее как фальшивый стек вызовов. `Visual Studio .NET` работала лучше, но иногда показывала странные плавающие стеки, даже когда у меня повсюду были отличные символы. Почесав немного затылок, я понял, в чем дело. `MiniDumpWriteDump` записывала стек вызовов для собственного потока, начинающийся глубоко внутри самой `MiniDumpWriteDump`. Хотя у меня и были отличные символы, изучать код было очень сложно.

Так как я записывал дампы, а не обрабатывал ошибку, я несколько недоумевал. Все файлы дампов, которые я записывал при ошибке, были прекрасно сформированы и читались обоими отладчиками. Конечно, чтобы `WinDBG` читал настоящие файлы дампа ошибки, я должен был использовать команды `.exsg`; `kr` для создания структуры информации об исключении и просмотра стека. Я подумал, что для получения файла минидампа с хорошими стеками вызовов можно реализовать похожую идею: подделать структуру `MINIDUMP_EXCEPTION_INFORMATION`, заполнив ее теми же значениями, которые имеют место при ошибке.

Вся проблема подделки структуры `MINIDUMP_EXCEPTION_INFORMATION` сводится к занесению правильной информации о регистрах в структуру `CONTEXT`, чтобы поддельная ошибка казалась отладчикам настоящей. После многих проб и ошибок я написал функцию `SnapCurrentProcessMiniDump`. Теперь при получении минидампа в любое время всегда будет выполняться правильный анализ стека. Вам, возможно, захочется изучить код листинга 13-5, потому что он работает довольно интересно.

Первая проблема заключалась в том, откуда брать регистры и какое значение надо присваивать адресу исключения. В конце концов я решил, что мне нужны те же значения регистров, которые имеют место при вызове моей функции `SnapCurrentProcessMiniDump`. Чтобы получить правильные значения регистров, я должен был добраться до них раньше кода, сгенерированного компилятором, поэтому я использовал соглашение вызова `naked`.

В итоге я создал структуру `CONTEXT` в стеке и написал на встроенном языке ассемблера копирование регистров в соответствующие поля структуры. Первую ошибку я допустил сам, потому что копировал 16-разрядные сегментные регистры в поля `CONTEXT`, упустив из вида, что поля для этих сегментов были 32-разряд-

ными. В результате я оставлял мусор в старших словах полей. Для исправления этой ошибки мне пришлось сначала копировать сегментные регистры в EAX и потом сохранять его в полях структуры. Значения EBP и ESP нужно было находить иначе, потому что с их помощью в начале функции создавался кадр стека, но и это не вызвало проблем. В структуру заносятся те же значения ESP и EBP, какими они были во время вызова `SnapCurrentProcessMiniDump`. Макрос `SNAPPROLOG` в листинге 13-5 представляет собой пролог, а `SNAPEPILOG` — эпилог, необходимые для функций с соглашением вызова `naked`. Единственный регистр, значение которого не заносится в `CONTEXT` во время пролога, — это регистр EIP, для которого потребовалось чуть больше работы.

Я думал, что регистров общего назначения хватит, но все равно не исключалась возможность того, что для правильного отображения пользовательской информации отладчику нужны и другие регистры, например, регистры для работы с числами с плавающей точкой или дополнительные регистры. Поэтому я решил получить и их значения, вызвав `GetThreadContext` для выполняющегося потока. Так как мой код не изменяет эти регистры, я получаю их действительные значения в момент вызова. Конечно, я передаю в функцию `GetThreadContext` адрес другой структуры `CONTEXT`, иначе я просто перезаписал бы уже сохраненные значения регистров. После получения регистров через `GetThreadContext` я копирую в итоговую структуру значения, сохраненные мной ранее в первой структуре.

Итак, я получил корректные значения регистров, имевшие место при вызове `SnapCurrentProcessMiniDump`. Сначала я хотел поместить в стек адрес возврата и как значение регистра EIP, и как адрес исключения. Получить адрес возврата уже просто, так как Microsoft задокументировала внутреннюю функцию `_ReturnAddress`. С ее помощью вы можете получить адрес возврата из любого места функции. Чтобы задействовать `_ReturnAddress`, вам надо добавить в свой код две следующих строки, чтобы компилятор не жаловался, что функция не определена. Я предпочитаю размещать эти строки в прекомпилированном заголовочном файле, чтобы они были доступны глобально.

```
extern "C" void * _ReturnAddress ( void ) ;  
#pragma intrinsic ( _ReturnAddress )
```

Благодаря тому, что остальные регистры у меня имеют те же значения, какие имели до вызова, очень немногие люди заметили бы разницу, если б я просто использовал адрес возврата как значение EIP и адрес исключения. Однако я поступил более предусмотрительно, изучив значения, отстоящие на несколько байт от адреса возврата, на предмет наличия идентификаторов операций 0xE8 и 0xFF, определяющих команды ближнего и дальнего вызовов соответственно. После поправки все регистры имеют абсолютно правильные значения, такие же, как и при вызове функции. Определение типа `CALL` вы можете увидеть в `CalculateBeginningOfCallInstruction`.

Остальные действия после получения нужных регистров заключаются в заполнении структуры `MINIDUMP_EXCEPTION_INFORMATION`, открытии описателя файла и вызове `MiniDumpWriteDump`. Все это вы можете увидеть в функции `CommonSnapCurrentProcessMiniDump` в листинге 13-5.

Открытие файла, созданного в SnapCurrentProcessMiniDump, аналогично открытию любого другого минидампа. Единственное различие в том, что отладчик сообщит номер исключения как 0 и покажет указатель команд на самой команде CALL. Теперь вы никак не оправдаете отсутствия минидампов. Создайте фоновый поток, ожидающий нужное событие, и при возникновении этого события во внешней программе вызывайте в своем потоке SnapCurrentProcessMiniDump для получения отличных дампов на всем протяжении выполнения своей программы.

Листинг 13-5. SnapCurrentProcessMiniDump и ее друзья из файла MINIDUMP.CPP

```
// Ниже приведены фрагменты из файла MINIDUMP.CPP, иллюстрирующие
// работу функции SnapCurrentProcessMiniDump.

// Расстояние (в байтах) от адреса возврата до команд
// ближнего и дальнего вызовов. Эти значения используются
// в функции CalculateBeginningOfCallInstruction.
#define k_CALLNEARBACK 5
#define k_CALLFARBACK 6

// Общий пролог для функций SnapCurrentProcessMiniDumpA и
// SnapCurrentProcessMiniDumpW, использующих соглашение naked.
#define SNAPPROLOG(Cntx)
__asm PUSH EBP /* Явное сохранение регистра EBP. */ \
__asm MOV EBP, ESP /* Формирование кадра стека. */ \
__asm SUB ESP, __LOCAL_SIZE /* Место для локальных переменных.*/ \
/* Копирование значений всех легкодоступных регистров. */ \
__asm MOV Cntx.Eax, EAX \
__asm MOV Cntx.Ebx, EBX \
__asm MOV Cntx.Ecx, ECX \
__asm MOV Cntx.Edx, EDX \
__asm MOV Cntx.Edi, EDI \
__asm MOV Cntx.Esi, ESI \
/* Я обнуляю весь регистр EAX, но записываю значения сегментов */ \
/* только в его младшее слово. Это гарантирует правильную */ \
/* инициализацию старших слов полей сегментных регистров */ \
/* в структуре CONTEXT, так как на самом деле они 32-разрядные. */ \
__asm XOR EAX, EAX \
__asm MOV AX, GS \
__asm MOV Cntx.SegGs, EAX \
__asm MOV AX, FS \
__asm MOV Cntx.SegFs, EAX \
__asm MOV AX, ES \
__asm MOV Cntx.SegEs, EAX \
__asm MOV AX, DS \
__asm MOV Cntx.SegDs, EAX \
__asm MOV AX, CS \
__asm MOV Cntx.SegCs, EAX \
__asm MOV AX, SS \
__asm MOV Cntx.SegSs, EAX \
/* Получение предыдущего значения EBP. */ \
```

```

__asm MOV  EAX , DWORD PTR [EBP]                                \
__asm MOV  Cntx.Ebp , EAX                                       \
/* Получение предыдущего значения ESP. */                       \
__asm MOV  EAX , EBP                                            \
/* Предыдущее значение регистра ESP на два                     */ \
/* двойных слова превышает значение EBP.                       */ \
__asm ADD  EAX , 8                                              \
__asm MOV  Cntx.Esp , EAX                                       \
/* Сохранение изменяемых регистров. */                         \
__asm PUSH ESI                                                  \
__asm PUSH EDI                                                  \
__asm PUSH EBX                                                  \
__asm PUSH ECX                                                  \
__asm PUSH EDX                                                  \

// Общий эпилог для функций SnapCurrentProcessMiniDumpA и
// SnapCurrentProcessMiniDumpW, использующих соглашение naked.
#define SNAPEPILOG(eRetVal)                                     \
__asm POP   EDX          /* Восстановление                     */ \
__asm POP   ECX          /* сохраненных регистров.             */ \
__asm POP   EBX                                                  \
__asm POP   EDI                                                  \
__asm POP   ESI                                                  \
__asm MOV   EAX , eRetVal /* Возвращаемое значение.           */ \
__asm MOV   ESP , EBP     /* Восстановление указателя стека.  */ \
__asm POP   EBP          /* Восстановление регистра EBP.      */ \
__asm RET                                /* Возврат в вызвавшую функцию. */

BSUMDRET CommonSnapCurrentProcessMiniDump ( MINIDUMP_TYPE eType      ,
                                             LPCWSTR          szDumpName ,
                                             PCONTEXT          pCtx      )
{
    // Надеемся на лучшее.
    BSUMDRET eRet = eDUMP_SUCCEEDED ;

    // Пытался ли я уже получить экспортируемую ф-цию MiniDumpWriteDump?
    if ( ( NULL == g_pfnMDWD ) && ( eINVALID_ERROR == g_eIMDLastError ))
    {
        if ( FALSE == IsMiniDumpFunctionAvailable ( ) )
        {
            eRet = g_eIMDLastError ;
        }
    }
    // Если указатель на MiniDumpWriteDump равен NULL, выполняется выход.
    if ( NULL == g_pfnMDWD )
    {
        eRet = g_eIMDLastError ;
    }

    if ( eDUMP_SUCCEEDED == eRet )

```

см. след. стр.

```

{
    // Вооружившись контекстом, имевшим место во время вызова
    // этой функции, я могу заняться действительной записью дампа.
    // Чтобы все работало должным образом, мне нужно создать
    // впечатление, что произошло исключение. Для этого надо
    // заполнить структуру MINIDUMP_EXCEPTION_INFORMATION.

    EXCEPTION_RECORD stExRec ;
    EXCEPTION_POINTERS stExpPtrs ;
    MINIDUMP_EXCEPTION_INFORMATION stExInfo ;

    // Обнуление всех отдельных значений структур.
    ZeroMemory ( &stExRec , sizeof ( EXCEPTION_RECORD ) ) ;
    ZeroMemory ( &stExpPtrs , sizeof ( EXCEPTION_POINTERS ) ) ;
    ZeroMemory ( &stExInfo , sizeof(MINIDUMP_EXCEPTION_INFORMATION));

    // Присвоение адресу исключения начала команды CALL.
    // Интересно, что код исключения задавать не требуется.
    // При открытии файла .DMP, созданного этим фрагментом
    // программы, в VS.NET код исключения будет иметь вид:
    // 0x00000000: The operation completed successfully.

    // Запрещение предупреждения C4312: 'type cast' : conversion
    // from 'DWORD' к типу 'PVOID' of greater size (преобразование
    // типа 'DWORD' к типу 'PVOID', имеющему больший размер).
    #pragma warning ( disable : 4312 )
    stExRec.ExceptionAddress = (PVOID)(pCtx->Eip) ;
    #pragma warning ( default : 4312 )

    // Заполнение структуры stExpPtrs (типа EXCEPTION_POINTERS).
    stExpPtrs.ContextRecord = pCtx ;
    stExpPtrs.ExceptionRecord = &stExRec ;

    // Наконец я заполняю структуру информации об исключении.
    stExInfo.ThreadId = GetCurrentThreadId ( ) ;
    stExInfo.ClientPointers = TRUE ;
    stExInfo.ExceptionPointers = &stExpPtrs ;

    // Создание файла для записи минидампа.
    HANDLE hFile = CreateFile ( szDumpName
                                ,
                                GENERIC_READ | GENERIC_WRITE ,
                                FILE_SHARE_READ
                                ,
                                NULL
                                ,
                                CREATE_ALWAYS
                                ,
                                FILE_ATTRIBUTE_NORMAL
                                ,
                                NULL
                                ) ;

    ASSERT ( INVALID_HANDLE_VALUE != hFile ) ;
    if ( INVALID_HANDLE_VALUE != hFile )
    {
        // Запись файла минидампа.
    }
}

```



```

        BOOL bRetVal = g_pfnMDWD ( GetCurrentProcess ( ) ,
                                    GetCurrentProcessId ( ) ,
                                    hFile
                                    ,
                                    eType
                                    ,
                                    &stExInfo
                                    ,
                                    NULL
                                    ,
                                    NULL
                                    ) ;

        ASSERT ( TRUE == bRetVal ) ;
        if ( TRUE == bRetVal )
        {
            eRet = eDUMP_SUCCEEDED ;
        }
        else
        {
            eRet = eMINIDUMPWRITEDUMP_FAILED ;
        }
        // Закрытие файла.
        VERIFY ( CloseHandle ( hFile ) ) ;
    }
    else
    {
        {
            eRet = eOPEN_DUMP_FAILED ;
        }
    }
    return ( eRet ) ;
}

BSUMDRET __declspec ( naked )
    SnapCurrentProcessMiniDumpW ( MINIDUMP_TYPE eType
                                ,
                                LPCWSTR szDumpName )

{
    // Место хранения значений регистров,
    // имевших место при вызове этой функции.
    CONTEXT stInitialCtx ;
    // Место хранения заключительных значений регистров.
    CONTEXT stFinalCtx ;
    // Возвращаемое значение.
    BSUMDRET eRet ;
    // Локальное возвращаемое значение типа Boolean.
    BOOL bRetVal ;

    // Выполнение пролога.
    SNAPPROLOG ( stInitialCtx ) ;

    eRet = eDUMP_SUCCEEDED ;

    // Проверка параметра-строки.
    ASSERT ( FALSE == IsBadStringPtr ( szDumpName , MAX_PATH ) ) ;
    if ( TRUE == IsBadStringPtr ( szDumpName , MAX_PATH ) )

```

см. след. стр.

```

{
    eRet = eBAD_PARAM ;
}

if ( eDUMP_SUCCEEDED == eRet )
{
    // Обнуление структуры заключительного контекста.
    ZeroMemory ( &stFinalCtx , sizeof ( CONTEXT ) ) ;

    // Я хочу получить все характеристики контекста.
    stFinalCtx.ContextFlags = CONTEXT_FULL |
                                CONTEXT_CONTROL |
                                CONTEXT_DEBUG_REGISTERS |
                                CONTEXT_EXTENDED_REGISTERS |
                                CONTEXT_FLOATING_POINT ;

    // Получение значений всех регистров для контекста данного потока.
    bRetVal = GetThreadContext ( GetCurrentThread ( ) , &stFinalCtx );
    ASSERT ( TRUE == bRetVal ) ;
    if ( TRUE == bRetVal )
    {
        COPYKEYCONTEXTREGISTERS ( stFinalCtx , stInitialCtx ) ;

        // Получение адреса возврата и адреса команды call,
        // вызвавшей данную функцию. Всем остальным регистрам
        // присвоены значения, которые они имели до вызова,
        // поэтому указатель команд устанавливается аналогично.
        UINT_PTR dwRetAddr = (UINT_PTR)_ReturnAddress ( ) ;
        bRetVal = CalculateBeginningOfCallInstruction ( dwRetAddr ) ;
        ASSERT ( TRUE == bRetVal ) ;
        if ( TRUE == bRetVal )
        {
            // Установка указателя команд на начало команды call.
            stFinalCtx.Eip = (DWORD)dwRetAddr ;

            // Вызов общей функции, выполняющей
            // фактическую запись минидампа.
            eRet = CommonSnapCurrentProcessMiniDump ( eType ,
                                                        szDumpName ,
                                                        &stFinalCtx ) ;
        }
        else
        {
            eRet = eGETTHREADCONTEXT_FAILED ;
        }
    }
}

// Эпилог.
SNAPEPILOG ( eRet ) ;
}

```

```

// Я должен был вынести этот блок за пределы функций
// SnapCurrentProcessMiniDumpA/W, так как они используют
// соглашение вызова naked и поэтому не могут работать с SEH.
BOOL CalculateBeginningOfCallInstruction ( UINT_PTR & dwRetAddr )
{
    BOOL bRet = TRUE ;
    // При обработке исключений я обеспечиваю полную защиту.
    // Мне нужно быть чрезвычайно внимательным, так как я читаю
    // стек и могу исказить его вершину. Я не хочу, чтобы вызов
    // функции SnapCurrentProcessMiniDump приводил к краху
    // приложения, поэтому я обрабатываю все возможные исключения.
    __try
    {
        BYTE * pBytes = (BYTE*)dwRetAddr ;

        if ( 0xE8 == *(pBytes - k_CALLNEARBACK) )
        {
            dwRetAddr -= k_CALLNEARBACK ;
        }
        else if ( 0xFF == *(pBytes - k_CALLFARBACK) )
        {
            dwRetAddr -= k_CALLFARBACK ;
        }
        else
        {
            bRet = FALSE ;
        }
    }
    __except ( EXCEPTION_EXECUTE_HANDLER )
    {
        bRet = FALSE ;
    }
    return ( bRet ) ;
}

```

Резюме

В этой главе я описал обработчики ошибок, включающие в себя обработчики исключений и фильтры необработанных исключений. Обработчики ошибок позволяют получить более подробную информацию об ошибках и оставить у пользователей при этом более благоприятное впечатление. Одно из условий более быстрой отладки состоит в своевременном получении необходимой информации — как раз для этого и нужны обработчики ошибок.

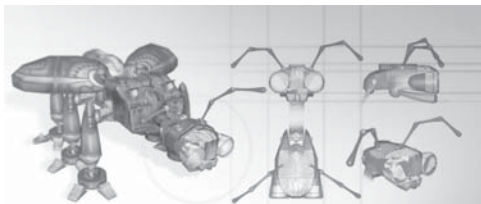
Исключения C++ и исключения SEH иногда путают. Исключения C++ входят в спецификацию языка C++, в то время как исключения SEH обеспечивает ОС; эти два типа обработки исключений абсолютно разные, хотя и тесно сплетены в деталях реализации.

Надеюсь, я смог показать вам всю голую правду об обработке исключений C++, и теперь вы дважды подумаете, прежде чем решитесь использовать их в своем

приложении. Во-первых, из-за дополнительных расходов они ухудшают быстродействие программ. Однако гораздо хуже то, что в реализации компиляторов Microsoft блок `catch (...)` «съедает» ошибки SEH. Фактически это делает блок `catch (...)` самой плохой конструкцией программирования, которую только можно вообразить. Кроме того, нужно избегать функции `_set_se_translator`, потому что она работает не так, как многие предполагают. Возможно, некоторые пуристы объектно-ориентированного программирования и языка C++ найдут мои взгляды на исключения C++ излишне жесткими, но мне кажется, что единственная чистота, которая имеет значение, — это своевременная разработка качественных программ. Если вы откажетесь от обработки исключений C++, ваши программы станут чистыми настолько, насколько это вообще возможно.

Своим существованием обработчики ошибок обязаны магической функции `SetUnhandledExceptionFilter`, позволяющей установить конечный фильтр исключений SEH, который обеспечивает получение управления прямо перед появлением диалогового окна сообщения об ошибке и записать бесценную информацию о причинах проблемы. Приведенные мной функции API `CrashHandler` облегчат установку фильтров необработанных исключений и выполнят за вас всю грязную работу по преобразованию информации об ошибке, благодаря чему вы сможете сосредоточиться на отображении информации и уникальных компонентах своего приложения.

В конце главы я рассказал про одну чрезвычайно полезную новую API-функцию — `MiniDumpWriteDump`. Она имеет несколько недостатков, но я о них позаботился, сделав создание и последующее изучение минидампов максимально удобным. Вооружившись минидампами, созданными в самые подходящие моменты, вы сможете решать любые проблемы своих клиентов.



Отладка служб Windows и DLL, загружаемых в службы

После драйверов устройств сложнее всего отлаживать код служб Microsoft Windows и DLL, загружаемых в службы. Может показаться, что, раз службы являются всего лишь процессами пользовательского режима без UI, то отлаживать их столь же легко, как и консольные приложения. Увы, все не так просто. На самом деле со службами Windows и DLL, загружаемыми в службы, связано столько подводных камней, особенно касающихся защиты в Windows, что при работе с ними вам может захотеться рвать на себе волосы. При появлении Microsoft Windows NT очень немногие разработчики писали службы или вообще знали, что они существуют. Однако в современном мире COM+, Microsoft Internet Information Services (IIS), расширений Microsoft Exchange Server и Windows Clustering многим разработчикам придется иметь дело со службами. И отлаживать их.

В этой главе я представляю обзор основных характеристик служб. Чтобы понять, как отлаживать службы и DLL, загружаемые в службы, такие как ISAPI-фильтры и расширения, надо знать, как службы работают. Затем я поясню аспекты, напрямую связанные с отладкой служб. По мере прохождения этапов отладки службы я буду отмечать моменты, касающиеся определенных технологий служб Microsoft.

Основы служб

Служба характеризуется тремя основными свойствами:

- служба должна работать постоянно, даже если в системе компьютера никто не зарегистрирован или при первоначальном запуске компьютера;

- служба не имеет UI;
- службу могут контролировать и управлять ею как локальные, так и удаленные клиенты.

Решая, как реализовать свое приложение: в виде службы или обычного приложения пользовательского режима, — спросите себя, предъявляются ли к проблеме, которую вы хотите решить, эти три требования. Если да, надо реализовать приложение как службу. А если вы решили написать службу (и хотите ее отлаживать), убедитесь, что вы четко понимаете работу служб. Сведений, приведенных в этом разделе, хватит, чтобы составить представление о том, с чем вам придется столкнуться. Если хотите узнать о службах больше, ознакомьтесь с прекрасной книгой Джефффри Рихтера (Jeffrey Richter) и Джейсона Кларка (Jason Clark) «Programming Server-Side Applications for Microsoft Windows 2000» (Microsoft Press, 2000 г.)¹.

Прекрасный пример случая, когда следует писать службу, — создание приложения, контролирующего источник бесперебойного питания (UPS). Все, что нужно делать ПО для UPS, — следить, когда UPS сообщит о сбое питания в сети, а когда заряд батареи подойдет к концу, программе следует инициировать управляемое выключение (controlled shutdown). Очевидно, что если ПО для UPS не работает постоянно (первый критерий в решении, должно ли приложение быть службой), выключения не произойдет, и, когда в UPS закончится заряд батарей, компьютер просто остановится. В ПО для UPS нет необходимости в UI (второй критерий), так как ему лишь надо выполняться в фоновом режиме, следя за UPS. Наконец, если вы работаете над системой бесперебойного питания для хранилищ данных, системные администраторы определенно захотят проверять состояние удаленных UPS (третий критерий).

Пока все довольно просто. Теперь приступим к работе служб. Первый аспект, о котором я расскажу, — специальные функции API, вызываемые для превращения обычного процесса пользовательского режима в службу.

API

Некоторые качества служб потребуют от вас определенных действий, чтобы приспособиться к ним. Во-первых, не важно, какую точку входа вы используете в службах: `main` или `WinMain`. Поскольку службы не имеют UI, точки входа для консольных приложений или приложений с GUI взаимозаменяемы.

Внутри `main` или `WinMain` прежде всего следует вызвать функцию `API StartServiceCtrlDispatcher`. Ей передается структура `SERVICE_TABLE_ENTRY`, в которой указывается имя и главная точка входа службы. Диспетчер управления службами (Service Control Manager, SCM), запускающий все службы, с которым в конечном счете общается `StartServiceCtrlDispatcher`, чтобы установить вашу службу, является средством ОС, которое, как следует из его названия, управляет всеми службами. Если ваша служба не вызовет `StartServiceCtrlDispatcher` в течение 30 секунд с момента запуска, SCM завершит ее работу. Как вы увидите ниже, такое ограничение по времени может сделать запуск отладки чуть интереснее.

¹ Рихтер Дж., Кларк Дж. Программирование серверных приложений для Microsoft Windows 2000. — М.: «Русская Редакция», 2001. — *Прим. перев.*

Когда вы вызываете SCM, он создает поток для вызова точки входа вашей службы. К точке входа службы предъявляется одно жесткое требование: нужно зарегистрировать обработчик через `RegisterServiceCtrlHandlerEx` и вызвать `SetServiceStatus` в течение 82 секунд с момента запуска. Если за это время служба не выполняет вызовов, SCM считает, что в службе произошел сбой, хотя и не завершает ее. Если в конце концов служба вызовет `RegisterServiceCtrlHandlerEx`, она продолжит выполнение в нормальном режиме. Считая, что произошел сбой, SCM должен бы завершить работу службы, но этого не происходит, — такое поведение, каким бы странным оно ни казалось, облегчает отладку выполняющейся службы.

`RegisterServiceCtrlHandlerEx` принимает еще другой указатель — на функцию-обработчик. SCM вызывает функцию-обработчик для управления рабочими характеристиками службы в таких операциях, как остановка, приостановка или возобновление.

Когда служба переходит между состояниями, запускаясь, останавливаясь и приостанавливаясь, она общается с SCM через функцию API `SetServiceStatus`. Большинству служб надо просто вызвать `SetServiceStatus` и инициализировать основное состояние, в которое они переходят, — в этой функции нет ничего особенного.

Я сгладил некоторые подробности, связанные с функциями API, но обычно вызовы `StartServiceCtrlDispatcher`, `RegisterServiceCtrlHandlerEx` и `SetServiceStatus` — все, что нужно ОС от вашей службы, чтобы обеспечить ее работоспособность. Заметьте, я ничего не упомянул о требованиях к коммуникационным протоколам, используемым службой для связи между написанным вами UI контроллера и вашей службой. К счастью, службы имеют доступ ко всем обычным функциям Windows API, так что вы вправе использовать проецируемые в память файлы (memory-mapped files), почтовые ящики (mail slots), именованные каналы (named pipes) и т. д. В службах вам действительно доступны те же варианты, что и в обычном межпроцессном взаимодействии. Самая сложная проблема со службами, как я говорил в начале главы, — это защита.

Защита

Если не указать иное, службы выполняются под специальной учетной записью `System`. Поскольку Windows для всех объектов реализует защиту на уровне пользователей, учетная запись `System` допустима для машины, а не для сети в целом. Следовательно, процесс под учетной записью `System` не имеет доступа к сетевым ресурсам. Для многих служб, скажем, для упомянутого выше примера с UPS, проблем защиты в процессе разработки может не возникнуть. Но, если вы, например, пытаетесь разделить проецируемую память от службы к клиентскому приложению с UI, а защита установлена неправильно, вы столкнетесь с ошибками нарушения прав доступа от клиентских приложений, когда они будут пытаться проецировать общую память.

К сожалению, отладкой проблем защиты не решить — вам придется обеспечить программирование служб и клиентских приложений с правильно настроенной защитой. Полное описание программирования защиты в Windows займет отдельную книгу, так что приготовьтесь провести некоторое время, планируя программирование защиты с самого начала разработки. Чтобы у вас сложилось представление о диапазоне нюансов с защитой в службах, настоятельно рекомендую ста-

тью Фрэнка Кима (Frank Kim) «Why Do Certain Win32 Technologies Misbehave in Windows NT Services?» в мартовском номере «Microsoft Systems Journal» за 1998 год. Есть и другие прекрасные ресурсы: рубрика Кейта Брауна (Keith Brown) «Security Briefs» в «Microsoft Systems Journal» и его книга «Programming Windows Security» (Addison-Wesley, 2000). Наконец, одна из лучших книг о реальном мире защиты Windows — «Writing Secure Code, Second Edition» Майкла Говарда (Michael Howard) и Дэвида Лебланка (David LeBlanc) (Microsoft Press, 2003)².

Теперь, пронесшись вихрем по службам, обратимся к сердцу этой главы — отладке служб.

Отладка служб

Как вы видели, уже одна уникальная природа служб означает, что вам придется сталкиваться с вопросами, не возникающими при разборке обычных приложений пользовательского режима. Учтите, что до сих пор речь шла лишь о минимальной функциональности, необходимой для службы. Я даже не касался фундаментальных требований обеспечения работы общих алгоритмов и реализаций с особыми элементами службы. Простейший и лучший способ отладки служб без риска быть раздавленным — подойти к отладке поэтапно.

В отладке служб два основных этапа:

- отладка базового кода;
- отладка службы.

Отладка базового кода

Прежде чем даже помыслить о том, чтобы запустить приложение как службу, запускайте и проверяйте приложение как стандартный исполняемый файл пользовательского режима до тех пор, пока весь базовый код не будет отлажен. После этого можно приступать к работе над специфическими проблемами служб.

При отладке базового кода следует отлаживать все на одной машине, работая под учетной записью разработчика; т. е. базовый код службы и весь клиентский код должны находиться на одной машине. Так что вам не придется заботиться о проблемах с защитой или сетью. Отладив логику, можно приступить к радостям других связанных со службами проблем, таких как защита и порядок инициализации служб.

Службы COM+

Если вы собираете службу COM+ с помощью Active Template Library (ATL), вам ничего не нужно делать с безопасностью. По умолчанию ATL запускается как исполняемый файл пользовательского режима пока вы не зарегистрируете свое приложение с параметром командной строки `-Service`.

² Говард М., Лебланк Д. Защищенный код. — М.: Русская Редакция, 2003. — *Прим. перев.*

ISAPI-фильтры и расширения

Экспортируемые функции, которые надо предоставить для фильтров и расширений, довольно просты, и можно легко создать тестовую программу, действующую как фиктивная IIS-система. Вы можете протестировать все свои базовые алгоритмы в контролируемой среде, так что их можно полностью отладить до запуска службы в IIS.

Exchange Server

Можно собирать службы Exchange Server, запускающиеся в виде консольных приложений, используя вспомогательные функции из WINWRAP.LIB. Запуск службы со стартовым параметром `notserv` вызовет ее исполнение в виде обычного процесса; `notserv` должен быть первым среди указанных параметров.

Отладка служб

После тестирования и отладки общей логики можете приступить к отладке вашего кода, выполняющегося как служба. Вся первоначальная отладка должна проходить в системе, где все под вашим контролем. В идеале нужна вторая машина рядом с главной машиной для разработки, которую можно использовать для первоначальной отладки. На второй машине должна быть та же по версии и особенностям система Windows, что рекомендуется пользователям для среды, в которой будет работать ваша служба. Цель отладки базового кода — проверка основной логики, тогда как предварительная отладка службы выполняется, чтобы «перетряхнуть» основной код, относящийся к службе. В ходе отладки вашего первого кода службы следует выполнить четыре задачи:

- включить Allow Service To Interact With Desktop;
- установить идентификационные данные службы;
- подключиться к службе;
- отладить стартовый код.

Включение Allow Service To Interact With Desktop

Независимо от типа отлаживаемой службы следует включить Allow Service To Interact With Desktop (Разрешить взаимодействие с рабочим столом) на вкладке Log On (Вход в систему) диалогового окна Properties (Свойства) службы. Хотя в службе не должно быть элементов UI, уведомления утверждений (assertion notifications), которые позволяют получить управление в отладчике, очень помогут. Уведомления утверждений в сочетании с прекрасным регистрирующим кодом (logging code), таким, какой предоставляет вам ATL для записи в журнал событий, могут облегчить отладку служб.

На начальных стадиях отладки я включаю диалог утверждений SUPERASSERT, чтобы быстро оценить общее состояние моего кода. (О SUPERASSERT см. главу 3.) Но по мере выполнения службы я устанавливаю параметры утверждений так, чтобы все утверждения проходили только через операторы трассировки.

Пока я не уверюсь в коде службы, я обычно оставляю параметр Allow Service To Interact With Desktop включенным. Одну мерзкую ошибку, встретившуюся в написанной мною как-то службе, я долго не мог обнаружить, поскольку я отклю-

чил его при выведенном информационном окне. Поскольку защита ОС не позволяет обычным службам выводить информационные окна, моя служба просто зависала. До того как отключить Allow Service To Interact With Desktop, я дважды удостоверяюсь, что моя служба (и все используемые ею DLL) не вызывает информационные окна, проверяя с помощью DUMPBIN/IMPORTS, что ни `MessageBoxA`, ни `MessageBoxW` не импортируются там, где я этого не жду.

Если для своих утверждений вы используете `SUPERASSERT`, вам повезло. Даже когда вы забываете отключить его диалоговые уведомления, прежде чем отобразить замечательное диалоговое окно с уведомлением, `SUPERASSERT` проверяет, что процесс выполняется с видимым рабочим столом. Вообще эта особенность столь полезна, что я инкапсулировал ее в функции `BSUIsInteractiveUser` (`BSUFUNCTIONS.CPP`) из `BugslayerUtil.DLL`.

Установка идентификационных данных службы

Чтобы избежать проблем с защитой при попытках заставить службу работать, можно установить идентификационные данные службы. По умолчанию все службы выполняются под учетной записью `System`, которая иногда называется учетной записью `LocalSystem`. Однако вы вправе настроить службу на работу под учетной записью пользователя с более высоким уровнем доступа, скажем, члена группы с расширенными правами.

В диалоговом окне `Properties` вашей службы щелкните вкладку `Log On`. Установите переключатель `This Account` (С учетной записью), щелкните кнопку `Browse` (Обзор) и выберите нужную учетную запись из диалогового окна `Select User` (Выбор: Пользователь). Выбрав пользователя, введите и подтвердите пароль для этой учетной записи. Для исполняемых служб COM+ установить идентификационные данные для регистрации позволяет также `DCOMCNFG.EXE`.

Подключение к службе

После запуска службы отладка обычно не так сложна. Все, что надо сделать, — подключиться к процессу службы из отладчика `Microsoft Visual Studio .NET`. В зависимости от службы и сложности кода подключение к службе из отладчика может оказаться единственным, что нужно сделать для отладки. Чтобы подключиться к активному процессу из отладчика `Visual Studio .NET`, сделайте так.

1. Запустите `DEVENV.EXE`.
2. Выбрав `Debug Processes` из меню `Tools`, откройте диалоговое окно `Processes`.
3. Установите флажок `Show System Processes` и щелкните кнопку `Refresh`, чтобы увидеть все процессы службы. Выберите из списка процессы, которые нужно отладить, и щелкните кнопку `Attach`. Если при щелчке кнопки `Attach` нажать любую клавишу `Ctrl`, вы автоматически начнете отладку неуправляемого кода, пропустив диалог `Attach To Process`.
4. Если появился диалог `Attach To Process`, убедитесь, что установлен вариант `Native` и щелкните `OK`.

Прекрасная новинка отладчика `Visual Studio .NET`: теперь вы можете не повторять все предыдущие действия каждый раз, когда надо подключиться к службе, потому что вы можете создать проект, выполняющий подключение к службе, ког-

да вы приступаете к отладке. Давайте создадим специальные проекты подключения (обратите внимание: следующие действия предполагают, что вы запускаете отладчик под учетной записью, имеющей все привилегии администратора и находящейся на машине в группе Debugger Users).

1. Соберите приложение. По завершении сборки закройте существующее решение.
2. Из меню File выберите Open Solution.
3. В диалоговом окне Open Solution измените значение в раскрывающемся списке Files Of Type на Executable Files и перейдите туда, где находится собранный EXE-файл службы. Выберите EXE-файл как решение и щелкните кнопку Open.
4. Сохраните решение, выбрав Save All из меню File, и в диалоговом окне Save File As присвойте ему имя вроде <проект>_Attach.SLN.
5. Щелкните правой кнопкой узел .EXE в окне Solution Explorer и из контекстного меню выберите команду Properties.
6. На странице Debugging диалогового окна Property Pages проекта установите поле Attach на Yes (рис. 14-1).

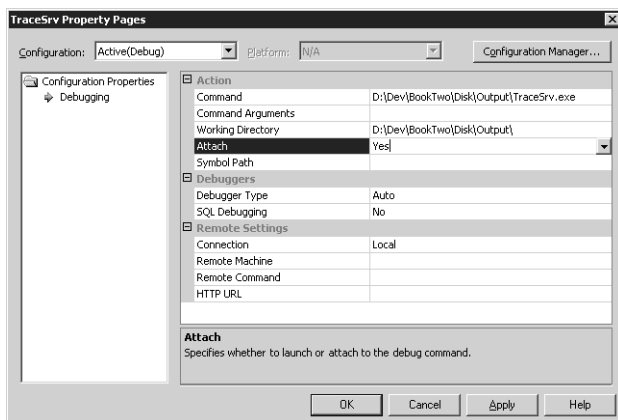


Рис. 14-1. Страница свойств Attach Debugging

7. Щелкните ОК в диалоговом окне Property Pages проекта.
8. Установите и запустите вашу службу обычным способом.
9. Когда будете готовы отлаживать службу, просто нажмите F5 в Visual Studio .NET с загруженным проектом подключения — и все!

В качестве альтернативного метода подключения отладчика можно вызвать функцию `API DebugBreak`. Когда появится диалоговое окно Application Error, просто щелкните кнопку Cancel (Windows 2000) или Debug (Microsoft Windows XP/Server 2003) и отлаживайте, как обычно. Помните: если вы собираете службу COM+, вызов `DebugBreak` следует выполнять за пределами любых COM-методов или инициаций свойств. Иначе COM поглотит исключение точки прерывания, генерируемое `DebugBreak`, и отладчик не будет подключен. Кроме того, не вызывайте `DebugBreak` как часть начального стартового кода службы — причины я объясню в разделе «Отладка стартового кода».

Другой способ подключения отладчика к вашей службе — если вы зарегистрированы в системе с правами администратора — использовать Task Manager. Вызовите Task Manager, выберите вкладку Processes, щелкните правой кнопкой процесс, который хотите отладить, и из контекстного меню выберите команду Debug. ОС позволяет легко подключить отладчик, если вам известно, какой процесс надо отладить.

Подключать отладчик к службам могут только пользователи с правами администратора на локальной машине, иначе при попытке отладить процесс, выполняющийся не под вашей учетной записью, команда Debug выведет информационное окно Unable To Attach Debugger (невозможно подключить отладчик).

ISAPI-фильтры и расширения IIS

До IIS 5 все ISAPI-фильтры выполнялись внутри INETINFO.EXE — главной IIS-службы, т. е. вы просто подключались к INETINFO.EXE и отлаживали единый процесс. В IIS 5 и выше из-за новой объединенной внепроцессной модели расширения выполняются в DLLHOST.EXE. ISAPI-фильтры по-прежнему выполняются внутри IIS-процесса INETINFO.EXE. Новая модель делает IIS гораздо стабильнее и, по утверждению Microsoft, гораздо более масштабируемыми. Единственная проблема с отладкой в том, что вы можете не знать, в каком процессе DLLHOST.EXE выполняется ваше расширение.

В документации IIS сказано, что следует настроить свои расширения на выполнение внутри IIS, чтобы иметь возможность их отлаживать. Единственная проблема изменения места выполнения ваших расширений в том, что развертывать их надо, используя объединенную внепроцессную модель. Поскольку я проповедую отладку в сценариях, близких к тем, в которых будут работать ваши пользователи, я покажу вам трюк, который позволит отлаживать расширения, даже когда они выполняются в DLLHOST.EXE, т. е. там, где они будут работать.

Но, прежде чем поговорить об отладчике, надо знать, как определить, в каком процессе выполняется ваш фильтр или расширение, так как выполняются несколько экземпляров DLLHOST.EXE. Во-первых, скачайте фантастическую бесплатную утилиту Process Explorer с Web-сайта Марка Руссиновича (Mark Russinovich) и Брюса Когсвелла (Bruce Cogswell) www.sysinternals.com. Я впервые упомянул Process Explorer в главе 2, потому что это прекрасный инструмент, которым можно определить, перемещались ли DLL, загруженные в ваше адресное пространство.

Process Explorer покажет вам описатели, открытые процессом, и — главное — какие DLL в какие процессы загружены. Чтобы найти вашу DLL с помощью Process Explorer, сначала нажмите Ctrl+D чтобы указать, что вы хотите просмотреть DLL, затем нажмите Ctrl+F и в диалоговом окне Process Explorer Search укажите имя файла вашей DLL в поле ввода DLL Substring. Щелкните кнопку Search, и Process Explorer покажет список имен и идентификаторов процессов (PID), в которые загружена ваша DLL. Имея PID, вы можете подключить отладчик Visual Studio .NET к процессу командой Debug Process из меню Tools. Не забудьте прочитать врезку о других возможностях Process Explorer, так как это один из лучших инструментов которые можно держать на жестком диске.

Если вы ищете инструмент, эквивалентный Process Explorer, но работающий из командной строки, это TLIST.EXE, поставляемый в комплекте Debugging Tools

for Windows (т. е. WinDBG). Он может показывать MTS-пакеты, а также в какие процессы какие DLL загружены. Запуск `TLIST -?` покажет все параметры командной строки, поддерживаемые `TLIST.EXE`. Ключ `-k` показывает все процессы, содержащие в себе MTS-пакеты, ключ `-m` — какие процессы содержат определенную DLL. Ключ `-m` поддерживает синтаксис регулярных выражений. Так, чтобы увидеть все модули, загружающие `KERNEL32.DLL`, следует указать `*KERNEL32.DLL` как шаблон.

Поскольку вы ищете загруженную DLL, вам, очевидно, придется убедиться, что она загружается, прежде чем отлаживать ее. Фильтры выполняются внутри `INET-INFO.EXE`, так что вы не можете подключить отладчик до запуска служб IIS. Так что, если вы хотите отладить инициализацию, вам не повезло. Если вы отлаживаете расширения, то, проявив изобретательность, вы сможете отладить инициализацию. Идея в том, чтобы создать фиктивное расширение и заставить IIS его загрузить, подключившись к вашему Web-сайту через Microsoft Internet Explorer, что вынудит IIS запустить объединенный внепроцессный исполняемый файл `DLLHOST.EXE`. Обнаружив PID нового `DLLHOST.EXE`, вы сможете подключить отладчик. Затем можно установить точку прерывания на `LdrpRunInitializeRoutines`, чтобы попасть прямо в `DllMain` вашего расширения. В своей рубрике «Under the Hood» («Microsoft Systems Journal», сентябрь 1999) Мэтт Питрек (Matt Pietrek) объясняет, как установить точку прерывания на `LdrpRunInitializeRoutines`. Установив точку прерывания, вы можете загружать настоящее расширение с помощью Internet Explorer и отлаживать инициализацию.

Отладка стартового кода

Самое сложное в отладке служб — отладка стартового кода. SCM будет ждать всего 30 секунд, чтобы служба запустилась и вызвала `StartServiceCtrlDispatcher`, показывая, что выполнение идет нормально. Хотя для процессора это время — почти целая жизнь, его легко можно потратить, пошагово выполняя код и следя за переменными.

Если все, чем вы располагаете, — это отладчик Visual Studio .NET, то единственный корректный способ отладить стартовый код вашей службы — использовать операторы трассировки. `DebugView` Марка Руссиновича (см. главу 3) позволяет видеть операторы по ходу работы службы. К счастью, стартовый код службы обычно проще, чем ее главный код, так что отладка с помощью операторов трассировки не слишком болезненна.

Для служб, не способных запускаться быстро, ограниченное время ожидания SCM может представлять проблему. Медленная аппаратная часть или природа вашей службы иногда могут диктовать большое время запуска. Если ваша служба предрасположена к превышению времени запуска, вам помогут два поля — `dwCheckPoint` и `dwWaitHint`, которые содержит структура `SERVICE_STATUS`, передаваемая `SetServiceStatus`.

Когда ваша служба запускается, вы вправе сообщить SCM, что вы переходите в состояние `SERVICE_START_PENDING`, поместить большое значение в поле `dwWaitHint` (время в мс) и установить поле `dwCheckPoint` в 0, чтобы SCM не использовал стандартные значения времени. Если при старте службы вам нужно больше времени, вы вправе повторять вызов `SetServiceStatus` сколько угодно, увеличивая поле `dwCheckPoint` перед каждым следующим вызовом.

Последнее, что я хотел сказать об отладке стартового кода: SCM будет добавлять записи в журнал событий, объясняя почему он не смог запустить определенную службу. В Event Viewer, найдите в столбце Source строку «Service Control Manager». Если вы также используете журнал событий для легкой трассировки, то среди записей SCM и вашей информации трассировки вы сможете найти решение многих проблем запуска. Если вы используете журнал событий, убедитесь, что взаимосвязи вашей службы установлены так, что ваша служба запускается после службы журнала событий.

Стандартный вопрос отладки

Почему каждому разработчику нужен Process Explorer?

Я уже говорил, что чудесная программа Марка Руссиновича Process Explorer позволяет легко выяснить, какой экземпляр DLLHOST.EXE загрузил DLL и определить, имеются ли в процессе перемещенные DLL. Однако Process Explorer способен на большее — например, быть прекрасным инструментом отладки, и я хочу уделить секунду рассказу о некоторых его замечательных функциях.

По умолчанию Process Explorer обновляется периодически, как Task Manager. Хотя это обновление прекрасно для общего мониторинга, из-за него вы можете пропустить некоторые детали при отладке. Лучше настроить Process Explorer на обновление вручную, выбрав меню View и установив Update Speed на Paused.

Наверное, лучший способ показать вам мощь Process Explorer — небольшая демонстрация. Вы можете повторять все операции, чтобы увидеть инструмент в действии. Первый шаг — запустить Process Explorer, указав далее NOTEPAD.EXE, так как я буду использовать его для демонстрации. Настройте Process Explorer на ручное обновление, выбрав меню View и установив Update Speed на Paused.

Первый трюк, который можно выполнять с помощью Process Explorer, — определение, какие DLL поступают в ваше адресное пространство вследствие определенной операции. В Process Explorer нажмите F5 чтобы обновить экран, выберите экземпляр NOTEPAD.EXE, запущенный секунду назад, и нажмите Ctrl+D, чтобы изменить вид на отображение DLL для Блокнота. Активизируйте Блокнот и выберите Open из его меню File. Оставьте диалоговое окно Open в Блокноте открытым и переключитесь в Process Explorer. Нажмите F5, чтобы обновить отображение в Process Explorer, и вы увидите несколько строк зеленого цвета, появившихся в отображении DLL для NOTEPAD.EXE (рис. 14-2). Зеленый цвет показывает, какие DLL поступили в адресное пространство с момента последнего обновления. Конечно, вы также можете увидеть, какие DLL покинули адресное пространство, переключившись обратно на Блокнот и закрыв диалоговое окно Open, а затем вернувшись в Process Explorer и обновив отображение кнопкой F5. Все DLL, покинувшие адресное пространство, отображаются красным. Эта возможность быстро увидеть, что приходит и уходит из ваших процессов, полезна для определения причин загрузки и выгрузки модулей. Выделение цветом, по-

казывающее, что было загружено и выгружено, также применяется к списку EXE-файлов в верхней половине экрана Process Explorer.

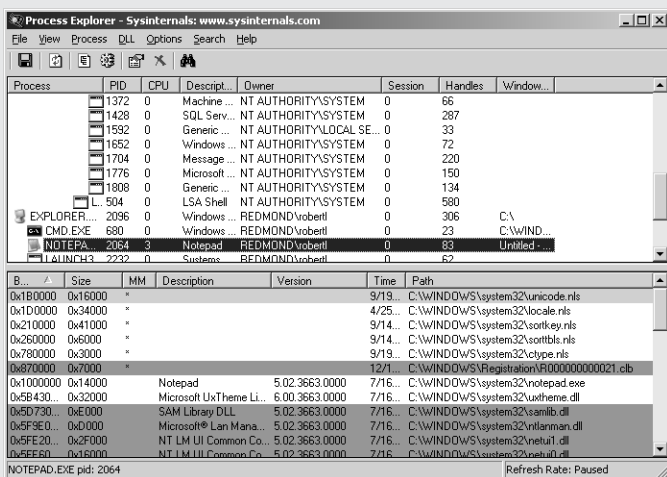


Рис. 14-2. Отображение DLL в Process Explorer, показывающее новые DLL, добавленные в процесс Блокнота

Второй трюк Process Explorer позволяет получать все виды интересной информации о процессе просто двойным щелчком этого процесса. Появляющееся диалоговое окно показывает четыре или пять вкладок в зависимости от процесса. Первая — Image — показывает путь и текущий каталог процесса, а также предлагает кнопку, позволяющую завершить процесс. Вторая — Performance — показывает важные данные о производительности, касающиеся процессора, памяти, ввода/вывода и GDI-описателей. Третья — Security — показывает группы для процессов и предоставленный доступ. Если процесс — хост или служба Microsoft Win32, вкладка Services показывает имена служб, выполняющихся в этом процессе. Последняя вкладка — Environment — показывает список активных для данного процесса переменных окружения. С помощью вкладок Security и Environment я находил некоторые очень интересные проблемы, касающиеся программирования защиты, так как Process Explorer — практически единственный инструмент, позволяющий легко увидеть эту информацию.

Последний трюк Process Explorer позволяет увидеть, какие описатели открыты в данный момент любым процессом! В прошлом я применял эту функцию для обнаружения большого количества разных проблем с описателями. В Process Explorer нажмите Ctrl+N, чтобы изменить нижнюю половину экрана на отображение описателей. Первый отображаемый столбец представляет значение описателя, а второй — тип описателя (объяснение см. в обсуждении `!handle` из главы 8). Третий столбец содержит биты доступа для описателя, а четвертый — имя объекта. Как сказано в главе 8, именование описателей критично для обнаружения проблем. Если вам нужны подробности о каком-то описателе, дважды щелкните его, чтобы увидеть

свойства описателя и больше, чем вы когда-либо хотели, узнать о конкретных значениях этого описателя, связанных с разрешениями.

Как и в отображении DLL, вы можете видеть описатели, создаваемые и закрываемые в процессе. Выберите экземпляр `NOTEPAD.EXE`, запущенный ранее. Нажмите `Ctrl+N`, чтобы перейти к отображению описателей, и обновите содержимое, нажав `F5`. Переключитесь на Блокнот и вновь откройте диалоговое окно `Open`. Когда оно откроется, переключитесь обратно на `Process Explorer` и опять обновите отображение. Все новые описатели в процессе Блокнота выделяются зеленым. Если вы закроете диалоговое окно `Open` Блокнота и еще раз обновите `Process Explorer`, все закрытые описатели будут выделены красным.

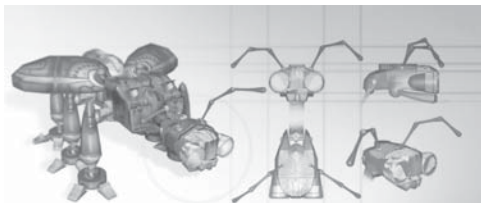
Я использовал отображение описателей в `Process Explorer` для поиска утечек описателей больше, чем могу сосчитать. По умолчанию `Process Explorer` покажет только те описатели, что имеют имена. Вы также можете увидеть все безымянные описатели, нажав `Ctrl+U`. Если вы отслеживаете проблемы с описателями, вам, вероятно, захочется просмотреть все описатели, чтобы видеть все типы, где может быть утечка.

Интересная особенность отображения описателей позволяет принудительно закрыть определенный описатель, щелкнув его правой кнопкой и выбрав `Close Handle`. Когда я спросил Марка, зачем он внес такую функцию, он ответил: «Потому что мог». Когда я засмеялся и сказал, что это было довольно опасно, он сказал, что это мое дело — заботиться о причинах наличия этой функции. Главная причина наугад закрывать описатели в `Process Explorer` — прокрасться в кабинет вашего менеджера и закрыть половину описателей `Outlook`, чтобы он не смог отправлять вам надоедливые сообщения по электронной почте. Я решил, что такой причины вполне достаточно!

Резюме

В этой главе рассказано о некоторых испытаниях и злоключениях, являющихся частью отладки служб `Windows` и `DLL`, загружаемых в службы. Службы обладают особым статусом в ОС, и вследствие проблем, связанных с безопасностью, вам необходимо понимать, что представляют собой службы и как они себя ведут. Отладка служб требует больше предварительного планирования, чем обычная отладка.

Первый шаг в отладке служб и любых `DLL`, загружаемых в службы, — отладка максимального количества базового кода при выполнении в виде обычного приложения. На втором этапе нужно обеспечить использование преимуществ среды для служб, таких как включение взаимодействия с рабочим столом и применение таких инструментов, как `Process Explorer`, для поиска информации, ускоряющей отладку.



Блокировка в многопоточных приложениях

Без сомнения, наибольшие проблемы при разработке современного ПО связаны с многопоточной блокировкой. Даже если вы думаете, что предусмотрели все, ваше многопоточное приложение может зависнуть, когда вы этого меньше всего ждете. Отладка многопоточных блокировок заметно осложняется тем, что после возникновения такой ошибки начинать отладку уже поздно.

В этой главе я опишу некоторые методы и хитрости, помогающие мне при разработке многопоточных программ. Я также представлю свою утилиту *Deadlock-Detection* — почти единственное средство, которое поможет найти причину ошибки и узнать, как избегать некоторых типов блокировки в будущем.

Советы и уловки, касающиеся многопоточности

Как вы знаете, одно из условий успешной отладки — планирование. При работе над многопоточными программами это вообще единственная возможность избежать ужасных блокировок. Все советы по планированию многопоточных приложений я могу систематизировать таким образом:

- не используйте многопоточность;
- не злоупотребляйте многопоточностью;
- делайте многопоточными только небольшие изолированные фрагменты программы;
- выполняйте синхронизацию на как можно более низком уровне;
- работая с критическими секциями, используйте спин-блокировку;
- не используйте функцию `CreateThread`;
- опасайтесь диспетчера памяти по умолчанию;

- получайте дампы в реальных условиях;
- уделяйте особое внимание обзору кода;
- тестируйте многопоточные приложения на многопроцессорных компьютерах;

Не используйте многопоточность

Этот совет может показаться шуткой, но на самом деле я абсолютно серьезен. Прежде чем сделать приложение многопоточным, убедитесь в отсутствии других приемлемых способов его организации. Включив в программу многопоточные фрагменты, можете смело добавлять в свой график минимум один дополнительный месяц на ее разработку и тестирование.

Если вы пишете объемное клиентское приложение, которое должно выполнять в фоновом режиме какую-то нетребовательную задачу, проверьте, можно ли реализовать ее через функцию `OnIdle` библиотеки MFC или периодическое фоновое событие таймера. Подойдя к проблеме творчески, вы скорее всего сможете найти способ избежать многопоточности и связанной с ней головной боли.

Не злоупотребляйте многопоточностью

Разрабатывая серверные приложения, нужно быть чрезвычайно внимательным, чтобы не создать чрезмерное число потоков. Одна из очень частых ошибок при написании серверных приложений состоит в обработке каждого соединения в отдельном потоке. Когда средняя группа разработчиков тестирует программу в самом напряженном режиме, создавая около 10 одновременных соединений, все идет по плану. При первом пробном запуске приложение может работать великолепно, но когда дело доходит до реальных задач, оно начинает тормозить из-за низкой масштабируемости.

При работе над серверными приложениями используйте преимущества пулов потоков, которые прекрасно поддерживаются Microsoft Windows 2000/XP/Server 2003 посредством семейства функций `QueueUserWorkItem`. Это позволяет выполнять тонкую настройку баланса между числом потоком и объемом работы. Программисты привыкли к обработке пулов потоков средствами Microsoft Internet Information Services (IIS) и COM+, однако разработка собственной системы пулинга потоков не относится к тем вещам, которые хорошо знакомы многим программистам, поэтому тщательно проанализируйте собственную ситуацию. При неправильном использовании пулов потоков блокировка становится гораздо более вероятной, чем можно представить.

Делайте многопоточными только небольшие изолированные фрагменты программы

Если многопоточности избежать не удастся, постарайтесь ограничить ее небольшими изолированными фрагментами. В объемных клиентских программах ее следует использовать только для выполнения небольших элементов работы, не связанных, как правило, с пользовательским интерфейсом. В качестве примера разумного использования многопоточности можно привести печать в фоновом режиме, потому что в это время пользовательский интерфейс вашей программы сможет принимать вводимые данные.

В случае же серверных приложений вы должны оценить, действительно ли дополнительные затраты на создание и выполнение потоков приведут к повышению быстродействия программы. Хотя потоки и гораздо «легче» процессов, они требуют большого объема работы. Поэтому убедитесь, что выгода от создания потоков оправдает все затраты. Так, многие серверные приложения должны обмениваться информацией с некоторой базой данных. Цена ожидания записи в базу может быть весьма высока. Если вам не требуется запись транзакций, вы можете создать для записи информации в базу данных отдельный объект пула потоков и продолжить выполнение других задач. Это позволит вам быстрее реагировать на запросы и выполнить больший объем работы.

Выполняйте синхронизацию на как можно более низком уровне

За время, прошедшее с появления первого издания данной книги, я заметил, что это правило многопоточности нарушается чаще, чем прочие. Синхронизацию кода следует выполнять на как можно более низком уровне. Это может казаться самым собой разумеющимся, однако я постоянно сталкиваюсь с ошибками, когда разработчики используют для синхронизации классы-оболочки C++, получающие объект синхронизации в конструкторе и освобождающие его в деструкторе. Вот пример такого класса (вы можете найти его на CD в файле CRITICALSECTION.H):

```
class CUseCriticalSection;

class CCriticalSection
{
public:
    :

    CCriticalSection ( DWORD dwSpinCount = 4000 )
    {
        InitializeCriticalSectionAndSpinCount ( &m_CritSec ,
                                                dwSpinCount );
    }
    ~CCriticalSection ( )
    {
        DeleteCriticalSection ( &m_CritSec );
    }

    friend CUseCriticalSection ;
public:
    :
    CRITICAL_SECTION m_CritSec ;
};

class CUseCriticalSection
{
public:
    :
    CUseCriticalSection ( const CCriticalSection & cs )
    {
        m_cs = &cs ;
    }
};
```

```

        EnterCriticalSection ( ( LPCRITICAL_SECTION)&(m_cs->m_CritSec));
    }

    ~CUseCriticalSection ( )
    {
        LeaveCriticalSection ( (LPCRITICAL_SECTION)&(m_cs->m_CritSec) );
        m_cs = NULL ;
    }

private :
    CUseCriticalSection ( void )
    {
        m_cs = NULL ;
    }
    const CCriticalSection * m_cs ;
};

```

С точки зрения объектно-ориентированного программирования все просто великолепно, но такая реализация пагубно сказывается на быстродействии программы. Объект-оболочка `CUseCriticalSection` создается в начале области видимости своего объявления и уничтожается, когда эта область заканчивается. Почти все программисты используют класс синхронизации так:

```

void DoSomethingMultithreaded ( )
{
    CUseCriticalSection ( g_lpCS ) ;

    for ( . . . )
    {
        CallSomeOtherFunction ( . . . ) ;
    }

    // Это единственный элемент данных, по-настоящему нуждающийся в защите.
    m_xFoo = z ;

    YetAnotherCallHere ( . . . ) ;
}

```

Конструктор получает критическую секцию после первой фигурной скобки, т. е. сразу же после пролога функции, в то время как деструктор вызывается только перед последней фигурной скобкой, перед эпилогом. Это значит, что критическая секция удерживается на протяжении всей функции `DoSomethingMultithreaded`, в том числе когда она вызывает другие функции, которым критическая секция не нужна. Такой подход просто убивает быстродействие.

Взглянув на `DoSomethingMultithreaded`, вы, вероятно, подумали: «Насколько ресурсоемким на самом деле может быть получение объекта синхронизации?» Если конкуренция за объект синхронизации отсутствует, затраты невелики. Однако, если один из потоков многопоточной программы не может получить объект синхронизации, затраты могут быть астрономическими!

Посмотрим, что происходит при вызове `WaitForSingleObject` для получения объекта синхронизации. Так как после чтения главы 7 ваши знания ассемблера приближаются к божественному уровню, вы можете сами проследить за всем в окне Disassembly: оно четко покажет все, о чем я буду рассказывать. Заметьте: я рассматриваю функцию `WaitForSingleObject` из Windows XP — в Windows 2000 она немного иная. Сама по себе `WaitForSingleObject` — это просто оболочка для `WaitForSingleObjectEx`, которая выполняет около 40 строк ассемблерных команд и вызывает две функции для присвоения значений некоторым данным. Незадолго до своего окончания `WaitForSingleObjectEx` вызывает функцию `NtWaitForSingleObject` из `NTDLL.DLL`. Итак, `WaitForSingleObject` — это оболочка для второй оболочки. Если вы дизассемблируете код, начиная с адреса памяти, по которому располагается `NtWaitForSingleObject` (для этого надо ввести в поле Address окна Disassembly выражение `{, ,ntdll}_NtWaitForSingleObject@12`), то узнаете, что на самом деле происходит вызов странной функции `ZwWaitForSingleObject`, которая также находится в `NTDLL.DLL` (в Windows 2000 на функции `NtWaitForSingleObject` вы остановитесь). Взглянув на дизассемблированную функцию `ZwWaitForSingleObject`, вы увидите нечто вроде:

```
_ZwWaitForSingleObject@12:
77F7F4A3  mov     eax, 10Fh
77F7F4A8  mov     edx, 7FFE0300h
77F7F4AD  call    edx
77F7F4AF  ret     0Ch
77F7F4B2  nop
```

Реальные действия происходят по адресу `0x7FFE0300`. Если вы посмотрите, что находится по этому адресу, то увидите:

```
7FFE0300  mov     edx, esp
7FFE0302  sysenter
7FFE0304  ret
```

Среднюю строку во фрагменте занимает магическая команда `SYSENTER`. Вы можете увидеть ее только в этом контексте и никогда — в своем коде, поэтому в главе 7 я ее не описывал. О роли этой команды можно догадаться по названию: она выполняет переключение из пользовательского режима в режим ядра. В Windows 2000 эту же функцию выполняет команда `INT 2E`. Зачем я все это? Просто я хотел показать, что `SYSENTER` отправляет поток в режим ядра, и подчеркнуть все затраты, связанные с выведением потока из очереди потоков, ожиданием и прочими действиями, необходимыми для координации потоков. Разумеется, при переключении в режим ядра, которое требуется для получения объекта ядра, переданного в `WaitForSingleObject`, выполняются тысячи команд, выводящих поток из очереди активных потоков и помещающих его в очередь ожидающих.

Внимательный читатель может подумать, что при вызове `WaitForSingleObject` для ожидания описателя ядра эти затраты неизбежны. Точно: описатели ядра, используемые для синхронизации процессов, выбора не оставляют. Поэтому большинство людей для внутренней синхронизации, которая не требует межпроцессной синхронизации, использует верную критическую секцию, как я показал выше на примере класса `CUseCriticalSection`. Почти все мы читали когда-то, что критические секции хороши тем, что они не требуют переключения в режим ядра. Все

так, однако большинство программистов забывает про одну важную деталь. Что, если получить критическую секцию не удастся? Очевидно, в таких случаях должна быть выполнена какая-то синхронизация. Так и есть: для этого служит описатель семафора (semaphore handle) Microsoft Win32.

Я привел это пространное описание, чтобы объяснить проблему чрезмерно долгого удержания объектов синхронизации. Мне попадались приложения, работу которых удавалось значительно ускорить, просто обнаружив участки конкуренции и удалив классы-оболочки. Я обнаружил, что гораздо лучше явно вызывать функции получения и освобождения объектов синхронизации только до и после фактического доступа к данным, даже если вам понадобится выполнять эти вызовы два, три или больше раз в одной функции. В случае критических секций это дает особенно большой рост быстродействия. Кроме того, использование синхронизации только для фактического доступа к данным — один из лучших методов защиты от случайной блокировки.

Еще раз: ничего плохого в классах-оболочках вроде `CUseCriticalSection` нет — проблема в их неправильном применении. Например, вполне допустимо:

```
void DoSomeGoodMultithreaded ( )
{
    for ( . . . )
    {
        CallSomeOtherFunction ( . . . ) ;
    }

    // Доступ к этому элементу данных нужно защитить,
    // но блокировка не должна быть слишком долгой.
    {
        CUseCriticalSection ( g_lpCS ) ;
        m_xFoo = z ;
    }

    YetAnotherCallHere ( . . . ) ;
}
```

В этом случае также используется вспомогательный класс `CUseCriticalSection`, однако благодаря ограничению его области видимости объект синхронизации приобретает и освобождается в одном локализованном месте и не удерживается слишком долго.

Работая с критическими секциями, используйте спин-блокировку

Как я уже говорил, критические секции — предпочтительный метод синхронизации, если она выполняется только внутри процесса. При этом вы получите большой прирост производительности, если будете помнить про спин-блокировку!

Когда-то Microsoft'овцы заинтересовались производительностью многопоточных приложений и разработали несколько сценариев тестирования, чтобы получить более подробную информацию. После долгих исследований они обнаружи-

ли один противоречащий интуиции, хотя и не новый в области компьютеринга факт: иногда гораздо выгоднее не выполнять операцию на самом деле, а просто подождать. Помните, когда мы только начинали, нам говорили никогда не ждать? Но в случае критических секций именно это и следует делать.

Обычно критические секции применяются для защиты небольших данных. Выше я говорил, что критическая секция защищается семафором и переключение в режим ядра для ее получения очень накладно. В первоначальном варианте функция `EnterCriticalSection` просто узнавала, можно ли получить критическую секцию. Если нет, `EnterCriticalSection` переключалась в режим ядра. Как правило, к тому моменту, когда поток успевает переключиться в режим ядра и обратно, оказывается, что другой поток уже освободил критическую секцию миллион компьютерных лет назад. Странный вывод сотрудников Microsoft заключался в том, что при работе на многопроцессорных системах надо проверять, доступна ли критическая секция, и, если нет, переходить в состояние спин-блокировки и ждать, а потом проверять ее доступность снова. Очевидно, что на однопроцессорных системах счетчик циклов спин-блокировки игнорируется. Если критическая секция недоступна и после второй проверки, выполняется переключение в режим ядра. Суть сказанного в том, что удержание потока в пользовательском режиме в пассивном состоянии все же много выгоднее, чем переключение в режим ядра.

Для присвоения значения счетчику циклов спин-блокировки критической секции служат две функции: `InitializeCriticalSectionAndSpinCount`, которую следует использовать вместо `InitializeCriticalSection`, и `SetCriticalSectionSpinCount`, позволяющая изменить первоначальное значение вашего счетчика или значение счетчика библиотечного кода, использующего только `InitializeCriticalSection`. Разумеется, для этого вам понадобится доступ к указателю на критическую секцию из своего кода.

Подобрать значение счетчика спин-блокировки может оказаться нелегко. Если у вас есть две-три недели для проработки всех сценариев, займите этим начинающих программистов — они все равно бездельничают. Однако большинству из нас не так везет. Я всегда инициализирую этот счетчик значением 4000. Именно оно используется в Microsoft для куч ОС, и я всегда находил свой код менее требовательным, чтобы уменьшать это число. С другой стороны, оно достаточно велико, чтобы почти всегда удерживать код в пользовательском режиме.

Не используйте функции `CreateThread/ExitThread`

Одна из самых коварных ошибок, допускаемых при разработке многопоточных приложений, связана с функцией `CreateThread`. Конечно, возникает вопрос: если потоки нельзя создавать при помощи `CreateThread`, как же их вообще создавать? Вместо `CreateThread` следует всегда использовать `_beginthreadex`, функцию создания потоков из стандартной библиотеки C. Как вы уже догадались, раз уж `CreateThread` дополняет функция `ExitThread` для завершения потока, `_beginthreadex` тоже имеет соответствующую функцию `_exitthreadex`, которую также нужно использовать вместо `ExitThread`.

Возможно, вы вызываете `CreateThread` в своей программе и не испытываете проблем. Увы, при этом возможны очень тонкие ошибки, потому что при использовании `CreateThread` не инициализируется стандартная библиотека C. Работа стан-

дартной библиотеки C основана на некоторых данных, отдельных для каждого потока, и определенные ее функции были разработаны до того, как стали нормой высокопроизводительные многопоточные приложения. Так, функция `strtok` хранит обрабатываемую строку в памяти отдельного потока. Функция `_beginthreadex` гарантирует наличие данных, отдельных для потоков, а также всех остальных вещей, нужных стандартной библиотеке C. Для гарантии правильной очистки потока вызывайте `_exitthreadex`, которая правильно освобождает ресурсы стандартной библиотеки C, если вам нужно преждевременно завершить поток.

`_beginthreadex` работает так же и принимает те же параметры, что и `CreateThread`. Поток завершается возвратом из функции потока или вызовом `_endthreadex`. Для преждевременного завершения потоков служит `_endthreadex`. Как и `CreateThread`, `_beginthreadex` возвращает описатель потока, который нужно затем передать `CloseHandle`, чтобы избежать утечки описателей.

В документации к `_beginthreadex` вы увидите функцию стандартной библиотеки C по имени `_beginthread`. Избегайте ее, как чумы, потому что, по-моему, ее поведение по умолчанию просто ошибочно. Описатель, возвращаемый `_beginthread`, кэшируется, поэтому при быстром завершении потока и его перезаписи другим потоком описатель может оказаться неверным. Даже в документации к `_beginthread` указано, что безопаснее использовать `_beginthreadex`. При обзоре кода отметьте все вызовы `_beginthread` и `_endthread`, чтобы изменить их затем на `_beginthreadex` и `_endthreadex` соответственно.

Опасайтесь диспетчера памяти по умолчанию

Одна из компаний хотела сделать серверное приложение максимально быстрым. Когда программисты обнаружили, что увеличение числа потоков, которое, по их мнению, должно было обеспечивать масштабируемость вычислительной мощности, не возымело эффекта, они обратились к нам. Одна из первых вещей, которые я сделал, заключалась в остановке программы в отладчике и изучении расположения каждого потока при помощи окна `Threads` (потоки).

Приложение интенсивно работало с библиотекой STL, которая, как я говорил при обсуждении WinDBG в главе 8, сама по себе может ухудшать быстродействие, выделяя огромные объемы памяти. Остановив серверное приложение, я хотел увидеть, какие потоки находились в системе управления памятью стандартной библиотеки C. У всех нас есть исходный код управления памятью (ведь вы устанавливаете исходный код стандартной библиотеки C при каждой установке Microsoft Visual Studio, да?), и я увидел, что всю систему управления памятью защищает одна критическая секция. Это всегда пугало меня, так как мне кажется, что это может приводить к проблемам с производительностью. Но когда я взглянул на клиентское приложение, то просто пришел в ужас: 38 из 50 потоков были заблокированы на критической секции системы управления памятью стандартной библиотеки C! Большая часть программы находилась в состоянии ожидания, ничего не делая! Стоит ли говорить, что это не вызвало у программистов особой радости.

Для большинства программ поставляемая Microsoft стандартная библиотека C подходит прекрасно, не вызывая проблем с памятью. Однако в более крупных серверных приложениях одна-единственная критическая секция может все испортить. Итак, прежде всего я хочу порекомендовать вам всегда тщательно обдумы-

вать использование STL и, если избежать этого не удастся, обратите внимание на STLPort версии 1 (см. главу 2). Ранее я уже указывал на многие проблемы с STL. В контексте крупных многопоточных приложений библиотека STL от Microsoft может приводить к появлению узких мест.

Более серьезная проблема: что делать с единственной критической секцией стандартной библиотеки C? Для ее решения нужно предоставить каждому потоку отдельную кучу, а не использовать единственную глобальную кучу для всех потоков. Это позволило бы потокам никогда не переключаться в режим ядра для выделения или освобождения памяти. Конечно, создания отдельной кучи для каждого потока недостаточно, так как порой память выделяется в одном потоке и освобождается в другом. К счастью, эта головоломка имеет три решения.

Первое — коммерческие системы управления памятью, обрабатывающие код работы с кучами отдельных потоков. Жаль, но цены на такие системы просто грабительские, и ваш начальник никогда не согласится на покупку. Второе решение обеспечивает значительное повышение производительности Windows 2000 и основано на усовершенствованиях, внесенных Microsoft в механизм работы куч ОС (куч, создаваемых функцией `HeapCreate` и используемых при помощи `HeapAlloc` и `HeapFree`). Чтобы задействовать преимущества кучи ОС, можно заменить все выделения памяти при помощи `malloc/free` соответствующими функциями `Heap*`. Что до функций `new` и `delete` языка C++, то для их замены нужно предоставить глобальные функции. Если ваша программа будет выполняться на многопроцессорных системах, то третье решение может заключаться в использовании великолепной библиотеки Hoard, написанной Эмери Бергером (Emery Berger) и предназначенной для управления памятью многопроцессорных компьютеров (<http://www.hoard.org>). Эта библиотека заменяет функции работы с памятью C и C++ и очень быстро работает на многопроцессорных системах. Если из-за дублирования символов у вас возникнут проблемы с ее компоновкой, укажите компоновщику LINK.EXE ключ командной строки `/FORCE:MULTIPLE`. Помните, что Hoard предназначена для многопроцессорных систем, поэтому на однопроцессорных компьютерах она может работать даже медленнее, чем диспетчер памяти по умолчанию.

Получайте дампы в реальных условиях

Один из наиболее огорчительных случаев имеет место, когда ваша программа блокируется в реальных условиях и, несмотря на все усилия, вы не можете воспроизвести ошибку. Однако, благодаря последним усовершенствованиям библиотеки DBGHELP.DLL, вы больше никогда не окажетесь в такой ситуации. Новые функции работы с минидампами позволяют сделать снимок блокировки и отладить ее в удобное для вас время. Функцию записи минидампа и мою улучшенную оболочку для нее, `SnapCurrentProcessMiniDump`, находящуюся в библиотеке BUGSLAYER-UTIL.DLL, я описал в главе 13.

Чтобы получить дамп в реальных условиях, нужно просто создать фоновый поток, который создает и ожидает некоторое событие. При возникновении события поток должен вызывать `SnapCurrentProcessMiniDump` и записывать дамп на диск. Соответствующая функция показана в следующем фрагменте псевдокода. Для ус-

тановки события создайте отдельный исполняемый файл и скажите пользователям запускать его в нужной ситуации.

```
DWORD WINAPI DumperThread ( LPVOID )
{
    HANDLE hEvents[2] ;
    hEvents[0] = CreateEvent ( NULL ,
                             TRUE ,
                             FALSE ,
                             _T ( "DumperThread" ) ) ;
    hEvents[1] = CreateEvent ( NULL ,
                             TRUE ,
                             FALSE ,
                             _T ( "KillDumperThread" ) ) ;

    int iRet = WaitForMultipleObjects ( 2 , hEvents , FALSE , INFINITE);
    while ( iRet != 1 )
    {
        // Возможно, каждому файлу следует присваивать уникальное имя.
        SnapCurrentProcessMiniDump ( MiniDumpWithFullMemory ,
                                     _T ( "Program.DMP" ) ) ;
        iRet = WaitForMultipleObjects ( 2 , hEvents , FALSE , INFINITE);
    }
    VERIFY ( CloseHandle ( hEvents[ 0 ] ) ) ;
    VERIFY ( CloseHandle ( hEvents[ 1 ] ) ) ;
    return ( TRUE ) ;
}
```

Уделяйте особое внимание обзору кода

Если вам на самом деле нужно включить в свое приложение многопоточные фрагменты, им нужно уделять повышенное внимание во время обзоров кода. При этом я советую назначать по одному человеку на каждый поток и каждый объект синхронизации. Обзор кода многопоточных приложений «многопоточен» во многих отношениях.

При обзоре кода представьте, что каждый поток выполняется с приоритетом реального времени на собственном процессоре, никогда не прерываясь. Просматривая код, каждый «наблюдатель за потоком» уделяет внимание только тем участкам, которые выполняются его потоком. Когда «наблюдатель за потоком» получает объект синхронизации, к нему подходит «наблюдатель за этим объектом». При освобождении объекта синхронизации «наблюдатель за объектом» уходит в нейтральный угол комнаты. Помимо представителей потоков и объектов, надо назначить нескольких программистов, наблюдающих за общей активностью потоков. Они должны оценивать общий ход выполнения программы и помогать искать места блокировки потоков.

Выполняя обзор кода, помните, что ваш процесс работает и с объектами синхронизации ОС, которые также могут привести к блокировке. В качестве примеров таких объектов можно привести критическую секцию процесса, описывае-

мую в разделе «Отладка: фронтовые очерки. Взаимоблокировка, не имеющая смысла», и печально известный мьютекс Win16 в Microsoft Windows 9x/Me. Обращайте внимание на все, что может вызвать конкуренцию в вашей программе.

Тестируйте многопоточные приложения на многопроцессорных компьютерах

Как я говорил, многопоточные приложения требуют более серьезного тестирования, чем однопоточные. Самый важный совет по поводу тестирования многопоточных приложений таков: тщательно тестируйте их на многопроцессорных компьютерах. Я имею в виду не просто выполнение нескольких тестов, а непрерывное тестирование с использованием всех возможных сценариев. Даже если ваша программа прекрасно работает на однопроцессорных машинах, ее запуск на многопроцессорном компьютере может указать на блокировки, о возможности которых вы даже не подозревали.

В идеале тестирование приложения на многопроцессорных компьютерах нужно выполнять каждый день. Если вы руководитель и в вашем отделе нет многопроцессорных систем, немедленно прекратите чтение и снабдите ими половину своих программистов и тестировщиков отдела контроля качества! Если вы — программист, не имеющий многопроцессорного компьютера, покажите эту главу своему начальнику и потребуйте у него нужное для работы оборудование! Я получил несколько писем, в которых люди утверждали, что это на самом деле помогало, поэтому не колеблясь идите к начальнику и скажите ему, чтобы компания предоставила вам такую систему. Если что, ссылайтесь на Джона Роббинса.

Отладка: фронтовые очерки

Как я спас нескольких людей от увольнения

Боевые действия

Когда вице-президент одной компании позвонил мне и сказал, что хотел бы нанять меня для решения проблемы с блокировкой, я понял, что работа предстоит нелегкая. Он был весьма раздражен и недоволен тем, что его компании пришлось прибегнуть к услугам консультанта. Он позвонил двум ведущим программистам, и мы вчетвером собрались на онлайн-конференцию. Вице-президент злился, что разработчики слишком долго бездействовали из-за этой ошибки. Представляю, как чувствовали себя эти два программиста, когда их грязное белье отправляли по телефону какому-то парню, которого они даже не знали. Как сказал вице-президент, они работали над переносом приложения «с настоящей ОС» (UNIX) на «эту (вырезано цензурой) детскую ОС под названием Windows», и это «вычеркнуло из его жизни целый год». Конечно, когда я спросил его, зачем им понадобилось переносить программу на другую платформу, он признал, что «это было нужно, чтобы остаться на плаву». Я невольно улыбнулся на своем конце провода!

см. след. стр.

Программисты прислали мне по электронной почте код, и мы начали изучать его, в то время как вице-президент с гордым видом расхаживал по своему кабинету. Когда я добрался до места блокировки, у меня тут же выступил холодный пот, а сердце забилося гораздо чаще. Я понимал, что если я скажу, что для исправления ошибки нужно только удалить буквы S, E, N и D и напечатать вместо них P, O, S и T, вице-президент просто сойдет с ума и, вполне возможно, уволит этих двух разработчиков.

Исход

Какое-то время я молчал, собираясь с мыслями. Наконец, глубоко вздохнув и сказав: «Ого, это очень, очень серьезно», — я сообщил вице-президенту, что на исправление проблемы нам потребуется несколько часов. Было бы лучше, если бы мы с программистами остались одни, потому что наверняка у него есть гораздо более важные дела, чем слушать телефонные разговоры, состоящие большей частью из перечисления всяких шестнадцатеричных чисел. Нам повезло: он купился на это, и я сказал программистам, что перезвоню им.

Перезвонив, я сообщил им, что они сделали одну очень частую ошибку, которую допускают многие разработчики программ для UNIX: дело в том, что возврат из функции, посылающей сообщения из одного потока в другой, в некоторых версиях UNIX выполняется сразу. Однако в Windows возврат из `SendMessage` не происходит, пока сообщение не будет обработано. Я нашел в их коде место, где поток, которому они посылали сообщение, уже был заблокирован на объекте синхронизации, поэтому `SendMessage` вызывала блокировку. Когда я сказал им, что для исправления проблемы нужно только заменить `SendMessage` вызовом `PostMessage`, настроение у них резко упало. Тогда я попытался их успокоить, сказав, что непонимание происходящего было вполне законным. Мы провели остаток дня, разбирая другие вопросы, такие как модификация базовых адресов DLL и создание приложений с полным набором отладочных символов. Вернувшись к телефонному разговору с вице-президентом, я сказал ему, что это была одна из самых хитрых ошибок, с которыми мне приходилось сталкиваться, но его программисты оказали мне поистине неоценимую помощь. В итоге все остались довольны. Вице-президент избавился от проблемы, программисты узнали много полезного, а я спас их от увольнения!

Полученный опыт

Если вы работаете над многопоточной программой и хотите передавать сообщения между потоками, тщательно обдумайте взаимодействие объектов синхронизации и сообщений. В подобной ситуации всегда старайтесь использовать `PostMessage`. Конечно, если вы передаете при помощи сообщений значения, превышающие по объему 32 бита, вызовы `PostMessage` не будут работать, потому что переданные вами параметры могут быть повреждены к тому времени, когда другой поток обработает сообщение. В таких случаях вызывайте `SendMessageTimeout` — она по крайней мере выполнит возврат

в какой-то момент времени, и вы сможете узнать, заблокирован ли другой поток и смог ли он обработать сообщение.

Отладка: фронтовые очерки

Взаимоблокировка, не имеющая смысла

Боевые действия

Разрабатывая приложение, программисты столкнулись с непонятной блокировкой. Поборовшись с ней пару дней, они обратились ко мне.

Их программа имела интересную архитектуру и была в высокой степени многопоточной. Блокировка происходила только в определенных случаях, причем всегда в процессе загрузки ряда DLL. Программа блокировалась при вызове функции `WaitForSingleObject`, проверявшей, смог ли поток создать некоторые совместно используемые объекты.

Программисты были опытными и уже перепроверили код несколько раз на пример потенциальных взаимоблокировок, однако остались в полном замешательстве. Я еще раз спросил, искали ли они взаимоблокировки, и они заверили меня, что да.

Исход

Я хорошо помню эту ситуацию, потому что она относится к тем случаям, когда я почувствовал себя героем через 5 минут после запуска отладчика. Как только программисты воспроизвели блокировку, я взглянул в окно Call Stack (стек вызовов) и заметил, что программа ожидала описателя потока внутри функции `DllMain`. При загрузке их программой определенной DLL в ее функции `DllMain` создавался другой поток. Прежде чем продолжить выполнение, `DllMain` вызывала `WaitForSingleObject` для подтверждения события, гарантирующего, что созданный поток смог правильно инициализировать некоторые важные общие объекты.

Разработчики не знали, что каждый процесс имеет критическую секцию процесса, которую ОС использует для синхронизации различных действий, происходящих за кулисами процесса. Помимо прочего, критическая секция процесса служит для сериализации выполнения `DllMain` при четырех вариантах ее вызова: `DLL_PROCESS_ATTACH`, `DLL_THREAD_ATTACH`, `DLL_THREAD_DETACH` и `DLL_PROCESS_DETACH`. На причину вызова `DllMain` указывает ее второй параметр.

Итак, вызов `LoadLibrary` заставлял ОС захватить критическую секцию процесса, чтобы ОС могла вызвать `DllMain` по причине `DLL_PROCESS_ATTACH`. После этого `DllMain` создавала второй поток. При создании процессом нового потока ОС всегда захватывает критическую секцию процесса для вызова функции `DllMain` каждой загруженной DLL по причине `DLL_THREAD_ATTACH`. В этой конкретной программе второй поток блокировался потому, что первый поток удерживал критическую секцию процесса. К несчастью, первый поток вызывал затем `WaitForSingleObject`, чтобы убедиться в правильной

см. след. стр.

инициализации вторым потоком совместно используемых объектов. Так как второй поток был заблокирован на критической секции процесса, удерживаемой первым потоком, а первый поток блокировался в ожидании второго потока, результатом была обычная взаимоблокировка.

Полученный опыт

Урок очевиден: чтобы избежать блокировки, связанной с объектами ядра, не вызывайте внутри `DllMain` функции `Wait*` или `EnterCriticalSection`, потому что критическая секция процесса блокирует остальные потоки. Как вы смогли убедиться, даже опытные программисты ошибаются в многопоточных программах, так что еще раз: проблемы подобного типа часто происходят там, где вы их ожидаете меньше всего.

Требования к DeadlockDetection

Вероятно, вы заметили, что выше я привел мало рекомендаций по поводу исправления блокировок. Большинство советов было профилактическими мерами и касались предотвращения блокировок, а не их разрешения. Всем известно, что исправить блокировки, используя отладчик, непросто. В этом разделе я предоставляю вам дополнительную помощь — утилиту `DeadlockDetection`.

Вот основные требования, которыми я руководствовался при ее разработке.

1. Указание точного места блокировки в пользовательском коде. От утилиты, которая только сообщает, что вызов `EnterCriticalSection` заблокирован, толку мало. Эффективное средство должно указывать адрес (а значит, и исходный файл и номер строки) блокировки, чтобы можно было быстро ее исправить.
2. Отображение объекта синхронизации, вызвавшего блокировку.
3. Вывод информации о заблокированной функции `Windows` и переданных в нее параметрах. Это позволило бы узнать значения тайм-аута и значения, переданные в функцию.
4. Определение потока, вызвавшего блокировку.
5. Утилита должна быть «легкой», чтобы как можно меньше влиять на пользовательскую программу.
6. Обработка выводимой информации должна быть расширяемой. Утилита должна поддерживать разные способы обработки информации, собранной в системе обнаружения блокировок, и давать возможность настройки и расширения вывода информации не только вам, но и другим программистам.
7. Средство должно обеспечивать легкую интеграцию с пользовательскими программами.

Работая с утилитами, подобными `DeadlockDetection`, следует помнить, что они неизбежно влияют на поведение исследуемого приложения. Можно рассматривать это как еще одно наглядное подтверждение принципа неопределенности Гейзенберга. `DeadlockDetection` сама может вызывать в ваших программах блокировки, которые вы иначе не обнаружили бы, потому что выполняемая ею работа

по сбору информации тормозит потоки. Я склонен считать это поведение одной из особенностей утилиты, потому что любая возможность блокировки в вашем коде указывает на ошибку, что является первым шагом к ее исправлению. Ошибки лучше всегда находить самому, чем оставлять такую радость своим клиентам.

Общие вопросы разработки DeadlockDetection

Чтобы DeadlockDetection удовлетворяла названным требованиям, я должен был ответить на ряд вопросов. Сначала я должен был определить, какие функции нужно отслеживать для воспроизведения полной истории блокировки (табл. 15-1).

Табл. 15-1. Функции, отслеживаемые утилитой DeadlockDetection

Тип	Функции
Функции работы с потоками	CreateThread, ExitThread, SuspendThread, ResumeThread, TerminateThread, _beginthreadex, _beginthread, _exitthreadex, _exitthread, FreeLibraryAndExitThread
Функции работы с критическими секциями	InitializeCriticalSection, InitializeCriticalSectionAndSpinCount, DeleteCriticalSection, EnterCriticalSection, LeaveCriticalSection, SetCriticalSectionSpinCount, TryEnterCriticalSection
Функции работы с мьютексами	CreateMutexA, CreateMutexW, OpenMutexA, OpenMutexW, ReleaseMutex
Функции работы с семафорами	CreateSemaphoreA, CreateSemaphoreW, OpenSemaphoreA, OpenSemaphoreW, ReleaseSemaphore
Функции работы с событиями	CreateEventA, CreateEventW, OpenEventA, OpenEventW, PulseEvent, ResetEvent, SetEvent
Функции блокировки	WaitForSingleObject, WaitForSingleObjectEx, WaitForMultipleObjects, WaitForMultipleObjectsEx, MsgWaitForMultipleObjects, MsgWaitForMultipleObjectsEx, SignalObjectAndWait
Специальные функции	CloseHandle, ExitProcess, GetProcAddress, LoadLibraryA, LoadLibraryW, LoadLibraryExA, LoadLibraryExW, FreeLibrary

Обдумав проблему сбора информации, необходимой для удовлетворения первых четырех требований, я понял, что мне нужно перехватывать функции из табл. 15-1 (устанавливать для них ловушки), регистрируя получение и освобождение объектов синхронизации. Перехват — нетривиальная задача; ее решение я рассмотрю в разделе «Перехват импортируемых функций». Для перехвата импортируемых функций код DeadlockDetection должен находиться в DLL, потому что ловушки работают только в том адресном пространстве, в котором создаются. Это значит, что пользователь должен загружать DLL утилиты DeadlockDetection в свое адресное пространство. Данное требование не такое уж и жесткое, если учесть все его достоинства. Реализованная в форме DLL, утилита допускала бы легкую интеграцию с пользовательской программой, что позволило бы удовлетворить требование 7.

Вы могли заметить, что я не включил в табл. 15-1 некоторые функции работы с сообщениями, способные вызывать блокировку, такие как SendMessage, PostMessage

и `WaitMessage`. Сначала я намеревался реализовать и их поддержку, однако когда я запустил под управлением `DeadlockDetection` классическую программу Чарльза Петцольда (Charles Petzold) «Hello World!» с графическим пользовательским интерфейсом, `DeadlockDetection` сообщила столько вызовов, что работа программы в конечном счете была нарушена. Чтобы сделать `DeadlockDetection` как можно компактнее и быстрее, мне пришлось отказаться от этих функций.

Решение проблемы сбора информации, удовлетворяющей требованиям 1–4, прямо вытекает из выбранного мной подхода внутрипроцессного перехвата функций. Это значит, что при любом вызове функций работы с потоками и функций синхронизации управление будет передаваться в код `DeadlockDetection` со всей нужной мне информацией.

Сделать `DeadlockDetection` максимально компактной и быстрой (требование 5) оказалось довольно трудно. Я старался, чтобы код был как можно более эффективным, однако в связи с заданными мной целями при этом возникли трудности. Так как вам лучше известно, какие типы объектов синхронизации вы используете в своей программе, я решил сгруппировать их, чтобы вы могли указать именно те функции, которые хотите перехватывать. Скажем, если вас интересует только блокировка на мьютексах, вы можете обрабатывать только функции работы с мьютексами.

Я позволяю во время выполнения указывать, какие наборы функций работы с объектами синхронизации вы хотите отслеживать. Кроме того, вы можете включать/отключать `DeadlockDetection` любое число раз. Вы даже можете назначить своей программе сочетание клавиш (accelerator) или специальный пункт меню, который включает/выключает всю систему `DeadlockDetection`. Такое ограничение области и времени действия необходимо для соответствия требованию 5 и помогает удовлетворить требованию 7.

После этого мне осталось разобраться только с требованием 6: обеспечить максимальную расширяемость обработки выводимой информации. Я хотел предоставить вам широкие возможности конфигурирования параметров вывода, а не навязывать какой-либо жестко закодированный формат. Отделив перехват функций и основную логику программы от кода вывода, я смог улучшить возможность повторного использования кода, потому что разработать только новый модуль вывода гораздо проще, чем переписывать ядро программы. Я назвал модули вывода расширениями `DeadlockDetection` или, сокращенно, `DeadDetExt`. `DeadDetExt` — это просто DLL, которые экспортируют несколько функций, вызываемых `DeadlockDetection` в случае необходимости.

Что ж, пришло время описать работу с `DeadlockDetection`.

Использование `DeadlockDetection`

Перед использованием `DeadlockDetection` нужно разместить в одном месте `DEADLOCKDETECTION.DLL`, ее файл инициализации и нужную библиотеку `DeadDetExt`. Файл инициализации — это простой файл `INI`, в котором должно быть указано хотя бы имя загружаемого файла `DeadDetExt`. Например, файл `DEADLOCKDETECTION.INI`, который загружает поставляемую вместе с утилитой библиотеку `TEXTFILEDDEXT.DLL`, содержит следующую информацию:


```
[Initialization]
; Единственное обязательное значение, имя файла DeadDetExt,
; который будет обрабатывать вывод.
ExtDll = "TextFileDDExt.dll"

; Если этот параметр равен 1, DeadlockDetection будет
; выполнять инициализацию в собственной функции DllMain,
; чтобы протоколирование могло быть начато как можно раньше.
StartInDllMain = 0

; Если StartInDllMain равняется 1, этот ключ задает первоначальные
; параметры DeadlockDetection. В нем указываются значения флагов DDOPT_*.
; InitialOpts = 0

; Список модулей, игнорируемых при перехвате функций
; синхронизации. IMM32.DLL – это DLL Input Method Editor (редактор
; методов ввода), которую Windows XP загружает во все процессы.
; Создавайте список в последовательном порядке, начиная с номера 1.
[IgnoreModules]
Ignore1=IMM32.DLL
```

Как вы можете увидеть по некоторым параметрам INI, DeadlockDetection может выполнять инициализацию при простом вызове LoadLibrary. Для проактивной отладки было бы неплохо, чтобы во время инициализации ваше приложение проверяло конкретный раздел реестра или переменную среды и вызывало при их наличии LoadLibrary с указанным именем DLL. Благодаря этому вам не нужно было бы использовать условную компиляцию, и у вас были бы средства чистой загрузки DLL в свое адресное пространство. Конечно, это подразумевает, что загружаемые вами таким образом DLL должны полностью инициализироваться в собственных функциях DllMain и не должны требовать вызова каких-нибудь других экспортируемых функций.

Чтобы вы могли указывать параметры инициализации DeadlockDetection в своем коде, а не при помощи файла INI, вам нужно включить в свою программу файл DEADLOCKDETECTION.H и скомпоновать ее с библиотекой DEADLOCKDETECTION.LIB. Если вы хотите инициализировать DeadlockDetection сами, вызовите в нужном месте функцию OpenDeadlockDetection, которая принимает единственный параметр — первоначальные флаги протоколирования. Все флаги DDOPT_* указаны в табл. 15-2. Вызывать OpenDeadlockDetection следует до того, как ваша программа начнет создавать потоки, чтобы вы могли записать всю важную информацию об объектах синхронизации.

Изменять параметры протоколирования можно в любой момент при помощи функции SetDeadlockDetectionOptions. Она принимает тот же набор объединенных при помощи операции ИЛИ флагов, что и OpenDeadlockDetection. Чтобы увидеть текущие параметры, вызовите GetDeadlockDetectionOptions. Во время выполнения программы можете изменять параметры протоколирования сколько вашей душе угодно. Для приостановления и возобновления протоколирования служат функции SuspendDeadlockDetection и ResumeDeadlockDetection соответственно.

Табл. 15-2. Параметры протоколирования DeadlockDetection

Флаг	Ограничивает протоколирование
DDOPT_WAIT	Функциями ожидания
DDOPT_THREADS	Функциями работы с потоками
DDOPT_CRITSEC	Функциями работы с критическими секциями
DDOPT_MUTEX	Функциями работы с мьютексами
DDOPT_SEMAPHORE	Функциями работы с семафорами
DDOPT_EVENT	Функциями работы с событиями
DDOPT_ALL	Регистрирует все перехваченные функции

Вместе с исходным кодом DeadlockDetection вы можете найти на диске мою библиотеку DeadDetExt под названием TEXTFILEDDEXT.DLL. Это относительно простое расширение записывает всю информацию в текстовый файл. При запуске DeadlockDetection вместе с TEXTFILEDDEXT.DLL расширение создает текстовый файл в том же каталоге, в котором находится выполняемая программа. Текстовый файл будет иметь имя выполняемой программы с расширением .DD. Например, при запуске программы DDSIMPTEST.EXE итоговый файл будет назван DDSIMPTEST.DD. Вот пример вывода, сгенерированного TEXTFILEDDEXT.DLL (листинг 15-1).

Листинг 15-1. Данные, выводимые утилитой DeadlockDetection при помощи TEXTFILEDDEXT.DLL

TID	Ret Addr	C/R	Ret Value	Function & Params
0x00000DF8	[0x004011B2]	(R)	0x00000000	InitializeCriticalSection 0x00404150
0x00000DF8	[0x004011CC]	(R)	0x000007C0	CreateEventA 0x00000000, 1, 0, 0x004040F0 [The Event Name]
0x00000DF8	[0x004011EF]	(R)	0x000007BC	CreateThread 0x00000000, 0x00000000, 0x00401000, 0x00000000, 0x00000000, 0x0012FF5C
0x00000DF8	[0x00401212]	(R)	0x000007B8	CreateThread 0x00000000, 0x00000000, 0x004010BC, 0x00000000, 0x00000000, 0x0012FF5C
0x00000DF8	[0x00401229]	(C)		EnterCriticalSection 0x00404150
0x000000A8	[0x00401030]	(C)		EnterCriticalSection 0x00404150
0x00000F04	[0x004010F3]	(R)	0x000007B0	OpenEventA 0x001F0003, 0, 0x004040BC [The Event Name]
0x00000DF8	[0x00401229]	(R)	0x00000000	EnterCriticalSection 0x00404150
0x00000DF8	[0x0040123E]	(C)		WaitForSingleObject 0x000007C0, INFINITE
0x00000F04	[0x00401121]	(C)		EnterCriticalSection 0x00404150

Заметьте: сведения об именах функций и их параметрах представлены в листинге 15-1 на нескольких строках, чтобы они помещались на странице. Информация выводится в таком порядке.

1. Идентификатор выполняемого потока.
2. Адрес возврата, показывающий, какая из ваших функций вызвала функцию синхронизации. При помощи утилиты CrashFinder из главы 12 можно просмотреть адреса возврата и узнать, как вы оказались в ситуации блокировки.

3. Индикатор вызова/возврата, который помогает определить действия, происшедшие до или после конкретных функций.
4. Возвращаемое функцией значение, если ваша программа его сообщает.
5. Имя функции синхронизации.
6. Список параметров функции синхронизации. Значения в квадратных скобках описывают данные в понятной людям форме. Особое внимание я уделил выводу строковых значений, но вы легко реализуете вывод более подробной информации, скажем, отдельных флагов.

Если при запуске вашей программы она заблокируется, завершите процесс и изучите файл вывода, чтобы узнать, какая функция синхронизации была вызвана последней. Для обновления информации TEXTFILEDDEXT.DLL сбрасывает файловые буферы в файл при каждом вызове функций `WaitFor*`, `EnterCriticalSection` и `TryEnterCriticalSection`.

Предупреждение: если вы включите полное протоколирование всех функций, почти мгновенно будут созданы очень большие файлы. Так, создав пару потоков при помощи приложения MTGDI из числа примеров к Visual C++, я за минуту или две сгенерировал 11-Мбайтный текстовый файл.

Реализация DeadlockDetection

Как видите, работать с DeadlockDetection довольно просто. Однако под простотой ее использования скрывается весьма сложная реализация. В первую очередь я хочу рассказать про перехват функций.

Перехват импортируемых функций

Способов перехвата вызываемых программой функций много. Можно выполнять поиск всех команд `CALL` и заменять их операнды собственным адресом, но этот подход сложен и подвержен ошибкам. К счастью, в случае DeadlockDetection мне нужно перехватывать импортируемые функции, поэтому их гораздо легче обрабатывать, чем команды `CALL`.

Импортируемая функция — это функция, которая располагается в DLL. Например, вызывая `OutputDebugString`, ваша программа вызывает функцию, находящуюся в `KERNEL32.DLL`. Когда я только начал писать программы для Win32, я думал, что вызов импортируемых функций аналогичен вызовам любых других функций: команда `CALL` или команда перехода передает управление по нужному адресу и начинает выполнение импортируемой функции. Единственное различие могло бы состоять в том, что в случае импортируемой функции загрузчик программ ОС должен был бы просмотреть исполняемый файл и исправить адреса, чтобы они соответствовали той области памяти, в которую будет загружена вызываемая DLL. Однако, взглянув на действительную реализацию вызовов импортируемых функций, я был поражен ее простотой и элегантностью.

Недостаток только что описанного мной подхода станет очевидным, если учесть наличие огромного числа API-функций и возможность вызова одной и той же функции во многих местах. Если бы загрузчик должен был найти и исправить каждый вызов, скажем, функции `OutputDebugString`, загрузка программы могла бы продолжаться вечно. Даже если б компоновщик создавал таблицу, где указы-

вал бы место каждого вызова `OutputDebugString`, загрузка программы была бы мучительно медленной из-за огромного объема работы, связанной с циклами и записью в память.

Так как же загрузчик сообщает программе о том, где находится импортируемая функция? Решение чертовски умно. Представив, куда направляются вызовы `OutputDebugString`, вы вскоре поймете, что каждый вызов должен обращаться к одному и тому же адресу памяти, по которому `OutputDebugString` была загружена. Конечно, ваша программа не может знать этот адрес заранее, поэтому все вызовы `OutputDebugString` выполняются посредством единственного косвенного адреса. При загрузке вашего исполняемого файла и нужных ему DLL загрузчик корректирует этот единственный косвенный адрес, чтобы он соответствовал итоговому адресу загрузки `OutputDebugString`. Чтобы косвенная адресация работала, компилятор генерирует при каждом вызове импортируемой функции переход к этому косвенному адресу. Косвенный адрес хранится в исполняемом файле в разделе `.idata` (или `import`). Если вы импортируете функцию, объявляя ее как `__declspec(dllimport)`, то вместо косвенного перехода будет косвенный вызов, что экономит несколько команд на каждом вызове функции.

Чтобы установить ловушку для импортируемой функции, нужно отыскать в исполняемом файле раздел импорта, найти адрес нужной функции и заменить его адресом функции-ловушки. Вам может показаться, что это потребует большого объема работы, однако все не так уж плохо, так как формат файлов Win32 Portable Executable (PE) организован очень разумно.

Метод установки ловушки для импортируемых функций описан в главе 10 великолепной книги Мэтта Питрека (Matt Pietrek) «Windows 95 System Programming Secrets» (IDG Books, 1995). Мэтт просто ищет для модуля раздел импорта и просматривает в цикле импортируемые функции, используя значение, возвращаемое функцией `GetProcAddress`. Обнаружив нужную функцию, он перезаписывает ее первоначальный адрес адресом функции-ловушки.

С момента издания книги Мэтта в 1995 г. в мире программирования произошли два небольших изменения. Во-первых, когда Мэтт писал свою книгу, большинство программистов не объединяло раздел импорта с другими разделами PE-файла. Поэтому, если раздел импорта располагается в памяти, доступной только для чтения, попытка перезаписи адреса функции приведет к нарушению доступа. Чтобы избежать этой ошибки, я перед записью адреса функции-ловушки устанавливаю защиту виртуальной памяти в состояние разрешения чтения и записи. Вторая, чуть более сложная проблема связана с невозможностью перехвата в некоторых случаях импортируемых функций в Microsoft Windows Me. Очень многие спрашивают меня о перехвате функций, поэтому я решил реализовать его и для Windows Me и рассказать, что происходит в этой ОС.

Работая с `DeadlockDetection`, вам хотелось бы иметь возможность перенаправления функций работы с потоками при любом запуске своей программы, даже когда она выполняется под управлением отладчика. Однако установка ловушек под управлением отладчика может представлять проблему, хотя на первый взгляд так не кажется. Получив адрес функции при помощи `GetProcAddress` в Windows XP или при выполнении программы в Windows Me вне отладчика, вы всегда сможете найти этот адрес в разделе импорта. Но в Windows Me адрес, возвращаемый функцией

`GetProcAddress` в программе, выполняемой под управлением отладчика, отличается от адреса, получаемого при выполнении вне отладчика. В первом случае `GetProcAddress` на самом деле возвращает отладочный шлюз (debug thunk) — специальную оболочку для действительного вызова.

Отладочный шлюз нужен потому, что Windows Me не выполняет копирование при записи (copy-on-write) для адресов, расположенных выше 2 Гб. Копирование при записи предполагает, что при записи в страницу разделяемой памяти ОС делает копию страницы и предоставляет ее процессу, выполняющему запись. Обычно Windows Me и Windows XP следуют одинаковым правилам, и все работает отлично. Однако для разделяемой памяти, находящейся выше 2 Гб, где в Windows Me загружаются все DLL системы, Windows Me не выполняет копирования при записи. Это значит, что при изменении памяти в DLL системы в результате установки точки прерывания или исправления функции изменение произойдет для всех процессов ОС, что вызовет ее крах, если измененная область будет использоваться другим процессом. Поэтому Windows Me прилагает серьезные усилия, чтобы помешать вам исказить эту память.

Отладочный шлюз, возвращаемый `GetProcAddress` при выполнении под отладчиком, — это средство, при помощи которого Windows Me предотвращает попытки отладки системных функций, расположенных выше 2 Гб. В целом отсутствие копирования при записи большинство программистов волновать не должно; оно представляет проблему только для тех, кто разрабатывает отладчики или желает корректно перехватывать функции независимо от того, выполняется программа под отладчиком или нет.

К счастью, получить действительный адрес импортируемой функции не так сложно — просто для этого требуется чуть поработать, избегая при этом `GetProcAddress`. Структура `IMAGE_IMPORT_DESCRIPTOR` в PE-файле, которая содержит всю информацию о функциях, импортируемых из конкретной DLL, имеет указатели на два массива в исполняемом файле — таблицы адресов импортируемых функций (import address table, IAT) (иногда их называют массивами данных шлюзов — thunk data array). Первый указатель указывает на действительную IAT, который загрузчик программ корректирует при загрузке исполняемого файла, второй — на исходную IAT, содержащую адреса импортируемых функций и не изменяемую загрузчиком. Итак, для обнаружения действительного адреса импортируемой функции нужно просто найти ее в исходной IAT; после этого надо записать адрес ловушки в соответствующий элемент действительной IAT, используемой программой. Благодаря этому ловушка будет работать всегда независимо от того, где она вызывается.

Всю работу, связанную с установкой ловушек, выполняет моя функция `HookImportedFunctionsByName` (табл. 15-3). Так как я хотел сделать перехват функций как можно более общим, я реализовал возможность одновременного перехвата нескольких функций, импортируемых из одной DLL. Как можно догадаться по имени, `HookImportedFunctionsByName` перехватывает только те функции, которые импортируются по имени. Для перехвата функций, экспортируемых по ординалу, я написал функцию `HookOrdinalExport`, но я не буду рассматривать ее в этой книге.


```

    // В отладочных компоновках выполняется глубокая проверка paHookArray.
#ifdef _DEBUG
    if ( NULL != paOrigFuncs )
    {
        ASSERT ( FALSE == IsBadWritePtr ( paOrigFuncs ,
                                           sizeof ( PROC ) * uiCount ) );

    }
    if ( NULL != pdwHooked )
    {
        ASSERT ( FALSE == IsBadWritePtr ( pdwHooked , sizeof ( UINT ) ));
    }

    // Проверка всех элементов массива перехватываемых функций.
    {
        for ( UINT i = 0 ; i < uiCount ; i++ )
        {
            ASSERT ( NULL != paHookArray[ i ].szFunc ) ;
            ASSERT ( '\\0' != *paHookArray[ i ].szFunc ) ;
            // Если адрес функции не равен NULL, выполняется его проверка.
            if ( NULL != paHookArray[ i ].pProc )
            {
                ASSERT ( FALSE == IsBadCodePtr ( paHookArray[i].pProc));
            }
        }
    }
}
#endif

// Дополнительная проверка параметров и установка кода ошибки.
if ( ( 0 == uiCount ) ||
      ( NULL == szImportMod ) ||
      ( TRUE == IsBadReadPtr ( paHookArray ,
                              sizeof ( HOOKFUNCDESC ) * uiCount ) ) )
{
    SetLastErrorEx ( ERROR_INVALID_PARAMETER , SLE_ERROR ) ;
    return ( FALSE ) ;
}
if ( ( NULL != paOrigFuncs ) &&
      ( TRUE == IsBadWritePtr ( paOrigFuncs ,
                              sizeof ( PROC ) * uiCount ) ) )
{
    SetLastErrorEx ( ERROR_INVALID_PARAMETER , SLE_ERROR ) ;
    return ( FALSE ) ;
}
if ( ( NULL != pdwHooked ) &&
      ( TRUE == IsBadWritePtr ( pdwHooked , sizeof ( UINT ) ) ) )
{
    SetLastErrorEx ( ERROR_INVALID_PARAMETER , SLE_ERROR ) ;
    return ( FALSE ) ;
}

```

```

// Здесь я проверяю, расположена ли данная системная DLL выше 2 Гб,
// в случае чего Windows 98 не позволит ее скорректировать.
if ( ( FALSE == IsNT ( ) ) && ( (DWORD_PTR)hModule >= 0x80000000 ) )
{
    SetLastErrorEx ( ERROR_INVALID_HANDLE , SLE_ERROR );
    return ( FALSE );
}

// СООБРАЖЕНИЯ ПО ПОВОДУ УЛУЧШЕНИЯ ПРОГРАММЫ
// Следует ли проверять каждый элемент массива
// перехватываемых функций в заключительных компоновках?

if ( NULL != paOrigFuncs )
{
    // Присвоение всем элементам массива paOrigFuncs значения NULL.
    memset ( paOrigFuncs , NULL , sizeof ( PROC ) * uiCount );
}
if ( NULL != pdwHooked )
{
    // Присвоение числу перехваченных функций значения 0.
    *pdwHooked = 0 ;
}

// Получение специфического дескриптора импорта.
PIMAGE_IMPORT_DESCRIPTOR pImportDesc =
    GetNamedImportDescriptor ( hModule , szImportMod );
if ( NULL == pImportDesc )
{
    // Запрошенный модуль не был импортирован. Не возвращать ошибку.
    return ( TRUE );
}

// ИСПРАВЛЕННАЯ ОШИБКА. Спасибо Аттиле Шепезвари (Attila Szepesváry)!
// Проверка того, что первый шлюз и исходный первый шлюз
// не равны NULL. Исходный первый шлюз может быть нулевым
// дескриптором импорта, что вызвало бы крах этой функции.
if ( ( NULL == pImportDesc->OriginalFirstThunk ) ||
      ( NULL == pImportDesc->FirstThunk ) )
{
    // Я возвращаю TRUE, потому что это аналогично случаю,
    // в котором запрошенный модуль не был импортирован.
    // Все в порядке!
    SetLastError ( ERROR_SUCCESS );
    return ( TRUE );
}

// Получение информации об исходном шлюзе для этой DLL.
// Я не могу использовать информацию, хранимую
// в pImportDesc->FirstThunk, так как загрузчик уже изменил
// этот массив во время коррекции импортируемых функций.

```



```

// Исходный шлюз предоставляет мне доступ к именам функций.
PIMAGE_THUNK_DATA pOrigThunk =
    MakePtr ( PIMAGE_THUNK_DATA      ,
              hModule                  ,
              pImportDesc->OriginalFirstThunk );
// Получение указателя на массив pImportDesc->FirstThunk, при
// помощи которого выполняется действительный перехват функций.
PIMAGE_THUNK_DATA pRealThunk = MakePtr ( PIMAGE_THUNK_DATA      ,
                                          hModule                  ,
                                          pImportDesc->FirstThunk );

// Поиск перехватываемых функций.
while ( NULL != pOrigThunk->u1.Function )
{
    // Выполняется поиск только тех функций, которые
    // импортируются по имени, но не по значению ординала.
    if ( IMAGE_ORDINAL_FLAG !=
          ( pOrigThunk->u1.Ordinal & IMAGE_ORDINAL_FLAG ))
    {
        // Изучение имени этой импортируемой функции.
        PIMAGE_IMPORT_BY_NAME pByName ;

        pByName = MakePtr ( PIMAGE_IMPORT_BY_NAME      ,
                            hModule                      ,
                            pOrigThunk->u1.AddressOfData );

        // Если имя начинается с NULL, оно пропускается.
        if ( '\\0' == pByName->Name[ 0 ] )
        {
            // ИСПРАВЛЕННАЯ ОШИБКА (спасибо Аттиле Шепезвари!)
            // Я забыл про увеличение указателей на шлюз!
            pOrigThunk++;
            pRealThunk++;
            continue ;
        }

        // Этот флаг показывает, перехватываю ли я функцию.
        BOOL bDoHook = FALSE ;

        // СООБРАЖЕНИЯ ПО ПОВОДУ УЛУЧШЕНИЯ ПРОГРАММЫ
        // Возможно, здесь следует реализовать двоичный поиск.
        // Я проверяю, есть ли имя этой импортируемой функции
        // в массиве перехватываемых функций. Возможно, paHookArray
        // следует держать отсортированным по именам функций, чтобы
        // можно было ускорить его просмотр при помощи двоичного
        // поиска. Однако передаваемый в эту функцию параметр
        // uiCount будет довольно небольшим, поэтому при поиске
        // каждой функции, импортируемой по szImportMod, вполне
        // допустимо просматривать весь массив paHookArray.
    }
}

```

см. след. стр.

```

for ( UINT i = 0 ; i < uiCount ; i++ )
{
    if ( ( paHookArray[i].szFunc[0] ==
          pByName->Name[0] ) &&
        ( 0 == strcmpi ( paHookArray[i].szFunc ,
                        (char*)pByName->Name ) ) )
    {
        // Если адрес функции равен NULL, выполняется выход;
        // в противном случае функция перехватывается.
        if ( NULL != paHookArray[ i ].pProc )
        {
            bDoHook = TRUE ;
        }
        break ;
    }
}

if ( TRUE == bDoHook )
{
    // Я обнаружил функцию, которую нужно перехватить. Теперь,
    // прежде чем перезаписать указатель на функцию, я должен
    // изменить защиту памяти, разрешив запись в нее. Заметьте,
    // что я выполняю запись в область действительного шлюза!

    MEMORY_BASIC_INFORMATION mbi_thunk ;

    VirtualQuery ( pRealThunk
                  ,
                  &mbi_thunk
                  ,
                  sizeof ( MEMORY_BASIC_INFORMATION ) ) ;

    if ( FALSE == VirtualProtect ( mbi_thunk.BaseAddress ,
                                   mbi_thunk.RegionSize ,
                                   PAGE_READWRITE
                                   ,
                                   &mbi_thunk.Protect ))
    {
        ASSERT ( !"VirtualProtect failed!" ) ;
        SetLastErrorEx ( ERROR_INVALID_HANDLE , SLE_ERROR ) ;
        return ( FALSE ) ;
    }

    // Сохранение исходного адреса в случае надобности.
    if ( NULL != paOrigFuncs )
    {
        paOrigFuncs[i] =
            (PROC)((INT_PTR)pRealThunk->u1.Function) ;
    }

    // Перехват функции.
    DWORD_PTR * pTemp = (DWORD_PTR*)&pRealThunk->u1.Function ;
    *pTemp = (DWORD_PTR)(paHookArray[i].pProc);
}

```

```

        DWORD dwOldProtect ;

        // Возвращение параметра защиты в состояние,
        // бывшее до перезаписи указателя на функцию.
        VERIFY ( VirtualProtect ( mbi_thunk.BaseAddress ,
                                   mbi_thunk.RegionSize ,
                                   mbi_thunk.Protect      ,
                                   &dwOldProtect          ) ) ;

        if ( NULL != pdwHooked )
        {
            // Увеличение общего числа перехваченных функций.
            *pdwHooked += 1 ;
        }
    }
}

// Увеличение указателей на обе таблицы.
pOrigThunk++ ;
pRealThunk++ ;
}

// Все OK!
SetLastError ( ERROR_SUCCESS ) ;
return ( TRUE ) ;
}

```

`HookImportedFunctionsByName` не должна быть слишком сложной для понимания. После тщательной профилактической проверки каждого параметра я вызываю вспомогательную функцию `GetNamedImportDescriptor`, выполняющую поиск `IMAGE_IMPORT_DESCRIPTOR` для запрошенного модуля. Получив указатели на исходную и действительную IAT, я просматриваю исходную IAT и изучаю каждую функцию, импортируемую по имени, чтобы узнать, есть ли она в списке `paHookArray`. Если функция имеется в списке перехватываемых функций, я просто разрешаю запись в область памяти действительной IAT, записываю вместо адреса действительной функции адрес ловушки и возвращаю защиту памяти в исходное состояние. В исходный код `BUGSLAYERUTIL.DLL` я включил функцию блочного теста для `HookImportedFunctionsByName`, которая поможет вам со всем разобраться, если вы не очень внимательно следили за происходящим.

Теперь, когда вы представляете механизм перехвата импортируемых функций, займемся реализацией остальной части `DeadlockDetection`.

Детали реализации

Одна из моих основных целей при реализации `DeadlockDetection` состояла в том, чтобы сделать утилиту максимально ориентированной на использование данных и таблиц. Поразмыслив о том, как выполняется перехват функций DLL, вы поймете, что его механизм почти идентичен для всех функций, указанных в табл. 15-1. Функция-ловушка вызывается, определяет, отслеживается ли ее класс функций,

вызывает действительную функцию и (если для этого класса включено протоколирование) записывает информацию и выполняет возврат. Я должен был написать ряд похожих функций-ловушек и хотел сделать их как можно проще. Сложные функции-ловушки — плодородная почва для ошибок, которые могут прокрасться в ваш код, даже когда вы пытаетесь все упростить. Чуть ниже я расскажу про одну неприятную ошибку в коде DeadlockDetection в первом издании этой книги.

Лучше всего показать эту простоту, обсудив написание DLL DeadDetExt. Библиотека DeadDetExt должна иметь три экспортируемых функции. Роль первых двух, DeadDetExtOpen и DeadDetExtClose, очевидна. Интерес представляет DeadDetProcessEvent, вызываемая каждой функцией-ловушкой при наличии информации для записи. DeadDetProcessEvent принимает единственный параметр — указатель на структуру DDEVENTINFO:

```
typedef struct tagDDEVENTINFO
{
    // Идентификатор, определяющий содержание оставшейся части структуры.
    eFuncEnum    eFunc            ;
    // Индикатор предварительного или заключительного вызова.
    ePrePostEnum ePrePost        ;
    // Адрес возврата. Он нужен для нахождения вызвавшей функции.
    DWORD        dwAddr          ;
    // Идентификатор вызвавшего потока.
    DWORD        dwThreadId      ;
    // Значение, возвращаемое при заключительных вызовах.
    DWORD        dwRetValue      ;
    // Информация о параметрах. Приводите этот элемент к указателю
    // на структуру, соответствующую функции, как описано ниже. При доступе
    // к параметрам обращайтесь с ними, как со значениями только для чтения.
    DWORD        dwParams        ;
} DDEVENTINFO , * LPDDEVENTINFO ;
```

Весь вывод для какой-либо функции из листинга 15-1 основан на информации, содержащейся в структуре DDEVENTINFO. Большинство полей DDEVENTINFO говорит само за себя, а вот dwParams требует пояснения. Это поле является на самом деле указателем на параметры в том порядке, в котором они расположены в памяти.

В главе 7 я рассказал о том, как параметры передаются в стек. Напомню, что параметры функций с соглашениями вызова __stdcall и __cdecl передаются справа налево, а стек растет по направлению от старших адресов памяти к младшим. Поле dwParams структуры DDEVENTINFO указывает на последний параметр в стеке, т. е. слева направо. Чтобы обеспечить легкое преобразование dwParams, я прибегнул к приведению типов.

В файле DEADLOCKDETECTION.H содержатся объявления typedef, описывающие списки параметров каждой перехватываемой функции. Например, если бы поле eFunc соответствовало значению eWaitForSingleObjectEx, то для получения параметров нужно было бы привести тип dwParams к LPWAITFORSINGLEOBJECTEX_PARAMS. Чтобы увидеть все это творческое приведение типов в действии, изучите код библиотеки TEXTFILEDDEXT.DLL (см. CD, прилагаемый к книге).

Хотя обработка вывода относительно проста, сбор информации может оказаться сложным. Мне требовалось, чтобы `DeadlockDetection` перехватывала функции синхронизации из табл. 15-1, но я не хотел, чтобы функции-ловушки изменяли поведение действительных функций. Я также хотел получать параметры и возвращаемые значения и с легкостью писать функции-ловушки на C/C++. Я провел за отладчиком и дизассемблером немало времени, пока мне удалось сделать это правильно.

Первоначально я сделал все функции-ловушки сквозными (pass-through function), чтобы они вызывали действительные функции непосредственно. Этот подход работал отлично. Затем я поместил параметры функций и возвращаемые ими значения в локальные переменные. Получение возвращаемого значения из действительной функции оказалось простым, но из-за того, что я начал реализацию `DeadlockDetection` на Visual C++ 6, у меня не было чистого способа получения адресов возврата в моих функциях-ловушках C/C++. Visual C++ .NET поддерживает внутреннюю (intrinsic) функцию `_ReturnAddress`, но в Visual C++ 6 такой возможности не было. Мне нужно было значение `DWORD` прямо перед текущим указателем стека. Увы, в обычном C/C++ пролог функции уже выполнил бы все свои действия к тому времени, когда я смог бы получить управление, и указатель стека имел бы не то значение, которое мне было нужно.

Вы можете подумать, что указатель стека — это просто смещение, определяемое числом локальных переменных, но это не всегда так. Компилятор Visual C++ выполняет великолепную оптимизацию, так что при различных конфигурациях флагов оптимизации указатель стека может иметь разные значения. Так, когда вы объявляете переменную как локальную, компилятор может оптимизировать работу с ней, сохранив в регистре, из-за чего она даже не появится в стеке.

Мне нужен был гарантированный способ получения указателя стека независимо от параметров оптимизации. В этот момент я начал думать, почему бы не объявить функции-ловушки как `__declspec(naked)` и не создать собственные пролог и эпилог? Это дало бы мне полный контроль над регистром ESP независимо от параметров оптимизации. Кроме того, это облегчило бы и получение адреса возврата и параметров, так как они находятся по смещениям `ESP+04h` и `ESP+08h` соответственно. Помните, что мои пролог и эпилог не представляют собой ничего сверхъестественного, поэтому я все же выполняю обычные команды `PUSH EBP` и `MOV EBP, ESP` в прологе и `MOV ESP, EBP` и `POP EBP` в эпилоге.

Решив объявлять все функции-ловушки как `__declspec(naked)`, я написал для обработки пролога и эпилога два макроса: `H00KFN_PROLOG` и `H00KFN_EPILOG`. Кроме того, я заблаговременно объявил в `H00KFN_PROLOG` некоторые общие локальные переменные, нужные всем функциям-ловушкам. В число этих переменных вошли значение последней ошибки, `dwLastError`, и структура информации о событии, `stEvtInfo`, передаваемая в DLL `DeadDetExt`. Переменная `dwLastError` — просто еще один признак состояния, который мне нужно сохранять при перехвате функций.

При помощи функции `SetLastError` Windows API возвращает специальный код ошибки, предоставляя более подробную информацию в случае неудачи функции. Этот код ошибки может быть настоящим благословением, потому что он сообщает о причине неудачи API-функции. Так, если `GetLastError` возвратит 122, вы будете знать, что причиной ошибки стал недостаточный размер переданного в функцию

буфера. Все возвращаемые ОС коды ошибок указаны в файле WINERROR.H. Проблема с функциями-ловушками в том, что во время своего выполнения они могут перезаписать значение последней ошибки. Это может привести к катастрофе, если значение последней ошибки используется вашей программой.

Если при вызове `CreateEvent` вы хотите узнать, был ли возвращаемый описатель создан или просто открыт, вы также можете использовать код последней ошибки: если `CreateEvent` просто открыла описатель, код будет иметь значение `ERROR_ALREADY_EXISTS`. Одно из важнейших правил перехвата функций гласит, что вы не можете изменять ожидаемое поведение функции, поэтому сразу же после вызова действительной функции я должен был вызвать `GetLastError`, чтобы моя функция-ловушка могла правильно установить код последней ошибки, возвращаемый действительной функцией. Общее правило написания функций-ловушек таково: сразу после вызова действительной функции нужно вызывать `GetLastError`, а непосредственно перед выходом из ловушки устанавливать код ошибки при помощи `SetLastError`.

Стандартный вопрос отладки

Если теперь доступна `ReturnAddress`, почему вы не использовали ее, а пошли на все эти проблемы?

Когда пришло время обновить `DeadlockDetection` для второго издания этой книги, я думал немного упростить свою жизнь и изменить макрос `HOOK_FN_PROLOG`, чтобы он использовал новую внутреннюю функцию `_ReturnAddress`. Это значило бы, что я могу избавиться от объявлений `naked`, привести функции к более нормальному виду и не создавать собственные пролог и эпилог. Однако существующие макросы дают мне одно большое преимущество: я могу обращаться с параметрами, как с блоками памяти, и передавать их прямо в функцию вывода. Если бы я применял стандартные функции, мне нужно было бы выполнять странное приведение типов для достижения того же результата. Кроме того, у меня был самый весомый аргумент: имевшийся код работал очень хорошо, и мне не хотелось его переписывать. Поэтому я оставил функции с соглашением `naked` прежними.

В этот момент я подумал, что все, кроме тестирования, сделано. Увы, во время первого теста я нашел ошибку: между вызовами ловушек я не сохранял регистры `ESI` и `EDI`, потому что в документации к встроенному ассемблеру сказано, что их сохранять не требуется. После решения этой проблемы казалось, что `DeadlockDetection` работает прекрасно. Однако когда я начал сравнивать регистры до, во время и после вызовов функций, я заметил, что я не возвращаю значения, сохраняемые действительными функциями в `EBX`, `ECX` и `EDX` и, что еще хуже, в регистре флагов. Хотя я не видел в этом никаких проблем и в документации говорилось, что эти регистры сохранять не требуется, я все же был озабочен тем, что мои функции-ловушки изменяли состояние приложения. Для сохранения значений регистров после вызовов действительных функций я объявил структуру `REGSTATE`, чтобы можно было восстанавливать регистры по возвращении из функции-ловушки. Для сохранения и восстановления регистров я создал два дополнительных

макроса, `REAL_FUNC_PRE_CALL` и `REAL_FUNC_POST_CALL`, которые размещаю до и после вызова действительной функции, выполняемого функцией-ловушкой.

После дополнительного тестирования я обнаружил еще одну проблему: в заключительных компоновках с полной оптимизацией программа по необъяснимой причине часто терпела крах. В конце концов я выяснил, что ошибки вызваны влиянием оптимизации на некоторые из моих функций-ловушек. Конечно, оптимизатор пытался помочь, но в итоге приносил больше вреда, чем пользы. Я очень внимательно подошел к работе с регистрами в своих ловушках и использовал только `EAX` или непосредственно память стека. Однако, несмотря на все меры предосторожности по сохранению регистров, я обнаружил, что в отладочных компоновках команды:

```
MOV DWORD PTR [EBP-018h] , 00000002h
MOV DWORD PTR [EBP-014h] , 00000002h
```

преобразовывались оптимизатором в:

```
PUSH 002h
POP EBX
MOV DWORD PTR [EBP-01Ch] , EBX
MOV DWORD PTR [EBP-018h] , EBX
```

Легко увидеть, что во втором фрагменте команда `POP EBX` искажает значение регистра. Чтобы помешать оптимизатору искажать регистры без моего ведома, я отключил оптимизацию для всех функций-ловушек, поместив директиву:

```
#pragma optimize("", off )
```

в начало каждого файла. Кроме того, отключение оптимизации упростило отладку, потому что генерируемый компилятором неоптимизированный код очень похож и в заключительных, и в отладочных компоновках.

В листинге 15-3 приведена заключительная версия внутреннего заголовочного файла `DD_FUNC.H`, в котором объявлены все специальные макросы для функций-ловушек. В комментарии в начале файла вы можете найти два примера функций-ловушек, поясняющих применение каждого специального макроса. Внимательно изучите пример `DDSimpleTest`, который можно найти на CD. Исследуйте вызовы функций на языке ассемблера полностью, потому что это единственный способ увидеть все выполняемые действия.

Листинг 15-3. Файл `DD_FUNC.H`

```
/*-----
Отладка приложений для Microsoft .NET и Microsoft Windows
Copyright © 1997-2003 John Robbins - All rights reserved.
-----
Прототипы для всех функций-ловушек и кода пролога/эпилога
-----*/

#ifndef _DD_FUNC_H
#define _DD_FUNC_H
/*////////////////////////////////////
```

см. след. стр.

Все функции-ловушки объявляются с соглашением `__declspec(naked)`, поэтому я должен сам написать пролог и эпилог. Я должен предоставить собственные пролог и эпилог по нескольким причинам.

1. Функции, написанные на С, не позволяют контролировать использование регистров и сохранение исходных регистров компилятором. Отсутствие контроля над регистрами означает, что получить адрес возврата почти невозможно. Для проекта `DeadlockDetection` адрес возврата очень важен.
2. Я хотел передавать параметры в функцию обработки из DLL расширения, не копируя при каждом вызове функции большие объемы данных.
3. Так как почти все функции-ловушки ведут себя похожим образом, я могу присвоить значения общим переменным, нужным во всех функциях.
4. Функции-ловушки не должны изменять возвращаемые значения, в том числе значение, возвращаемое `GetLastError`. Собственные пролог и эпилог позволяют мне значительно упростить возвращение правильного значения. Кроме того, я должен восстанавливать значения регистров в то состояние, в каком они были после вызова действительной функции.

Базовая функция-ловушка требует только двух макросов: `HOOKFN_STARTUP` и `HOOKFN_SHUTDOWN`.

Как вы можете видеть, это здорово облегчает работу!

```

BOOL NAKEDDEF DD_InitializeCriticalSectionAndSpinCount (
                                LPCRITICAL_SECTION lpCriticalSection,
                                DWORD dwSpinCount )
{
    HOOKFN_STARTUP ( eInitializeCriticalSectionAndSpinCount ,
                    DDOPT_CRITSEC ,
                    0 ) ;

    InitializeCriticalSectionAndSpinCount ( lpCriticalSection ,
                                           dwSpinCount ) ;

    HOOKFN_SHUTDOWN ( 2 , DDOPT_CRITSEC ) ;
}

```

Если надо выполнить специальную обработку и вы не хотите делать чего-то, что нельзя выполнить, используя обычные макросы, вам помогут макросы:

```

HOOKFN_PROLOG
REAL_FUNC_PRE_CALL
REAL_FUNC_POST_CALL
HOOKFN_EPILOG

```

Пример функции, использующей указанные макросы:

```

HMODULE NAKEDDEF DD_LoadLibraryA ( LPCSTR lpLibFileName )
{
    // Все локальные переменные должны быть объявлены

```



```

// до макроса HOOKFN_PROLOG. Он создает фактический
// пролог функции и автоматически определяет некоторые
// важные переменные, такие как stEvtInfo (DDEVENTINFO).
HOOKFN_PROLOG ( ) ;

// Перед вызовом действительной функции нужно указать
// макрос REAL_FUNC_PRE_CALL, чтобы регистры имели те
// же значения, что и при вызове функции-ловушки.
REAL_FUNC_PRE_CALL ( ) ;
// Вызов действительной функции.
LoadLibraryA ( lpLibFileName ) ;
// Макрос для сохранения регистров после вызова действительной
// функции. Благодаря этому я могу присвоить регистрам нужные
// значения перед выходом из ловушки, сделав ее "невидимой".
REAL_FUNC_POST_CALL ( ) ;

// Операции, специфичные для ловушки LoadLibraryA.
if ( NULL != stEvtInfo.dwRetValue )
{
    HookAllLoadedModules ( ) ;
}

// Этот макрос создает эпилог функции, восстанавливая
// регистры и стек. Значение, передаваемое макросу, - это
// число регистров, переданных в функцию. Вырезая и вставляя
// данный макрос в другие функции, будьте очень внимательны,
// чтобы не допустить ошибку "наследования при редактировании"!
HOOKFN_EPILOG ( 1 ) ;
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/*//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Структура состояния регистров. Я использую эту структуру, чтобы
// гарантировать, что ВСЕ регистры по возвращении будут иметь такие
// же значения, какие были после выполнения действительной функции.
// Обратите внимание, что EBP и ESP обрабатываются во время пролога.
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
typedef struct tag_REGSTATE
{
    DWORD   dwEAX ;
    DWORD   dwEBX ;
    DWORD   dwECX ;
    DWORD   dwEDX ;
    DWORD   dwEDI ;
    DWORD   dwESI ;
    DWORD   dwEFL ;
} REGSTATE , * PREGSTATE ;

/*//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Макросы для сохранения и восстановления ESI между

```

```

// вызовами функций в отладочных компоновках.
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
#ifdef _DEBUG
#define SAVE_ESI()      __asm PUSH ESI
#define RESTORE_ESI()  __asm POP  ESI
#else
#define SAVE_ESI()
#define RESTORE_ESI()
#endif

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Общий пролог для всех функций DD_*.
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
#define HOOKFN_PROLOG() \
/* Все функции-ловушки автоматически получают */ \
/* три одинаковых локальных переменных. */ \
DDEVENTINFO stEvtInfo ; /* Информация о событии для функции. */ \
DWORD dwLastError ; /* Значение последней ошибки. */ \
REGSTATE stRegState ; /* Состояние регистров, нужное для их */ \
/* правильного восстановления. */ \
{ \
__asm PUSH EBP /* Всегда явно сохраняйте EBP. */ \
__asm MOV EBP , ESP /* Настройка кадра стека. */ \
__asm MOV EAX , ESP /* Получение указателя стека для подсчета */ \
/* адреса возврата и адреса параметров. */ \
SAVE_ESI ( ) /* Сохранение ESI в отлад. компоновках. */ \
__asm SUB ESP , __LOCAL_SIZE /* Место для локальных переменных. */ \
__asm ADD EAX , 04h + 04h /* Нужно учесть команду PUSH EBP */ \
/* и адрес возврата. */ \
/* Сохранение начала параметров в стеке. */ \
__asm MOV [stEvtInfo.dwParams] , EAX \
__asm SUB EAX , 04h /* Вернуться к адресу возврата. */ \
__asm MOV EAX , [EAX] /* Теперь EAX содержит адрес возврата. */ \
/* Сохранение адреса возврата. */ \
__asm MOV [stEvtInfo.dwAddr] , EAX \
__asm MOV dwLastError , 0 /* Инициализация dwLastError. */ \
/* Инициализация информации о событии. */ \
__asm MOV [stEvtInfo.eFunc] , eUNINITIALIZEDFE \
__asm MOV [stRegState.dwEDI] , EDI /* Сохранение двух регистров, */ \
__asm MOV [stRegState.dwESI] , ESI /* которые нужно сохранять */ \
/* между вызовами функций. */ \
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Общий эпилог для всех функций DD_*. INumParams – это число
// параметров функции, используемое для восстановления
// правильного состояния стека после вызова ловушки.
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
#define HOOKFN_EPILOG(iNumParams) \
{ \

```

```

SetLastError ( dwLastError );          /* Установка кода последней      */
/* ошибки действительной ф-ции.      */
__asm ADD    ESP , __LOCAL_SIZE        /* Добавление к ESP размера      */
/* локальных переменных.          */
__asm MOV    EBX , [stRegState.dwEBX] /* Восстановление всех регистров, */
__asm MOV    ECX , [stRegState.dwECX] /* чтобы этот вызов был          */
__asm MOV    EDX , [stRegState.dwEDX] /* прозрачным для перехваченной  */
__asm MOV    EDI , [stRegState.dwEDI] /* функции.                      */
__asm MOV    ESI , [stRegState.dwESI] \
__asm MOV    EAX , [stRegState.dwEFL] \
__asm SAHF                                     \
__asm MOV    EAX , [stRegState.dwEAX] \
RESTORE_ESI ( )                             /* Восстановление ESI в          */
/* отладочных компоновках          */
__asm MOV    ESP , EBP                     /* Восстановление ESP.          */
__asm POP    EBP                         /* Восстановление EBP.          */
__asm RET    iNumParams * 4               /* Восстановление стека для      */
/* функций с соглашением stdcall.  */
}

/*//////////////////////////////////////
// Макрос REAL_FUNC_PRE_CALL нужно размещать НЕПОСРЕДСТВЕННО
// *ПЕРЕД* ЛЮБЫМ вызовом действительной функции, обрабатываемым
// ловушкой. Этот макрос гарантирует, что регистры EDI и ESI будут
// иметь те же значения, что и при вызове функции-ловушки.
//////////////////////////////////////*/
#define REAL_FUNC_PRE_CALL() \
{ \
__asm MOV    EDI , [stRegState.dwEDI] /* Восстановление EDI.          */
__asm MOV    ESI , [stRegState.dwESI] /* Восстановление ESI.          */
}

/*//////////////////////////////////////
// Макрос REAL_FUNC_POST_CALL нужно размещать СРАЗУ ЖЕ *ПОСЛЕ*
// ЛЮБОГО вызова действительной функции, обрабатываемого ловушкой.
// Он сохраняет значения всех регистров после вызова действительной
// функции, чтобы эпилог функции-ловушки мог вернуть те же значения,
// что и действительная функция.
//////////////////////////////////////*/
#define REAL_FUNC_POST_CALL() \
{ \
__asm MOV    [stRegState.dwEAX] , EAX /* Сохранение значения EAX.    */
__asm MOV    [stRegState.dwEBX] , EBX /* Сохранение значения EBX.    */
__asm MOV    [stRegState.dwECX] , ECX /* Сохранение значения ECX.    */
__asm MOV    [stRegState.dwEDX] , EDX /* Сохранение значения EDX.    */
__asm MOV    [stRegState.dwEDI] , EDI /* Сохранение значения EDI.    */
__asm MOV    [stRegState.dwESI] , ESI /* Сохранение значения ESI.    */
__asm XOR    EAX , EAX               /* Обнуление EAX.              */
__asm LAHF                                     /* Загрузка флагов в AH.       */
__asm MOV    [stRegState.dwEFL] , EAX /* Сохранение флагов.         */

```

см. след. стр.

```

}
dwLastError = GetLastError ( ) ;      /* Сохр. кода последней ошибки.*/\
{
__asm MOV    EAX , [stRegState.dwEAX] /* Восстановление EAX          */\
/* Установка возвращаемого          */\
/* значения.                        */\
__asm MOV    [stEvtInfo.dwRetVal] , EAX
}

/*//////////////////////////////////////
// Удобный макрос для заполнения структуры информации о событии
//////////////////////////////////////*/
#define FILL_EVENTINFO(eFn)                \
    stEvtInfo.eFunc      = eFn            ; \
    stEvtInfo.ePrePost   = ePostCall ;    \
    stEvtInfo.dwThreadId = GetCurrentThreadId ( )

/*//////////////////////////////////////
// Макросы для второй версии программы, ЗНАЧИТЕЛЬНО
// облегчающие определение функций-ловушек
//////////////////////////////////////*/
// Объявляйте его в начале каждой функции-ловушки.
// eFunc      - значение перечисления, соответствующее функции.
// SynchClassType - значение флага DDOPT_*, указывающее на класс
//                обрабатываемой вами функции.
// bRecordPreCall - выполняет запись информации об этой функции.
#define HOOKFN_STARTUP(eFunc,SynchClassType,bRecordPreCall) \
    HOOKFN_PROLOG ( ) ; \
    if ( TRUE == DoLogging ( SynchClassType ) ) \
    { \
        FILL_EVENTINFO ( eFunc ) ; \
        if ( TRUE == (int)bRecordPreCall ) \
        { \
            stEvtInfo.ePrePost = ePreCall ; \
            ProcessEvent ( &stEvtInfo ) ; \
        } \
    } \
    REAL_FUNC_PRE_CALL ( ) ;

/*//////////////////////////////////////
// Макрос завершения функции-ловушки.
// iNumParams - число параметров, переданных функции.
// SynchClassType - класс функции синхронизации.
//////////////////////////////////////*/
#define HOOKFN_SHUTDOWN(iNumParams,SynchClass) \
    REAL_FUNC_POST_CALL ( ) ; \
    if ( TRUE == DoLogging ( SynchClass ) ) \
    { \
        stEvtInfo.ePrePost = ePostCall ; \
        ProcessEvent ( &stEvtInfo ) ; \
    }

```

```

}
HOOKFN_EPILOG ( iNumParams );

/*//////////////////////////////////////
// Объявление соглашения вызова для всех функций DD_*.
////////////////////////////////////*/
#define NAKEDDEF __declspec(naked)

/*//////////////////////////////////////
// ВАЖНОЕ ПРИМЕЧАНИЕ! ВАЖНОЕ ПРИМЕЧАНИЕ!
// Следующие прототипы выглядят, как функции с соглашением __cdecl, но на
// самом деле это не так: они все __stdcall! Использование правильного
// соглашения вызова гарантируется моими прологом и эпилогом!
////////////////////////////////////*/

//////////////////////////////////////
// Функции, перехват которых обязателен, иначе система не будет
// работать.
BOOL DD_FreeLibrary ( HMODULE hModule );
VOID DD_FreeLibraryAndExitThread ( HMODULE hModule ,
                                   DWORD dwExitCode );
HMODULE DD_LoadLibraryA ( LPCSTR lpLibFileName );
HMODULE DD_LoadLibraryW ( LPCWSTR lpLibFileName );
HMODULE DD_LoadLibraryExA ( LPCSTR lpLibFileName ,
                           HANDLE hFile ,
                           DWORD dwFlags );
HMODULE DD_LoadLibraryExW ( LPCWSTR lpLibFileName ,
                           HANDLE hFile ,
                           DWORD dwFlags );

VOID DD_ExitProcess ( UINT uExitCode );

FARPROC DD_GetProcAddress ( HMODULE hModule , LPCSTR lpProcName );

//////////////////////////////////////
// Функции работы с потоками
HANDLE DD_CreateThread (LPSECURITY_ATTRIBUTES lpThreadAttributes ,
                       DWORD dwStackSize ,
                       LPTHREAD_START_ROUTINE lpStartAddress ,
                       LPVOID lpParameter ,
                       DWORD dwCreationFlags ,
                       LPDWORD lpThreadId );
VOID DD_ExitThread ( DWORD dwExitCode );
DWORD DD_SuspendThread ( HANDLE hThread );
DWORD DD_ResumeThread ( HANDLE hThread );
BOOL DD_TerminateThread ( HANDLE hThread , DWORD dwExitCode );

// Ниже приведены функции работы с потоками стандартной библиотеки C.
// Они обрабатываются правильно, так как используют соглашение __cdecl.
uintptr_t

```

см. след. стр.

```

    DD_beginthreadex ( void *          security          ,
                      unsigned         stack_size        ,
                      unsigned ( __stdcall *start_address )( void * ) ,
                      void *          arglist           ,
                      unsigned         initflag          ,
                      unsigned *       thrddaddr         ) ;

uintptr_t
    DD_beginthread ( void( __cdecl *start_address )( void * ) ,
                    unsigned  stack_size                      ,
                    void *    arglist                         ) ;

VOID DD_endthreadex ( unsigned retval ) ;
VOID DD_endthread  ( void ) ;

////////////////////////////////////
// Функции ожидания и специальные функции
DWORD DD_WaitForSingleObject ( HANDLE hHandle
                              ,
                              DWORD  dwMilliseconds ) ;
DWORD DD_WaitForSingleObjectEx ( HANDLE hHandle
                                ,
                                DWORD  dwMilliseconds ,
                                BOOL    bAlertable    ) ;
DWORD DD_WaitForMultipleObjects( DWORD    nCount
                                ,
                                CONST HANDLE * lpHandles
                                ,
                                BOOL    bWaitAll
                                ,
                                DWORD    dwMilliseconds ) ;
DWORD DD_WaitForMultipleObjectsEx( DWORD    nCount
                                ,
                                CONST HANDLE * lpHandles
                                ,
                                BOOL    bWaitAll
                                ,
                                DWORD    dwMilliseconds ,
                                BOOL    bAlertable    ) ;
DWORD DD_MsgWaitForMultipleObjects ( DWORD    nCount
                                    ,
                                    LPHANDLE pHandles
                                    ,
                                    BOOL    fWaitAll
                                    ,
                                    DWORD    dwMilliseconds,
                                    DWORD    dwWakeMask    ) ;
DWORD DD_MsgWaitForMultipleObjectsEx ( DWORD    nCount
                                    ,
                                    LPHANDLE pHandles
                                    ,
                                    DWORD    dwMilliseconds
                                    ,
                                    DWORD    dwWakeMask
                                    ,
                                    DWORD    dwFlags
                                    ) ;
DWORD DD_SignalObjectAndWait ( HANDLE hObjectToSignal ,
                              HANDLE hObjectToWaitOn ,
                              DWORD  dwMilliseconds ,
                              BOOL    bAlertable     ) ;
BOOL DD_CloseHandle ( HANDLE hObject ) ;

////////////////////////////////////
// Функции работы с критическими секциями
VOID DD_InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection);
BOOL DD_InitializeCriticalSectionAndSpinCount (
    LPCRITICAL_SECTION lpCriticalSection,

```

```

                                DWORD          dwSpinCount    );
VOID DD_DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection );
VOID DD_EnterCriticalSection ( LPCRITICAL_SECTION lpCriticalSection );
VOID DD_LeaveCriticalSection ( LPCRITICAL_SECTION lpCriticalSection );
DWORD DD_SetCriticalSectionSpinCount (
                                LPCRITICAL_SECTION lpCriticalSection,
                                DWORD          dwSpinCount    );
BOOL DD_TryEnterCriticalSection ( LPCRITICAL_SECTION lpCriticalSection);

////////////////////////////////////
// Функции работы с мьютексами
HANDLE DD_CreateMutexA ( LPSECURITY_ATTRIBUTES lpMutexAttributes ,
                        BOOL          bInitialOwner    ,
                        LPCSTR        lpName          ) ;
HANDLE DD_CreateMutexW ( LPSECURITY_ATTRIBUTES lpMutexAttributes ,
                        BOOL          bInitialOwner    ,
                        LPCWSTR       lpName          ) ;
HANDLE DD_OpenMutexA ( DWORD dwDesiredAccess ,
                      BOOL bInheritHandle ,
                      LPCSTR lpName        ) ;
HANDLE DD_OpenMutexW ( DWORD dwDesiredAccess ,
                      BOOL bInheritHandle ,
                      LPCWSTR lpName        ) ;
BOOL DD_ReleaseMutex ( HANDLE hMutex ) ;

////////////////////////////////////
// Функции работы с семафорами
HANDLE
    DD_CreateSemaphoreA ( LPSECURITY_ATTRIBUTES lpSemaphoreAttributes ,
                        LONG          lInitialCount    ,
                        LONG          lMaximumCount    ,
                        LPCSTR        lpName          ) ;
HANDLE
    DD_CreateSemaphoreW ( LPSECURITY_ATTRIBUTES lpSemaphoreAttributes ,
                        LONG          lInitialCount    ,
                        LONG          lMaximumCount    ,
                        LPCWSTR       lpName          ) ;
HANDLE DD_OpenSemaphoreA ( DWORD dwDesiredAccess ,
                          BOOL bInheritHandle ,
                          LPCSTR lpName        ) ;
HANDLE DD_OpenSemaphoreW ( DWORD dwDesiredAccess ,
                          BOOL bInheritHandle ,
                          LPCWSTR lpName        ) ;
BOOL DD_ReleaseSemaphore ( HANDLE hSemaphore ,
                          LONG lReleaseCount ,
                          LPLONG lpPreviousCount ) ;

////////////////////////////////////
// Функции работы с событиями
HANDLE DD_CreateEventA ( LPSECURITY_ATTRIBUTES lpEventAttributes ,

```

см. след. стр.

```

        BOOL                bManualReset        ,
        BOOL                bInitialState       ,
        LPCSTR              lpName              ) ;
HANDLE DD_CreateEventW ( LPSECURITY_ATTRIBUTES lpEventAttributes ,
        BOOL                bManualReset        ,
        BOOL                bInitialState       ,
        LPCWSTR             lpName              ) ;
HANDLE DD_OpenEventA ( DWORD dwDesiredAccess ,
        BOOL                bInheritHandle     ,
        LPCSTR              lpName              ) ;
HANDLE DD_OpenEventW ( DWORD dwDesiredAccess ,
        BOOL                bInheritHandle     ,
        LPCWSTR             lpName              ) ;
BOOL DD_PulseEvent ( HANDLE hEvent ) ;
BOOL DD_ResetEvent ( HANDLE hEvent ) ;
BOOL DD_SetEvent ( HANDLE hEvent ) ;

#endif // _DD_FUNCS_H

```

В первой версии DeadlockDetection я допустил глупейшую ошибку «наследования при редактировании». Я создал функции-ловушки для LoadLibraryA и LoadLibraryW и понял, что мне также нужны ловушки для LoadLibraryExA и LoadLibraryExW. Как типичный программист, я вырезал функции LoadLibraryA/W, вставил их в нужное место кода и отредактировал с учетом особенностей LoadLibraryExA/W. Если вы посмотрите на макрос HOOKFN_EPILOG, то увидите, что он принимает некоторое значение, а именно число параметров функции. Наверное, вы уже догадались, что случилось: я забыл изменить это значение с 1 на 3, поэтому вызовы LoadLibraryExA/W удаляли из стека два лишних элемента и приводили к краху программ!

Изучив код всех функций-ловушек, я понял, что он по сути везде одинаков. Для инкапсуляции общих действий я создал макросы HOOKFN_STARTUP и HOOKFN_SHUTDOWN. Как видно по имени, макрос HOOKFN_STARTUP размещается в начале функции-ловушки и заботится о прологе, а также выполняет необходимое протоколирование до вызова действительной функции. Он принимает следующие параметры: перечисление функции, флаг DDOPT_*, показывающий, к какой группе относится данная функция, и булев флаг, выполняющий предварительное протоколирование, если имеет значение TRUE. Предварительное протоколирование предназначено для функций, которые потенциально могут вызывать блокировку, таких как WaitForSingleObject. Макрос HOOKFN_SHUTDOWN принимает число параметров функции и тот же флаг DDOPT_*, что передается в HOOKFN_STARTUP. Конечно, чтобы не допустить ту же ошибку, которую я сделал в случае ловушек LoadLibraryExA/W, я проверил число параметров HOOKFN_SHUTDOWN.

Я хочу упомянуть еще несколько деталей. Во-первых, DeadlockDetection всегда активна в вашем приложении, даже если вы приостанавливаете протоколирование. Вместо того чтобы устанавливать и удалять ловушки, я оставляю функции перехваченными и изучаю некоторые внутренние флаги, чтобы определить, как ловушка должна себя вести. Поддержание всех функций в перехваченном состоянии упрощает переключение между разными протоколируемыми функциями в

период выполнения, но несколько снижает эффективность вашей программы. Я чувствовал, что реализация перехвата и его отмены «на лету» привела бы к появлению дополнительных ошибок.

Во-вторых, `DeadlockDetection` перехватывает функции при загрузке DLL в вашу программу с помощью `LoadLibrary`. Однако `DeadlockDetection` может получить контроль только после выполнения в DLL функции `DllMain`, поэтому, если какие-то объекты синхронизации создаются или используются во время `DllMain`, `DeadlockDetection` может их упустить.

В-третьих, `DeadlockDetection` также перехватывает функции `GetProcAddress` и `ExitProcess`. Перехват `GetProcAddress` полезен, если ваша программа или элемент управления сторонней фирмы, который может привести к блокировке, вызывает `GetProcAddress` для нахождения метода синхронизации в период выполнения.

Я перехватываю `ExitProcess` потому, что при завершении приложения мне нужно отменить перехват функций и завершить `DeadlockDetection`, чтобы она не вызвала крах или зависание вашей программы. Так как при завершении программы контролировать порядок выгрузки DLL невозможно, вы с легкостью можете попасть в ситуацию, когда DLL утилиты `DeadlockDetection`, такая как `DeadDetExt`, выгружается до самой `DeadlockDetection`. К счастью, очень немногие программы обрабатывают несколько потоков после вызова `ExitProcess`.

Перехват `ExitProcess` реализован в файле `DEADLOCKDETECTION.CPP`. Из-за огромной важности правильного завершения `DeadlockDetection` я принудительно перехватываю все вызовы `ExitProcess` даже в игнорируемых модулях. Это позволяет исключить неожиданные ошибки, при которых функции синхронизации остаются перехваченными после завершения `DeadlockDetection`.

Наконец, вместе с `DeadlockDetection` вы можете найти на CD несколько тестовых программ. Все они включены в главное решение `DeadlockDetectionTests` и компонируются вместе с `DEADLOCKDETECTION.DLL`. Они помогут вам понять работу `DeadlockDetection`.

Что после `DeadlockDetection`?

`DeadlockDetection` — достаточно полная утилита, и я успешно применял ее для отслеживания многих многопоточных блокировок. Однако, как всегда, я предлагаю вам обдумать возможности расширения `DeadlockDetection`, чтобы сделать ее еще полезнее. Вот некоторые мои идеи по этому поводу.

- Создайте отдельное приложение для работы с файлом `DEADLOCKDETECTION.INI`. Будет еще лучше, если оно позволит указывать DLL `DeadDetExt` и будет проверять, что эта DLL экспортирует корректные функции.
- Вы можете оптимизировать функции-ловушки, если они не выполняют протоколирования. В этом случае нужно копировать не все значения регистров.
- На данный момент `DeadlockDetection` просто пропускает перехват некоторых DLL, которые ей указаны. Было бы великолепно, если бы вы разработали механизм пропуска DLL с учетом выполняемой программы.

Отладка: фронтовые очерки

Проблема фиксации транзакций при использовании пула объектов COM

Боевые действия

Мой хороший друг Питер Иерарди (Peter Ierardi) рассказал об одной интересной многопоточной ошибке. Он работал над крупным проектом DCOM, включавшим многопоточную службу DCOM для координации транзакций базы данных. Служба DCOM управлял транзакциями, создавая пул ориентированных на базу данных внутривещественных COM-объектов, применявшихся для записи и чтения данных из реляционной СУБД. Взаимодействие компонентов осуществлялось при помощи Microsoft Message Queue Server (MSMQ). Несмотря на явную фиксацию транзакций, данные, похоже, не записывались в базу. Служба DCOM повторяла попытку от трех до пяти раз, и только после этого данные появлялись, как по щучьему велению. Очевидно, лишние попытки снижали быстродействие приложения, а то, что данные не хотели записываться в базу данных, вызывал тревогу.

Исход

После нескольких тяжелых сеансов отладки Питер обнаружил, что служба DCOM выполнял чтение/запись при помощи отдельных несинхронизированных потоков. Чтение происходило до того, как отдельный экземпляр COM-объекта базы записывал данные. Во время отладки это поведение было далеко не очевидным, потому что отладчик навязывал правильный отсчет времени и синхронизацию. В конце концов Питер обнаружил проблему, правильно отметив объекты в журнале событий.

Полученный опыт

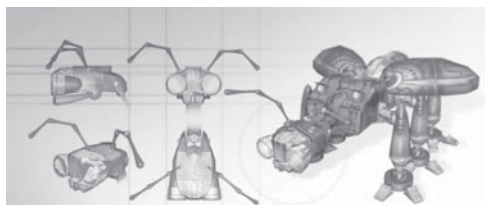
Как сказал Питер, эта ошибка помогла ему извлечь один очень важный урок: работая над крупномасштабным распределенным приложением, нельзя предполагать, что среда отладки правильно воспроизведет среду выполнения заключительной версии программы. Он решил проблему, добавив нужный код синхронизации и включив взаимодействие компонентов (которые первоначально взаимодействовали по отдельности при помощи MSMQ), в транзакции записи в базу данных, чтобы сообщения отправлялись только после фиксации транзакций.

Ошибка Питера состояла в том, что MSMQ выполнял операции чтения/записи, как легко догадаться, гораздо быстрее, чем база данных. Хотя Питер и другие члены его группы тщательно проработали и спланировали все многопоточные фрагменты, они все же не до конца осознавали, насколько быстрее в реальном мире будут выполняться определенные операции, не подвластные их приложению.

Резюме

Разрабатывать многопоточные приложения трудно, и в этой области встречаются одни из самых сложных ошибок. В данной главе я обсудил методы, которые помогут вам избегать блокировок с самого начала проекта. Как я подчеркнул в начале главы, при программировании многопоточного приложения особую важность приобретает заблаговременное планирование, поэтому, приступая к работе над такой программой, вы должны предоставить своей группе достаточно времени и ресурсов для тщательного и правильного ее проектирования. Однако, когда вы столкнетесь с неизбежными многопоточными блокировками, не нужно паниковать — в этой главе я представил утилиту `DeadlockDetection`, которая скажет вам, какие потоки заблокированы на каком объекте синхронизации.

Наконец (и важность этого момента нельзя переоценить), при программировании многопоточных приложений вы должны разрабатывать, выполнять и тестировать их на многопроцессорных компьютерах, иначе многопоточность лучше вообще не использовать — так вы избежите некоторых чрезвычайно серьезных ошибок.



Автоматизированное тестирование

В главе 3 я рассказал про блочные тесты и объяснил, почему они так важны для разработки высококачественного кода. Если вы работаете преимущественно над внутренней логикой приложения, блочные тесты могут быть довольно просты. На CD в каталоге `BugslayerUtil\Tests` вы можете найти все блочные тесты, использованные мной при разработке `BUGSLAYERUTIL.DLL`. Почти все они — консольные приложения, прекрасно справляющиеся со своими обязанностями.

К сожалению, тестирование пользовательского интерфейса (UI) гораздо сложнее независимо от того, является ли приложение «толстым клиентом» Microsoft .NET или работает на базе браузера. В этой главе я расскажу про мою утилиту `Tester`, которая поможет вам автоматизировать тестирование кода UI. По сравнению с версией `Tester`, включенной в первое издание книги, новая утилита `Tester` уже приближается к возможностям полного коммерческого средства регрессивного тестирования. `Tester` на самом деле применяют очень многие группы разработчиков, чем я весьма польщен. Она не только проще в работе многих коммерческих систем, но и гораздо дешевле.

Проклятие блочного тестирования: UI

Абсолютно уверен, что если разработчики программ для Microsoft Windows и получают туннельный синдром, то это вызвано не написанием исходного кода, а многократными нажатиями одних и тех же комбинаций клавиш при тестировании приложений. После пятидесяти тысяч нажатия `Alt+F`, О запястья сковываются сильнее, чем арматура, залитая бетоном. При отсутствии средства, автоматизирующего доступ к различным функциям ваших приложений, обычно необходимо следовать некоторому сценарию, чтобы гарантировать, что блочное тестирова-

ние проведено в достаточном объеме. Тестирование программ вручную при помощи сценариев мгновенно надоедает, значительно повышая вероятность человеческих ошибок.

Автоматизация блочного тестирования позволяет уменьшить объем работы с клавиатурой и дает возможность быстрой проверки состояния кода. Очень жаль, но аналога программы Recorder из состава Microsoft Windows 3.0 и 3.1 в 32-разрядных ОС нет. Если вы не работали со старыми версиями Windows, я поясню сказанное: Recorder записывала ваши манипуляции с мышью и клавиатурой в файл, который позднее можно было воспроизвести, симитировав физические события мыши и клавиатуры. В настоящее время доступны некоторые программы сторонних фирм, обеспечивающие автоматизацию работы с приложением и другие возможности (например, сравнение экранов с проверкой каждого пиксела и поддержку базы данных о времени проведения тех или иных тестов), но я все равно хотел разработать что-то более простое и дружественное к программистам. Так родилась идея Tester.

Задумав утилиту автоматизации тестирования, я некоторое время размышлял о том, какие именно возможности мне хотелось бы получить. Сначала я решил разработать утилиту вроде Recorder. Во времена Windows 3.0 у меня был целый набор REC-файлов для выполнения моих тестов. Однако Recorder имел большой недостаток: он не поддерживал условные тесты. Если во время тестирования мое приложение сообщало об ошибке, Recorder просто продолжал работу, проигрывая записанные нажатия клавиш клавиатуры и мыши и полностью игнорируя страдания моей программы. Однажды благодаря Recorder я умудрился уничтожить половину ОС: я тестировал собственное расширение WINFILE.EXE, и, когда в нем возникла ошибка, Recorder проиграл команду удаления файлов для всего каталога System. Мое новое средство автоматизации тестирования непременно должно было поддерживать конструкцию `if...then...else`.

Очевидно, что для этого мне нужен был некоторый вид языка. Разработка собственного языка тестирования казалась заманчивым интеллектуальным упражнением, но вскоре я пришел к выводу, что я больше заинтересован в полезном отладочном средстве, а не в проектировании языка и работе с YACC и FLEX. Я почти сразу понял, что Tester нужно реализовать как объект COM: благодаря этому программисты могли бы создавать тесты на предпочтительном для них языке, а я мог бы сосредоточиться на программировании функций регрессивного тестирования, а не на разработке нового языка. Лично я предпочитаю создавать тесты на языках сценариев, таких как Microsoft Visual Basic Scripting Edition (VBScript) и Microsoft JScript, потому что они не требуют компиляции. Однако различные реализации механизма сценариев Microsoft Windows Scripting Host (WSH) имеют некоторые ограничения, на которые я укажу ниже. Сейчас я хотел бы обсудить требования, которыми я руководствовался при создании Tester.

Требования к Tester

Я хотел, чтобы Tester очень хорошо делал две вещи: записывал нажатые вами комбинации клавиш и проигрывал их обратно вашему приложению, ускоряя проведение блочного тестирования. Если вы когда-либо изучали коммерческие сред-

ства регрессивного тестирования, то знаете, насколько различные функции они могут поддерживать: от простого управления окном до проверки самых сложных и причудливых свойств окна. Я хотел сосредоточиться на потребностях разработчиков во время блочного тестирования и сделать Tester простым в использовании. Вот какими были основные требования к Tester.

1. Возможность управления им при помощи любого языка, поддерживающего COM.
2. При получении строки нажатых клавиш в формате, используемом классом `System.Windows.Forms.SendKeys`, Tester должен уметь проигрывать ее активному окну.
3. Tester должен поддерживать возможность нахождения любых окон верхнего уровня или дочерних окон по их заголовку или классу.
4. При получении любого `HWND` Tester должен быть способен получить все свойства окна.
5. Tester должен уведомлять пользовательский сценарий о создании/уничтожении конкретного окна, чтобы сценарий мог обработать потенциальные условия ошибки или выполнить дополнительную обработку окна. Tester не должен ограничивать возможность расширения кода, позволяя разработчикам удовлетворить любые свои потребности.
6. Tester должен уметь записывать нажатия клавиш в строки, совместимые с его модулем воспроизведения.
7. Сохраняемые сценарии Tester должны быть полными, т. е. готовыми к запуску.
8. Пользователь должен иметь возможность редактирования автоматически сгенерированного сценария перед его сохранением.
9. Tester должен гарантировать правильность потока воспроизведения информации, присваивая фокус конкретным окнам, в том числе любым дочерним элементам управления.

Tester поддерживает практически полный набор функций, однако он, наверное, не решит всех задач, поставленных перед вашим отделом контроля качества, состоящим из 20 человек. Я просто хотел создать средство, которое позволило бы нам, программистам, автоматизировать блочное тестирование. Думаю, Tester отвечает этим требованиям. Он очень помог мне при разработке WDBG, отладчика с графическим пользовательским интерфейсом (GUI), описанного в главе 4. Самый приятный аспект работы с Tester при создании WDBG заключался в том, что он избавил меня от многих тысяч нажатий клавиш. Как видите, я уже добрался до 16 главы и все еще могу двигать пальцами!

Использование Tester

Работать с Tester относительно просто. Сначала я расскажу про объект Tester и его использование в сценариях, а затем перейду к обсуждению записи сценариев при помощи программы `TESTREC.EXE`. Разобравшись с объектом, который Tester предоставляет вашим сценариям, вы сможете создавать их более эффективно.

Сценарии Tester

Принцип работы сценариев прост: вы создаете несколько объектов Tester, запускаете приложение или находите его основное окно, воспроизводите ему несколько нажатий клавиш, проверяете результаты и завершаете сценарий. В листинге 16-1 представлен пример сценария VBScript, который запускает NOTEPAD.EXE, вводит несколько строк текста и закрывает Блокнот (все примеры сценариев, приведенные в этой главе, вы можете найти на CD).

Листинг 16-1. Сценарий MINIMAL.VBS поясняет работу с часто используемыми объектами Tester

```
' Минимальный пример сценария Tester на VBScript. Он просто запускает
' Notepad, вводит несколько строк текста и закрывает Notepad.

' Получение системного объекта и объекта ввода.
Dim tSystem
Dim tInput
Dim tWin
Set tSystem = WScript.CreateObject ( "Tester.TSystem" )
Set tInput = WScript.CreateObject ( "Tester.TInput" )

' Запуск Notepad.
tSystem.Execute "NOTEPAD.EXE"

' Подождать 3 секунды.
tSystem.Sleep 3.0

' Попытка найти основное окно Notepad.
Set tWin = tSystem.FindTopTWindowByTitle ( "Untitled - Notepad" )
If ( tWin Is Nothing ) Then
    MsgBox "Unable to find Notepad!"
    WScript.Quit
End If

' Гарантия того, что Notepad работает в активном режиме.
tWin.SetForegroundTWindow

' Ввод чего-нибудь.
tInput.PlayInput "Be all you can be!~~~~"
' Еще раз.
tInput.PlayInput "Put on your boots and parachutes....~~~~"
' И еще.
tInput.PlayInput "Silver wings upon their chests.....~~~~"
' Подождать 3 секунды.
tSystem.Sleep 3.0

' Закрывать Notepad.
tInput.PlayInput "%FX"
tSystem.Sleep 2.0
tInput.PlayInput "{TAB}~"

' Сценарий выполнен!
```

В листинге 16-1 вы можете увидеть три объекта, чаще всего используемых Tester. Объект TSystem позволяет находить окна верхнего уровня, запускать приложения и приостанавливать тестирование. Объект TWindow, возвращаемый в листинге 16-1 методом FindTopTWindowByTitle, — главная рабочая лошадка и является оболочкой для HWND, включающей все виды свойств окна, которые вам могут понадобиться. Кроме того, TWindow умеет находить все дочерние окна, относящиеся к конкретному родительскому окну. Последний объект в листинге 16-1 — объект TInput, поддерживающий единственный метод PlayInput для воспроизведения нажатий клавиш окну с фокусом.

В листинге 16-2 представлен тест на языке VBScript, поясняющий работу с объектом TNotify. Одна из самых сложных проблем при разработке сценариев автоматизации тестирования связана с появлением неожиданного окна, такого как информационное окно ASSERT. Объект TNotify позволяет предоставить обработчик таких непредвиденных событий. Несложный сценарий, показанный в листинге 16-2, просто следит за любыми окнами, содержащими в заголовке слово «Notepad». Скорее всего класс TNotify вам понадобится нечасто, но порой он по-настоящему нужен.

Листинг 16-2. Сценарий HANDLERS.VBS демонстрирует использование объекта TNotify

```
' Тест VBScript, иллюстрирующий обработку оконных уведомлений

' Константы, передаваемые в метод TNotify.AddNotification.
Const antDestroyWindow    = 1
Const antCreateWindow     = 2
Const antCreateAndDestroy = 3

Const ansExactMatch       = 0
Const ansBeginMatch       = 1
Const ansAnyLocMatch      = 2

' Получение системного объекта и объекта ввода.

Dim tSystem
Dim tInput
Set tSystem = WScript.CreateObject ( "Tester.TSystem" )
Set tInput = WScript.CreateObject ( "Tester.TInput" )
' Переменная для объекта TNotify.
Dim Notifier

' Создание объекта TNotify.
Set Notifier = _
    WScript.CreateObject ( "Tester.TNotify" , _
        "NotepadNotification" )

' Добавление интересующих меня уведомлений. В данном случае мне
' нужны уведомления об уничтожении и создании окна. Все возможные
' комбинации уведомлений вы найдете в исходном коде TNotify.
```



```

Notifier.AddNotification antCreateAndDestroy , _
                        ansAnyLocMatch      , _
                        "Notepad"

' Запуск Notepad.
tSystem.Execute "NOTEPAD.EXE"

' Пауза на 1 секунду.
tSystem.Sleep 1.0

' Поскольку модель разделенных потоков небезопасна с точки зрения
' потоков, я использую в схеме уведомлений таймер. Однако сообщение
' может быть заблокировано, поскольку вся обработка ограничивается
' одним потоком. Эта функция позволяет вручную проверить уведомления
' о создании и уничтожении окна.
Notifier.CheckNotification

' Информационное окно процедуры NotepadNotification_CreateWindow
' вызывает блокировку, поэтому код завершения Notepad не будет
' выполнен, пока оно не будет закрыто.
tInput.PlayInput "%FX"
tSystem.Sleep 1.0

' Еще одна проверка уведомлений.
Notifier.CheckNotification

' Я даю TNotify шанс на перехват сообщения об уничтожении окна.
tSystem.Sleep 1.0

' Отключение уведомлений. Если при работе с WSH этого
' не сделать, объект не будет уничтожен, и уведомления
' в таблице уведомлений останутся в активном состоянии.
WScript.DisconnectObject Notifier

Set Notifier = Nothing

WScript.Quit

Sub NotepadNotificationCreateWindow ( tWin )
    MsgBox ( "Notepad was created!!" )
End Sub

Sub NotepadNotificationDestroyWindow ( )
    MsgBox ( "Notepad has gone away...." )
End Sub

```

Время от времени нужно вызывать метод `CheckNotification` объекта `TNotify` (причины этого я объясню в разделе «Реализация Tester»). Периодический вызов метода `CheckNotification` гарантирует поступление уведомлений, даже если в выбранном

вами языке нет цикла сообщений. Листинг 16-2 иллюстрирует использование информационных окон (message box) в процедурах уведомлений, однако вам, вероятно, не захочется включать в реальные сценарии вызовы информационных окон, потому что они могут вызвать проблемы, неожиданно изменяя окно с фокусом.

Помните также, что я позволяю задать только ограниченное число уведомлений — пять, поэтому вам не следует использовать `TNotify` для общих задач сценариев, таких как ожидание появления окна сохранения файла. `TNotify` следует применять только для обработки неожиданных окон. Вы можете определить свои обработчики уведомлений и параметры поиска текста в заголовке окна так, что будете получать уведомления для окон, в которых вы не заинтересованы. Скорее всего вы будете получать нежелательные уведомления при использовании общих строк, таких как «Notepad», и указании, что строка может находиться в любом месте заголовка окна. Для избежания нежелательных уведомлений методу `AddNotification` объекта `TNotify` надо передавать как можно более специфичную информацию. Кроме того, в процедурах обработки события `CreateWindow` следует изучать получаемый объект `TWindow`, чтобы можно было проверить, то ли это окно, которое вам нужно. В процедурах события `DestroyWindow`, обрабатывающих общие уведомления, следует выполнять поиск открытых окон, чтобы гарантировать, что интересующее вас окно больше не существует.

На CD есть и другие примеры, которые помогут вам лучше разобраться в работе с `Tester`. `NPAD_TEST.VBS` — это более полный тест `VBScript`, включающий некоторые повторно используемые блоки. `PAINTBRUSH.JS` иллюстрирует воспроизведение манипуляций с мышью, не зависящее от разрешения экрана. Его выполнение требует некоторого времени, однако результат того стоит. `TesterTester` — это основной блочный тест для COM-объекта `Tester`. Эта программа написана на `C#` и расположена в каталоге `Tester\Tester\Tests\TesterTester`. Изучив ее, вы получите представление о том, как использовать `Tester` вместе с `.NET`. Кроме того, пример `TesterTester` демонстрирует работу с объектом `TWindows` — коллекцией объектов `TWindow`.

Хотя я предпочитаю писать свои блочные тесты на языках `JScript` и `VBScript`, я понимаю, что иногда это довольно трудно. Языки сценариев не позволяют контролировать тип переменных и не включают редактор `IntelliSense`, такой как редактор `C#` в `Visual Studio .NET`, поэтому вам придется вернуться к старому стилю отладки — «запустить и ошибиться». Языки сценариев нравятся мне в первую очередь тем, что они не требуют компиляции тестов. Если вы работаете с гибкой средой сборки программы, в которой легко создавать другие двоичные файлы в дополнение к главному приложению, вы можете применять `.NET`, создавая тесты вместе со сборкой своего приложения. Конечно, `Tester` не ограничивает вас простейшими языками тестирования. Если вам удобней писать тесты на `C` или `Microsoft Macro Assembler (MASM)` — пожалуйста.

Использовать объекты `Tester` довольно просто, однако реальная работа состоит в планировании тестов. Ваши тесты должны быть максимально конкретными и простыми. Когда я только приступал к автоматизации своих блочных тестов в начале карьеры, я пытался включить в них побольше функций. Теперь каждый мой сценарий тестирует только одну операцию. В качестве хорошего примера такого

сценария можно привести тест открытия файла. Для повторного использования сценариев вы можете объединить их самыми разными способами. Например, как только вы напишете сценарий открытия файла, вы сможете включить его в три различных теста: тест открытия существующего файла, тест открытия несуществующего файла и тест открытия поврежденного файла. Как и при разработке обычных программ, следует избегать включения в тесты жестко закодированных строк. Это не только облегчит интернационализацию сценария, но и упростит его адаптацию к очередной версии системы меню и комбинаций управляющих клавиш (accelerator).

При разработке сценариев Tester нужно предусмотреть и способы проверки того, что сценарии на самом деле работают. Если вам нечем заняться, можете просто сидеть и следить за их выполнением, сравнивая результаты каждого запуска. Однако лучше было бы записывать основные состояния и точки сценария, чтобы сравнение результатов можно было проводить автоматически. Если для выполнения сценариев вы используете CSCRIPT.EXE, можете перенаправить вывод в файл методом `WScript.Echo`. По завершении сценария вы можете сравнить разные версии полученных файлов утилитой нахождения различий версий (такой как WinDiff) и узнать, корректно ли выполнен сценарий. Помните: записываемая информация должна быть нормализованной и не зависящей от деталей конкретного запуска сценария. Так, если вы работаете над приложением, загружающим из сети курсы акций, не следует записывать в файл время последнего обновления курса.

Что сказать об отладке сценариев Tester? Tester не включает собственного интегрированного отладчика, поэтому вам понадобятся другие средства отладки, доступные для языка, на котором написан сценарий. Отлаживая сценарий, старайтесь не прерываться на вызове метода `PlayInput` объекта `Tinput`, потому что при остановке на этом методе нажатия клавиш будут воспроизведены неправильному окну. Для решения этой потенциальной проблемы я обычно перед каждым вызовом `PlayInput` перемещаю окно, которому посылаю нажатия клавиш, на вершину z-порядка, вызывая метод `SetForegroundTWindow` объекта `TWindow`. Это позволяет мне прерваться на вызове `SetForegroundTWindow` и проверить состояние приложения, не вызывая ошибок воспроизведения нажатий клавиш.

Запись сценариев

Теперь вы понимаете работу объектов Tester и знаете, как их вызывать из собственных сценариев, поэтому я хочу перейти к рассмотрению программы TESTREC.EXE, которую вы будете применять для записи взаимодействия со своими приложениями. Запустив TESTREC.EXE в первый раз, вы заметите, что это текстовый редактор, который уже при запуске генерирует некоторый объем кода. По умолчанию сценарии создаются на языке JScript, но чуть ниже я покажу, как изменить его на VBScript. Для начала записи нужно нажать на панели инструментов кнопку Record (запись) или клавиши `Ctrl+R`.

При записи сценария TESTREC.EXE минимизируется и изменяет заголовок на «RECORDING!», давая вам знать о происходящем. Остановить запись можно несколькими способами. Самый простой — сделать TESTREC.EXE активной программой, нажав `Alt+Tab` или выбрав ее на панели задач. Запись сценария также остановится при нажатии `Ctrl+Break` или `Ctrl+Alt+Delete`; первый вариант упоминается в

документации к функциям-ловушкам, а второй позволяет принудительно завершить все активные ловушки записи журнала (journaling hooks), при помощи которых TESTREC.EXE колдует.

Прежде чем приступить к записи своих многочисленных сценариев, вы должны составить некоторый план, чтобы полностью задействовать преимущества Tester. Хотя Tester умеет обрабатывать и воспроизводить манипуляции с мышью, ваши сценарии будут гораздо более надежными, если все действия вы будете выполнять при помощи клавиатуры. Одно из достоинств Tester в том, что при записи сценария он внимательно следит за тем, какому окну принадлежит фокус. По умолчанию перед обработкой одиночных и двойных щелчков мыши Tester генерирует код, устанавливающий фокус на окно верхнего уровня. Кроме того, при записи действий с клавиатурой Tester отслеживает комбинацию Alt+Tab, также устанавливая фокус.

Так как запись всех событий мыши может привести к получению просто огромного сценария, по умолчанию TESTREC.EXE записывает только одиночные, двойные щелчки и перемещение курсора на каждые 50 пикселей при нажатой клавише. Конечно, я позволяю точно указать параметры записи сценариев. Диалоговое окно Script Recording Options (параметры записи сценариев) программы TESTREC.EXE, показанное на рис. 16-1, доступно при нажатии Ctrl+T или выборе пункта Script Options (параметры сценария) из меню Scripts (сценарии). На рисунке все пункты имеют значения по умолчанию.

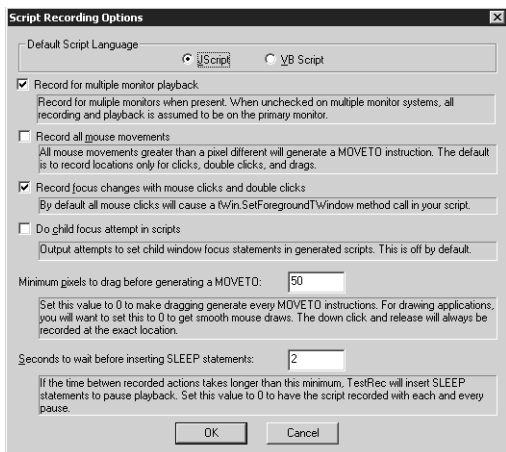


Рис. 16-1. Параметры записи сценариев Tester

В самой верхней части окна Script Recording Options вы можете выбрать язык записи новых сценариев: JScript или VBScript. Установленный по умолчанию флажок Record For Multiple Monitor Playback (запись сценария для воспроизведения на нескольких мониторах), включает в сценарий вызовы метода TSystem.CheckVirtualResolution, настраивающие размер экрана для последующих записываемых действий. Если этот флажок убрать, при нажатии кнопок мыши и доступе к точкам вне основного монитора запись будет прерываться. Возможно, вам следует отключить эту функцию, если вы планируете запускать записанные сценарии на несколь-

ких компьютерах. Однако при записи сценариев, с которыми будете работать только вы, лучше оставить этот флажок установленным, чтобы позже можно было задействовать все удобства нескольких мониторов.

Если вы создаете сценарий, включающий интенсивную работу с мышью, и желаете записать все движения мыши между нажатием и отпусканием ее клавиш, задайте в поле Minimum Pixels To Drag Before Generating A MOVETO (минимальное число пикселей, после которого при перетаскивании генерируется команда MOVETO) значение 0. Если вы собираетесь записать много нажатий кнопок мыши без передачи фокуса другим приложениям, снимите флажок Record Focus Changes With Mouse Clicks And Double Clicks (записывать изменения фокуса при одиночных и двойных щелчках мыши). Благодаря этому TESTREC.EXE не будет генерировать код, форсирующий установку фокуса при каждом щелчке, что сделает ваши сценарии гораздо меньше.

При заданном флажке Do Child Focus Attempt In Scripts (выполнять попытку установки фокуса на дочернем окне) в сценарий добавляется код, пытающийся установить фокус на конкретном элементе управления или дочернем окне, в котором вы щелкаете. По умолчанию этот параметр отключен, так как я генерирую команды установки фокуса на окне верхнего уровня. Такие приложения, как Notepad имеют только одно дочернее окно, однако многие другие программы характеризуются глубоко вложенной иерархией окон, и отслеживание дочерних окон может быть сложным, когда все родительские окна не имеют заголовков и уникальных классов. Изучите, например, иерархию окон редактора Visual Studio .NET при помощи утилиты Spy++. Я обнаружил, что установка фокуса на окне верхнего уровня перед генерированием кода нажатия кнопки мыши, как правило, работает отлично.

Наконец, параметр Seconds To Wait Before Inserting SLEEP Statements (число секунд перед включением команд SLEEP) автоматически включает в сценарий паузы, превышающие указанное значение в секундах. Обычно вы будете хотеть, чтобы сценарии выполнялись максимально быстро, однако дополнительная пауза поможет скоординировать сценарии.

Сценарии Tester поддерживают тот же формат записи и воспроизведения данных, что и класс .NET System.Windows.Forms.SendKeys, кроме параметра повторения клавиш. Для обработки событий мыши я расширил формат, включив в него команду повторения, а также модификаторы формата, необходимые для использования вместе с ней клавиш Ctrl, Alt и Shift. Формат команд мыши, передаваемых методу TInput.PlayInput, описан в табл. 16-1.

Табл. 16-1. Команды мыши, передаваемые методу TInput.PlayInput

Команда	Использование
MOVETO	{MOVETO x , y}
BTNDOWN	{BTNDOWN btn , x , y}
BTNUP	{BTNUP btn , x , y}
CLICK	{CLICK btn , x , y}
DBLCLICK	{DBLCLICK btn , x , y}
SHIFT DOWN	{SHIFT DOWN}
SHIFT UP	{SHIFT UP}

см. след. стр.

Табл. 16-1. Команды мыши ... *(продолжение)*

Команда	Использование
CTRL DOWN	{CTRL DOWN}
CTRL UP	{CTRL UP}
ALT DOWN	{ALT DOWN}
ALT UP	{ALT UP}
	btn: LEFT, RIGHT, MIDDLE
	x: координата экрана X
	y: координата экрана Y

Есть некоторые функции мыши, которые я не смог реализовать. Во-первых, это обработка колесика. Я использовал ловушку записи журнала для перехвата действий с клавиатурой и мышью и смог получать сообщения о вращении колесика. Но, увы, из-за ошибки в функции-ловушке нет возможности узнать направление вращения колесика. Во-вторых, я не смог реализовать обработку новых клавиш X1 и X2, имеющихся на мыши Microsoft Explorer. Соответствующие сообщения WM_XBUTTONDOWN* содержат данные о нажатой клавише в старшем слове wParam. Так как сообщение WM_MOUSEWHEEL хранит направление вращения колесика там же, но функция-ловушка эту информацию не получает, я сомневаюсь, чтобы в случае кнопки X ситуация чем-нибудь отличалась.

Реализация Tester

Вы уже представляете, как записывать и воспроизводить при помощи Tester сценарии автоматизации тестирования, поэтому я перейду к некоторым более существенным вопросам его реализации. Просуммировав объем исходных и двоичных файлов Tester, включая TESTER.DLL и TESTREC.EXE, вы увидите, что Tester — самая крупная, а кроме того, и самая сложная утилита в этой книге из-за COM, рекурсивного синтаксического разбора и фоновых таймеров.

Уведомления и воспроизведение файлов в TESTER.DLL

В первом издании книги я реализовал TESTER.DLL на Visual Basic 6, потому что в то время язык и среда разработки Visual Basic 6 пользовались при работе с COM наибольшей популярностью. Однако я не хотел требовать от вас установки Visual Basic 6 только для компиляции одной DLL для COM. Прежде всего я решил перенести код TESTER.DLL на платформу .NET. Так как модуль воспроизведения событий клавиатуры был написана на C++, я решил, что будет проще переписать часть Tester, реализованную на Visual Basic 6, на C++, задействовав при этом преимущества новой технологии программирования COM на базе атрибутов (attributed COM programming).

В целом модель COM на базе атрибутов очень удобна, но мне потребовалось некоторое время для обнаружения атрибута `idl_quote`, необходимого для поддержки объявлений интерфейсов. Очень приятным сюрпризом при работе с COM на базе атрибутов оказалась чистота комбинирования языков IDL/ODL и кода C++. Кроме того, благодаря значительно улучшенным мастерам облегчилось добавле-

ние интерфейсов и их методов и свойств. Я хорошо помню, сколько времени ушло на решение проблем с мастерами в предыдущих версиях Visual Studio.

Когда я только задумывал утилиту автоматизированного воспроизведения, я полагал, что могу использовать функцию `SendKeys` языка Visual Basic 6. Однако после тестирования я обнаружил, что такая реализация неудовлетворительна, так как она некорректно посылает события клавиатуры таким программам, как Microsoft Outlook. Это означало, что мне нужно было написать собственную функцию, которая правильно посылала бы события клавиатуры и позволяла в будущем реализовать воспроизведение событий мыши. К счастью, я натолкнулся на функцию `SendInput`, поддерживаемую технологией Microsoft Active Accessibility (MSAA), и заменил ею все предыдущие низкоуровневые функции обработки событий, такие как `keybd_event`. Кроме того, функция `SendInput` помещает всю вводимую информацию в поток ввода клавиатуры или мыши в виде непрерывного блока, гарантируя, что вводимые вами данные не будут перемешаны с посторонней пользовательской информацией. Эта возможность была особенно привлекательной для Tester.

Как только я узнал, как правильно посылать нажатия клавиш, мне нужно было разработать формат их ввода. Функция `SendKeys` языка Visual Basic 6 или класс `System.Windows.Forms.SendKeys` платформы .NET уже предоставляли отличный формат ввода, поэтому я решил воспроизвести его в своей функции `PlayInput`. Я использовал все, кроме кода повтора клавиш, и, как уже говорилось, расширил формат, включив в него поддержку воспроизведения событий мыши. В коде синтаксического разбора нет ничего особо интересного, но если вам захочется его изучить, вы найдете его на CD в файле `Tester\Tester\ParsePlayInputString.CPP`. Если вам захочется увидеть его в действии, можете проработать в отладчике программу `ParsePlayKeysTest` из каталога `Tester\Tester\Tests\ParsePlayKeysTest`. Как можно догадаться по имени программы, это один из блочных тестов для Tester DLL.

Объекты `TWindow`, `TWindows` и `TSystem` просты, и вы сможете разобраться в их работе по исходному коду. Эти три класса являются по сути оболочками для соответствующих API-функций Windows. Единственный более-менее интересный аспект их реализации заключался в написании кода, гарантирующего, что методы `TWindow.SetFocusTWindow` и `TSystem.SetSpecificFocus` могут сделать окно активным. Для этого они до установки фокуса выполняют присоединение к потоку вывода при помощи API-функции `AttachThreadInput`.

С некоторыми интересными проблемами я столкнулся при написании класса `TNotify`. Когда я только начал думать о том, что нужно для определения создания или уничтожения окна с конкретным заголовком, я не ожидал, что создать такой класс будет настолько трудно. Кроме того, я обнаружил, что уведомления о создании окон невозможно сделать надежными, не приложив героических усилий.

Моя первая идея заключалась в реализации системной ловушки для приложений компьютерной профессиональной подготовки [computer-based training (CBT) hook]. В документации SDK подразумевается, что ловушка CBT — лучший метод перехвата событий создания и уничтожения окон. Я быстро написал пример, но вскоре столкнулся с неприятностями. Когда моя ловушка получала уведомление `HCBT_CREATEWND`, я не всегда мог узнать заголовок окна. По непродолжительном раз-

мышлении проблема начала обретать смысл: вероятно, ловушка CBT вызывалась во время обработки сообщения `WM_CREATE`, и очень немногие окна имели в этот момент установленные заголовки. Единственным типом окон, заголовки которых я мог надежно получать при помощи уведомления `NCBT_CREATEWND`, были диалоговые окна. В то же время с уничтожением окон при использовании ловушки CBT все было в порядке.

Просмотрев остальные типы ловушек, я расширил свой пример и попробовал их в действии. Как я и подозревал, простое отслеживание `WM_CREATE` не обеспечивало надежного получения заголовка окна. Один друг предложил мне наблюдать только за сообщениями `WM_SETTEXT`, так как именно его в конечном счете используют для установки заголовка почти все окна. Конечно, если вы рисуете в клиентской области окна или выполняете битовый перенос (*bit blitting*), вы не будете использовать сообщение `WM_SETTEXT`. По ходу дела я заметил одну интересную деталь: некоторые программы, в частности Microsoft Internet Explorer, посылают сообщения `WM_SETTEXT` с одним и тем же текстом много раз подряд.

Поняв, что мне нужно следить за сообщениями `WM_SETTEXT`, я внимательней рассмотрел типы ловушек, которые мог установить. В конце концов наилучшим вариантом оказалась ловушка вызова оконной процедуры (`WH_CALLWNDPROC`). Она позволяет с легкостью наблюдать за сообщениями `WM_CREATE` и `WM_SETTEXT`, а также за сообщениями `WM_DESTROY`. Сначала я полагал, что с `WM_DESTROY` будут некоторые проблемы, так как думал, что заголовок окна может быть уничтожен до получения этого сообщения. К счастью, оказалось, что заголовок окна корректен вплоть до получения сообщения `WM_NCDESTROY`.

Рассмотрев плюсы и минусы обработки сообщений `WM_SETTEXT` только для тех окон, которые еще не имеют заголовка, я решил поступить проще и обрабатывать все сообщения `WM_SETTEXT`. Альтернативным вариантом могло бы быть создание конечного автомата для отслеживания созданных окон и времени установки ими своих заголовков; это решение казалось безошибочным, однако в то же время сложным в реализации. Недостаток обработки всех сообщений `WM_SETTEXT` в том, что вы можете получить много уведомлений о создании одного окна. Например, если вы установите обработчик `TNotify` для окон, содержащих в заголовке слово «Notepad», вы будете получать уведомления не только при запуске `NOTEPAD.EXE`, но и при каждом открытии им нового файла. В итоге я предпочел смириться с не самой лучшей реализацией, но не проводить многие дни за отладкой «правильного» решения. Кроме того, написание ловушки охватывало реализацию итогового класса `TNotify` только на четверть; остальные три четверти были посвящены уведомлению пользователя о создании и уничтожении окон.

Выше я упоминал, что использование объекта `TNotify` связано с некоторыми неудобствами: время от времени вы должны вызывать метод `CheckNotification`. Необходимость периодического вызова `CheckNotification` объясняется тем, что `Tester` поддерживает только модель разделенных потоков, которая не может быть многопоточной; так что мне нужен был механизм проверки создания и уничтожения окон, работающий в том же потоке, что и оставшаяся часть `Tester`.

Рассмотрев некоторые аспекты механизма уведомлений, я ограничил требования следующими базовыми факторами.

- Ловушка `WM_CALLWNDPROC` должна быть системной, поэтому ее необходимо реализовать в ее собственной DLL.
- Очевидно, что DLL приложения `Tester` не подходит на эту роль, так как я не хочу размещать всю эту DLL и, соответственно, весь код COM в каждом адресном пространстве на компьютере пользователя. Это значит, что DLL ловушки, наверное, должна устанавливать что-то вроде флага, который DLL приложения `Tester` могла бы читать, узнавая об удовлетворении нужного условия.
- `Tester` не может быть многопоточным, поэтому мне нужно выполнять всю обработку в одном потоке.

Первое следствие из этих требований заключалось в том, что функцию-ловушку нужно было написать на C. Так как функция-ловушка загружается во все адресные пространства, ее DLL не может вызывать функции из `TESTER.DLL`, написанные при помощи разделенных потоков COM. Поэтому мой код должен был периодически проверять результаты работы ловушки.

Если вы писали программы для 16-разрядных ОС Windows, то знаете, что какая-нибудь фоновая обработка в однопоточной среде с невытесняющей многозадачностью — прекрасная работа для API-функции `SetTimer`. Благодаря `SetTimer` вы можете выполнять фоновую задачу, сохраняя приложение однопоточным. С этой целью я включил в объект `TNotify` уведомление таймера, указывающее на создание или уничтожение интересующих меня окон.

Фоновая обработка при помощи `TNotify` интересна тем, что процедура таймера казалась решением на все случаи жизни, однако на самом деле она обычно работает только при наличии `TNotify`. В зависимости от объема сценария и от того, реализован ли в выбранном вами языке цикл сообщений, сообщение `WM_TIMER` может до вас не добраться, поэтому вам нужно вызывать метод `CheckNotification`, который также проверяет данные ловушки.

Все эти подробности могут казаться запутанными, но вы удивитесь, как мало кода понадобилось для фактической реализации `Tester`. В листинге 16-3 показан код функции-ловушки из файла `TNOTIFYHLP.CPP`. С точки зрения `Tester`, файл `TNOTIFY.CPP` — это модуль, в котором находится процедура таймера и код COM, необходимый для объекта. Класс `TNotify` имеет несколько методов C++, которые объект `TNotify` может использовать для возбуждения событий и определения интересующих пользователя типов уведомлений. Один из интересных фрагментов кода функции-ловушки — глобально разделяемый сегмент данных, `.HOOKDATA`, содержащий массив данных об уведомлениях. При изучении кода помните, что данные об уведомлениях глобальны, в то время как все остальные данные уникальны для каждого процесса.

Листинг 16-3. TNOTIFYHLP.CPP

```
/*-----
Отладка приложений для Microsoft .NET и Microsoft Windows
Copyright © 1997-2003 John Robbins - All rights reserved.
-----*/
#include "stdafx.h"

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

см. след. стр.

```

        Определения и константы с областью видимости файла
        //////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
// Максимальное число слотов уведомлений
static const int TOTAL_NOTIFY_SLOTS = 5 ;
// Имя мьютекса
static const LPCTSTR k_MUTEX_NAME  = _T ( "TNotifyHlp_Mutex" ) ;
// Наибольшее время ожидания мьютекса
static const int k_WAITLIMIT = 5000 ;

// Здесь я определяю свою директиву TRACE, потому что не хочу
// размещать BugslayerUtil.DLL в каждом адресном пространстве.
#ifdef _DEBUG
#define TRACE    ::OutputDebugString
#else
#define TRACE    __noop
#endif

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
// Объявления typedef с областью видимости файла
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
// Структура данных интересующего нас окна.
typedef struct tag_TNOTIFYITEM
{
    // PID процесса, создавшего структуру
    DWORD   dwOwnerPID ;
    // Тип уведомления
    int     iNotifyType ;
    // Параметр сравнения заголовка
    int     iSearchType ;
    // Описатель создаваемого окна
    HWND    hWndCreate ;
    // Флаг, указывающий на уничтожение окна
    BOOL    bDestroy    ;
    // Строка заголовка
    TCHAR   szTitle [ MAX_PATH ] ;
} TNOTIFYITEM , * PTNOTIFYITEM ;

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
// Глобальные переменные с областью видимости файла
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
// Эти данные **НЕ** являются общими для процессов,
// поэтому каждый процесс получает собственную их копию.
// HINSTANCE этого модуля. Установка глобальных системных
// ловушек должна выполняться при помощи DLL.
static HINSTANCE g_hInst = NULL ;

// Мьютекс, защищающий таблицу g_NotifyData
static HANDLE g_hMutex = NULL ;

// Описатель ловушки. Я не включил его в раздел общих

```

```

// данных, потому что при выполнении нескольких сценариев
// процессы могут устанавливать собственные ловушки.
static HHOOK g_hHook = NULL ;

// Число элементов, добавленных в таблицу этим процессом. Это
// число нужно для того, чтобы я знал, как обрабатывать ловушку.
static int g_iThisProcessItems = 0 ;

/*//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Прототипы функций с областью видимости файла
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
// Наша функция-ловушка
LRESULT CALLBACK CallWndRetProcHook ( int nCode ,
                                       WPARAM wParam ,
                                       LPARAM lParam ) ;

// Внутренняя функция проверки
static LONG_PTR __stdcall CheckNotifyItem ( HANDLE hItem , BOOL bCreate ) ;

/*//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Данные, общие для всех экземпляров ловушек
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/
#pragma data_seg ( ".HOOKDATA" )
// Таблица уведомлений
static TNOTIFYITEM g_shared_NotifyData [ TOTAL_NOTIFY_SLOTS ] =
{
    { 0 , 0 , 0 , NULL , 0 , '\0' } ,
    { 0 , 0 , 0 , NULL , 0 , '\0' } ,
    { 0 , 0 , 0 , NULL , 0 , '\0' } ,
    { 0 , 0 , 0 , NULL , 0 , '\0' } ,
    { 0 , 0 , 0 , NULL , 0 , '\0' }
} ;
// Счетчик использованных слотов уведомлений
static int g_shared_iUsedSlots = 0 ;
#pragma data_seg ( )

/*//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// ЗДЕСЬ НАЧИНАЕТСЯ ВНЕШНЯЯ РЕАЛИЗАЦИЯ
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*/

extern "C" BOOL WINAPI DllMain ( HINSTANCE hInst ,
                                DWORD dwReason ,
                                LPVOID /*lpReserved*/ )
{
#ifdef _DEBUG
    BOOL bCHRet ;
#endif

    BOOL bRet = TRUE ;
    switch ( dwReason )

```

см. след. стр.

```

{
    case DLL_PROCESS_ATTACH :
        // Присвоение значения глобальному описателю модуля.
        g_hInst = hInst ;
        // Мне не нужны уведомления, связанные с потоками.
        DisableThreadLibraryCalls ( g_hInst ) ;
        // Создание мьютекса для данного процесса. Мьютекс
        // создается, но его получение пока не выполняется.
        g_hMutex = CreateMutex ( NULL , FALSE , k_MUTEX_NAME ) ;
        if ( NULL == g_hMutex )
        {
            TRACE ( _T ( "Unable to create the mutex!\n" ) ) ;
            // Если я не могу создать мьютекс, продолжение
            // невозможно, и загрузка DLL завершилась неудачей.
            bRet = FALSE ;
        }
        break ;
    case DLL_PROCESS_DETACH :

        // Имеет ли этот процесс какие-нибудь элементы
        // в массиве уведомлений? Если да, я их удаляю,
        // чтобы не оставлять осиротевшие элементы.
        if ( 0 != g_iThisProcessItems )
        {
            DWORD dwProcID = GetCurrentProcessId ( ) ;
            // В этом случае мне не нужно получать мьютекс,
            // потому что только один поток может вызывать
            // DLL по причине DLL_PROCESS_DETACH.

            // Нахождение элементов, относящихся к этому процессу.
            for ( INT_PTR i = 0 ; i < TOTAL_NOTIFY_SLOTS ; i++ )
            {
                if ( g_shared_NotifyData[i].dwOwnerPID == dwProcID )
                {
#ifdef _DEBUG
                    TCHAR szBuff[ 50 ] ;
                    wsprintf ( szBuff ,
                        _T("DLL_PROCESS_DETACH removing : #d\n"),
                        i ) ;
                    TRACE ( szBuff ) ;
#endif
                    // И их удаление.
                    RemoveNotifyTitle ( (HANDLE)i ) ;
                }
            }
        }

        // Закрытие описателя мьютекса.
#ifdef _DEBUG
        bCHRet =

```

```

#endif
        CloseHandle ( g_hMutex );
#ifdef _DEBUG
        if ( FALSE == bCHRet )
        {
            TRACE ( _T ( "!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n" ) );
            TRACE ( _T ( "CloseHandle(g_hMutex) " ) \
                _T ( "failed!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n" ) );
            TRACE ( _T ( "!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n" ) );
        }
#endif
        break ;
    default :
        break ;
}
return ( bRet ) ;
}

HANDLE TNOTIFYHLP_DLLINTERFACE __stdcall
AddNotifyTitle ( int      iNotifyType ,
                 int      iSearchType ,
                 LPCTSTR szString )
{
    // Проверка корректности типа уведомления.
    if ( ( iNotifyType < ANTN_DESTROYWINDOW ) ||
        ( iNotifyType > ANTN_CREATEANDDESTROY ) )
    {
        TRACE (
            _T( "AddNotify Title : iNotifyType is out of range!\n" ) );
        return ( INVALID_HANDLE_VALUE );
    }
    // Проверка корректности метода сравнения заголовка.
    if ( ( iSearchType < ANTS_EXACTMATCH ) ||
        ( iSearchType > ANTS_ANYLOCMATCH ) )
    {
        TRACE (
            _T( "AddNotify Title : iSearchType is out of range!\n" ) );
        return ( INVALID_HANDLE_VALUE );
    }
    // Проверка корректности заголовка.
    if ( TRUE == IsBadStringPtr ( szString , MAX_PATH ) )
    {
        TRACE ( _T( "AddNotify Title : szString is invalid!\n" ) );
        return ( INVALID_HANDLE_VALUE );
    }

    // Ожидание получения мьютекса.
    DWORD dwRet = WaitForSingleObject ( g_hMutex , k_WAITLIMIT );
    if ( WAIT_TIMEOUT == dwRet )

```

см. след. стр.

```

{
    TRACE ( _T( "AddNotifyTitle : Wait on mutex timed out!!\n" ));
    return ( INVALID_HANDLE_VALUE );
}

// Если все слоты использованы, выполняется выход.
if ( TOTAL_NOTIFY_SLOTS == g_shared_iUsedSlots )
{
    ReleaseMutex ( g_hMutex );
    return ( INVALID_HANDLE_VALUE );
}

// Нахождение первого свободного слота.
for ( INT_PTR i = 0 ; i < TOTAL_NOTIFY_SLOTS ; i++ )
{
    if ( _T ( '\0' ) == g_shared_NotifyData[ i ].szTitle[ 0 ] )
    {
        break ;
    }
}

// Добавление данных.
g_shared_NotifyData[ i ].dwOwnerPID = GetCurrentProcessId ( ) ;
g_shared_NotifyData[ i ].iNotifyType = iNotifyType ;
g_shared_NotifyData[ i ].iSearchType = iSearchType ;
lstrcpy ( g_shared_NotifyData[ i ].szTitle , szString );

// Увеличение счетчика использованных слотов.
g_shared_iUsedSlots++ ;

// Увеличение счетчика элементов этого процесса.
g_iThisProcessItems++ ;

TRACE ( _T( "AddNotifyTitle - Added a new item!\n" ) ) ;

ReleaseMutex ( g_hMutex );

// Если это первый запрос об уведомлениях, устанавливается ловушка.
if ( NULL == g_hHook )
{
    g_hHook = SetWindowsHookEx ( WH_CALLWNDPROCRET ,
                                CallWndRetProcHook ,
                                g_hInst ,
                                0 ) ;
}

#ifdef _DEBUG
if ( NULL == g_hHook )
{
    TCHAR szBuff[ 50 ] ;
    wsprintf ( szBuff ,
               _T ( "SetWindowsHookEx failed!!!! (0x%08X)\n" ),

```

```

        GetLastError ( ) ) ;
        TRACE ( szBuff ) ;
    }
#endif
}

return ( (HANDLE)i ) ;

}

void TNOTIFYHLP_DLLINTERFACE __stdcall
RemoveNotifyTitle ( HANDLE hItem )
{
    // Проверка описателя.
    INT_PTR i = (INT_PTR)hItem ;
    if ( ( i < 0 ) || ( i > TOTAL_NOTIFY_SLOTS ) )
    {
        TRACE ( _T ( "RemoveNotifyTitle : Invalid handle!\n" ) ) ;
        return ;
    }

    // Получение мьютекса.
    DWORD dwRet = WaitForSingleObject ( g_hMutex , k_WAITLIMIT ) ;
    if ( WAIT_TIMEOUT == dwRet )
    {
        TRACE ( _T ( "RemoveNotifyTitle : Wait on mutex timed out!\n"));
        return ;
    }

    if ( 0 == g_shared_iUsedSlots )
    {
        TRACE ( _T ( "RemoveNotifyTitle : Attempting to remove when " ) \
            _T ( "no notification handles are set!\n" ) ) ;
        ReleaseMutex ( g_hMutex ) ;
        return ;
    }

    // Перед удалением чего-либо нужно убедиться в том, что индекс
    // указывает на элемент NotifyData, имеющий корректное значение.
    // Если бы я этого не делал, неоднократный вызов данной функции
    // с одним и тем же параметром приводил бы к сбою счетчика
    // использованных слотов.
    if ( 0 == g_shared_NotifyData[ i ].dwOwnerPID )
    {
        TRACE ( _T ( "RemoveNotifyTitle : ") \
            _T ( "Attempting to double remove!\n" ) ) ;
        ReleaseMutex ( g_hMutex ) ;
        return ;
    }
}

```

```

// Удаление элемента из массива.
g_shared_NotifyData[ i ].dwOwnerPID   = 0 ;
g_shared_NotifyData[ i ].iNotifyType  = 0 ;
g_shared_NotifyData[ i ].hWndCreate   = NULL ;
g_shared_NotifyData[ i ].bDestroy     = FALSE ;
g_shared_NotifyData[ i ].iSearchType  = 0 ;
g_shared_NotifyData[ i ].szTitle[ 0 ] = _T ( '\0' ) ;

// Уменьшение счетчика использованных слотов.
g_shared_iUsedSlots- ;

// Уменьшение счетчика элементов данного процесса.
g_iThisProcessItems- ;

TRACE ( _T ( "RemoveNotifyTitle - Removed an item!\n" ) ) ;

ReleaseMutex ( g_hMutex ) ;

// Если это последний элемент данного
// процесса, ловушка процесса удаляется.
if ( ( 0 == g_iThisProcessItems ) && ( NULL != g_hHook ) )
{
    if ( FALSE == UnhookWindowsHookEx ( g_hHook ) )
    {
        TRACE ( _T ( "UnhookWindowsHookEx failed!\n" ) ) ;
    }
    g_hHook = NULL ;
}

}

HWND TNOTIFYHLP_DLLINTERFACE __stdcall
CheckNotifyCreateTitle ( HANDLE hItem )
{
    return ( (HWND)CheckNotifyItem ( hItem , TRUE ) ) ;
}

BOOL TNOTIFYHLP_DLLINTERFACE __stdcall
CheckNotifyDestroyTitle ( HANDLE hItem )
{
    return ( (BOOL)CheckNotifyItem ( hItem , FALSE ) ) ;
}

/*//////////////////////////////////////
// ЗДЕСЬ НАЧИНАЕТСЯ ВНУТРЕННЯЯ РЕАЛИЗАЦИЯ
//////////////////////////////////////*/

static LONG_PTR __stdcall CheckNotifyItem ( HANDLE hItem , BOOL bCreate )
{
    // Проверка описателя.

```



```
INT_PTR i = (INT_PTR)hItem ;
if ( ( i < 0 ) || ( i > TOTAL_NOTIFY_SLOTS ) )
{
    TRACE ( _T ( "CheckNotifyItem : Invalid handle!\n" ) );
    return ( NULL );
}

LONG_PTR lRet = 0 ;

// Получение мьютекса.
DWORD dwRet = WaitForSingleObject ( g_hMutex , k_WAITLIMIT ) ;
if ( WAIT_TIMEOUT == dwRet )
{
    TRACE ( _T ( "CheckNotifyItem : Wait on mutex timed out!\n" ) );
    return ( NULL );
}

// Если все слоты пусты, делать нечего.
if ( 0 == g_shared_iUsedSlots )
{
    ReleaseMutex ( g_hMutex ) ;
    return ( NULL );
}

// Проверка запрошенного элемента.
if ( TRUE == bCreate )
{
    // Если HWND не равен NULL, выполняется возврат
    // его значения и его обнуление в таблице.
    if ( NULL != g_shared_NotifyData[ i ].hWndCreate )
    {
        lRet = (LONG_PTR)g_shared_NotifyData[ i ].hWndCreate ;
        g_shared_NotifyData[ i ].hWndCreate = NULL ;
    }
}
else
{
    if ( FALSE != g_shared_NotifyData[ i ].bDestroy )
    {
        lRet = TRUE ;
        g_shared_NotifyData[ i ].bDestroy = FALSE ;
    }
}

ReleaseMutex ( g_hMutex ) ;

return ( lRet ) ;
}

static void __stdcall CheckTableMatch ( int    iNotifyType ,
```

см. след. стр.

```

                                HWND    hWnd    ,
                                LPCTSTR szTitle )
{
    // Получение мьютекса.
    DWORD dwRet = WaitForSingleObject ( g_hMutex , k_WAITLIMIT ) ;
    if ( WAIT_TIMEOUT == dwRet )
    {
        TRACE ( _T ( "CheckTableMatch : Wait on mutex timed out!\n" ) ) ;
        return ;
    }

    // Таблица не должна быть пустой, но никогда
    // нельзя быть ни в чем уверенным.
    if ( 0 == g_shared_iUsedSlots )
    {
        ReleaseMutex ( g_hMutex ) ;
        TRACE ( _T ( "CheckTableMatch called on an empty table!\n" ) ) ;
        return ;
    }

    // Просмотр элементов таблицы.
    for ( int i = 0 ; i < TOTAL_NOTIFY_SLOTS ; i++ )
    {
        // Не пуст ли этот элемент и совпадают ли типы уведомлений?
        if ( ( _T ( '\0' ) != g_shared_NotifyData[ i ].szTitle[ 0 ] ) &&
            ( g_shared_NotifyData[ i ].iNotifyType & iNotifyType ) )
        {
            BOOL bMatch = FALSE ;
            // Сопоставление заголовка окна
            // с аналогичным полем элемента таблицы.
            switch ( g_shared_NotifyData[ i ].iSearchType )
            {
                case ANTS_EXACTMATCH :
                    // Это просто.
                    if ( 0 == lstrcmp ( g_shared_NotifyData[i].szTitle ,
                                        szTitle ) )
                    {
                        bMatch = TRUE ;
                    }
                    break ;
                case ANTS_BEGINMATCH :
                    if ( 0 ==
                        _tcsncmp ( g_shared_NotifyData[i].szTitle ,
                                szTitle ,
                                _tcslen(g_shared_NotifyData[i].szTitle)))
                    {
                        bMatch = TRUE ;
                    }
                    break ;
            }
        }
    }
}

```

```

        case ANTS_ANYLOCMATCH :
            if ( NULL != _tcsstr ( szTitle
                                   ,
                                   g_shared_NotifyData[i].szTitle ))
            {
                bMatch = TRUE ;
            }
            break ;
        default :
            TRACE ( _T ( "CheckTableMatch invalid " ) \
                    _T ( "search type!!!\n" ) ) ;
            ReleaseMutex ( g_hMutex ) ;
            return ;
            break ;
    }
    // Ну, и каковы результаты?
    if ( TRUE == bMatch )
    {
        // Если это уведомление об уничтожении окна,
        // соответствующее поле получает значение "1".
        if ( ANTN_DESTROYWINDOW == iNotifyType )
        {
            g_shared_NotifyData[ i ].bDestroy = TRUE ;
        }
        else
        {
            // Иначе устанавливается значение HWND.
            g_shared_NotifyData[ i ].hWndCreate = hWnd ;
        }
    }
}

ReleaseMutex ( g_hMutex ) ;
}

LRESULT CALLBACK CallWndRetProcHook ( int    nCode ,
                                       WPARAM wParam ,
                                       LPARAM lParam )
{
    // Буфер для хранения заголовка окна
    TCHAR szBuff[ MAX_PATH ] ;

    // Прежде чем что-либо сделать, я всегда передаю сообщение
    // следующей ловушке, чтобы не забыть сделать это потом.
    // Передав сообщение, я могу спокойно заняться своими делами.
    LRESULT lRet = CallNextHookEx ( g_hHook , nCode , wParam , lParam ) ;

    // В документации запрещается обрабатывать сообщение при
    // отрицательных значениях параметра nCode. Не будем спорить.
    if ( nCode < 0 )
    {

```

см. след. стр.

```

        return ( lRet );
    }

    // Получение структуры сообщения. Интересно, зачем нужны три
    // (или больше) различных структуры сообщений? Почему нельзя
    // было использовать структуру MSG для всех сообщений/ловушек?
    PCWPRETSTRUCT pMsg = (PCWPRETSTRUCT)lParam;

    // Нет заголовка - нет работы.
    LONG lStyle = GetWindowLong ( pMsg->hwnd , GWL_STYLE );
    if ( WS_CAPTION != ( lStyle & WS_CAPTION ) )
    {
        return ( lRet );
    }

    // Сообщения WM_DESTROY используются и диалоговыми, и обычными
    // окнами. Нужно просто получить заголовок окна и проверить
    // наличие соответствующего элемента в таблице уведомлений.
    if ( WM_DESTROY == pMsg->message )
    {
        if ( 0 != GetWindowText ( pMsg->hwnd , szBuff , MAX_PATH ) )
        {
            CheckTableMatch ( ANTN_DESTROYWINDOW , pMsg->hwnd , szBuff );
        }
        return ( lRet );
    }

    // С созданием окна все не так просто, как с его уничтожением.

    // Получение класса окна. Если это подлинное диалоговое
    // окно, мне нужно только сообщение WM_INITDIALOG.
    if ( 0 == GetClassName ( pMsg->hwnd , szBuff , MAX_PATH ) )
    {
#ifdef _DEBUG
        TCHAR szBuff[ 50 ];
        wsprintf ( szBuff
                    , _T ( "GetClassName failed for HWND : 0x%08X\n" ) ,
                    pMsg->hwnd
                    );
        TRACE ( szBuff );
#endif
        // Продолжение не имеет смысла.
        return ( lRet );
    }
    if ( 0 == lstrcmpi ( szBuff , _T ( "#32770" ) ) )
    {
        // Мне нужно проверять только сообщение WM_INITDIALOG.
        if ( WM_INITDIALOG == pMsg->message )
        {
            // Получение заголовка диалогового окна.
            if ( 0 != GetWindowText ( pMsg->hwnd , szBuff , MAX_PATH ) )

```

```

        {
            CheckTableMatch ( ANTN_CREATEWINDOW ,
                             pMsg->hwnd      ,
                             szBuff          ) ;
        }
    }
    return ( lRet ) ;
}

// Я разобрался с диалоговыми окнами. Теперь
// мне нужно позаботиться о других окнах.

if ( WM_CREATE == pMsg->message )
{
    // Очень немногие окна устанавливают заголовок
    // при обработке сообщения WM_CREATE. Однако некоторые
    // поступают именно так, и они не используют WM_SETTEXT,
    // поэтому я должен выполнить соответствующую проверку.
    if ( 0 != GetWindowText ( pMsg->hwnd , szBuff , MAX_PATH ) )
    {
        CheckTableMatch ( ANTN_CREATEWINDOW ,
                         pMsg->hwnd      ,
                         szBuff          ) ;
    }
}
else if ( WM_SETTEXT == pMsg->message )
{
    // Я всегда обрабатываю WM_SETTEXT, поскольку именно так
    // программы устанавливают заголовки. К сожалению, похоже,
    // некоторые приложения, такие как Internet Explorer, вызывают
    // WM_SETTEXT несколько раз с одним заголовком. Чтобы не усложнять
    // эту функцию, я просто сообщаю WM_SETTEXT вместо поддержки
    // странных, тяжелых в отладке структур данных, необходимых
    // для слежения за окнами, которые уже вызывали WM_SETTEXT раньше.
    if ( NULL != pMsg->lParam )
    {
        CheckTableMatch ( ANTN_CREATEWINDOW      ,
                         pMsg->hwnd              ,
                         (LPCTSTR)pMsg->lParam ) ;
    }
}

return ( lRet ) ;
}

```

Некоторые аспекты реализации TNotify казались довольно сложными, поэтому я был приятно удивлен тем, как мало проблем я испытал на самом деле. Если вы хотите усовершенствовать код функции-ловушки, знайте, что отлаживать системные ловушки очень непросто. Для этого лучше всего использовать удаленную отладку (см. главу 5). Еще один способ отладки системных ловушек заключается в отладке в стиле printf. Программа DebugView, которую можно загрузить с сайта

www.sysinternals.com, позволит вам видеть все вызовы `OutputDebugString`, указывающие на состояние вашей ловушки.

Реализация TESTREC.EXE

После создания `Tester DLL` мне нужно было разработать программу `TESTREC.EXE`, которая записывала бы события клавиатуры и мыши. Если дело касается записи вводимой информации, в ОС Windows, есть только один чистый способ сделать это: использовать ловушку записи журнала. В ловушках записи журнала нет ничего увлекательного, кроме проблемы правильной обработки сообщения `WM_CAN-`

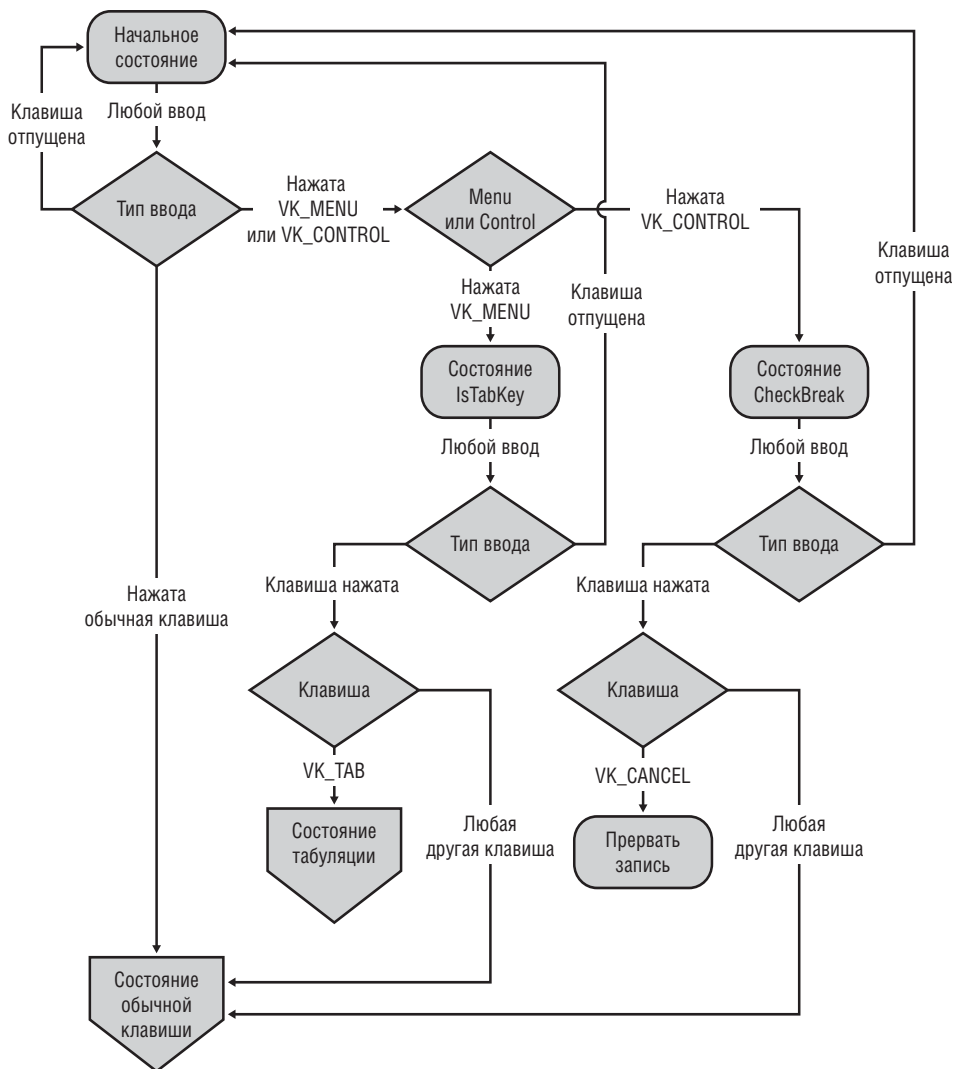


Рис. 16-2. Конечные автоматы начала записи событий клавиатуры, обработки клавиши `Tab` и комбинации `Ctrl+Break`

CELJOURNAL. Когда пользователь нажимает Ctrl+Alt+Delete, ОС завершает все активные ловушки записи журнала. Это очень грамотное решение, так как возможность записи нажатий клавиш при вводе пароля открывала бы огромную брешь в системе безопасности. Чтобы скрыть детали реализации обработки WM_CANCELJOURNAL, я написал фильтр, отслеживающий это сообщение. Все подробности работы функции-ловушки вы можете увидеть в файле HOOKCODE.CPP, находящемся в каталоге Tester\TestRec.

Обработка ввода с клавиатуры

Запись событий клавиатуры сводится главным образом к правильной обработке нажатий клавиш Shift, Ctrl и Alt. Прежде чем я опишу некоторые аспекты борьбы с нажатиями отдельных клавиш, изучите рисунки 16-2 — 16-4, на которых представлен упрощенный граф всех состояний клавиатуры, обрабатываемых кодом записи сценария.

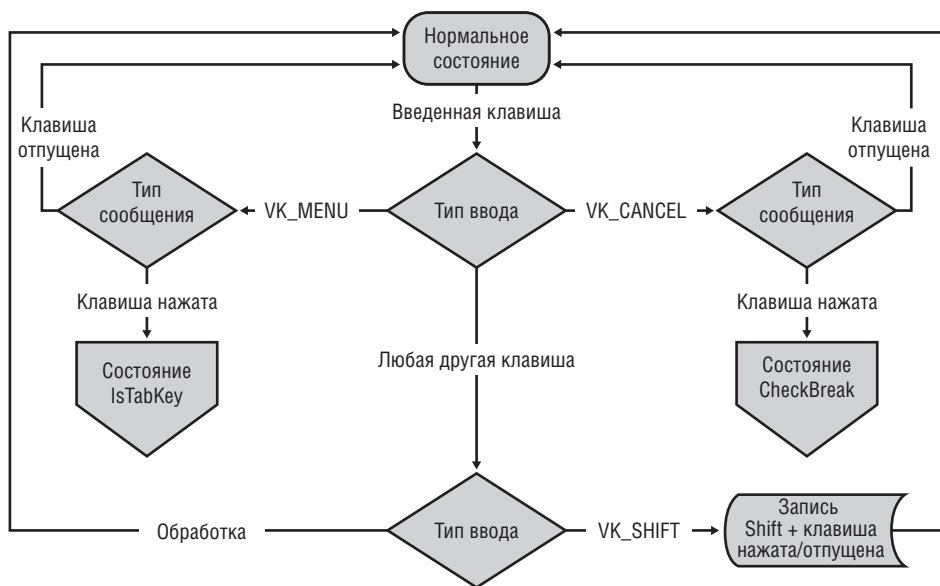


Рис. 16-3. Конечный автомат нормальной обработки событий клавиатуры

Первая проблема записи событий клавиатуры заключается в получении их из функции-ловушки в понятной человеку форме. Если вы никогда не испытывали радость работы с виртуальными и скан-кодами, вы не знаете, что такое настоящие трудности! Кроме того, я обнаружил, что некоторые данные, получаемые ловушкой записи журнала довольно сильно отличались от того, что я ожидал.

Последний раз я работал с клавиатурой на этом уровне во времена MS-DOS (похоже, я выдал свой возраст!). Поэтому я внес в проблему некоторые дополнительные недоразумения. Например, когда я впервые ввел восклицательный знак, я ожидал увидеть, что именно этот символ и поступит в функцию-ловушку. Однако вместо этого я получил символ Shift, за которым следовала единица. Именно так восклицательный знак вводится с клавиатур US English. Однако я хотел, чтобы все воспроизводимые мной последовательности нажатых клавиш были пре-

дельно понятны. Последовательность SendKeys «+1» с технической точки зрения верна, но при этом нужно проделать некоторую умственную гимнастику, чтобы понять, что на самом деле это символ «!».

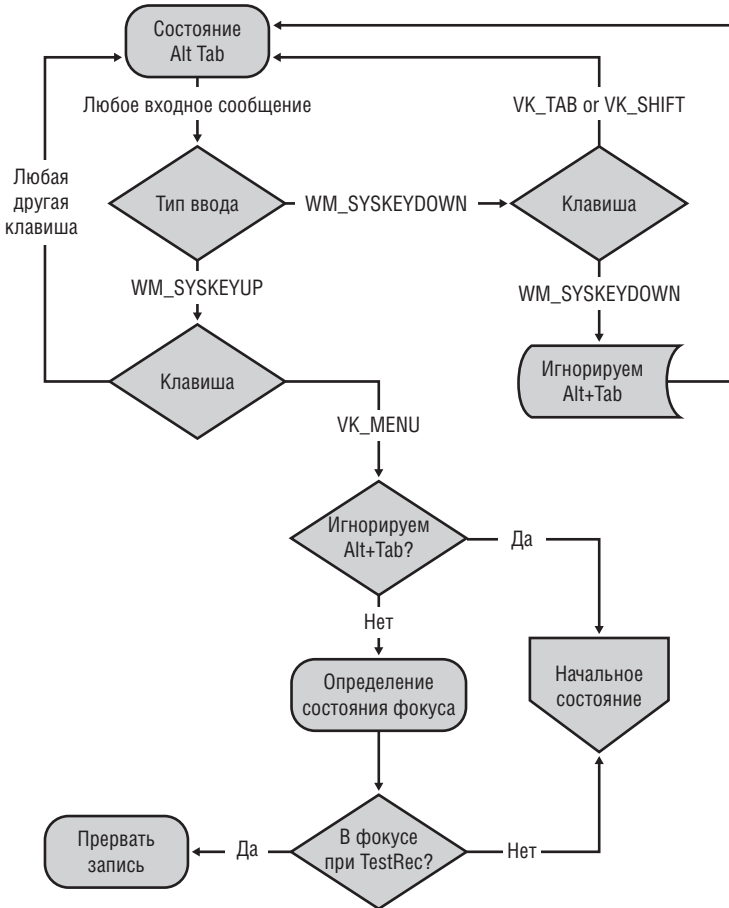


Рис. 16-4. Конечный автомат обработки комбинации Alt+Tab

Чтобы программа TESTREC.EXE была максимально полезна, я должен был реализовать некоторый специальный механизм обработки, который позволил бы сделать выводимые строки понятными. Проще говоря, я должен был проверить состояние клавиатуры, узнать, нажата ли клавиша Shift, и, если да, преобразовать символ в понятную форму. К счастью, для получения действительного символа у нас есть API-функции GetKeyboardState и ToUnicode. Чтобы понять суть обработки нажатий клавиш, изучите функцию CRecordingEngine::NormalKeyState из листинга 16-4.

Листинг 16-4. CRecordingEngine::NormalKeyState

```

void CRecordingEngine::NormalKeyState ( PEVENTMSG pMsg )
{
    // Состояние, в которое будет выполнен переход

```



```

// после обработки поступившего сообщения.
eKeyStates eShiftToState = eNormalKey ;

UINT vkCode = LOBYTE ( pMsg->paramL ) ;

#ifdef _DEBUG
{
    STATETRACE ( _T("RECSTATE: Normal : ") ) ;
    if ( ( WM_KEYDOWN == pMsg->message ) ||
          ( WM_SYSKEYDOWN == pMsg->message ) )
    {
        STATETRACE ( _T( "KEYDOWN " ) ) ;
    }
    else
    {
        STATETRACE ( _T( "KEYUP " ) ) ;
    }
    TCHAR szName [ 100 ] ;
    GetKeyNameText ( pMsg->paramH << 16 , szName , 100 ) ;
    CharUpper ( szName ) ;
    STATETRACE ( _T( " %c %d %04X (%s)\n" ) ,
                  vkCode
                  ,
                  vkCode
                  ,
                  vkCode
                  ,
                  szName
                  ) ;
}
#endif

// Проверка того, что нажатая клавиша
// не вызовет прекращения записи сообщений.
switch ( vkCode )
{
    case VK_CONTROL :
        // Нажатие CTRL может произойти при нажатой клавише ALT.
        if ( ( WM_KEYDOWN == pMsg->message ) ||
              ( WM_SYSKEYDOWN == pMsg->message ) )
        {
            eShiftToState = eCheckBreak ;
            STATETRACE ( _T( "RECSTATE: Looking for BREAK key\n") );
        }
        else
        {
            m_cKeyBuff += _T( "{CTRL UP}" ) ;
            m_iKeyBuffKeys++ ;
        }
        m_iKeyBuffKeys++ ;
        break ;
    case VK_MENU :
        if ( ( WM_KEYDOWN == pMsg->message ) ||

```

см. след. стр.

```

        ( WM_SYSKEYDOWN == pMsg->message ) )
    {
        eShiftToState = eIsTabKey ;
        STATETRACE ( _T("RECSTATE: Looking for TAB key\n") ) ;
    }
    else
    {
        m_cKeyBuff += _T( "{ALT UP}" ) ;
        m_iKeyBuffKeys++ ;
    }
    m_iKeyBuffKeys++ ;
    break ;

case VK_SHIFT :
    if ( ( WM_KEYDOWN == pMsg->message ) ||
        ( WM_SYSKEYDOWN == pMsg->message ) )
    {
        // При нажатии SHIFT этот блок выполняется только один раз!
        if ( FALSE == m_bShiftDown )
        {
            // Нажата клавиша SHIFT, выполняется установка флагов.
            m_bShiftDown = TRUE ;
            m_bShiftDownInString = FALSE ;
        }
    }
    else
    {
        // Если раньше я записал {SHIFT DOWN},
        // мне нужно сопоставить его с {SHIFT UP}.
        if ( TRUE == m_bShiftDownInString )
        {
            m_cKeyBuff += _T( "{SHIFT UP}" ) ;
            m_iKeyBuffKeys++ ;

            m_bShiftDownInString = FALSE ;
        }
        // Клавиша SHIFT отпущена.
        m_bShiftDown = FALSE ;
    }
    break ;
default :
    // Это обычная клавиша.

    // Если это сообщение не о нажатии, я ничего не делаю.
    if ( ( WM_KEYDOWN == pMsg->message ) ||
        ( WM_SYSKEYDOWN == pMsg->message ) )
    {
        //TRACE ( "vkCode = %04X\n" , vkCode ) ;

        // Есть ли строка, соответствующая этому виртуальному коду?

```

```

if ( NULL != g_KeyStrings[ vkCode ].szString )
{
    // Нажата ли клавиша SHIFT? Если да,
    // перед записью обрабатываемой клавиши
    // мне нужно записать {SHIFT DOWN}.
    if ( ( TRUE == m_bShiftDown          ) &&
        ( FALSE == m_bShiftDownInString ) )
    {
        m_cKeyBuff += _T ( "{SHIFT DOWN}" );
        m_iKeyBuffKeys++;
        m_bShiftDownInString = TRUE ;
    }
    // Добавление клавиши в список.
    m_cKeyBuff += g_KeyStrings[ vkCode ].szString ;
}
else
{
    // Я должен преобразовать нажатую клавишу в ее
    // символьный эквивалент. Для правильного
    // преобразования таких последовательностей, как
    // "{SHIFT DOWN}1{SHIFT UP}" в "!", мне нужно
    // получить состояние клавиатуры и вызвать
    // функцию ToAscii.

    // Сначала нужно получить состояние клавиатуры.
    BYTE byState [ 256 ] ;

    GetKeyboardState ( byState ) ;

    // А теперь выполнить преобразование.
    TCHAR cConv[3] = { _T ( '\0' ) } ;
    TCHAR cChar ;

#ifdef _UNICODE
    int iRet = ToUnicode ( vkCode
                          ,
                          pMsg->paramH
                          ,
                          byState
                          ,
                          (LPWORD)&cConv
                          ,
                          sizeof ( cConv ) /
                          sizeof ( TCHAR )
                          ,
                          0
                          ) ;

#else
    int iRet = ToAscii ( vkCode
                       ,
                       pMsg->paramH
                       ,
                       byState
                       ,
                       (LPWORD)&cConv
                       ,
                       0
                       ) ;

#endif

    if ( 2 == iRet )
    {

```

см. след. стр.

```

        // Это национальный символ.
        ASSERT ( !"I gotta handle this!" );
    }

    // Если символ не был преобразован,
    // cChar не используется!
    if ( 0 == iRet )
    {
        cChar = (TCHAR)vkCode ;
    }
    else
    {
        // Прежде чем записать символ, мне нужно узнать,
        // нажата ли клавиша CTRL. Если да, то функции
        // ToAscii/ToUnicode возвращают управляющий
        // код ASCII. Так как мне нужен символ, я должен
        // выполнить некоторую дополнительную работу.
        SHORT sCtrlDown =
            GetAsyncKeyState ( VK_CONTROL ) ;
        if ( 0xFFFF8000 == ( 0xFFFF8000 & sCtrlDown ) )
        {
            // Клавиша CTRL нажата, поэтому мне нужно
            // узнать состояние клавиш CAPSLOCK и SHIFT.
            BOOL bCapsLock =
                ( 0xFFFF8000 == ( 0xFFFF8000 &
                    GetAsyncKeyState ( VK_CAPITAL))) ;
            if ( TRUE == bCapsLock )
            {
                // Если нажаты клавиши CAPSLOCK и SHIFT,
                // используем символ нижнего регистра.
                if ( TRUE == m_bShiftDown )
                {
                    // Запрещение предупреждения 'variable' : conversion from 'type' to 'type'
                    // of greater size (преобразование к типу, имеющему больший размер).
                    #pragma warning ( disable : 4312 )
                    cChar = (TCHAR)
                        CharLower ( (LPTSTR)vkCode );
                    #pragma warning ( default : 4312 )
                }
                else
                {
                    // Символ верхнего регистра.
                    cChar = (TCHAR)vkCode ;
                }
            }
            else
            {
                // Клавиша CAPSLOCK не нажата,
                // поэтому проверяется только
                // клавиша SHIFT.

```

```

        if ( TRUE == m_bShiftDown )
        {
            cChar = (TCHAR)vkCode ;
        }
        else
        {
// Запрещение предупреждения 'variable' : conversion from 'type' to 'type'
// of greater size (преобразование к типу, имеющему больший размер).
#pragma warning ( disable : 4312 )
            cChar = (TCHAR)
                CharLower ( (LPTSTR)vkCode );
#pragma warning ( default : 4312 )
        }
    }
    else
    {
        // Клавиша CTRL не нажата, поэтому я могу
        // сразу использовать преобразованную клавишу.
        cChar = cConv[ 0 ] ;
    }
}

switch ( cChar )
{
    // Квадратные и фигурные скобки и тильды
    // требуют особой обработки. Все остальные
    // клавиши просто добавляются в буфер вывода.
    case _T ( '[' ) :
        m_cKeyBuff += _T ( "{[" ) ;
        break ;
    case _T ( ']' ) :
        m_cKeyBuff += _T ( "}" ) ;
        break ;
    case _T ( '~' ) :
        m_cKeyBuff += _T ( "~" ) ;
        break ;
    case _T ( '{' ) :
        m_cKeyBuff += _T ( "{" ) ;
        break ;
    case _T ( '}' ) :
        m_cKeyBuff += _T ( "}" ) ;
        break ;
    default :
        m_cKeyBuff += cChar ;
}
}
// Увеличение числа обработанных клавиш.
m_iKeyBuffKeys++ ;

```

```

        if ( ( m_iKeyBuffKeys > 20          ) ||
            ( m_cKeyBuff.Length ( ) > 50 ) )
        {
            DoKeyStrokes ( TRUE ) ;
        }

        break ;
    }

    // Установка состояния, в которое выполняется
    // переход после обработки этой клавиши.
    eCurrKeyState = eShiftToState ;
}

```

Специальным образом обрабатывается только комбинация Alt+Tab. Я мог записывать фактические нажатия клавиш Alt и Tab, однако это могло бы привести к проблемам при следующем запуске сценария, так как мне нужно было бы размещать окно приложения в том же месте z-порядка и иметь то же число запущенных приложений. Поэтому вместо записи нажатий клавиш я перехожу в состояние ожидания того, когда вы отпустите клавишу Alt, а при следующем вводе какой-либо журнальной информации я определяю, какое приложение имеет фокус, и генерирую соответствующий код сценария.

Обработка ввода мыши

Когда я собрался реализовать поддержку мыши, я по-настоящему удивился тому, что мой первоначальный механизм обработки событий клавиатуры не позволял добавить обработку событий мыши. Первоначальный вариант моего кода обработки клавиатуры оптимизировал обработку Ctrl, Shift и Alt, гарантируя, что один метод PlayInput включает полную команду, начиная с нажатия одной из клавиш Ctrl, Shift или Alt и заканчивая ее отпусканием. Когда я стал думать о поддержке событий мыши, я понял, что это может приводить к генерированию метода PlayInput, включающего десятки тысяч символов! Даже для начала обработки ввода мыши мне нужно было изменить код записи событий клавиатуры, чтобы он генерировал специальные коды, такие как {ALT DOWN} и {ALT UP}. Это было нужно для того, чтобы сгенерированные команды мог выполнять любой язык сценариев.

Позабывшись об обработке Ctrl, Alt и Shift, я должен был разобраться с одиночными, двойными щелчками и перетаскиванием. Очень интересно, что ловушка записи журнала, при помощи которой я регистрирую события клавиатуры и мыши, получает только сообщения WM_XBUTTONDOWN и WM_XBUTTONUP. Я был бы очень рад получать сообщения WM_XBUTTONDOWNBLCLK, так как это сделало бы мою жизнь гораздо проще. Обработка событий мыши отчаянно требовала создания конечного автомата, подобного тому, что я разработал для обработки событий клавиатуры. На рис. 16-5 и 16-6 показан конечный автомат обработки ввода мыши, который я реализовал в файлах RECORDINGENGINE.H/.CPP. Помните, что я должен был выполнять такое отслеживание состояния для каждой клавиши. Слоты 0 и 1 нужны для слежения за предыдущим событием с целью сравнения.

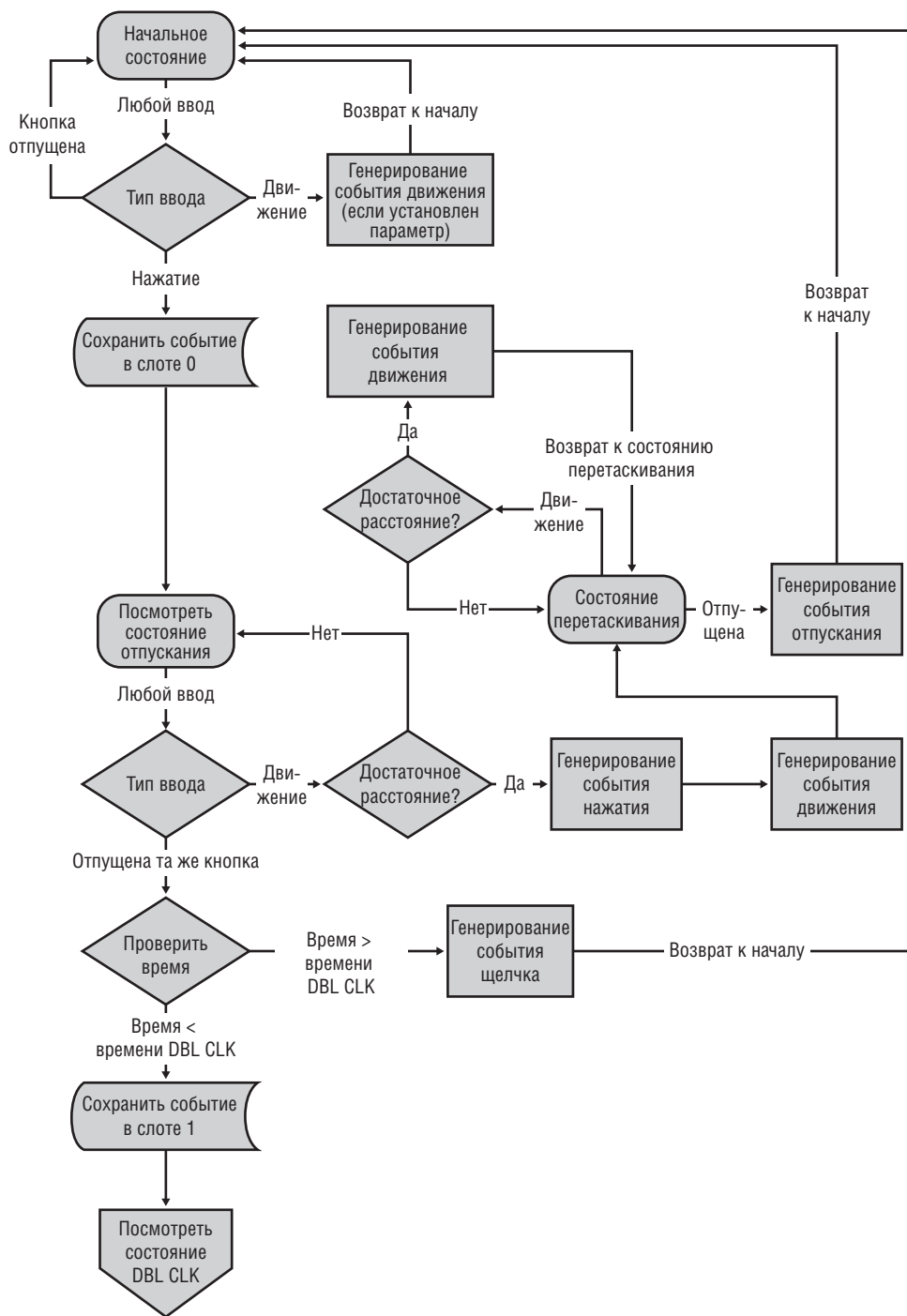


Рис. 16-5. Конечный автомат нормальной обработки событий мыши

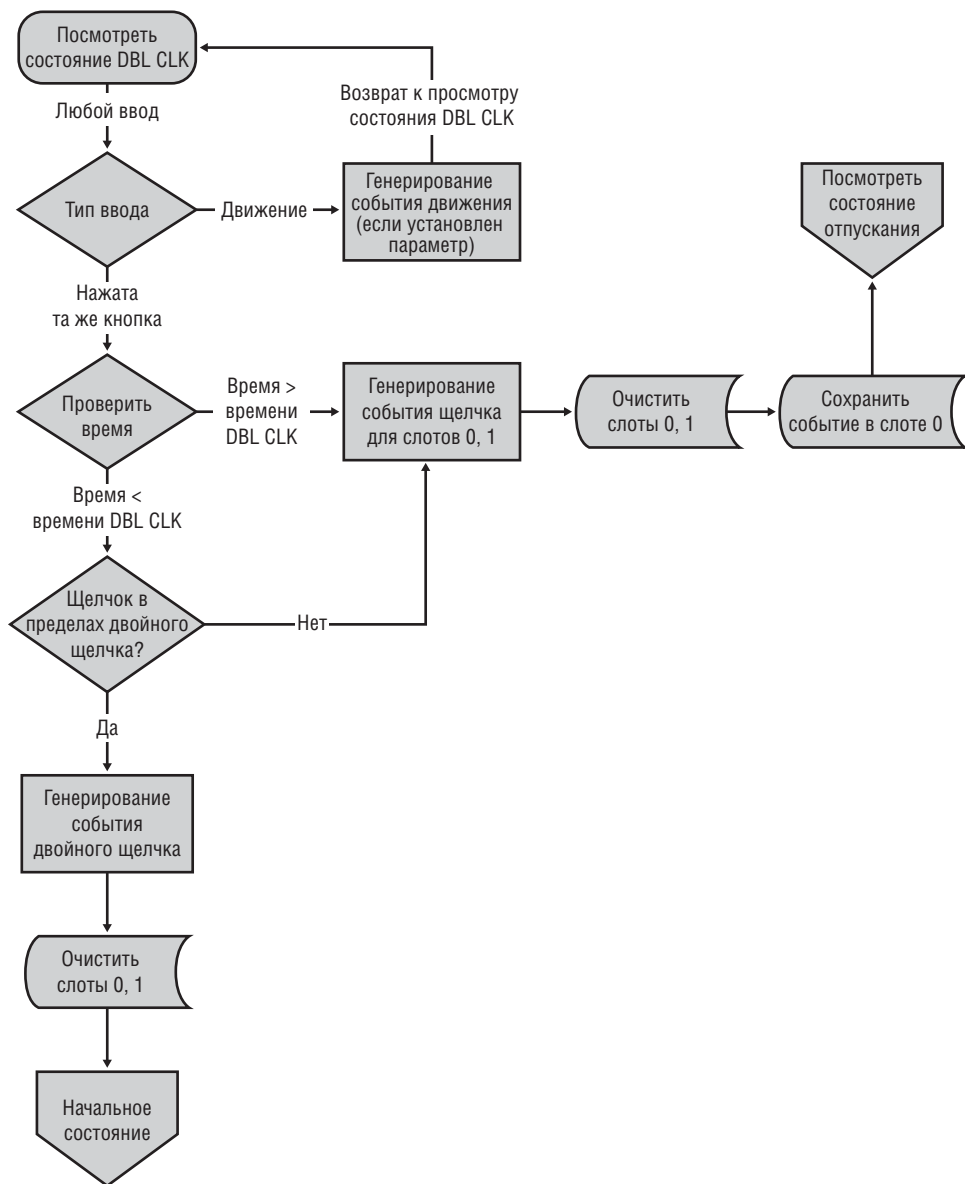


Рис. 16-6. Конечный автомат обработки двойного щелчка

После написания кода записи событий мыши все казалось правильным, пока я не занялся серьезным тестированием — сразу же возникли проблемы. Запись сценариев рисования в Microsoft Paint работала отлично, но при их воспроизведении я столкнулся с неприятностями. Например, при воспроизведении нарисованной вручную окружности сначала появлялась прямая линия, а затем вырисовывалась оставшаяся часть окружности. Я тщательно изучил код записи и воспроизведения сценариев, но ошибок не нашел. Как оказалось, сценарий слишком бы-

стро передавал команды `MOVE0`, что вызывало переполнение входной очереди ОС Windows и отбрасывание избыточных сообщений. Следовательно, я должен был замедлить обработку сообщений мыши, обеспечив достаточное время для выполнения всех соответствующих событий. Так как для воспроизведения команд я использовал функцию `SendInput`, я сначала подумал о том, чтобы задать время для каждого события мыши в структуре `INPUT`, предоставив дополнительное время для их обработки. Это не сработало, и я обнаружил, что задание достаточно долгого времени переводит компьютер в режим энергосбережения, что в первый раз меня довольно сильно удивило.

Тогда я попробовал другой способ. Я решил, что, если мой код записывает вводимые команды в массив структур `INPUT`, указатель на который передается функции `SendInput`, я могу изучать массив по одному элементу и делать дополнительную паузу при обнаружении событий мыши. Длительность пауз я определил экспериментально. После ряда проб я обнаружил, что лучше всего делать паузы на 25 миллисекунд до и после каждого события мыши. Это означает, что записанные сценарии будут воспроизводиться гораздо медленнее по сравнению с тем, когда вы их записывали.

Что после Tester?

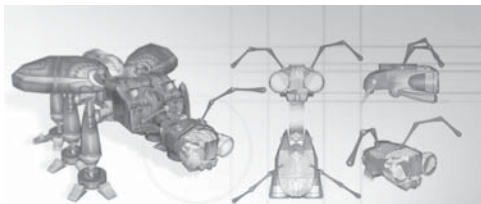
Как я уже говорил, Tester хорошо справляется с двумя вещами: с записью сценариев и воспроизведением записанных событий. Если у вас имеется необходимое вдохновение, вы можете усовершенствовать Tester (равно как и все остальные утилиты из этой книги). Вот некоторые возможные способы улучшения Tester:

- Добавьте классы-оболочки, такие как `TListBox`, `TTreeControl` и `TRadioButton`, чтобы вы могли проверять состояния и данные элементов управления. Такие классы позволят проверять элементы управления и писать более сложные сценарии. Возможно, для облегчения этой задачи понадобится изучить интерфейсы MSAA.
- Реализуйте поддержку оперативного ввода проверочного и другого необходимого кода в сценарий во время его записи.
- Сделайте программу TestRec более дружелюбной к пользователю во время разработки сценариев и создайте систему помощи. Например, можно реализовать хранение сведений о поддерживаемых объектах Tester, что поможет другим разработчикам писать собственные сценарии.
- Реализуйте возможность записи сценариев на компилируемых языках, таких как C# и Visual Basic .NET (или Microsoft Visual Object Assembler, если Microsoft когда-нибудь его создаст).
- В настоящее время TestRec и Tester поддерживают только клавиатуру US English. Если хотите, вы можете сделать оба приложения по-настоящему интернациональными.

Резюме

Блочное тестирование UI иногда вызывает проблемы. В этой главе я представил полезную утилиту, `Tester`, которая позволяет автоматизировать тестирование, записывая манипуляции с клавиатурой и мышью и проигрывая их вашему приложению. По функциональности `Tester` немного не дотягивает до коммерческих средств регрессивного тестирования, но даже в этом случае он благоприятно скажется на ваших запястьях.

Надеюсь, `Tester` покажет вам, насколько ценными могут быть средства автоматизации выполнения программ. Если ваше приложение довольно сложно, я рекомендую приобрести коммерческое средство регрессивного тестирования, чтобы ускорить проведение блочного тестирования. Потратив некоторое время на планирование использования средства регрессивного тестирования, вы сможете создать систему, позволяющую программистам писать сценарии, которые сотрудники отдела контроля качества смогут включать в автоматизированные тесты контроля качества всей программы. Если вы сделаете все правильно, вам будет казаться, что основные тесты контроля качества пишут себя сами.



Стандартная отладочная библиотека C и управление памятью

Даже после выхода первого издания этой книги я получал и получаю массу вопросов по поводу искажений и утечек памяти. Если бы программисты просто прекратили использовать память в своих программах, они избежали бы массы проблем. Все так. А если бы мы прекратили дышать, мы никогда не страдали бы от легочных болезней. Память — эликсир жизни программ C и C++, поэтому, если вы и впрямь хотите что-то сделать с искажениями и утечками памяти, а не просто мечтать, чтобы они исчезли, нужно позаботиться об их проактивной обработке. Первый шаг в этом направлении — изучение отладочной библиотеки C, разработанной Microsoft.

Сложность отладки памяти имеет легендарный статус, и именно она была одним из главных факторов, побудивших Microsoft разработать платформу .NET. Благодаря сборщику мусора CLR программисты могут избавиться от многих аспектов работы с памятью, что устраняет, наверное, около 50% ошибок, с которыми приходится сталкиваться в мире Microsoft Win32/Win64. Однако, если для ваших приложений очень важно быстрое действие, вам еще долго придется писать их на C++ и быть готовым к возможным ошибкам при работе с памятью.

C/C++-программисты от таких проблем ничем не защищены. Эти языки обеспечивают почти полную свободу программирования, но при этом позволяют вам не только выстрелить себе в ногу, но и полностью отстрелить ее даже при небольшой ошибке. К счастью, разработчики стандартной библиотеки C (C run-time, CRT library) не оставили наши муки без внимания и создали миллион Интернет-лет назад удивительное средство — отладочную библиотеку CRT (debug CRT, DCRT library), включенную в состав сред Microsoft, начиная с Visual C++ 4.

Странно, но похоже, что многие C/C++-программисты не подозревают о существовании этой библиотеки. Ее таинственность объясняется тем, что по умолчанию многие ее свойства отключены. Однако, как только вы установите соответствующие флаги, вы сразу поймете, сколь богатые возможности от вас ускользали. В этой главе я сначала представлю библиотеку DCRT и опишу два ее расширения, MemDumperValidator и MemStress, которые предоставят вам еще более развитые возможности. Кратко рассмотрев DCRT, я расскажу о разных аспектах отладки памяти, такие как кучи ОС, отслеживание записи по случайным адресам и использование бесплатных, но крайне полезных инструментов от Microsoft: PageHeap и Application Verifier. Наконец, как я и обещал в главе 2, я опишу удивительные ключи проверки ошибок в период выполнения (/RTCx) и ключ безопасности (/GS), повышающие эффективность отладки памяти и безопасность программ, написанных на Visual C++, на недостижимый раньше уровень.

Особенности стандартной отладочной библиотеки C

Главное достоинство библиотеки DCRT — удивительные возможности слежения за памятью куч. Она позволяет следить за всей памятью, выделяемой в отладочных компоновках при помощи стандартных функций C/C++, таких как `new`, `malloc` и `calloc`, а также за записью данных до начала выделенного блока памяти (`underwrite`) и после его окончания (`overwrite`). Обо всех этих ошибках сообщает сама DCRT посредством утверждений (`assertion`). DCRT также следит за утечками памяти, сообщая о них при завершении программы посредством функции `OutputDebugString`, вывод которой появляется в окне Output отладчика. Если вы работали над приложениями, использующими библиотеку Microsoft Foundation Class (MFC), то сталкивались при завершении своих программ с отчетами об утечке памяти; их посылала вам библиотека DCRT. MFC подключает некоторые ее функции автоматически.

Другое полезное свойство библиотеки DCRT — ее подсистема сообщений (мы с вами назвали бы ее трассировщиком), обеспечиваемая макросами `_RPTn` и `RPTFn` и утверждениями. О поддержке библиотекой DCRT утверждений и их использовании я писал в главе 3. Как я говорил, утверждения DCRT очень полезны, но они уничтожают значение последней ошибки, что может приводить к различному поведению отладочных и заключительных компоновок. Я советую применять для своих утверждений макрос `SUPERASSERT`, код которого включен в `BUGSLAYERUTIL.DLL`.

Еще одна приятная особенность библиотеки DCRT в том, что ее исходный код поставляется вместе с компилятором. Список всех ее файлов см. в табл. 17-1. Если при установке Microsoft Visual Studio .NET вы установили исходный код библиотеки CRT, что я очень рекомендую, то сможете найти весь исходный код библиотек CRT и DCRT в подкаталоге <каталог установки Visual Studio .NET>\VC7\CRT\SRC.

Табл. 17-1. Исходные файлы стандартной отладочной библиотеки C

Исходный файл	Описание
DBGDEL.CPP	Определение глобального отладочного оператора <code>delete</code> .
DBGHEAP.C	Определения всех отладочных функций работы с кучами.
DBGHOOK.C	Заглушка функции-ловушки выделения памяти.
DBGINT.H	Объявления внутренних данных и функций отладочной библиотеки.
DBGNEW.CPP	Определение глобального отладочного оператора <code>new</code> .
DBGRT.C	Определения отладочных функций подсистемы сообщений.
CRTDBG.H	Заголовочный файл, который вы будете включать в свои программы. Он находится в стандартном каталоге включаемых файлов.

Стандартный вопрос отладки

Зачем мне стандартная отладочная библиотека C, если я использую средство обнаружения ошибок наподобие BoundsChecker?

Такие инструменты обнаружения ошибок, как BoundsChecker компании Compuware или Purify от Rational Software автоматически обрабатывают запись данных до начала и после окончания выделенной памяти, а также ее утечки. Если вы работаете с одним из этих средств, вам может казаться, что использование библиотеки DCRT не стоит затрат времени и усилий. С технической точки зрения это верно, однако, чтобы гарантировать нахождение всех проблем с памятью, отладочную компоновку приложения нужно всегда выполнять под управлением средства обнаружения ошибок. Это должны делать не только вы и ваши коллеги по группе, но и, если вы следовали моим советам, приведенным в главе 2, даже сотрудники отдела контроля качества. Не думаю, чтобы все люди были такими ответственными.

Библиотека DCRT подобна хорошей страховке от пожара или кражи. Все мы надеемся, что такая страховка нам не понадобится, но порой она может спасти нас от разорения. Не упускайте ни одной возможности проверить данные своей программы. Библиотека DCRT не вызывает значительного снижения быстродействия программ и в то же время может указать на некоторые очень коварные ошибки. Вам следует использовать ее всегда, даже если вы применяете все средства обнаружения ошибок в мире.

Использование стандартной отладочной библиотеки C

Чтобы вы начали как можно раньше извлекать выгоду из слежения за памятью, библиотеку DCRT нужно прежде всего подключить. Для этого в главный прекомпилированный заголовочный файл (или любой другой заголовочный файл, включаемый во все исходные файлы проекта) надо добавить такую строку, указав ее перед всеми директивами `#include`:

```
#define _CRTDBG_MAP_ALLOC
```

После всех остальных заголовочных файлов нужно включить файл CRTDBG.H. Благодаря определению `_CRTDBG_MAP_ALLOC` вызовы обычных функций выделения и освобождения памяти будут перенаправляться их специальным версиям, записывающим при каждой такой операции сведения об исходном файле и номере строки.

После этого нужно включить средства проверки кучи, обеспечиваемые библиотекой DCRT. Как я уже упоминал, большинство из них по умолчанию отключено. В документации утверждается, что они отключены для уменьшения объема кода и повышения быстродействия программы. Конечно, для заключительных компоновок это очень важно, но не забывайте, что назначение отладочных компоновок как раз в нахождении ошибок! Увеличение объема и снижение быстродействия отладочных компоновок не играют большой роли. Поэтому без колебаний включайте все средства библиотеки DCRT, которые, по вашему мнению, могут пригодиться. Для их включения нужно передать функции `_CrtSetDbgFlag` набор флагов, объединенных операцией ИЛИ (табл. 17-2).

Табл. 17-2. Флаги стандартной отладочной библиотеки C

Флаг	Описание
<code>_CRTDBG_ALLOC_MEM_DF</code>	Подключает механизмы отладочного выделения памяти и использование идентификаторов блоков памяти. Это единственный флаг, установленный по умолчанию.
<code>_CRTDBG_CHECK_ALWAYS_DF</code>	Выполняет проверку всей памяти при каждом запросе об ее выделении и освобождении. Установка этого флага позволяет обнаруживать запись данных вне блока памяти как можно раньше после возникновения ошибки.
<code>_CRTDBG_CHECK_CRT_DF</code>	После установки этого флага во всех операциях обнаружения утечек и изменений блоков памяти проверяются блоки <code>_CRT_BLOCK</code> . Как правило, устанавливать его следует только при проблемах с функциями библиотеки CRT. При этом вы будете получать сообщения о выделении памяти библиотекой CRT. Так как она должна иметь выделенную память вплоть до истинного завершения вашей программы, что происходит после вывода сообщений об утечках памяти, то после установки этого флага вы увидите массу ложных сообщений об утечках.
<code>_CRTDBG_DELAY_FREE_MEM_DF</code>	После установки этого флага действительное освобождение памяти не выполняется. Блоки продолжают храниться во внутреннем списке кучи, но заполняются значениями 0xDD, благодаря чему вы легко можете узнать освобожденную память, изучая ее в отладчике. Этот флаг позволяет вам проверить свою программу в условиях нехватки памяти. Кроме того, библиотека DCRT следит за тем, чтобы все ячейки освобожденных блоков памяти оставались равными 0xDD, что поможет вам обнаружить попытки повторного доступа к ним. Устанавливайте этот флаг всегда, но помните, что требования вашей программы к памяти при этом легко могут удвоиться, потому что освобожденная память не возвращается в кучу.
<code>_CRTDBG_LEAK_CHECK_DF</code>	Проверяет утечки памяти в конце программы. Установка этого флага просто обязательна.

Включив в программу директивы `#include` и `#define` и вызвав `_CrtSetDbgFlag`, вы получите полный доступ ко всем функциям библиотеки DCRT, что поможет контролировать использование памяти и получать нужную информацию. Вы можете вызывать эти функции в любой момент. Многие из них приспособлены для использования в утверждениях, так что вы можете свободно «рассыпать» их по своему коду для раннего обнаружения проблем с памятью.

Ошибка в DCRT

Если вы следуете инструкциям предыдущего раздела, ваш исходный код будет похож на листинг 17-1. В начале листинга вы видите определение `_CRTDBG_MAP_ALLOC` и заголовочный файл `CRTDBG.H`, между которыми включаются все остальные заголовочные файлы. В самом начале `main` вызывается функция `_CrtSetDbgFlag` для настройки DCRT. Далее я выделяю три блока памяти: один при помощи `malloc` и два при помощи `new`, — и все три случая вызывают утечку памяти.

Листинг 17-1. Подключение библиотеки DCRT и механизмов отслеживания утечек памяти

```
// Эту директиву define нужно указывать до включения
// любых заголовочных файлов.
#define _CRTDBG_MAP_ALLOC
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tchar.h>

// Файл CRTDBG.H включается после всех остальных заголовочных файлов.
#include <crtdbg.h>
void main ( void )
{
    // Включение всех механизмов проверки кучи.
    _CrtSetDbgFlag ( _CRTDBG_ALLOC_MEM_DF |
                    _CRTDBG_CHECK_ALWAYS_DF |
                    _CRTDBG_DELAY_FREE_MEM_DF |
                    _CRTDBG_LEAK_CHECK_DF );

    // Выделение памяти.
    TCHAR * pNew = new TCHAR[ 200 ] ;
    TCHAR * pNew2 = new TCHAR[ 200 ] ;
    TCHAR * pMemLeak = (TCHAR*)malloc ( 100 ) ;

    _tcscpy ( pNew , _T ( "New'd memory..." ) );
    _tcscpy ( pNew2 , _T ( "More new'd memory..." ) );
    _tcscpy ( pMemLeak , _T ( "Malloc'd memory..." ) );
}
```

Так как в листинге 17-1 я включил проверку утечек памяти, то при выполнении программы вы увидите в окне Output нечто вроде:

```

Detected memory leaks!
Dumping objects ->
NewProblem.cpp(22) : {62} normal block at 0x002F2E58, 100 bytes long.
Data: <M a l l o c ' d > 4D 00 61 00 6C 00 6C 00 6F 00 63 00 27 00 64 00
G:\vsnet\Vc7\include\crtDBG.h(692) :
{61} normal block at 0x002F2C88, 400 bytes long.
Data: <M o r e   n e w > 4D 00 6F 00 72 00 65 00 20 00 6E 00 65 00 77 00
G:\vsnet\Vc7\include\crtDBG.h(692) :
{60} normal block at 0x002F4DC0, 400 bytes long.
Data: <N e w ' d   m e > 4E 00 65 00 77 00 27 00 64 00 20 00 6D 00 65 00
Object dump complete.

```

Формат вывода информации об утечках памяти очень удобен: двойной щелчок выведенного номера строки вызовет автоматический переход к этой строке в исходном файле. Первая утечка происходит в строке 22 файла NEWPROBLEM.CPP (см. листинг 17-1), и двойной щелчок действительно переносит вас на строку, выполняющую `malloc`. Однако, дважды щелкнув номер строки, указанной в списке второй, вы попадете в файл CRTDBG.H (строка 692). Вы увидите вызов `new`, но это определенно не мой исходный код. Так как в моей программе несколько вызовов `new`, можно сделать вывод, что все вызовы `new` отображаются как исходящие из файла CRTDBG.H. Легко понять, что это не очень поможет при поиске утечек памяти в большой программе!

Проблема заключается в объявлении `new` в файле CRTDBG.H:

```

inline void* __cdecl operator new[](size_t s)
{ return ::operator new[](s, _NORMAL_BLOCK, __FILE__, __LINE__); }

```

Если вы никогда не сталкивались с синтаксисом размещения (`placement syntax`) оператора `new`, представленный фрагмент может сначала показаться странным. Оператор `new` — очень специфическая конструкция языка C++, так как он может принимать самые разнообразные параметры. Видно, что во время вызова `::operator new` ему передаются три дополнительных параметра. Эта версия `new` определяется в файле CRTDBG.H. Кажется, причина неприятностей неочевидна. Макросы `__FILE__` и `__LINE__` расширяются при компиляции в имя исходного файла и номер строки в нем. Однако, как вы увидели, они расширяются не в ваши файлы и номера строк. Проблема — в первом слове представленного фрагмента: `inline`. В заключительных компоновках объявление функции `inline` означает, что компилятору рекомендуется не вызывать функцию, а разместить в месте ее вызова сам код функции. Но при этом нужно помнить, что в отладочных компоновках функции `inline` не расширяются и рассматриваются как действительные функции. Соответственно макросы `__FILE__` и `__LINE__` в нашем случае расширяются в CRTDBG.H и 692.

Так как это накладывает на использование DCRT серьезные ограничения, я должен был найти способ сделать так, чтобы все обнаруженные утечки указывали на выделение памяти в истинном исходном коде. Сначала я хотел изменить CRTDBG.H, но потом решил, что это не очень здорово: все-таки это системный заголовочный файл. Поразмыслив, я в конце концов написал следующий макрос, который нужно включать в прекомпилированный заголовочный файл сразу после включения CRTDBG.H. Он просто преобразует любой вызов `new` в версию `new` с синтаксисом размещения, принимающую дополнительные параметры.


```

#ifdef _DEBUG
#ifndef NEW_INLINE_WORKAROUND
#define NEW_INLINE_WORKAROUND new ( _NORMAL_BLOCK , \
                                   __FILE__ , __LINE__ )

#define new NEW_INLINE_WORKAROUND
#endif
#endif // _DEBUG

```

Наверное, внимательные читатели догадались, что в моем макросе `NEW_INLINE_WORKAROUND` скрыта проблема: если у вас есть класс, определяющий оператор `new`, мой макрос приведет к путанице объявлений. Скорее всего вам не приходится определять оператор `new` в своих классах, но такие библиотеки, как MFC и STL, это точно делают. Вся хитрость — в его определении только внутри прекомпилированного заголовочного файла, во включении в этот файл только таких библиотек, как MFC и STL, и в использовании улучшенных директив `#pragma push_macro/#pragma pop_macro`. Точные рекомендации на этот счет см. в разделе «Стандартный вопрос отладки: что включать в прекомпилированные заголовочные файлы?».

Если в вашу программу входят классы, определяющие оператор `new`, вы тоже можете использовать макрос `NEW_INLINE_WORKAROUND`, если согласитесь добавить в класс несколько директив и немного кода. В следующем фрагменте я привожу упрощенный пример класса, содержащего все необходимое для отличной работы `NEW_INLINE_WORKAROUND`:

```

// Сохранение определения оператора new в стеке макроса.
#pragma push_macro ( "new" )
// Отмена определения new нужна для правильного объявления класса.
#ifdef new
#undef new
#endif

class TestClass
{
public :

// Синтаксис размещения new с прототипом, нужным
// для записи правильной информации об исходном
// файле и номере строки при выделении памяти.
#ifdef _DEBUG
    // iSize          - размер выделяемой памяти.
    // iBlockType     - тип блока DCRT.
    // lpszFileName   - имя исходного файла.
    // nLine          - номер строки в исходном файле.
    static void * operator new ( size_t nSize          ,
                                int iBlockType         ,
                                char * lpszFileName     ,
                                int nLine              )

    {
        // Любой нужный вам код.

        // Действительное выделение памяти при помощи _malloc_dbg

```

```

// и передача всех параметров для записи места этой операции.
return ( _malloc_dbg ( nSize
                      ,
                      iBlockType
                      , lpszFileName
                      , nLine
                      ) );
}
#endif // _DEBUG
};

// Восстановление сохраненного значения
// макроса new (т. е. NEW_INLINE_WORKAROUND).
#pragma pop_macro ( "new" )

```

Своим существованием такое решение обязано новым директивам `#pragma push_macro` и `#pragma pop_macro`, которые сохраняют текущее определение макроса во внутреннем стеке компилятора и восстанавливают сохраненное значение соответственно. Вы должны будете заключать в них любой класс, в котором выполняется перегрузка оператора `new`, потому что директивы `#pragma` не могут быть автоматизированы при помощи макросов. Дополнительный оператор `new` будет вызываться макросом `NEW_INLINE_WORKAROUND` для изменения фактического выделения памяти. Применять дополнительный оператор `new` немного неудобно, но так вы хоть получите полные отчеты обо всех утечках памяти. Чтобы увидеть, как все это работает, изучите проект `FixedNewProblem` на CD.

Стандартный вопрос отладки

Что включать в прекомпилированные заголовочные файлы?

Как я упоминал при обсуждении решения ошибки в DCRT, для получения сообщений об утечках памяти, выделяемой при помощи оператора `new`, и отображения верной информации о номере строки исходного кода, необходимо наличие правильных прекомпилированных заголовочных файлов. К тому же это не только облегчит отладку памяти, но и ускорит компиляцию программ. Прекомпилированный заголовочный файл — это по сути записанное на диск дерево грамматического разбора для файлов, указанных в файле `.H` (традиционно называемом `STDAFX.H`). Поэтому вы компилируете его только раз, а не при каждой компиляции файла `.C/.CPP`.

Вот правила создания прекомпилированного заголовочного файла.

1. Включайте в него все заголовочные файлы библиотек CRT/компилятора.
2. Если в директиве `#include` вы заключаете имя файла в угловые скобки, имена ваших заголовочных файлов должны указываться в кавычках.
3. Включайте в него все заголовочные файлы сторонних фирм, такие как файлы для `BUGSLAYERUTIL.DLL`.
4. Включайте в него все заголовочные файлы вашего проекта, которые не изменялись более одного-двух месяцев.

Полезные функции DCRT

Одна из наиболее полезных функций библиотеки DCRT — `_CrtCheckMemory` — просматривает всю выделенную вами память и проверяет, не выполняете ли вы запись данных вне выделенных блоков и не используете ли вы ранее освобожденные блоки. Даже одна эта функция оправдывает использование всей библиотеки DCRT. Один из великолепных методов обнаружения проблем с памятью состоит в «разбрасывании» вызовов `ASSERT (_CrtCheckMemory ())`; по всему коду программы. Так вы сможете находить ошибки записи данных вне выделенных блоков памяти максимально близко к месту их возникновения.

Другой набор функций позволяет с легкостью проверять корректность любого блока памяти. Отладочные функции `_CrtIsValidHeapPointer`, `_CrtIsValidMemoryBlock` и `_CrtIsValidPointer` прекрасно подходят для проверки параметров. Эти функции вместе с `_CrtCheckMemory` обеспечивают великолепные возможности проверки памяти.

Еще один полезный набор функций библиотеки DCRT включает функции изучения состояния памяти: `_CrtMemCheckpoint`, `_CrtMemDifference` и `_CrtMemDumpStatistics`. Благодаря им вы можете легко выполнять сравнение кучи до и после вызова какой-нибудь функции, определяя момент, когда что-то начинает работать не так. Скажем, если вы используете стандартную библиотеку в группе, вы можете записывать состояние кучи до и после вызовов библиотечных функций для обнаружения утечек памяти и определения объема памяти, необходимого для конкретной операции.

Сахарной глазурью на пирожном проверки памяти является возможность установки ловушки, благодаря которой вы можете узнавать про каждый вызов функций выделения и освобождения памяти. Если ловушка выделения памяти возвращает `TRUE`, выделение памяти можно продолжить, если же `FALSE` — выделение памяти завершается неудачей. Когда я впервые обнаружил эту функциональность, я сразу же понял, что, приложив небольшие усилия, я получу инструменты тестирования кода в по-настоящему сложных граничных условиях, которые иначе было бы очень сложно воспроизвести. Результатом этого является модуль `MemStress` из состава `BUGSLAYERUTIL.DLL`. Он дает вам возможность принудительно вызывать неудачи выделения памяти в ваших программах, про что я расскажу ниже.

И, наконец, вишенка на сахарной глазури: библиотека DCRT позволяет устанавливать ловушку для функций записи дампа памяти и перечислять клиентские блоки (выделенную вами память). Вы можете заменить функции дампа памяти, используемые по умолчанию, собственными функциями, знающими о ваших данных все. После этого вы сможете получать не таинственный дамп памяти по умолчанию (который не только сложен в понимании, но и менее полезен), а точное содержание блока памяти, отформатированное так, как вам угодно. MFC предоставляет для этого функцию `Dump`, но она работает только с классами, унаследованными от `CObject`. Я уверен, что если мы с вами в чем-то похожи, вы также не можете смириться с написанием программ только при помощи MFC и хотели бы получить более общие функции создания дампов памяти, охватывающие различные типы кода.

Как можно догадаться, перечисление клиентских блоков позволяет перечислять выделенные вами блоки памяти. Опираясь на эту прекрасную возможность,

можно создать некоторые очень интересные утилиты. Так, в функциях MemDumper-Validator из библиотеки BUGSLAYERUTIL.DLL я при перечислении клиентских блоков вызываю ловушки записи дампов, чтобы можно было создавать дампы и выполнять проверку многих типов выделенной памяти за одно действие. Это очень мощное средство, позволяющее реализовать глубокую проверку памяти, а не только проверку записи данных вне выделенного блока. Под глубокой проверкой я понимаю алгоритм, который знает формат данных блока памяти и проверяет корректность всех элементов блока с учетом их формата.

Выбор правильной стандартной отладочной библиотеки C для вашего приложения

Некоторое замешательство по поводу использования библиотек CRT при создании программ для Microsoft Windows связано с выбором правильной библиотеки. Существует шесть версий библиотеки CRT, подразделяющихся на две основных категории: отладочные (DCRT) и заключительные (CRT). В каждой категории имеется однопоточная статическая библиотека, многопоточная статическая библиотека и многопоточная DLL.

При работе со статическими версиями библиотек CRT библиотечные функции компонуется прямо в вашу программу; именно эти версии используются по умолчанию для приложений, создаваемых без помощи мастеров MFC. Преимущество этого подхода в том, что вам не придется поставлять вместе со своей программой динамическую библиотеку CRT, а недостаток — в огромном увеличении объема двоичных файлов и рабочего набора. Названия двух вариантов статической библиотеки CRT — однопоточной и многопоточной — говорят сами за себя. Если вы создаете DLL и хотите задействовать статическую библиотеку CRT, вам следует выполнять компоновку только с ее многопоточной версией, иначе многопоточные приложения не смогут работать с вашей DLL, так как однопоточные статические библиотеки CRT небезопасны с точки зрения потоков.

DLL-версии библиотек CRT — MSVCRT(D).DLL — позволяют импортировать библиотечные функции CRT. Благодаря этим DLL вы можете уменьшить размер своих двоичных файлов, а значит, и рабочий набор программы. Поскольку другие приложения будут загружать одни и те же DLL, ОС сможет предоставить нескольким процессам совместный доступ к страницам кода DLL, и вся система станет работать быстрее. Однако этот вариант имеет и недостаток: вполне возможно, что вам придется распространять еще одну DLL вместе со своей программой.

Чрезвычайно важно, чтобы вы выбрали какую-то одну версию библиотеки CRT для всех двоичных файлов, загружаемых в адресное пространство своей основной программы. Если некоторые ваши DLL будут обращаться к статической библиотеке CRT, а другие — к динамической, вы не только израсходуете дополнительное адресное пространство из-за дублирования кода, но и создадите плодородную почву для одной из самых коварных ошибок памяти, на отладку которой могут потребоваться месяцы. При выделении памяти в куче из одной DLL и освобождении этой памяти во второй DLL, использующей другую версию библиотеки CRT, ваша программа сможет с легкостью потерпеть крах, потому что освобождающая память DLL не будет знать, откуда взялась выделенная память. Не относитесь к куче

с пренебрежением: одновременное выполнение разных версий библиотеки CRT влечет за собой различную обработку памяти куч.

Я всегда использую динамические версии библиотек CRT и советую вам делать то же самое. Выгода от уменьшения рабочего набора и сокращения основных двоичных файлов перевешивает все прочие соображения. Очень редко — скажем, при разработке игр, когда я уверен в том, что многопоточность мне не понадобится и когда чрезвычайную важность приобретает быстроедействие, — я могу рассмотреть применение однопоточных статических версий для избежания затрат на механизмы многопоточной блокировки.

Для работы с динамическими версиями библиотек CRT я создал библиотеку BUGSLAYERUTIL.DLL. Код расширений MemDumperValidator и MemStress, про которые я рассказываю в этой главе, также хранится в BUGSLAYERUTIL.DLL. Эти модули расширения тоже ожидают, что вы будете работать с их DLL-версиями. Однако, если вы захотите использовать их в своем приложении не в виде DLL, вы можете отобрать исходные файлы MEMDUMPERVALIDATOR.CPP, MEMDUMPERVALIDATOR.H, MEMSTRESS.CPP, MEMSTRESSCONSTANTS.H и MEMSTRESS.H, изменить указанный метод компоновки функций и включить их в свое приложение.

И еще одна деталь — она касается использования BUGSLAYERUTIL.DLL. В зависимости от того, как вы выделяете память, вы можете столкнуться с замедлением работы своей программы. Разрабатывая расширение MemDumperValidator, я хотел обеспечить всю полноту отслеживания и проверки памяти, для чего включил в библиотеке DCRT все соответствующие флаги, в том числе _CRTDBG_CHECK_ALWAYS_DF, который приказывает библиотеке DCRT просматривать и проверять все фрагменты памяти кучи при каждом выделении и освобождении памяти. Если вы выделяете в своей программе тысячи небольших блоков памяти, задержка будет очевидной, однако это ясно укажет вам на желательность изменения алгоритма обработки данных. Большое число выделений небольших фрагментов памяти плохо сказывается на быстродействии и требует исправления. Если вы не сможете изменить код, что ж, отключите этот флаг, вызвав функцию _CrtSetDbgFlag.

Использование MemDumperValidator

Расширение MemDumperValidator здорово упрощает отладку памяти. Библиотека DCRT по умолчанию сообщает об утечках памяти и записи данных вне выделенных блоков. Оба этих сообщения могут пригодиться, но, когда они выглядят следующим образом, тип «вытекшей» памяти определить очень сложно:

```
Detected memory leaks
Dumping objects ->
TestProc.cpp(104) : {596} normal block at 0x008CD5B0,
    24 bytes long.
Data: < k          w k > 90 6B 8C 00 B0 DD 8C 00 00 00 80 77 90 6B 8C 00
Object dump complete.
```

Как я уже говорил, гораздо лучше было бы иметь дополнительную информацию — скажем, глубокая проверка памяти помогает обнаружить запись данных по случайным адресам, что очень сложно в противном случае. Именно такую более подробную отладочную информацию и предоставляет вам MemDumperValidator

в своих отчетах об утечках памяти, поддерживая к тому же ряд дополнительных методов проверки памяти. А чем больше информации вы будете иметь при отладке, тем быстрее вы ее выполните.

MemDumperValidator использует идентификаторы блоков памяти библиотеки DCRT, позволяющие связать тип блока с набором функций, которым известно его содержание. Всем блокам, выделяемым библиотекой DCRT, присваиваются идентификаторы (табл. 17-3). Тип блока передается как параметр в функции выделения памяти библиотеки DCRT: `_nh_malloc_dbg (new)`, `_malloc_dbg (malloc)`, `_calloc_dbg (calloc)` и `_realloc_dbg (realloc)`.

Табл. 17-3. Идентификаторы блоков памяти

Идентификатор блока	Описание
<code>_NORMAL_BLOCK</code>	Обычный вызов <code>new</code> , <code>malloc</code> или <code>calloc</code> приводит к созданию нормального блока. После определения <code>_CRTDBG_MAP_ALLOC</code> вся память в куче выделяется по умолчанию в форме нормальных блоков, которые при этом ассоциируются с номером выделившей память строки и ее исходным файлом.
<code>_CRT_BLOCK</code>	Блоки памяти, выделяемые внутри многих функций стандартной библиотеки, отмечаются как блоки CRT, чтобы их можно было обрабатывать отдельно. Это позволяет не проверять их при поиске утечек и других операциях проверки памяти. Ваше приложение никогда не должно выделять, перераспределять или освобождать блок типа CRT.
<code>_CLIENT_BLOCK</code>	Чтобы программа следила за специфическим типом памяти, можно вызывать отладочные функции выделения памяти, передавая им специальное значение клиентского блока. Вы можете следить за подтипами клиентских блоков, помещая 16-разрядное значение в старшие 16 битов значения клиентского блока: <pre>define CLIENT_BLOCK_VALUE(x) \ (_CLIENT_BLOCK (x<<16)) _heap_alloc_dbg (10 , CLIENT_BLOCK_VALUE(0xA), __FILE__ , __LINE__) ;</pre> Для записи дампов клиентских блоков памяти можно указать функцию-ловушку, вызвав функцию <code>_CrtSetDumpClient</code> . Ловушка будет вызываться каждый раз, когда функция библиотеки DCRT захочет сделать дамп клиентского блока. Кроме того, функция <code>_CrtDoForAllClientObjects</code> позволяет перечислить клиентские блоки, выделенные в данный момент. MFC использует идентификатор клиентских блоков для всех классов, унаследованных от <code>CObject</code> . MemDumperValidator использует ловушку записи дампов клиентских блоков.
<code>_FREE_BLOCK</code>	В нормальных условиях вызов функции освобождения памяти приводит к удалению блока памяти из списков отладочной кучи. Однако, если через функцию <code>_CrtSetDbgFlag</code> установить флаг <code>_CRTDBG_DELAY_FREE_MEM_DF</code> , память не освобождается, а остается выделенной и заполняется значениями <code>0xDD</code> .

Табл. 17-3. Идентификаторы блоков памяти *(продолжение)*

Идентификатор блока	Описание
<code>_IGNORE_BLOCK</code>	Если временно отключить функции отслеживания памяти библиотеки DCRT, все выделенные в это время блоки памяти будут отмечены как игнорируемые блоки.

После того как вы зададите использование расширения `MemDumperValidator` для класса или типа данных C, библиотека DCRT будет вызывать `MemDumperValidator` для создания дампа блока памяти. Это расширение будет анализировать значение блока и вызывать соответствующую функцию записи дампа, если, конечно, такая функция будет обнаружена. Проверка блоков памяти выполняется аналогично за исключением того, что библиотека DCRT вызывает соответствующие функции проверки.

Описать `MemDumperValidator` просто — сложнее привести его в рабочее состояние. В листинге 17-2 показан файл `MEMDUMPERVALIDATOR.H`, выполняющий за вас большую часть инициализации. Включив в свою программу файл `BUGSLAYERUTIL.H`, вы автоматически включите и `MEMDUMPERVALIDATOR.H`.

Листинг 17-2. `MEMDUMPERVALIDATOR.H`

```

/*-----
Отладка приложений для Microsoft .NET и Microsoft Windows
Copyright © 1997-2003 John Robbins - All rights reserved.
-----*/

#ifdef _MEMDUMPERVALIDATOR_H
#define _MEMDUMPERVALIDATOR_H

// Нужно включать не MEMDUMPERVALIDATOR.H, а BUGSLAYERUTIL.H.
#ifdef _BUGSLAYERUTIL_H
#error "Include BUGSLAYERUTIL.H instead of this file directly!"
#endif // _BUGSLAYERUTIL_H

#ifdef __cplusplus
extern "C" {
#endif // __cplusplus

// Эта библиотека применяется только в отладочных компоновках.
#ifdef _DEBUG

////////////////////////////////////
// Объявления typedef для функций записи дампа и проверки памяти
////////////////////////////////////
// Функция записи дампа памяти. Единственный ее параметр – указатель на блок
// памяти. Эта функция может выводить данные блока памяти в любом формате,
// но ради согласованности в ней следует использовать тот же механизм отчетов,
// что применяется в оставшейся части библиотеки DCRT.
typedef void (*PFNMEMDUMPER)(const void *);
// Функция проверки памяти. Первый ее параметр – указатель

```

см. след. стр.

```
// на проверяемый блок памяти, а второй – указатель на данные
// о контексте, переданный в функцию ValidateAllBlocks.
typedef void (*PFNMEMVALIDATOR)(const void * , const void *) ;

////////////////////////////////////
// Полезные макросы
////////////////////////////////////
// Макрос установки значения подтипа клиентского блока. Этот макрос
// обеспечивает единственный надежный способ присвоения значения
// полю dwValue в структуре BSMDVINFO, описанной ниже.
#define CLIENT_BLOCK_VALUE(x) (_CLIENT_BLOCK|(x<<16))
// Макрос получения подтипа блока.
#define CLIENT_BLOCK_SUBTYPE(x) ((x >> 16) & 0xFFFF)

////////////////////////////////////
// Объявления, нужные для инициализации функций записи дампа и проверки
// памяти в соответствии со специфическим подтипом клиентского блока
////////////////////////////////////
typedef struct tag_BSMDVINFO
{
    // Подтип клиентского блока. Он должен устанавливаться при помощи
    // описанного выше макроса CLIENT_BLOCK_VALUE. Чтобы увидеть
    // присвоение значения этому полю, см. функцию AddClientDV.
    unsigned long    dwValue    ;
    // Указатель на функцию записи дампа.
    PFNMEMDUMPER     pfnDump    ;
    // Указатель на функцию проверки памяти.
    PFNMEMVALIDATOR  pfnValidate ;
} BSMDVINFO , * LPBSMDVINFO ;

/*-----
ФУНКЦИЯ: AddClientDV
ОПИСАНИЕ:
    Добавляет в список функции записи дампа и проверки клиентского блока.
    Если поле dwValue в структуре BSMDVINFO равно 0, задается следующее значение
    в списке. Значение dwValue после вызова этой функции всегда должно передаваться
    функции _malloc_dbg в качестве значения подтипа клиентского блока.
    Если значение подтипа устанавливается через CLIENT_BLOCK_VALUE,
    для передачи значения в _malloc_dbg можно использовать макрос.
    Заметьте: соответствующей функции удаления добавленных данных нет.
    Зачем идти на риск совершения ошибок в отладочном коде? Когда дело касается
    обнаружения проблем, быстроедействие не имеет особого значения.
ПАРАМЕТРЫ:
    lpBSMDVINFO - указатель на структуру BSMDVINFO
ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ:
    1 - функции записи дампа и проверки клиентского блока добавлены правильно.
    0 - функции записи дампа и проверки клиентского блока не были добавлены.
-----*/
BUGSUTIL_DLLINTERFACE int __stdcall
AddClientDV (LPBSMDVINFO lpBSMDVInfo);
```



```

/*-----
ФУНКЦИЯ: ValidateAllBlocks
ОПИСАНИЕ:
    Проверяет всю память, выделенную из локальной кучи.
    Просматривает все клиентские блоки и вызывает индивидуальные
    функции проверки для их различных подтипов.
    Вероятно, лучше всего вызывать эту функцию при помощи
    макроса VALIDATEALLBLOCKS, описанного ниже.
ПАРАМЕТРЫ:
    pContext - указатель на информацию о контексте,
                передаваемый в каждую функцию проверки.
ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ:
    Нет
-----*/

BUGSUTIL_DLLINTERFACE void __stdcall
    ValidateAllBlocks ( void * pContext ) ;

#ifdef __cplusplus
////////////////////////////////////////////////////////////////
// Вспомогательные макросы для классов C++
////////////////////////////////////////////////////////////////
// Объявляйте этот макрос в своем классе так же, как макросы MFC.
#define DECLARE_MEMDEBUG(classname)
public :
    static BSMDVINFO  m_stBSMDVINFO ;
    static void ClassDumper ( const void * pData ) ;
    static void ClassValidator ( const void * pData ,
                                const void * pContext ) ;
    static void * operator new ( size_t nSize )
    {
        if ( 0 == m_stBSMDVINFO.dwValue )
        {
            m_stBSMDVINFO.pfnDump      = classname::ClassDumper ;
            m_stBSMDVINFO.pfnValidate = classname::ClassValidator ;
            AddClientDV ( &m_stBSMDVINFO ) ;
        }
        return ( _malloc_dbg ( nSize
                                ,
                                (int)m_stBSMDVINFO.dwValue ,
                                __FILE__ ,
                                __LINE__ ) ) ;
    }
    static void * operator new ( size_t nSize
                                ,
                                char * lpszFileName ,
                                int    nLine
                                )
    {
        if ( 0 == m_stBSMDVINFO.dwValue )
        {
            m_stBSMDVINFO.pfnDump      = classname::ClassDumper ;
            m_stBSMDVINFO.pfnValidate = classname::ClassValidator ;
            AddClientDV ( &m_stBSMDVINFO ) ;
        }
    }
}

```

см. след. стр.

```

    }
    return ( _malloc_dbg ( nSize
                          ,
                          (int)m_stBSMDVINFO.dwValue ,
                          lpszFileName
                          ,
                          nLine
                          ) );
}
static void * operator new ( size_t nSize
                          ,
                          int /*iBlockType*/ ,
                          char * lpszFileName
                          ,
                          int nLine
                          )
{
    if ( 0 == m_stBSMDVINFO.dwValue )
    {
        m_stBSMDVINFO.pfnDump = classname::ClassDumper ;
        m_stBSMDVINFO.pfnValidate = classname::ClassValidator ;
        AddClientDV ( &m_stBSMDVINFO ) ;
    }
    return ( _malloc_dbg ( nSize
                          ,
                          (int)m_stBSMDVINFO.dwValue ,
                          lpszFileName
                          ,
                          nLine
                          ) );
}
static void operator delete ( void * pData )
{
    _free_dbg ( pData , (int)m_stBSMDVINFO.dwValue ) ;
}
static void __cdecl operator delete ( void * _P ,
                                     int
                                     ,
                                     char *
                                     ,
                                     int
                                     )
{
    ::operator delete ( _P ) ;
}
static void __cdecl operator delete[] ( void * _P ,
                                       int
                                       ,
                                       char *
                                       ,
                                       int
                                       )
{
    ::operator delete[] ( _P ) ;
}

// Объявляйте этот макрос в начале своего файла .CPP.
#define IMPLEMENT_MEMDEBUG(classname)
    BSMDVINFO classname::m_stBSMDVINFO = { 0 , 0 , 0 }

// Макрос для отладочного выделения памяти. Если определен
// макрос DEBUG_NEW, можно использовать его.
#ifdef DEBUG_NEW
#define MEMDEBUG_NEW DEBUG_NEW
#else

```

```

#define MEMDEBUG_NEW new ( __FILE__ , __LINE__ )
#endif

// Конец директивы #ifdef __cplusplus.

////////////////////////////////////
// Вспомогательные макросы C
////////////////////////////////////

// Используйте этот макрос для выделения памяти в стиле C.
// Единственная проблема с языком C заключается в том,
// что вам придется возиться со структурой BSMDVINFO.
#define INITIALIZE_MEMDEBUG(lpBSMDVINFO , pfnD , pfnV )
{
    ASSERT ( FALSE == IsBadWritePtr ( lpBSMDVINFO ,
                                     sizeof ( BSMDVINFO ) ) ) ;
    ((LPBSMDVINFO)lpBSMDVINFO)->dwValue = 0 ;
    ((LPBSMDVINFO)lpBSMDVINFO)->pfnDump = pfnD ;
    ((LPBSMDVINFO)lpBSMDVINFO)->pfnValidate = pfnV ;
    AddClientDV ( lpBSMDVINFO ) ;
}

// Макросы, соответствующие функциям выделения памяти в стиле C.
// С ними будет легче работать, если создать для них оболочки,
// позволяющие не запоминать, какие структуры BSMDVINFO
// передавать в каждую функцию.
#define MEMDEBUG_MALLOC(lpBSMDVINFO , nSize)
    _malloc_dbg ( nSize ,
                  ((LPBSMDVINFO)lpBSMDVINFO)->dwValue ,
                  __FILE__ ,
                  __LINE__ )
#define MEMDEBUG_REALLOC(lpBSMDVINFO , pBlock , nSize)
    _realloc_dbg( pBlock ,
                  nSize ,
                  ((LPBSMDVINFO)lpBSMDVINFO)->dwValue ,
                  __FILE__ ,
                  __LINE__ )
#define MEMDEBUG_EXPAND(lpBSMDVINFO , pBlock , nSize )
    _expand_dbg( pBlock ,
                  nSize ,
                  ((LPBSMDVINFO)lpBSMDVINFO)->dwValue ,
                  __FILE__ ,
                  __LINE__ )
#define MEMDEBUG_FREE(lpBSMDVINFO , pBlock)
    _free_dbg ( pBlock ,
                ((LPBSMDVINFO)lpBSMDVINFO)->dwValue )
#define MEMDEBUG_MSIZEL(lpBSMDVINFO , pBlock) \
    _msize_dbg ( pBlock , ((LPBSMDVINFO)lpBSMDVINFO)->dwValue )

// Макрос для вызова функции ValidateAllBlocks.

```

```

#define VALIDATEALLBLOCKS(x)    ValidateAllBlocks ( x )

#else                            // Макрос _DEBUG не определен.

#ifdef __cplusplus
#define DECLARE_MEMDEBUG(classname)
#define IMPLEMENT_MEMDEBUG(classname)
#define MEMDEBUG_NEW new
#endif                          // __cplusplus

#define INITIALIZE_MEMDEBUG(lpBSMDVINFO , pfnD , pfnV )

#define MEMDEBUG_MALLOC(lpBSMDVINFO , nSize) \
        malloc ( nSize )
#define MEMDEBUG_REALLOC(lpBSMDVINFO , pBlock , nSize) \
        realloc ( pBlock , nSize )
#define MEMDEBUG_EXPAND(lpBSMDVINFO , pBlock , nSize) \
        _expand ( pBlock , nSize )
#define MEMDEBUG_FREE(lpBSMDVINFO , pBlock) \
        free ( pBlock )
#define MEMDEBUG_MSIZELP(lpBSMDVINFO , pBlock) \
        _msize ( pBlock )

#define VALIDATEALLBLOCKS(x)

#endif                          // _DEBUG

#ifdef __cplusplus
}
#endif                          // __cplusplus

#endif                          // _MEMDUMPERVALIDATOR_H

```

Использование MemDumperValidator в программах C++

К счастью, настроить класс C++ для его обработки MemDumperValidator'ом довольно просто. Для этого нужно добавить перед объявлением нужного класса директиву `#pragma push_macro ("new")` и отменить определение `new`. После объявления надо восстановить определение `new`, применив директиву `#pragma pop_macro ("new")`. В объявлении класса C++ просто укажите макрос `DECLARE_MEMDEBUG` с именем класса в качестве параметра. Этот макрос чем-то похож на «магические» макросы MFC: он тоже расширяется в пару объявлений данных и метода. Изучая листинг 17-2, вы заметите шесть встраиваемых функций для `new` и `delete`, обрабатывающих любой тип вызова этих операторов с синтаксисом размещения. Если какой-то из этих операторов определен в вашем классе, извлеките код из расширенных операторов и поместите его в операторы вашего класса.

В файле реализации класса C++ нужно указать макрос `IMPLEMENT_MEMDEBUG` с именем класса в качестве параметра. Этот макрос подготавливает статическую переменную для вашего класса. Макросы `DECLARE_MEMDEBUG` и `IMPLEMENT_MEMDEBUG` расширяются

только в отладочных компоновках, поэтому их не нужно включать в блок условной компиляции.

Указав оба макроса в корректном месте, вам нужно реализовать два метода, которые будут записывать дампа и проверять память вашего класса. Прототипы этих методов указаны чуть ниже. Очевидно, что их нужно заключать в блок условной компиляции, чтобы они не компилировались в заключительных компоновках.

```
static void ClassDumper ( const void * pData ) ;  
static void ClassValidator ( const void * pData,  
                           const void * pContext ) ;
```

Параметр `pData`, одинаковый для обоих методов, — это указатель на блок памяти объекта. Для получения готового к использованию указателя вам нужно только привести `pData` к типу указателя на ваш класс. Что бы вы ни делали при записи дампа или проверке памяти, рассматривайте значение `pData` как супер-только-для-чтения, или вы с легкостью внесете в свой код столько ошибок, сколько собирались предотвратить. Второй параметр метода `ClassValidator` — `pContext` — это параметр контекста, передаваемый вами в функцию `ValidateAllBlocks`. Подробнее о функции `ValidateAllBlocks` см. раздел «Глубокая проверка».

Что до реализации метода `ClassDumper`, то я могу дать лишь два совета. Во-первых, старайтесь использовать макросы `_RPTn` и `_RPTFn` библиотеки `DCRT`, чтобы ваш форматированный вывод записывался в дамп там же, где и остальная информация библиотеки `DCRT`. Во-вторых, завершайте свой вывод комбинацией «возврат каретки/перевод строки» (CR/LF), потому что макросы библиотеки `DCRT` не выполняют форматирования.

Подключение функций записи дампа и проверки памяти для класса C++ кажется почти тривиальным. А структуры данных C, которые вам также хотелось бы записывать в понятные и удобные дампы? Увы, их обработка требует большей работы.

Использование `MemDumperValidator` в программах C

Вы недоумеваете, зачем беспокоиться о поддержке C? Все просто: на этом языке написана масса используемых мной и вами программ. Как хотите, но некоторые из этих приложений и модулей тоже работают с памятью.

Чтобы использовать `MemDumperValidator` в программе C, нужно сначала объявить структуру `BSMDVINFO` для каждого типа памяти, который вы хотите проверять и записывать в дамп. Макросы C++ объявляют методы записи дампа и проверки памяти автоматически, однако в C кое-что придется сделать самостоятельно. Помните: все макросы, про которые я здесь говорю, должны получать указатель на специфическую структуру `BSMDVINFO`.

Прототипы функций записи дампа и проверки памяти C аналогичны прототипам методов C++ — нет только ключевого слова `static`. Как и при объявлении уникальных структур `BSMDVINFO` для блоков памяти, реализацию всех функций записи дампа и проверки C можно поместить в один файл.

Прежде чем вы начнете выделять, записывать и проверять память в программах C, вы должны сообщить расширению `MemDumperValidator` о подтипе клиентского блока и функциях записи дампа и проверки памяти. Эта информация пе-

редается расширению `MemDumperValidator` при помощи макроса `INITIALIZE_MEM_DEBUG`, который в качестве параметров принимает указатель на специфическую структуру `BSMDVINFO` и указатели на функции записи дампа и проверки. Вам нужно будет выполнять этот макрос перед выделением любого блока памяти соответствующего типа.

Наконец (и в этом смысле работать с памятью в C++ гораздо легче, чем в C), для выделения, освобождения, перераспределения, расширения или получения размера блока вы должны использовать набор макросов, передающих значение блока нужной функции работы с памятью. Так, если ваша структура `BSMDVINFO` называется `stdvBlockInfo`, нужно выделить блоки памяти в программе C:

```
MEMDEBUG_MALLOC ( &stdvBlockInfo , sizeof ( x ) ) ;
```

В конце листинга 17-2 содержатся все макросы для функций работы с памятью языка C. Можно запомнить структуры `BSMDVINFO` для каждого типа выделения памяти, но это непрактично, поэтому для обработки структур `BSMDVINFO` лучше написать макросы-оболочки — тогда вам нужно передавать в свои макросы-оболочки только обычные параметры функций работы с памятью.

Глубокая проверка

Польза записи дампов при помощи `MemDumperValidator` несомненна, тогда как назначение метода проверки не столь очевидно, пусть даже он позволяет выполнять глубокую проверку блока памяти. Зачастую функция проверки может быть даже пустой, если класс содержит только несколько переменных-строк. И все же функция проверки памяти все равно может оказаться бесценной, предоставляя великолепные отладочные возможности. Одна из причин того, что я начал использовать глубокую проверку, заключалась в получении второго уровня проверки данных для набора разработанных мной базовых классов. Хотя функция проверки не должна заменять вам старую добрую проверку параметров и вводимой информации, она может предоставить дополнительное подтверждение корректности данных. На основе глубокой проверки можно создать и вторую линию обороны против записи информации по случайным адресам памяти.

Самый очевидный способ использования функции проверки — контроль сложных структур данных после проведения над ними каких-либо операций. Я, например, как-то попал в сложную ситуацию, когда из-за ограничений памяти мне нужно было сделать так, чтобы две отдельных ссылающихся на себя структуры данных работали с одними и теми же объектами, находящимися в выделенной памяти. После заполнения структур большим набором данных я изучил при помощи функции проверки памяти отдельные блоки кучи и убедился в правильных значениях ссылок. Конечно, я мог написать код просмотра каждой структуры данных, но я знал, что любой написанный мной код будет потенциальным источником ошибок. Функция проверки памяти позволила мне применить для изучения выделенных блоков уже протестированный код и проверить структуры данных с разных позиций, так как блоки памяти располагались в порядке выделения, а не в отсортированном порядке.

Хотя в C настройка выделения памяти сложнее, чем в C++, применение функции проверки памяти в обоих языках одинаково. Для этого нужно только вызвать

макрос `VALIDATEALLBLOCKS`. Он расширяется в отладочных компоновках в вызов функции `ValidateAllBlocks`. Параметром макроса может быть любое значение, которое вы хотите передать зарегистрированным функциям проверки памяти. Раньше я указывал через этот параметр глубину выполняемой функцией проверки. Помните: `ValidateAllBlocks` передает это значение каждой зарегистрированной функции проверки, поэтому вам нужно согласовать его между всеми членами вашей группы.

Чтобы увидеть функции расширения `MemDumperValidator` в действии, изучите программу `Dump` (листинг 17-3; каталог `BUGSLAYERUTIL\TESTS\DUMP` на CD). `Dump` демонстрирует все действия, нужные для использования расширения. Хотя я не привел соответствующего кода, расширение `MemDumperValidator` хорошо работает с MFC, так как MFC вызывает любую зарегистрированную клиентскую функцию-ловушку записи дампа. `MemDumperValidator` позволяет вам получить в свое распоряжение лучшее обоих миров!

Листинг 17-3. DUMP.CPP

```
/*-----
Отладка приложений для Microsoft .NET и Microsoft Windows
Copyright © 1997-2003 John Robbins - All rights reserved.
-----*/

#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include <tchar.h>
#include "BugslayerUtil.h"

#pragma push_macro ( "new" )
#ifdef new
#undef new
#endif

class TestClass
{
public:
    TestClass ( void )
    {
        _tcscopy ( m_szData , _T ( "TestClass constructor data!" ) );
    }
    ~TestClass ( void )
    {
        m_szData[ 0 ] = _T ( '\\0' );
    }

    // Объявления механизмов отладки памяти для классов C++.
    DECLARE_MEMDEBUG ( TestClass );
private:
    TCHAR m_szData[ 100 ];
};
```

см. след. стр.

```

} ;

#pragma pop_macro ( "new" )

// Этот макрос создает статическую структуру BSMDVINFO.
IMPLEMENT_MEMDEBUG ( TestClass ) ;

// Методы, которые вы должны реализовать
// для записи дампов и проверки памяти.
#ifdef _DEBUG
void TestClass::ClassDumper ( const void * pData )
{
    TestClass * pClass = (TestClass*)pData ;
    _RPT1 ( _CRT_WARN
            " TestClass::ClassDumper : %S\n"
            , pClass->m_szData ) ;
}
void TestClass::ClassValidator ( const void * pData
                                , const void *
                                )
{
    // Выполняйте здесь проверку данных.
    TestClass * pClass = (TestClass*)pData ;
    _RPT1 ( _CRT_WARN
            " TestClass::ClassValidator : %S\n"
            , pClass->m_szData ) ;
}
#endif

typedef struct tag_SimpleStruct
{
    TCHAR szName[ 256 ] ;
    TCHAR szRank[ 256 ] ;
} SimpleStruct ;

// Функции записи дампа и проверки памяти для простых строк.
void DumperOne ( const void * pData )
{
    _RPT1 ( _CRT_WARN , " Data is : %S\n" , pData ) ;
}

void ValidatorOne ( const void * pData , const void * pContext )
{
    // Выполняйте здесь проверку данных строки.
    _RPT2 ( _CRT_WARN
            , " Validator called with : %s : 0x%08X\n"
            , pData
            , pContext ) ;
}

// Функции записи дампа и проверки памяти для структуры.
void DumperTwo ( const void * pData )

```



```

{
    _RPT2 ( _CRT_WARN
        " Data is Name : %S\n"
        "      Rank : %S\n"
        ((SimpleStruct*)pData)->szName ,
        ((SimpleStruct*)pData)->szRank  ) ;
}

void ValidatorTwo ( const void * pData , const void * /*pContext*/ )
{
    // Выполняйте здесь проверку полей структур.
    _RPT2 ( _CRT_WARN
        " Validator called with :\n"
        " Data is Name : %s\n"
        "      Rank : %s\n"
        ((SimpleStruct*)pData)->szName ,
        ((SimpleStruct*)pData)->szRank  ) ;
}

// К сожалению, функции C нуждаются в собственных структурах
// BSMDVINFO. При работе над реальными программами вам следует
// определять эти структуры как внешние ссылки и создавать
// для макросов MEMDEBUG собственные макросы-оболочки.
static BSMDVINFO g_dvOne ;
static BSMDVINFO g_dvTwo ;

void main ( void )
{
    _tprintf ( _T ( "At start of main\n" ) ) ;

    // Инициализация механизмов отладки памяти для типа 1.
    INITIALIZE_MEMDEBUG ( &g_dvOne , DumperOne , ValidatorOne ) ;
    // Инициализация механизмов отладки памяти для типа 2.
    INITIALIZE_MEMDEBUG ( &g_dvTwo , DumperTwo , ValidatorTwo ) ;

    // Выделение памяти для объекта C++ при помощи
    // оператора new, определенного в макросе MEMDEBUG.
    TestClass * pstClass ;
    pstClass = new TestClass ( ) ;

    // Выделение памяти для двух типов C.
    TCHAR * p = (TCHAR*)MEMDEBUG_MALLOC ( &g_dvOne , 20 ) ;
    _tcscpy ( p , _T ( "VC VC" ) ) ;

    SimpleStruct * pSt =
        (SimpleStruct*)MEMDEBUG_MALLOC ( &g_dvTwo ,
                                         sizeof ( SimpleStruct ) ) ;

    _tcscpy ( pSt->szName , _T ( "Pam" ) ) ;
    _tcscpy ( pSt->szRank , _T ( "CINC" ) ) ;
}

```

```
// Проверка всех блоков списка.
VALIDATEALLBLOCKS ( NULL ) ;

_tprintf ( _T ( "At end of main\n" ) ) ;

// Дамп каждого блока создается при проверке утечек памяти.

}
```

Реализация MemDumperValidator

Реализация функций MemDumperValidator оказалась довольно простой. Первая неожиданная проблема, с которой я должен был справиться, была в том, что в библиотеке DCRT не был документирован способ получения значений блоков памяти функциями-ловушками. Функциям-ловушкам передается только указатель на данные пользователя, а не на весь блок памяти, выделяемый библиотекой DCRT. К счастью, в исходном коде библиотеки DCRT я нашел точный механизм выделения блоков памяти. Каждый блок памяти выделяется как структура `_CrtMemBlockHeader`, определенная в файле `DBGINT.H`.

В файле `DBGINT.H` есть еще макросы доступа к `_CrtMemBlockHeader` через указатель на данные пользователя и доступа к данным пользователя через указатель `_CrtMemBlockHeader`. Чтобы получить эту информацию, я скопировал структуру `_CrtMemBlockHeader` и макросы в заголовочный файл `CRTDBG_INTERNALS.H` (листинг 17-4). Создание копии определения структуры — не лучший метод, так как определение может измениться, но тут все в порядке, поскольку структура `_CrtMemBlockHeader` не изменялась в библиотеке DCRT, начиная с Visual C++ 4, и все же это не значит, что она не изменится в будущих версиях Visual C++. Если вы собираетесь применять MemDumperValidator, вы должны будете следить за появлением всех пакетов обновлений и основных версий компилятора и проверять, не изменились ли в них внутренние структуры данных.

Листинг 17-4. CRTDBG_INTERNALS.H

```
/*-----
Отладка приложений для Microsoft .NET и Microsoft Windows
Copyright (c) 1997-2003 John Robbins - All rights reserved.
-----*/

#ifndef _CRTDBG_INTERNALS_H
#define _CRTDBG_INTERNALS_H
#define nNoMansLandSize 4

typedef struct _CrtMemBlockHeader
{
    struct _CrtMemBlockHeader * pBlockHeaderNext    ;
    struct _CrtMemBlockHeader * pBlockHeaderPrev    ;
    char *                      szFileName          ;
    int                          nLine               ;
}
```

```

size_t      nDataSize      ;
int         nBlockUse      ;
long        lRequest       ;
unsigned char gap[nNoMansLandSize] ;
/* после чего располагаются:
 * unsigned char      data[nDataSize];
 * unsigned char      anotherGap[nNoMansLandSize];
 */
} _CrtMemBlockHeader;

#define pbData(pblock) ((unsigned char *) \
                        ((_CrtMemBlockHeader *)pblock + 1))
#define pHdr(pblock) (((_CrtMemBlockHeader *)pblock)-1)

#endif      // _CRTDBG_INTERNALS_H

```

Если вам удобнее работать с DBGINT.H напрямую, можете заменить определение структуры в файле CRTDBG_INTERNALS.H директивой `#include DBGINT.H`. При этом вам понадобится добавить выражение «\$(VCInstallDir)VC7\CRT\SRC» в системную переменную среды INCLUDE и список включаемых файлов, для доступа к которому нужно открыть диалоговое окно Options (свойства), папку Projects (проекты) и выбрать страницу свойств VC++ Directories (каталоги VC++). Не все программисты устанавливают исходный код библиотеки CRT, хотя делать это следовало бы, поэтому я решил включить определение структуры непосредственно.

`_CrtMemBlockHeader` также позволяет извлечь более подробную информацию из структур `_CrtMemState`, заполняемых функцией `_CrtMemCheckpoint`, потому что первый элемент в `_CrtMemState` — указатель на `_CrtMemBlockHeader`. Надеюсь, в следующую версию библиотеки DCRT войдут настоящие функции доступа к информации о блоке памяти.

Просматривая исходный код в файле MEMDUMPERVALIDATOR.CPP из проекта BUGSLAYERUTIL.DLL (он находится на CD), вы заметите, что для внутреннего управления памятью я использовал простые API-функции семейства `HeapCreate`. Я сделал это, поскольку функции создания дампа и функции-ловушки, применяемые для работы с библиотекой DCRT, были бы реентерабельными при использовании функций стандартной библиотеки. Заметьте, что я не имею в виду многопоточную реентерабельность. Если бы моя ловушка выделяла память при помощи `malloc`, она была бы реентерабельной, потому что ловушки вызываются при каждом выделении памяти.

Инициализация и завершение в программах C++

Завершив реализацию `MemDumperValidator` и начав его тестирование, я с удовлетворением отметил, что все работает так, как было запланировано. Однако при рассмотрении всех способов, которыми программа может выделять память в куче, я покрылся холодным потом. При выделении памяти статическими конструкторами могли возникнуть проблемы. Взглянув на первоначальный код `MemDumperValidator`, я обнаружил серьезный пробел в своей логике.

Иногда, хоть и не очень часто, память выделяется до достижения точки входа в программу. Поэтому мне нужно было гарантировать, что нужные флаги устанавливаются функцией `_CrtSetDbgFlag` до любого выделения памяти.

Работая над `MemDumperValidator`, я ни в коем случае не хотел, чтобы вы вызывали до использования библиотеки некоторую функцию инициализации. Нам хватает проблем со структурами `BSMDVINFO` при программировании на C. Я хотел сделать `MemDumperValidator` как можно более автоматизированным, чтобы работать с ним было удобно большинству программистов.

К моей радости, замешательство не продлилось слишком долго, потому что я вспомнил о директиве `#pragma init_seg`, благодаря которой можно управлять порядком инициализации и уничтожения статических значений. Эта директива может принимать значения `compiler`, `lib`, `user`, `section name` и `funcname`. Важными являются первые три.

Значение `compiler` зарезервировано для компиляторов Microsoft; все объекты этой группы создаются первыми, а уничтожаются последними. Объекты, отмеченные как `lib`, создаются во вторую очередь, а уничтожаются предпоследними. Наконец, отмеченные как `user` создаются последними, а уничтожаются первыми.

Так как код `MemDumperValidator` должен инициализироваться до вашего кода, я мог просто указать `lib` в директиве `#pragma init_seg` и покончить со всем этим. Однако при создании своих библиотек вы также отмечаете их как сегменты `lib` (и правильно), поэтому мне нужен был другой способ инициализации своего кода в первую очередь. Чтобы справиться с этим непредвиденным обстоятельством, я указываю в директиве `#pragma init_seg` значение `compiler`. Хотя при инициализации сегментов всегда нужно следовать правилам, применение в отладочном коде значения `compiler` вполне безопасно.

Описанная идея инициализации работает только в коде C++, поэтому в `MemDumperValidator` входит специальный статический класс `AutoMatic`, который просто вызывает функцию `_CrtSetDbgFlag`. Я вынужден был пойти на все это, так как это единственный способ установки флагов DCRT до инициализации любых других библиотек. Кроме того, как вы увидите ниже, для преодоления некоторых ограничений проверки утечек памяти, свойственных библиотеке DCRT, я должен был реализовать кое-какие специфические действия в деструкторе класса. Пусть `MemDumperValidator` имеет интерфейс C, но я все равно воспользовался преимуществами C++ для инициализации этого расширения и своевременного приведения его в рабочее состояние.

И куда же подевались все сообщения об утечках памяти?

Наконец, я справился со всеми проблемами инициализации и заставил `MemDumperValidator` работать. Я был доволен всем за одним исключением: когда программа, вызывавшая утечку памяти, завершала работу, я не видел красиво отформатированных данных, выводимых моими функциями записи дампов. Вместо этого отображались стандартные старые дампы библиотеки DCRT. Я отследил «пропавшие» отчеты об утечках памяти и с удивлением обнаружил, что функции завершения библиотеки DCRT вызывали `_CrtSetDumpClient` с параметром `NULL`, аннулируя, таким образом, перед вызовом `_CrtDumpMemoryLeaks` мою ловушку записи дам-

пов. Я огорчился, но скоро понял, что завершающую проверку утечек памяти я должен был выполнять сам. Подходящее место для этого у меня уже имелось.

Выше я говорил, что для инициализации класса `AutoMatic` до вашего кода и вызова его деструктора после вашего кода я использовал директиву `#pragma init_seg(compiler)`, поэтому мне нужно было просто проверить в деструкторе утечку памяти и отключить после этого флаг `_CRTDBG_LEAK_CHECK_DF`, чтобы библиотека DCRT не генерировала собственный отчет. Этот подход имеет единственный недостаток: при компоновке программы с ключом `/NODEFAULTLIB` вы должны гарантировать, что выбранная вами библиотека CRT компоуется раньше `BUGSLAYERUTIL.LIB`. Библиотеки CRT не подчиняются директиве `#pragma init_seg(compiler)`, вследствие чего нет никакой гарантии, что данные `BUGSLAYERUTIL.LIB` будут инициализироваться первыми и уничтожаться последними, а значит, вы сами должны позаботиться о правильном порядке компоновки.

Очищение всех установленных ловушек записи дампа библиотекой DCRT не лишено смысла. Если бы ваша ловушка записи дампа использовала какие-то функции CRT, такие как `printf`, она могла бы нарушить завершение вашей программы, потому что во время вызова `_CrtDumpMemoryLeaks` библиотека находится в середине процесса прекращения своей работы. Если вы следуете указанным правилам и всегда компоуете свою программу сначала с библиотекой DCRT и только потом со всеми остальными библиотеками, все будет в порядке, потому что функции `MemDumperValidator` отключаются до завершения работы библиотеки DCRT. Тем не менее для избежания проблем используйте в своих функциях записи дампов только макросы `_RPTn` и `_RPTFn`, потому что `_CrtDumpMemoryLeaks` работает только с этими макросами.

Использование MemStress

Пора добавить в вашу жизнь каплю стресса. Как хотите, но стресс может быть полезным. Увы, подвергнуть стрессу приложения Win32 сейчас гораздо сложнее, чем раньше. Во времена 16-разрядных ОС Windows мы могли выполнять наши приложения под управлением `STRESS.EXE`, полезной программы из SDK. Она позволяла вам мучить свое приложение всеми способами, в том числе отнимать у него дисковое пространство или пространство кучи интерфейса графических устройств (GDI) и расходовать описатели файлов. Даже ее значок был великолепен: слон, идущий по канату.

Чтобы испытать приложения Win32 в стрессовых условиях, можно установить ловушку для системы выделения памяти библиотеки DCRT и управлять выделением памяти. Расширение `MemStress` дает вам возможность испытать выделение памяти на языке С или С++ (написание кода расходования дискового пространства я оставил вам). Чтобы сделать `MemStress` простым в использовании, я написал при помощи Windows Forms интерфейсную часть, позволяющую точно указать условия, в которых вы хотели бы проверить свою программу.

Расширение `MemStress` позволяет форсировать неудачи выделения памяти, опираясь на различные критерии: для всех выделений памяти, при каждом *n*-ом выделении памяти, после выделения *x* байт, при запросе более *y* байт, для всех выделений памяти из исходного файла и из конкретной строки исходного файла. Кроме того, вы можете указать расширению `MemStress` выводить при каждом

выделении памяти окно, спрашивающее, хотите ли вы, чтобы это конкретное выделение памяти завершилось неудачей, а также установить флаги библиотеки DCRT, влияющие на выполнение вашего приложения. На CD находится программа MemStressDemo; этот написанный при помощи MFC пример позволяет экспериментировать с параметрами пользовательского интерфейса MemStress и увидеть соответствующие результаты, а еще выполняет функцию блочного теста для MemStress.

Использовать расширение MemStress относительно просто. Для этого вы должны включить в свою программу файл BUGSLAYERUTIL.H и вызвать макрос `MEMSTRESSINIT`, передав ему имя своей программы. Для прекращения работы ловушки выделения памяти служит макрос `MEMSTRESSTERMINATE`. Вы можете подключать и останавливать ловушку при работе своей программы сколько угодно.

После компиляции своей программы запустите пользовательский интерфейс MemStress, нажмите на кнопку Add Program (добавить программу) и введите то же имя, что вы указали в макросе `MEMSTRESSINIT`. Выбрав условия ошибки, нажмите на кнопку Save Program Settings (сохранение настроек программы) для сохранения конфигурации в файле MEMSTRESS.INI. После этого вы можете запускать свою программу и изучать ее поведение при неудачном выделении памяти.

К расширению MemStress следует относиться очень избирательно. Так, указав, чтобы неудачей завершались все выделения блоков памяти, превышающих 100 байт, и включив макрос `MEMSTRESSINIT` в функцию `InitInstance` своего приложения MFC, вы скорее всего нарушите работу MFC, так как она не сможет инициализироваться. Самые лучшие результаты вы получите, если ограничите работу MemStress ключевыми областями своего кода, чтобы их можно было протестировать по отдельности.

Основная часть реализации MemStress касается чтения и обработки файла MEMSTRESS.INI, в котором хранятся все параметры для отдельных программ. С точки зрения библиотеки DCRT, особую важность представляет вызов функции `_CrtSetAllocHook` при инициализации MemStress, потому что он устанавливает функцию-ловушку выделения памяти `AllocationHook`. Если ловушка выделения памяти возвращает `TRUE`, обработка запроса на выделение памяти может продолжаться. Возвращая `FALSE`, ловушка выделения памяти может заставить библиотеку DCRT провалить запрос на выделение памяти. Библиотека DCRT предъявляет к ловушке выделения памяти только одно строгое требование: если тип блока, определяемый параметром `nBlockUse`, имеет значение `_CRT_BLOCK`, функция-ловушка должна возвращать `TRUE`, чтобы выделение могло увенчаться успехом.

Ловушка выделения памяти получает управление при вызове любого типа функции выделения памяти. Эти типы, передаваемые ловушке в первом ее параметре, могут иметь значения `_HOOK_ALLOC`, `_HOOK_REALLOC` и `_HOOK_FREE`. Если моя ловушка `AllocationHook` получает тип `_HOOK_FREE`, я пропускаю весь код, определяющий успешную или неудачную обработку запроса на выделение памяти. При получении типов `_HOOK_ALLOC` и `_HOOK_REALLOC` моя функция `AllocationHook` выполняет ряд операторов `if`, определяя, выполняется ли какое-нибудь из условий неудачи. Если хотя бы одно из условий выполняется, я возвращаю `FALSE`.

Интересные проблемы с MemStress

При тестировании MemStress на консольном примере все работало отлично, и я был очень доволен. Однако, закончив работу над программой MemStressDemo, основанной на MFC, я столкнулся с одной странной проблемой. Если я приказывал MemStress спрашивать меня, хочу ли я, чтобы выделение памяти провалилось, я слышал несколько звуковых сигналов, и MemStressDemo прекращала свою работу. Ошибка воспроизводилась при каждом запуске программы, но я никак не мог найти ее причин, что стало меня не на шутку раздражать.

После нескольких запусков я, наконец, получил информационное окно, однако оно находилось не в центре экрана, а в правом нижнем углу. Когда окна появляются в правом нижнем углу экрана, вы можете быть почти уверены в том, что столкнулись с ситуацией, в которой вызов API-функции `MessageBox` почему-то стал реентерабельным. Я предположил, что где-то в середине `MessageBox` вызывалась моя ловушка выделения памяти. Для проверки этой гипотезы я установил точку прерывания на первой команде `AllocationHook` и «перешагнул» (step over) через вызов `MessageBox`. Все подтвердилось: отладчик остановился на точке прерывания.

Я приступил к изучению стека и увидел, что вызов API-функции `MessageBox` почему-то проходил через MFC. Пробираясь через код и наблюдая за тем, что происходит, я попал в функцию `_AfxActivationWndProc` на строку, вызывавшую `CWnd::FromHandle`. Этот вызов приводил к выделению памяти, нужной для того, чтобы MFC могла создать `CObject`. Я был слегка удивлен тем, как я там оказался, однако комментарии в коде гласили, что `_AfxActivationWndProc` служит для обработки активизации диалоговых окон и их закрашивания в серый цвет. MFC использует ловушку приложенный компьютерной профессиональной подготовки (computer-based training (CBT) hook) для перехвата создания окон в адресном пространстве процесса. При создании нового окна — в моем случае простого информационного окна (message box) — MFC создает подкласс окна с его собственной оконной процедурой.

Когда я понял суть проблемы, то пришел в еще большее замешательство, потому что не знал, как с ней справиться. Поскольку реентерабельность имела место в одном потоке, я не мог использовать объект синхронизации, такой как семафор, потому что это привело бы к блокировке потока. Поразмыслив, я решил, что мне нужен флаг рекурсии, указывающий на реентерабельность `AllocationHook`, но он должен быть отдельным для каждого потока. У меня уже была критическая секция, защищающая `AllocationHook` от многопоточной реентерабельности.

Сформулировав проблему таким образом, я понял, что мне нужна только переменная в локальной памяти потока, которую я проверял бы в начале `AllocationHook`. Если бы ее значение превышало 0, это указывало бы на реентерабельность `AllocationHook` во время обработки `MessageBox`, и в этом случае я должен был бы немедленно покидать функцию. Я быстро реализовал динамичное решение на основе локальной памяти потока, и уровень моего беспокойства значительно снизился, так как все начало работать так, как я и планировал.

Я думал, что теперь все будет в порядке, но не тут то было. При тестировании кода, вызывавшего неудачу выделения памяти для конкретного файла и строки, имя исходного файла имело значение `NULL`, а номер строки — 0. Я писал программу MemStressDemo при помощи MFC и полагал, что она будет правильно использо-

вать функции выделения памяти CRTDBG.H для передачи в них исходного файла и номера строки. Увы, это было не так.

Я понял, что в начале STDAFX.H нужно указать определение `_CRTDBG_MAP_ALLOC`, а в конце включить файл CRTDBG.H. Конечно, стоило мне скомпилировать такой вариант программы, число ошибок, указывавших на переопределения и неожиданные константы, меня просто сразило. Как я только ни пытался заставить приложение MFC скомпилироваться и использовать правильные версии функций выделения памяти! Спустя некоторое время я обнаружил, что единственный приемлемый вариант решения этой проблемы состоял в извлечении определений из CRTDBG.H и включении их в файл STDAFX.H вручную. Чтобы в точности узнать, что именно вам нужно делать, можете изучить файл STDAFX.H программы Mem-StressDemo.

Кучи операционной системы

Возможно, вы думали, что я уже рассказал про систему куч все, однако есть еще один набор куч, про который вам непременно следует знать, — это кучи ОС. При запуске приложения под управлением отладчика Windows включает проверку кучи ОС. Это не отладочная куча стандартной библиотеки C — это куча Windows для тех куч, что создаются при помощи API-функции `HeapCreate`. Куча стандартной библиотеки C — отдельная сущность. Кучи ОС интенсивно используются процессами, например, для преобразования строк ANSI в формат Unicode при работе с функциями ANSI, поэтому вы можете видеть информацию о кучах ОС при нормальных операциях, вот почему их так важно рассмотреть. Если вы подключаете отладчик к своему приложению позднее, а не сразу начинаете выполнение программы под отладчиком, вы не активизируете проверку куч ОС. Очень часто меня спрашивают: «Вне отладчика моя программа работает отлично, но в отладчике она вызывает исключение пользовательской точки прерывания. Почему моя программа не работает?» Ответ: из-за проверки куч ОС.

При включенной проверке куч ОС ваше приложение будет работать медленнее, потому что при вызове функций работы с кучей ОС будет выполнять соответствующую проверку. В число примеров к этой книге я включил программу `Heaper` (листинг 17-5), повреждающую кучу. Запустив `Heaper` в отладчике, вы увидите, что она дважды вызывает `DebugBreak` на первой функции `HeapFree`. Будет также выведена информация об ошибке, пример которой приведен ниже. Да, вывод останавливается на буквах «of a» и не отображает размер блока, что могло бы быть весьма полезным. Если вы запустите эту программу вне отладчика, она проработает до своего завершения безо всяких проблем.

```
HEAP[Heaper.exe]: Heap block at 00311E98 modified at 00311EAA past
requested size of a
```

Листинг 17-5. HEAPER.CPP — пример повреждения кучи Windows

```
void main(void)
{
    // Создание кучи операционной системы.
    HANDLE hHeap = HeapCreate ( 0 , 128 , 0 ) ;
```



```
// Выделение памяти для 10-байтового блока.
LPVOID pMem = HeapAlloc ( hHeap , 0 , 10 );

// Запись 12 байт в 10-байтовый блок (запись после конца блока).
memset ( pMem , 0xAC , 12 );

// Выделение нового 20-байтового блока.
LPVOID pMem2 = HeapAlloc ( hHeap , 0 , 20 );

// Запись данных на 1 байт до начала второго блока.
char * pUnder = (char *) ( (DWORD_PTR)pMem2 - 1 );
*pUnder = 'P' ;

// Освобождение первого блока. Этот вызов HeapFree приведет
// к срабатыванию точки прерывания в отладочном коде кучи ОС.
HeapFree ( hHeap , 0 , pMem );

// Освобождение второго блока. Заметьте: этот
// вызов не приводит к сообщениям о проблеме.
HeapFree ( hHeap , 0 , pMem2 );

// Освобождение несуществующего блока.
// Этот вызов также не приводит к проблемам.
HeapFree ( hHeap , 0 , (LPVOID)0x1 );

HeapDestroy ( hHeap );

}
```

Если вы используете собственные кучи ОС или хотите, чтобы приложение включило проверку куч ОС при выполнении вне отладчика, вы можете установить дополнительные флаги для получения более подробного диагностического вывода. Небольшая утилита GFLAGS.EXE из пакета Debugging Tools for Windows поможет установить некоторые глобальные флаги, которые Windows проверяет при первом запуске приложения. На рис. 17-1 показаны параметры GFLAGS.EXE для HEAPER.EXE, программы из листинга 17-5. Многие из параметров System Registry (системный реестр) и Kernel Mode (режим ядра) глобальны, поэтому вам нужно быть чрезвычайно внимательным при их изменении, так как это может оказать значительное влияние на производительность системы или вообще нарушить ее работу. Изменять параметры Image File Options (настройки файла образа), показанные на рис. 17-1, гораздо безопасней, потому что они ограничены только одним исполняемым файлом. Кстати, несмотря на всю полезность GFLAGS.EXE, для проверки повреждений кучи можно использовать и средство Application Verifier, о котором я расскажу ниже.

Наконец, чтобы завершить рассказ про GFLAGS.EXE, я хочу обратить ваше внимание на один очень полезный параметр Show Loader Snaps (показывать снимки загрузчика). Если вы отметите этот флажок и запустите свою программу, вы увидите, где Windows загружает DLL и как она собирается настраивать импортируе-

мые функции для вашего приложения, что называется деланием снимков (snapping). Если вам нужно точно узнать, что делает загрузчик Windows при загрузке приложения (если у вас есть проблема), этот флажок следует отключить. Подробнее о снимках загрузчика см. раздел «Under the Hood» Мэтта Питрека (Matt Pietrek) в «Microsoft Systems Journal» (1999, сентябрь).

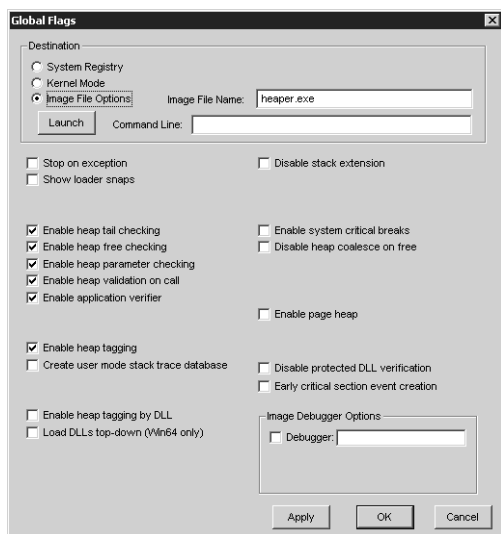


Рис. 17-1. Параметры `GFLAGS.EXE` для программы `HEAPER.EXE`

Советы по отслеживанию проблем с памятью

Теперь вы должны хорошо понимать доступные вам отладочные системы памяти, поэтому я могу перейти к методам нахождения тех проблем с памятью, которые случаются только в готовых системах и не желают показываться в отладчике.

Обнаружение записи в неинициализированную память

Нет ничего хуже ошибки, которая происходит из ниоткуда и не соответствует никакой ветви выполнения кода. Если вы попали в такую ситуацию, она объясняется скорее всего записью в неинициализированную память, также известной как запись по случайному адресу (wild write). Причина таких ошибок таится в неинициализированном указателе, который по воле случая указывает на корректную область памяти. Обычно это происходит со стековыми указателями или, иначе говоря, с локальными переменными. Так как при выполнении вашей программы стек постоянно изменяется, ничто не сможет сказать вам, куда указывает неинициализированный указатель, вот почему он может казаться случайным.

Когда нас просят помочь решить подобную коварную проблему, мы всегда встречаем совершенно растерянных программистов, утверждающих, что попробовали буквально все способы ее обнаружения. Так как они уже проделали «все», они отчаянно желают узнать, что мы наколдуюм. Я отвечаю на это, что мы собираемся применить запатентованный и зарегистрированный Магический Способ

Отладки Неинициализированной Памяти по Методике Ниндзя (MNUMDT). Все собираются посмотреть на это и ерзают в креслах, ожидая чуда, особенно когда я упоминаю, что MNUMDT сработает, только если мне помогут двое наименее опытных разработчиков группы.

В этот момент ко мне подходят двое младших программистов, всем своим видом показывающие, что намерены с достоинством нести свою ношу, и я описываю им MNUMDT:

- каждый из нас берет на себя по одной трети кода;
- каждый должен внимательно прочитать каждую строку кода, отыскивая все объявления указателей;
- при нахождении объявления неинициализированного указателя мы инициализируем его значением `NULL`;
- при обнаружении каждого вызова выделения памяти не для классов мы добавляем после него вызов `memset` или `ZeroMemory` для обнуления памяти;
- после любого освобождения памяти мы снова присваиваем указателю нулевое значение;
- при нахождении вызова `memset` или операции копирования строк мы должны проверить, что каждая операция правильно подсчитывает размер блока памяти;
- каждая переменная члена класса должна инициализироваться в конструкторе(ax).

Выслушивая правила MNUMDT, парни готовы выбежать из комнаты, но я не даю им сделать этого. Если вы думаете, что это грубое насилие, вы абсолютно правы: так и есть.

Я имел дело с тысячами ошибок, вызванных неинициализированными указателями, и знаю, что тут не поможет никакая отладка. Вы просто потратите время. Отладка станет гораздо эффективнее, если перед ее началом выполнить только что описанные мной действия. Очень вероятно, что проблему вызывает один из указателей, которые вы проинициализируете. После этого программа не сможет исказить память и продолжить свое выполнение, а тут же потерпит крах, пытаясь записать данные по указателю `NULL`.

Некоторые думают, что это не сработает, но я могу привести сотни случаев, когда программисты неделями пытались отыскать проблему и не получали никаких результатов. Когда они обращались к нам, мы находили проблему за день или два. Иногда разработчики пытаются перехитрить сами себя, не желая прибегать к такому грубому подходу, однако он просто великолепен в подобных ситуациях. Почему я прибегаю к помощи наименее опытных программистов? Да потому, что они менее высокомерны, чем старшие разработчики, и, находясь перед лицом всех своих коллег, чрезвычайно ответственно подходят к возложенным на них задачам.

Нахождение записи данных после окончания блока

Запись данных после окончания блока памяти следует искать не только при помощи DCRT и таких средств, как `BoundsChecker` компании Compuware, но и при тестировании программы. Увы, не все программисты относятся к тестированию так же серьезно, как вы, поэтому вы все равно будете сталкиваться в их коде с этими мерзкими ошибками. Кроме того, некоторые случаи записи данных по оконча-

нии блока происходят только в наиболее напряженных условиях. Ситуация ухудшается еще и тем, что, когда такая ошибка происходит в заключительной компоновке, в конце блока памяти нет пустого пространства, что может привести к коварному искажению данных, которое неделями может оставаться незамеченным, пока не потерпит крах важное серверное приложение.

Один из недостатков проверки записи данных после блока памяти в DCRT в том, что о них сообщается, только когда операция с памятью выполнит соответствующую проверку. Было бы лучше, если б программа, допустившая такую ошибку, тут же прекращала свою работу. Программисты Microsoft также желали получить эту возможность, поэтому несколько лет назад они выпустили средство PageHeap.

PageHeap тесно взаимодействует с ОС и использует уникальный трюк для немедленного обнаружения записи после выделенного блока. Когда вы выделяете 16 байт, программа PageHeap на самом деле выделяет 8 кб! Сначала она выделяет 4-кбайтную страницу, наименьший блок памяти, к которому применяются права доступа. PageHeap предоставляет права на чтение этой страницы и ее запись. Сразу за страницей для чтения/записи PageHeap выделяет следующую страницу, отмечая ее как не имеющую доступа. PageHeap выполняет некоторые манипуляции с указателями и предоставляет вашей программе адрес, находящийся за 16 байт до конца первой страницы. Таким образом, когда вы попытаетесь записать данные в 17-ый байт, начиная от этого адреса, вы попадете на страницу с запрещенным доступом и немедленно вызовете нарушение доступа. Формат памяти, выделяемой PageHeap, показан на рис. 17-2.

Память для чтения/записи	Возвращаемый 16-байтный блок	Память, не имеющая доступа
0x00310000	0x00310FF0	0x00311000

Рис. 17-2. *Память, выделяемая программой PageHeap*

Как вы можете представить, PageHeap использует гораздо больше памяти, чем нужно. Однако эта цена не имеет значения, если вы сможете найти запись данных вне блока. Если вы работаете над крупным приложением, установите в тестовый компьютер как можно больше памяти. Можете «позаимствовать» пару модулей из компьютера своего начальника — все равно он этого не заметит.

Рассказывая о PageHeap, я обязательно должен упомянуть, что все возвращенные вам указатели выравниваются по границе 16 байт. Это значит, что если вы выделяете 10 байт, то для попадания на страницу с запрещенным доступом вы должны будете записать в память 7 дополнительных байт. Пусть это вас не смущает: при выходе за пределы блока памяти обычно выполняется запись не 1–2 байт, а довольно приличного их числа, поэтому на эффективности PageHeap это не скажется. Это также значит, что PageHeap заслуживает внимания, только когда вы работаете с заключительной компоновкой своей программы. Отладочная компоновка сама по себе включает дополнительное пространство до и после выделяемых блоков памяти, поэтому в таких случаях PageHeap ничего не обнаружит.

Пакет Application Compatibility Toolkit

Я мог бы привести вам огромное описание подключения PageHeap при помощи странного средства командной строки, встроенного в GFLAGS, но есть способ получше. Пакет Application Compatibility Toolkit (ACT) не только вносит функциональность PageHeap прямо в Visual Studio .NET, но и предоставляет некоторые прекрасные средства обнаружения ошибок, про которые вам следует знать. ACT предназначен в первую очередь для обеспечения выполнения программ под Microsoft Windows XP/Server 2003, но он также включает программу Application Verifier (AppVerifier). Она-то нам и нужна.

Пакет ACT вы можете установить с CD, прилагаемого к книге, или загрузить его последнюю версию с сайта <http://www.microsoft.com/windowsxp/appexperience/default.asp>. В документации говорится, что AppVerifier из версии ACT 2.6, доступной на момент написания этой книги, может работать под управлением Microsoft Windows 2000 SP3 и более поздних версий ОС, но я смог запустить эту программу только на Windows XP/Server 2003. Я так и не добился правильной работы AppVerifier с Windows 2000. Кроме того, некоторые из тестов и ошибок не выводят информации, хотя в документации утверждается обратное. В оставшейся части обсуждения я буду полагать, что вы выполняете AppVerifier под Windows XP/Server 2003 с правами администратора (это необходимое условие).

AppVerifier включает отдельный исполняемый файл (APPVERIFEXE) и надстройку (VSAPPVERIF.DLL). Надстройка AppVerifier, включенная в ACT версии 2.6, интегрируется в панель инструментов Debug среды Visual Studio .NET 2002, но не Visual Studio .NET 2003. К счастью, благодаря опыту, полученному при работе с надстройками в главе 9, я смог обнаружить, как заставить AppVerifier работать и во втором случае. Если вы собираетесь использовать более позднюю версию AppVerifier, вероятно, она интегрируется в Visual Studio .NET 2003 сама, так что вы можете пропустить описание следующих действий.

Установив ACT, откройте командную строку и перейдите в подкаталог <каталог установки ACT>\Applications. Зарегистрируйте DLL надстройки AppVerifier командой `REGSVR32 VSAPPVERIF.DLL`, чтобы внести в реестр информацию о компонентах COM. Далее вы должны сообщить об этой надстройке среде Visual Studio .NET 2003. В каталоге AppVerifierAddIn на CD находится .REG-файл AppVerifierAddInReg.reg. Чтобы выполнить его, дважды щелкните его значок в Windows Explorer или введите в командной строке выражение `REGEDIT AppVerifierAddInReg.REG`.

Если вы беспокоитесь о том, может ли перенос надстройки, написанной для предыдущей версии Visual Studio .NET, в более новую версию этой среды привести к неприятностям, я отвечу, что это возможно. Если бы надстройка была написана при помощи .NET, использование предыдущих версий CLR могло бы вызвать проблемы. Однако AppVerifier написана только на C++, поэтому вы ничем не рискуете. Я знаю это совершенно точно, так как я проверил VSAPPVERIF.DLL при помощи REGASM, и REGASM сообщил, что эта надстройка не является сборкой .NET. Конечно, я все же протестировал VSAPPVERIF.DLL и проверил все ее параметры, чтобы гарантировать полную безопасность. Запустив Visual Studio .NET с AppVerifier, не имея прав администратора, вы увидите странное окно сообщения об ошибке — «Installer Error» (ошибка программы установки) — с текстом «Error: insufficient

permissions to run this program. Administrator access needed» (ошибка: недостаточно прав для выполнения программы. Необходимы права администратора).

Как только вы установите АСТ или зарегистрируете AppVerifier вручную, в панель инструментов Debug среды Visual Studio .NET будет добавлено несколько новых кнопок, которых можно не заметить. Новый вид панели Debug см. на рис. 17-3. Вам нужно будет настроить панель инструментов Debug так, чтобы она присутствовала не только при отладке, поэтому установить параметры надстройки AppVerifier можно лишь перед началом отладки. Один из главных принципов работы AppVerifier состоит в том, что при возникновении проблемы она вызывает функцию `DebugBreak`, поэтому программу всегда следует выполнять под управлением отладчика. Встроив надстройку в Visual Studio .NET, вы избавитесь от мучений, связанных с Windows NT Symbolic Debugger (NTSD) или WinDBG.

Панель Debug

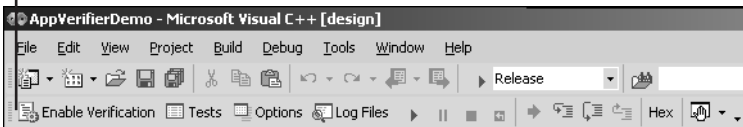


Рис. 17-3. Панель инструментов Debug среды Visual Studio .NET после правильного конфигурирования надстройки AppVerifier

При работе с AppVerifier нужно прежде всего подключить один недокументированный параметр, обеспечивающий более эффективное обнаружение ошибок памяти. Загрузив проект в Visual Studio .NET, щелкните на панели Debug новую кнопку Options (параметры) и установите в диалоговом окне Options флажок Use Full Page Heap (использовать кучу полных страниц), как показано на рис. 17-4. В самой по себе программе APPVERIFEXE этот параметр недоступен, но чем более тщательную проверку вы выполняете, тем лучше. На рис. 17-4 вы можете заметить, что я не устанавливал флажок Break In The Debugger After Each Logged Event (выходить в отладчик после регистрации каждого события). Установив его, вы будете останавливаться буквально каждые 15 наносекунд, когда надстройка AppVerifier будет сообщать о чем-нибудь.

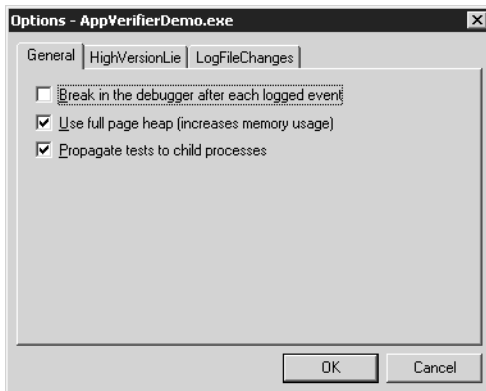


Рис. 17-4. Диалоговое окно Options надстройки AppVerifier в Visual Studio .NET

Если щелкнуть на панели инструментов Debug кнопку Tests (тесты), появится диалоговое окно AppVerifier Test Settings (параметры тестов AppVerifier), в котором вы можете указать, какие тесты выполнять (табл. 17-4). Кроме того, я подключаю все тесты, показанные на рис. 17-5. Пункты, указываемые в диалоговом окне AppVerifier Test Settings, относятся к отдельному процессу, поэтому их нужно устанавливать для каждого приложения, которое вы хотите выполнить под управлением AppVerifier.

Табл. 17-4. Описание тестов надстройки AppVerifier

Тест	Описание
Detect heap corruptions (обнаружение повреждений кучи)	Выполняет систематичную проверку кучи и добавляет после каждого выделенного блока памяти «защитную» страницу, нужную для обнаружения выхода за пределы блока.
Check lock usage (проверка механизмов блокировки)	Находит частые ошибки, связанные с критическими секциями.
Detect invalid handle usage (обнаружение неправильного использования описателей)	В АСТ 2.6 этот тест проверяет, не имеют ли описатели значения NULL/INVALID_HANDLE_VALUE и корректен ли параметр TLS (локальная память потока).
Check for adequate stack (проверка объема стека)	Проверяет наличие достаточного объема стека в службах Win32. Вероятно, вам не придется использовать этот тест.
Log start and stop (регистрация запуска и прекращения работы программы)	Этот параметр просто регистрирует информацию при запуске или прекращении работы приложения. Это помогает сделать журналы регистрации более понятными при изучении тестовых данных.
Checks system path usage (проверка использования системных путей)	Отслеживает попытки приложения получить информацию о пути к файлам и определяет, использует ли программа жестко закодированные пути или нестандартные способы получения этой информации.
Checks version handling (проверка обработки версий)	В прошлом многие приложения создавались только для конкретных версий Windows. Этот тест возвращает очень большой номер версии, когда приложение пытается определить, под управлением какой версии Windows оно выполняется.
Checks registry usage (проверка использования реестра)	Отслеживает работу приложения с системным реестром, проверяя, не выполняет ли оно какие-нибудь неуместные или опасные вызовы. Все обнаруженные проблемы записываются в журнал. Этот тест помогает гарантировать будущую совместимость программ, потому что он проверяет использование специфических разделов реестра, которые могут быть удалены или изменены в будущих версиях ОС Windows.
Logs changes to Windows File Protection files (регистрировать изменение файлов Windows File Protection)	Регистрирует попытки изменения файлов, защищенных механизмом Windows File Protection.
Logs DirectX file checks (регистрировать проверку файлов DirectX)	Регистрирует все попытки приложения выполнить проверку версий библиотек DirectX.

см. след. стр.

Табл. 17-4. Описание тестов надстройки AppVerifier *(продолжение)*

Тест	Описание
Logs registry changes (регистрировать изменения реестра)	Записывает в XML-файл все изменения, вносимые приложением в реестр.
Logs file system changes (регистрировать изменения файловой системы)	Записывает в XML-файл все изменения, вносимые приложением в файловую систему. Если программа попытается записать файлы в папки %windir% или Program Files, вы получите предупреждение.
Logs calls made to obsolete APIs (регистрировать вызовы устаревших API-функций)	Регистрирует все вызовы API-функций, отмеченные в Microsoft Platform SDK как устаревшие. При использовании этого теста изучайте в журнале только те записи, в которых встречаются ваши двоичные файлы. ОС сама выполняет достаточно много вызовов устаревших функций.
Logs installations of kernel-mode drivers (регистрировать установку драйверов режима ядра)	Регистрирует все попытки программы установить драйвер режима ядра.
Logs potential security issues (регистрировать потенциальные проблемы с безопасностью)	Определяет и регистрирует потенциальные проблемы с безопасностью при использовании дескрипторов безопасности DACL со значением NULL и вызовов API-функций создания процесса.
RPC Checks (проверки RPC)	Определяет некорректное использование RPC. Применяется только при работе с Windows Server 2003. Вероятно, вам никогда не понадобится включать этот тест.

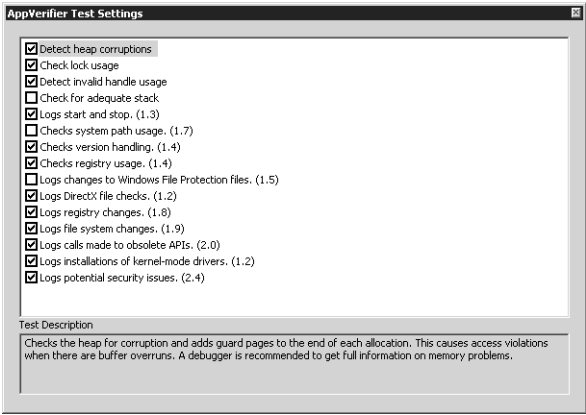


Рис. 17-5. Диалоговое окно AppVerifier Test Settings

Чтобы подключить AppVerifier для вашего приложения, нужно всего лишь нажать кнопку Enable Verification (включить проверку). Определять тесты и параметры нужно до нажатия Enable Verification, потому что после этого кнопки Tests и Options становятся неактивными. В пакет АСТ входит демонстрационная программа, но ее действия непонятны, а исходного кода нет, поэтому мне пришлось быстро написать небольшую программу AppVerifierDemo, которую вы можете использовать для изучения многих ошибок, которые находит AppVerifier. Чтобы увидеть нахождение ошибок в действии откройте проект AppVerifierDemo (он есть на CD).

Некоторые из кнопок — например, двойная инициализация критической секции — ошибок не вызывают: АСТ 2.6 документирует эту ошибку, но на самом деле не генерирует ее. Уделите особое внимание разделу PageHeap Errors (ошибки PageHeap), который покажет вам, как прекрасно PageHeap справляется с обнаружением выхода за пределы блока и других неприятных проблем с памятью.

Выполнив свое приложение пару раз, вы можете изучить журналы, сохраненные AppVerifier. Хотя большинство ошибок, обнаруженных надстройкой AppVerifier, приводит к немедленному нарушению доступа, некоторые появляются только в журнале, который открывается щелчком кнопки Log Files (файлы журналов) на панели инструментов Debug. Изучая журналы, убедитесь, что установлен параметр Show All (показывать все), иначе вы увидите не всю информацию.

Потрясающие ключи компилятора

Как я упоминал в главе 2, компилятор Microsoft Visual C++ .NET поддерживает новые ключи, которые необходимо устанавливать всегда. В этом разделе я опишу ключи /RTCx и /GS и объясню, почему они так важны.

Ключи проверки ошибок в период выполнения

Даже если бы единственной новой функцией, добавленной в компилятор Visual C++ .NET, были ключи /RTCx (Run-Time Error Checks, проверка ошибок во время выполнения), я бы всем говорил, что переход на эту версию просто обязателен. Как можно догадаться по названию, четыре ключа RTC следят за вашим кодом и вызывают отладчик, если при выполнении программы возникают определенные ошибки. На рис. 17-6 показана ошибка, возникшая в результате того, что какой-то код выполнил запись после окончания локальной переменной. Как вы можете увидеть по информационному окну, показана и искаженная локальная переменная. К тому же рисунок ничего не говорит о том, что информационное окно появляется в конце функции, в которой произошла ошибка, благодаря чему поиск и исправление проблемы становятся тривиальной задачей! Между прочим, ключи RTC допускаются применять только в отладочных компоновках.

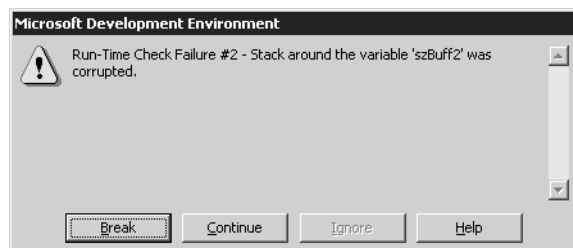


Рис. 17-6. Сообщение об ошибке при установленном ключе /RTCx

Первый ключ, /RTCs, чье сообщение об ошибке показано на рис. 17-6, выполняет в период выполнения программы разнообразные виды проверки стека. Во-первых, он инициализирует все локальные переменные ненулевыми значениями. Это помогает находить те неприятные проблемы заключительных компоновок, которые не проявляются в отладочных компоновках, например, наличие в стеке

неинициализированного указателя. Все локальные переменные получают значения 0xCC, что соответствует коду операции INT 3 (точка прерывания).

Во-вторых, ключ /RTCs выполняет проверку указателя стека при каждом вызове функции, обнаруживая несоответствия соглашений вызова. Так, если функция объявлена с соглашением `__cdecl`, но экспортируется как `__stdcall`, при возвращении из функции `__stdcall` стек будет поврежден. Если вы уже давно следите за ключами компиляторов, вы уже догадались, что первые два варианта проверки стека ключом /RTCs аналогичны функции ключа /GZ в Microsoft Visual C++ 6.

К нашей всеобщей радости, Microsoft расширила ключ /RTCs, и теперь он также проверяет запись данных до начала и после границ многобайтовых локальных переменных, таких как массивы. Он делает это, добавляя 4 байта к началу и концу этих массивов и проверяя в конце функции, сохранили ли эти байты первоначальное значение 0xCC. Локальная проверка работает со всеми многобайтовыми локальными переменными, кроме тех, что требуют от компилятора добавления дополнительных байтов. Обычно дополнительные байты добавляются только при использовании директивы `__declspec(align)`, ключа /Zp, выравнивающего члены структуры, или директивы `#pragma pack(n)`.

Второй ключ, /RTCu, проверяет использование ваших переменных и выводит предупреждение, если вы вызываете какую-нибудь из них, не проинициализировав. Если вы много лет охотитесь на насекомых, важность этого ключа может быть вызывать у вас удивление. Так как все ответственные читатели этой книги (и вы в том числе) уже компилируют свой код на уровне диагностики 4 (/W4) и рассматривают все предупреждения как ошибки (/WX), вы можете быть уверены в том, что предупреждения компилятора C4700 и C4701 укажут вам на стопроцентное и вероятное использование неинициализированных переменных соответственно. Ну, а благодаря ключу /RTCu об этих видах ошибок могут узнать и более легкомысленные программисты. В связи с ключом /RTCu интересно отметить, что код, проверяющий неинициализированные переменные, включается в программу, если компилятор обнаруживает условие C4700 или C4701. Третий ключ, /RTC1, просто объединяет ключи /RTCu и /RTCs.

Последний ключ, /RTCs, обнаруживает отбрасывание данных при операциях присваивания — например, если вы пытаетесь присвоить значение 0x101 переменной типа `char`. Как и в случае ключа /RTCu, если вы компилируете программу с ключами /W4 и /WX, отбрасывание данных приведет к ошибке C4244 во время компиляции. При получении ошибки /RTCs вам нужно или применить к нужным вам битам маску, или выполнить приведение типа. Поле Basic Runtime Checks (базовые виды проверки периода выполнения) диалогового окна Property Pages (рис. 17-7) позволяет установить только /RTCu, /RTCs или /RTC1. Чтобы включить ключ /RTCs, нужно выбрать значение в поле Smaller Type Check (проверка при преобразовании к меньшему типу), расположенному над полем Basic Runtime Checks.

Сначала я не мог понять, почему /RTCs не устанавливается ключом /RTC1. Однако потом я понял, что /RTCs может сообщать об ошибках при абсолютно корректном коде C, например:

```
char LoByte(int a)
{
    return ((char)a) ;
}
```

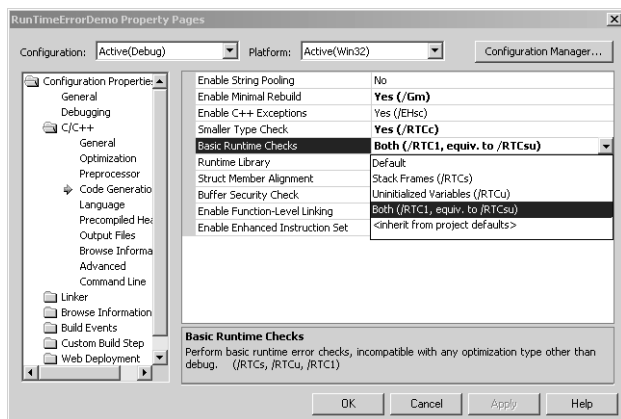


Рис. 17-7. Установка ключей /RTCx

Если бы ключ /RTCs входил в состав /RTC1, люди могли бы думать, что все ключи проверки ошибок в периода выполнения могут поднимать ложную тревогу. Тем не менее я все равно всегда устанавливаю ключ /RTCs, так как хочу знать обо всех потенциальных проблемах при выполнении своей программы.

Теперь обратимся к уведомлениям, которые вы получаете в случае обнаружения ошибки. При выполнении ваших программ вне отладчика код проверки в период выполнения выводит обычное диагностическое информационное окно стандартной библиотеки C. Если вы пишете службы или модули, не имеющие пользовательского интерфейса, вы должны будете перенаправить сообщение информационного окна при помощи функции `_CrtSetReportMode`, передав ей в качестве типа сообщения значение `_CRT_ASSERT`. Возможно, вы думали, что ключи /RTCx имеют единственный стандартный механизм уведомления пользователя, но это не так. При выполнении программы под отладчиком имеет место совершенно иной механизм уведомлений.

Если вы посмотрите на диалоговое окно Exceptions (исключения) интегрированной среды разработки Visual Studio .NET, вы можете заметить несколько новых классов исключений. В данный момент нам интересен класс Native Run-Time Checks (проверка в период выполнения). Открыв его в окне Exceptions, вы увидите четыре разных исключения, соответствующих ключам /RTCx. Наверное, вы уже догадались, что, работая под управлением отладчика и сталкиваясь с проверкой в период выполнения, ваша программа генерирует специальное исключение, чтобы отладчик мог его обработать.

Управление выводом проверки в период выполнения

Во многих ситуациях вас устроит информация, выводимая по умолчанию, однако в некоторых случаях вам захочется обрабатывать вывод данных об ошибке самостоятельно. Пример собственного обработчика ошибок приведен в листинге 17-6. Список параметров, передаваемых в обработчик проверки ошибок в период выполнения, отличается тем, что он может включать переменное число параметров. Очевидно, эта гибкость показывает, что в будущем Microsoft планирует добавить много других видов проверки ошибок в период выполнения. Ваш

собственный обработчик получает те же параметры, что и версия по умолчанию, поэтому вы можете выводить сведения об ошибках, соответствующих им переменных и другую информацию. Как видите, вы должны сами выбирать способ информирования пользователя. Код листинга 17-6 находится на CD в примере RTCHandle, так что вы можете изменять обработку ошибок, как вам будет угодно.

Листинг 17-6. Создание собственных отчетов об ошибках /RTCs

```

/*//////////////////////////////////////////////////////////////
ФУНКЦИЯ:  HandleRTCFailure
ОПИСАНИЕ:
    Обработчик ошибок периода выполнения (Run Time Checking, RTC), работающий
    при выполнении HE под отладчиком. При выполнении под управлением отладчика
    эта функция игнорируется. Поэтому вы никак не можете ее отладить!
ПАРАМЕТРЫ:
    iError    - тип ошибки, сообщаемый функции _RTC_SetErrorType.
                Обратите внимание, что я не использую этот параметр.
    szFile    - имя исходного файла, в котором произошла ошибка.
    iLine     - номер строки ошибки.
    szModule  - модуль ошибки.
    szFormat  - строка формата в стиле printf для переменного списка
                параметров. Обратите внимание, что я использую этот
                параметр только для получения их значений.
    ...       - "переменный" список параметров. Здесь могут передаваться
                только два значения. Первое - идентификатор ошибки RTC:
                    1 = /RTCs
                    2 = /RTCs
                    3 = /RTCu
                Второе - указатель на строку, выводимую отладчиком. Это
                значение важно только для ключей /RTCs и /RTCu, так как
                они показывают переменную, вызвавшую проблему.
ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ:
    TRUE      - после возврата из этой функции выполняется вызов _DebugBreak.
    FALSE     - выполнение продолжается.
//////////////////////////////////////////////////////////////
// Отключение проверок периода выполнения для этой функции,
// нужное для предотвращения ее рекурсивного вызова.
#pragma runtime_checks("", off)

// Критическая секция, защищающая функцию HandleRTCFailure
// от реентерабельности.
CRITICAL_SECTION g_csRTCLock ;

int HandleRTCFailure ( int      /*iError*/ ,
                      const char * szFile ,
                      int      iLine ,
                      const char * szModule ,
                      const char * szFormat ,
                      ...
                      )
{

```

```

// Нет реентерабельности кода!
EnterCriticalSection ( &g_csRTCLock ) ;

// Получение двух параметров переменного списка. Я полагаю,
// что в будущем будет добавлено множество проверок RTC.
va_list vl ;

va_start ( vl , szFormat ) ;

// Первый параметр – идентификатор ошибки.
_RTC_ErrorNumber RTCErrNum = va_arg ( vl , _RTC_ErrorNumber ) ;

// Второй – дополнительное описание ошибки.
char * szErrorVariableDesc = va_arg ( vl , char * ) ;

va_end ( vl ) ;

TCHAR szBuff [ 512 ] ;

// Получение описания ошибки по ее идентификатору.
const char *szErr = _RTC_GetErrDesc ( RTCErrNum ) ;

// Нужно убедиться, что szFile и szModule не равны NULL.
if ( NULL == szFile )
{
    szFile = "Unknown File" ;
}
if ( NULL == szModule )
{
    szModule = "Unknown Module" ;
}

// Если ошибка соответствует ключу /RTCs или /RTCu, я могу вывести
// полезную информацию, в том числе интересующую нас переменную!
if ( 1 != RTCErrNum )
{
    _stprintf ( szBuff
                ,
                _T ( "%S\n%S\nLine #d\nFile:%S\nModule:%S" ) ,
                szErr
                ,
                szErrorVariableDesc
                ,
                iline
                ,
                szFile
                ,
                szModule
                ) ;
}
else
{
    // Формирование строки.
    _stprintf ( szBuff
                ,
                _T ( "%S\nLine #d\nFile:%S\nModule:%S" ) ,
                szErr
                ,

```

см. след. стр.

```

        iLine
        szFile
        szModule
    }

    int iRes = TRUE ;
    if ( IDYES == MessageBox ( GetForegroundWindow ( )
                               szBuff
                               _T ( "Run Time Check Failure" )
                               MB_YESNO | MB_ICONQUESTION
                             ) )
    {
        // Возврат 1 означает, что по окончании этой функции
        // будет вызвана функция DebugBreak.
        iRes = 1 ;
    }
    else
    {
        iRes = 0 ;
    }

    // Выход из критической секции.
    LeaveCriticalSection ( &g_csRTCLock ) ;

    return ( iRes ) ;
}
#pragma runtime_checks("", restore)

```

Установка собственного обработчика ошибок тривиальна; для этого нужно просто передать указатель на него в `_RTC_SetErrorFunc`. Есть несколько других функций, помогающих обрабатывать ошибки в период выполнения. Первая, `_RTC_GetErrDesc`, получает строку, описывающую конкретную ошибку. `_RTC_NumErrors` возвращает общее число ошибок, поддерживаемых текущими версиями компилятора и стандартной библиотеки. Кроме того, есть еще и функция `_RTC_SetErrorType`, которую я нахожу немного опасной. Она позволяет отключить обработку ошибок для отдельных или всех специфических проверок в период выполнения.

Так как проверки в период выполнения основаны на возможностях стандартной библиотеки C, вы можете подумать, что, если ваша программа не использует стандартную библиотеку C, вы утрачиваете все преимущества ключей RTC. Если вам интересно, зачем вам когда-нибудь может понадобиться программировать без стандартной библиотеки C, вспомните про ATL и макрос `_ATL_MIN_CRT`. Если вы не используете стандартную библиотеку C, то при определенном макросе `__MSVC_RUNTIME_CHECKS` нужно вызвать функцию `_RTC_Initialize`. Вы также должны предоставить функцию `_CRT_RTC_INIT`, возвращающую указатель на ваш собственный обработчик ошибок.

Когда я только начал писать собственные обработчики вывода, я столкнулся с небольшой проблемой. Я не мог отладить собственный обработчик! Немного подумав об этом, вы поймете причину. Как я уже сказал, код проверки в период выполнения может определить, выполняете ли вы программу под управлением отладчика, и вывести информацию или в отладчике, или при помощи обычного

диагностического диалогового окна стандартной библиотеки C. Когда вы выполняете программу под отладчиком, код проверки в период выполнения знает это и просто генерирует для отладчика специальный код исключения, полностью избегая вашего обработчика вывода.

Стандартный вопрос отладки

Как убедиться в том, что при обработке строк я не допустил ошибок?

Наверное, самым частым источником ошибок и проблем с безопасностью является обработка старых добрых строк, оканчивающихся на 0. Проблема заключается в определении этих функций в стандартной библиотеке C: они никак не позволяют указать длину буфера. Так, функция `strcpy` принимает указатели на два буфера и вслепую копирует данные из буфера ввода в буфер вывода, совершенно не предполагая, что буфер вывода может быть вдвое меньшим. Это не только может привести к записи данных вне выделенного блока памяти, но и создает огромную брешь в защите, при помощи которой авторы вирусов перезаписывают в стеке адрес возврата для передачи управления на свой код.

Вы можете до потери сознания просматривать программу в поисках этих ошибок и все же упустить их. К счастью, некоторые умные люди из Microsoft поняли, что пора изменить положение дел. Новая библиотека STRSAFE призвана обезопасить обработку строк. STRSAFE входит в Platform SDK за июль 2002 года и включена в Visual Studio .NET 2003.

Чтобы работать с этой библиотекой, нужно включить в прекомпилированный заголовок файл STRSAFE.H, который предоставит вам доступ ко многим новым функциям, принимающим длину буфера вывода и гарантирующим, что число скопированных символов не превысит указанное вами. Включив STRSAFE.H, вы при первой же компиляции программы обнаружите, что уязвимые к ошибкам функции начнут возмущаться и компиляция не удастся, пока вы их не исправите.

Мне очень жаль, но STRSAFE появилась после того, как я написал почти весь код для этой книги. Работая над своими проектами, вы поймете, что настройка STRSAFE очень похожа на действия, нужные для преобразования кода ANSI в формат Unicode. Это требует некоторого времени, но затраты того стоят. Ко времени поступления этой книги в продажу или чуть позже я переработаю все программы из нее и выложу их на веб-сайте компании Wintellect.

Ключ проверки безопасности буфера

Проверки в период выполнения очень полезны, но есть и еще один ключ, который также следует устанавливать всегда. Это ключ `/GS`, выполняющий проверку безопасности буфера (Buffer Security Check). В отличие от ключей `/RTCx` его следует устанавливать и для отладочных, и для заключительных компоновок. Ключ `/GS` отслеживает адреса возврата из функции и предотвращает его перезапись, что часто делают вирусы и троянские кони для передачи управления на свой код. Для

этого он резервирует в стеке дополнительное пространство перед адресом возврата. При входе в функцию код пролога сохраняет в этом месте результат выполнения операции ИСКЛЮЧАЮЩЕЕ ИЛИ над адресом возврата и специальным значением (`security cookie`). Это значение подсчитывается во время загрузки модулей, чем достигается его уникальность для отдельных модулей. При возврате из функции специальная функция `_security_check_cookie` проверяет, не изменилось ли сохраненное значение. При обнаружении различия выводится информационное окно, и программа завершается. Если вы хотите посмотреть этот механизм в действии, изучите в исходном коде стандартной библиотеки С файлы `SECCINIT.C`, `SECCOOK.C` и `SECFAIL.C`.

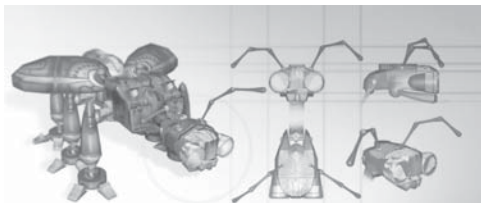
Обеспечения безопасности ключу `/GS` мало, поэтому он еще оказывает нам огромную помощь при отладке. Ключи `/RTCx` отслеживают множество ошибок, но случайную перезапись адреса возврата они все же иногда пропускают. Благодаря `/GS` вы получаете проверку таких ситуаций и в отладочных компоновках. Конечно, сотрудники Microsoft при разработке этого ключа не забывали про нас, поэтому вы можете заменить функцию вывода информационного окна по умолчанию собственным обработчиком, вызвав функцию `_set_security_error_handler`. Если вы повредите стек, ваш обработчик должен выполнять после записи ошибки вызов `ExitProcess`.

Резюме

Стандартная отладочная библиотека С предоставляет массу великолепных возможностей, если, конечно, вы подключите их в своем приложении. Так как использования памяти в программах С и С++ избежать невозможно, мы не должны упускать ни одного шанса облегчить решение проблем, которые обязательно при этом возникнут. В этой главе я привел самую важную информацию о библиотеке DCRT и представил две написанных мной утилиты, `MemDumperValidator` и `MemStress`, которые помогут получить более подробную информацию о памяти, используемой вашими программами, и позволят оптимизировать тестирование программ в стрессовых условиях.

Расширяемость библиотеки DCRT просто удивительна. Если вы всерьез занимаетесь отладкой более года или двух, вы, возможно, уже разрабатывали в прошлом что-то похожее на нее. Надеюсь, что я смог доказать вам мощь библиотеки DCRT. Советую разработать другие утилиты и модули, которые облегчат отладку проблем с памятью.

Мы также обсудили методы и инструменты обработки самых отвратительных проблем с памятью: записи посредством неинициализированных указателей и записи вне выделенных блоков памяти. Пусть применение грубой силы кажется очень неэлегантным способом решения этих проблем, но только так вы получите реальный шанс на успех. Когда дело доходит до записи данных после окончания блока памяти, их отслеживание поможет облегчить инструмент `PageHeap` из состава `AppVerifier`, одного из приложений пакета `Application Compatibility Toolkit`. Хотя `AppVerifier` не лишен недостатков, я уверен, что в будущем Microsoft их исправит. Наконец, огромную помощь в избавлении от ошибок оказывают новые ключи компилятора: ключи проверки ошибок в период выполнения и ключ проверки безопасности буферов.



FastTrace: высокопроизводительная утилита трассировки серверных приложений

Все мы знаем, что быстродействие — основное требование, предъявляемое к серверным приложениям. От скорости их выполнения напрямую зависит масштабируемость программы. Мы создаем приложения, которые должны обрабатывать тысячи и даже миллионы отдельных запросов, при этом любое дополнительное действие может иметь огромные последствия. Что делает серверные приложения еще «интереснее», так это то, что они в высокой степени многопоточны, из-за чего причины низкой производительности иногда найти очень трудно.

Писать серверные приложения трудно — отлаживать же трудней вдвое. При поиске некоторых ошибок — особенно проблем с производительностью — в клиентских приложениях мы можем изучать их работу непосредственно. В случае серверных приложений мы имеем дело с кодом, скрывающимся в самых потаенных уголках памяти, поэтому нам остается исследовать его только косвенно, по времени отклика, а также при помощи отладчиков и таких средств, как PerfMon, которые указывают только на отдельные аспекты общего состояния программы. Вообще-то все еще хуже: будьте готовы к тому, что никакие нетривиальные ошибки никогда не обнаружат себя в удобной управляемой среде ваших систем контроля качества, а покажутся только в джунглях готовой программы.

Для отладки серверных приложений у нас есть старое спасительное средство — трассировка. Это единственный способ, позволяющий увидеть общую картину, особенно при запуске готовой программы в реальных условиях. Мне дово-

дилось работать над проектами, включавшими самые невероятные системы трассировки. На их разработку тратилось много времени и усилий, чтобы программисты могли получить представление о действительной работе готовой системы и повысить свои шансы на обнаружение и исправление ошибок.

К сожалению, когда дело касается серверных приложений, баланс между «отлаживаемостью» и производительностью очень тонок. Я могу привести несколько случаев из нашей консультативной практики, когда нас приглашали решить проблемы с быстродействием программы и оказывалось, что они связаны с самими системами трассировки. Интересно, что разработчики об этом даже не подозревали.

На собственном опыте убедившись в плохой производительности многих систем трассировки, я захотел решить эту проблему раз и навсегда. Для этой главы я написал программу FastTrace, призванную обеспечить трассировку в каком угодно объеме, не вызывая значительного снижения быстродействия. Прежде чем перейти к обсуждению использования и реализации FastTrace, я хочу объяснить, в чем же заключается недостаток большинства методов трассировки.

Фундаментальная проблема и ее решение

Самая крупная проблема с серверными приложениями объясняется тем, что нам, людям, трудно представить себе множество вещей одновременно. Наш мозг организован линейным образом. Чтобы облегчить отладку серверных приложений, мы неосознанно организуем вывод трассировочной информации последовательно. А вот большинство современных серверов являются многопроцессорными, и многие приложения имеют до 20 и более потоков, так что многие из этих линейных процессов на самом деле выполняются параллельно. Пытаясь внести порядок в кажущийся хаос, мы выводим трассировочные данные всего приложения в один-единственный файл.

Если результаты трассировки многопоточной программы сохраняются в одном файле, вы попадаете в ситуацию, показанную на рис. 18-1. Именно линейная регистрация трассировочных вызовов для всех потоков приводит к возникновению узких мест (*bottle neck*¹). Как я уже говорил в главе 15, вся суть разработки многопоточных программ сводится к тому, чтобы потоки были максимально заняты и как можно меньше простаивали.

Очевидно, что для создания самого быстрого средства трассировки в западном мире требуется такой линейный способ вывода информации, который не влиял бы на нормальный ход выполнения программы. Если бы каждый поток выполнял собственную трассировку, все было бы прекрасно. Это и делает утилита FastTrace: она предоставляет каждому потоку отдельный файл вывода, что избавляет потоки от перехода в состояние ожидания или блокировки. После сохранения трассировочной информации нескольких потоков в файлах журнала вы можете объединить файлы и увидеть истинную последовательность трассировки. В разделе «Реализация FastTrace» вы увидите, что ничего сложного в этом нет.

¹ Буквально «бутылочное горлышко». — *Прим. пер.*



Рис. 18-1. Типичная система трассировки серверных приложений

Использование FastTrace

Детали реализации FastTrace я скрыл, поэтому для ее использования вам нужно только скомпоновать свое приложение с библиотекой FASTTRACE.DLL и вызывать в нужных случаях одну из ее экспортируемых функций, названную, конечно же, FastTrace. Вот ее прототип:

```
FASTTRACE_DLLINTERFACE void FastTrace ( LPCTSTR szFmt      ,
                                         ...                ) ;
```

Трассировочный вывод будет направлен в файлы, которые хранятся в том же каталоге, что и файл выполняющегося процесса. Имена файлов имеют формат:

<Имя EXE-файла>_<ID процесса>_<ID потока>.FTL

Открыв какой-нибудь файл, вы заметите, что информация хранится в двоичном виде. Каждая выводимая FastTrace строка сохраняется с соответствующим ей номером; это нужно для соблюдения правильного порядка строк при объединении файлов. Кроме того, каждая строка может содержать метку времени.

Чтобы сделать FastTrace простой и быстрой, я решил ограничить длину строк 80 символами. При выполнении отладочной компоновки на попытку записи более длинной строки вам укажет диагностическое сообщение SUPERASSERT. Если вы хотите использовать более длинные строки, нужно просто переопределить значение MAX_FASTTRACE_LEN в файле FastTrace.H и перекомпоновать FastTrace.

При помощи функции SetFastTraceOptions, принимающей указатель на структуру FASTTRACEOPTIONS, вы можете передать FastTrace три команды. Первая команда позволяет подключить сквозную запись. Это удобно, если вы ищете ошибку и хотите гарантировать, что вся трассировочная информация записывается в файл как можно быстрее. Очевидно, что это замедляет быстроедействие, поэтому следует использовать данную функцию только при необходимости. Вторая команда под-ключает создание меток времени для каждой строки. По умолчанию их создание

отключено с целью небольшого повышения производительности. Вы можете включать/отключать его когда угодно. Наконец, последняя команда позволяет призвать FastTrace вызвать функцию отладочного вывода, прототип которой соответствует функции `OutputDebugString`. По умолчанию FastTrace не вызывает `OutputDebugString`; как я объяснил в главе 4, эта функция генерирует исключение и будет замедлять ваше приложение. Однако вам, вероятно, хотелось бы видеть эти исключения при выполнении отладочных компоновок, поэтому я предоставляю и такую возможность.

Наконец, я реализовал две функции, предназначенные для управления трассировкой. Первая из них — `FlushFastTraceFiles` — как можно догадаться по ее имени, записывает все буферы FastTrace на диск. Возможно, вам захочется создать фоновый поток, следящий за работой серверного приложения и периодически «сбрасывающий» трассировочную информацию на диск в те моменты, когда оно не занято. Так вы сможете гарантировать, что нужная информация будет в файлах даже при аварийном завершении приложения.

Вторая называется `SnapFastTraceFiles`. Серверные приложения не имеют точно определенной точки завершения, поэтому вам нужен способ, позволяющий изучать трассировочную информацию в любое время. `SnapFastTraceFiles` закрывает все открытые файлы трассировки и переименовывает их, чтобы можно было определить соответствующие «снимки». Переименование файлов происходит согласно такой схеме:

```
SNAP_<Номер снимка>_<Имя EXE-файла>_<ID процесса>_<ID потока>.FTL
```

Номер снимка представляет собой уникальное для каждого процесса десятичное число, благодаря чему вы сможете увидеть всю цепь своих снимков. После закрытия и переименования всех активных файлов трассировки `SnapFastTraceFiles` снова открывает файлы для каждого потока, чтобы не пропустить никакой информации. Как и в случае с `FlushFastTraceFiles`, вам, вероятно, захочется реализовать возможность вызова `SnapFastTraceFiles` в нужный вам момент. Помните: для защиты внутренних структур данных и `SnapFastTraceFiles`, и `FlushFastTraceFiles` используют критическую секцию, что приводит к временной блокировке всех трассировочных вызовов.

Объединение журналов трассировки

Как я уже говорил, файлы журнала имеют двоичный формат, поэтому для изучения отдельных файлов или их слияния с целью исследования линейного потока трассировочных вызовов потребуется программа `FastTraceLog.EXE`. Для вывода дампа журнала на экран нужно только передать ей в командной строке выражение `-d <отдельный файл журнала>`. В результате вы увидите порядковый номер, дату/время (если было включено создание меток времени) и сохраненную в указанный момент строку трассировочной информации.

Слияние, или объединение файлов журнала чуть сложнее. Для этого `FastTraceLog.EXE` нужно передать в командной строке выражение `-c <SNAP_номер снимка (необязательный параметр)>_<имя EXE-файла>_<ID процесса>`. Например, выполнив тестовую программу `FTSimpTest.EXE`, вы увидите, что она генерирует файлы трассировки с такими именами:

```
FTSimpTest_2720_0400.FTL  
FTSimpTest_2720_1644.FTL  
FTSimpTest_2720_2332.FTL  
FTSimpTest_2720_2368.FTL  
FTSimpTest_2720_2424.FTL  
FTSimpTest_2720_2560.FTL  
FTSimpTest_2720_2584.FTL  
FTSimpTest_2720_2640.FTL  
FTSimpTest_2720_2688.FTL
```

Для объединения этих файлов отдельного потока нужно выполнить команду:

```
FastTraceLog.exe -c FTSimpTest_2720
```

В итоге вы получите текстовый файл — в нашем случае с именем FTSimpTest_2720.TXT. Информация в текстовом файле очень похожа на дамп, и в приведенном ниже фрагменте вы можете увидеть, что до последней строки создание меток времени было включено, а перед ней отключено:

```
[0x0B3C 57 1/1/2003 17:52:47:205]  
Hello from CThread -> 3!  
[0x0B50 58 1/1/2003 17:52:47:205]  
Hello from CThread -> 3!  
[0x0B50 59 1/1/2003 17:52:47:486]  
Hello from CThread -> 4!  
[0x0B20 60 1/1/2003 17:52:47:486]  
Hello from CThread -> 4!  
[0x0B20 61 1/1/2003 17:52:47:767]  
Hello from CThread -> 5!  
[0x0830 62]  
THIS SHOULD BE THE LAST LINE!
```

В квадратных скобках выводится идентификатор сгенерировавшего сообщение потока, порядковый номер и необязательная метка даты/времени записи сообщения, а на следующей строке — само трассировочное сообщение. Я с гордостью сообщаю, что сделал метку даты/времени интернациональной, благодаря чему вы увидите дату в том формате, к которому привыкли!

Реализация FastTrace

Работать с FastTrace совсем просто, да и реализовать ее было не намного сложнее. Главное чудо FastTrace — поддержка памяти для каждого потока. Каждый поток имеет свой объект класса записи в файл и записывает информацию только в один файл. Реализация основных механизмов FastTrace почти не требует пояснений. Исходный код см. на CD в проекте FastTrace.

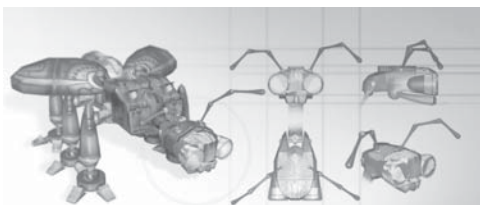
При реализации FastTrace я принял лишь два нетривиальных решения. Приступив к ее разработке, я собирался хранить журналы трассировки для отдельных потоков в виде текстовых файлов. В этом случае можно было бы с легкостью читать файлы без посторонней помощи. Однако, представив, сколько мне пришлось бы корпеть над механизмом их объединения, я понял, что мне нужен другой способ. В итоге я решил хранить каждую запись с двоичной меткой времени и кон-

кретной строкой трассировочной информации. Сначала я хотел реализовать поддержку строк различной длины, но потом решил, что не стоит замедлять трассировку, подсчитывая при каждой трассировочной команде число символов в строке. Кроме того, читать строки переменной длины было бы гораздо сложнее.

Второе интересное решение было связано с форматом хранения меток времени. Если вы когда-нибудь заглядывали в справочный раздел, описывающий их хранение в ОС, то знаете, что количество форматов просто огромно. Изучив их подробнее, я решил использовать формат `FILETIME`, потому что он занимает только 8 байт, в то время как формат `SYSTEMTIME` требует 16 байт. Кроме того, я рассмотрел их обработку и обнаружил, что самый быстрый способ получения времени обеспечивает функция `GetSystemTimeAsFileTime`.

Резюме

Сложность отладки серверных приложений, особенно что касается тех ошибок, которые обнаруживают себя только в готовых системах, означает, что трассировка — один из самых важных методов, имеющихся в вашем распоряжении. Однако, учитывая влияние трассировки на производительность программы, нужно позаботиться о том, чтобы она была максимально быстрой. В этой главе я представил средство трассировки, которое должно оказаться достаточно хорошим даже для самых требовательных серверных программ.



Утилита Smooth Working Set

Как утверждалось в фильме «Годзилла» и как вы сами можете убедиться по объему спама в вашем почтовом ящике, «размер имеет значение». У всех разработчиков просто слюнки текут при мысли о более быстрых компьютерах с большим объемом памяти, но даже в этом случае нам нужно беспокоиться о размере наших программ. Старая пословица «компактный код — хороший код» ничуть не утратила своей актуальности в современном мире, в котором оперативная память компьютеров разработчиков часто превышает 512 Мб, а серверов — 2 Гб. Если объем памяти, имеющейся в вашем распоряжении, кажется вам бесконечным, это не значит, что ее нужно использовать полностью!

После устранения явных проблем с производительностью программы, таких как циклическое вычисление числа π с точностью до миллиардного разряда, наступает самое время позаботиться о минимизации рабочего набора. Рабочий набор — это объем памяти, выделенной для выполняемых в текущий момент блоков вашей программы. Чем меньше вы сделаете рабочий набор, тем быстрее будет работать ваше приложение благодаря уменьшению числа ошибок страниц. Ошибка страницы происходит при доступе к блоку вашей программы, который или не находится в кэш-памяти (мягкая ошибка страницы), или не находится в памяти вообще и должен быть загружен с жесткого диска (жесткая ошибка страницы). Как мне однажды сказал один мудрый человек, «ошибки страниц способны испортить вам весь день!».

Объем текущего рабочего набора своей программы вы можете увидеть в столбце Mem Usage (память), выбрав в окне Task Manager (диспетчер задач) вкладку Processes (процессы). Сведения о рабочем наборе показывают также многие другие диагностические и информационные средства, такие как PerfMon. Убедившись, что ваши алгоритмы экономно расходуют память, вам следует попытаться уменьшить объем памяти, занимаемой самим кодом. В начале этой главы я подробнее опишу ошибки страниц во время выполнения программы и объясню, почему они так нежелатель-

ны. После этого, как вы уже, наверное, догадались, я представлю программу Smooth Working Set (SWS), которая сделает оптимизацию рабочего набора тривиальной.

Оптимизация рабочего набора

Вероятно, вы всегда интуитивно понимали, пусть и не пытаясь вдаваться в подробности, что при компиляции и компоновке двоичных файлов компоновщик просто размещает в итоговом файле сначала функции первого OBJ-файла, потом второго и т. д. Иначе говоря, функции размещаются в порядке компоновки, а не в порядке выполнения. Однако с данным процессом связана одна проблема: при этом не принимается во внимание расположение чаще всего вызываемых функций.

На рис. 19-1 — пример того, что может произойти: ОС поддерживает только шесть функций фиксированного размера на страницу памяти и в любой момент времени может хранить в памяти только четыре страницы; кроме того, 10 вызываемых функций — самые часто вызываемые функции программы.

При загрузке этой программы в память ОС, не долго думая, загружает четыре первых страницы двоичного файла. В ходе выполнения первая функция вызывает функцию 2, которая по стечению обстоятельств располагается на той же странице. Однако функция 2 вызывает функцию 3, которая находится на странице 5. Так как страницы 5 в памяти нет, происходит ошибка страницы. Теперь ОС должна выгрузить одну из загруженных страниц. Так как страница 4 не изменялась, ОС решает выгрузить ее и загружает на ее место страницу 5. Теперь можно выполнить функцию 3. Увы, функция 3 вызывает функцию 4, находящуюся на только что выгруженной странице 4, поэтому у нас происходит вторая ошибка страницы. ОС выгружает страницу 3, к которой дольше всего не было обращений, возвращает в память страницу 4 и выполняет функцию 4. Как вы можете видеть, после этого произойдет еще одна ошибка страницы, поскольку функция 4 вызывает функцию 5 — только что выгруженную. Я мог бы продолжить этот процесс дальше, но уже и так ясно, что при ошибках страниц происходит очень много работы.

Главное, что обработка ошибок страниц требует массу времени. Программа на рис. 19-1 вместо выполнения «отсиживается» в коде ОС. Если бы мы могли указать компоновщику порядок размещения функций на страницах, мы избежали бы многих дополнительных затрат, связанных с обработкой этих ошибок.

На рис. 19-2 показана та же программа после перемещения самых часто используемых функций в начало двоичного файла. Вся программа занимает тот же объем памяти (4 страницы), но благодаря объединению часто вызываемых функций они не приводят ни к каким ошибкам страниц, в результате чего приложение будет работать быстрее.

Процесс обнаружения и упорядочения наиболее часто вызываемых функций называется оптимизацией рабочего набора и состоит из двух этапов. На первом нужно определить, какие функции вызываются чаще всего, а на втором компоновщику нужно задать порядок размещения функций в файле, чтобы все они были расположены соответствующим образом.

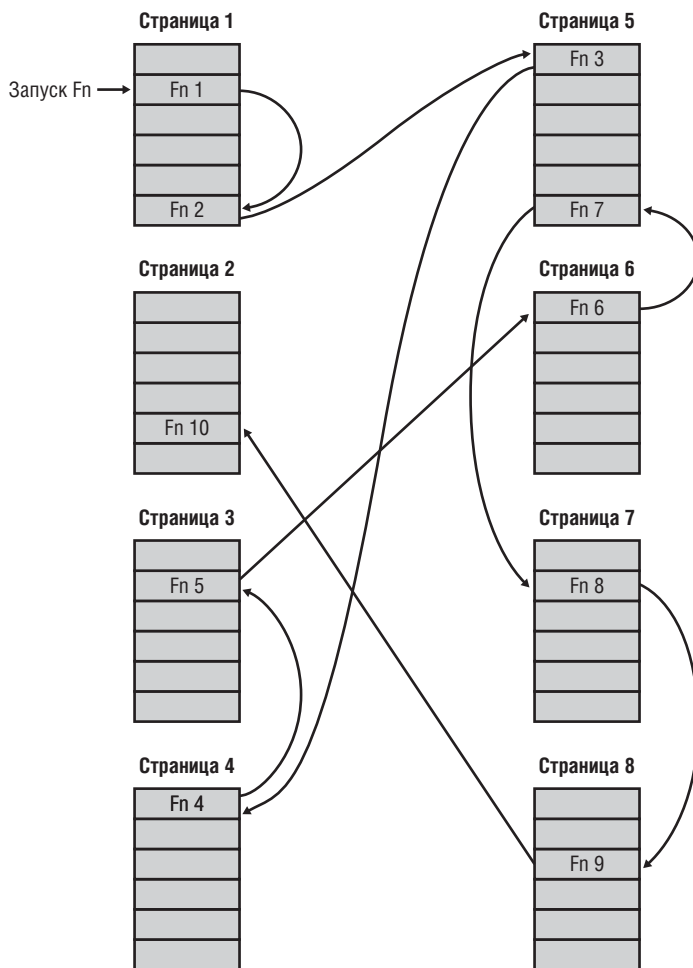


Рис. 19-1. Неоптимизированная система

Вся работа, связанная с подсчетом числа вызовов функций, скрыта в SWS; подробнее о необходимых для этого действиях см. раздел «Работа с SWS». Указать компоновщику порядок размещения функций проще простого. Для этого LINK.EXE поддерживает ключ командной строки `/ORDER`. В качестве параметра после ключа `/ORDER` просто указывается текстовый файл, называемый файлом порядка, в котором нужно привести список всех функций в желательном для вас порядке. Интересно, что имена всех функций должны быть указаны в файле порядка в полном, расширенном виде. Идея, лежащая в основе SWS, по сути заключается в создании файла порядка для вашей программы.

Возможно, у тех из вас, кто давно работает с Microsoft Windows, где-то в сознании сейчас замаячили буквы WST. WST означает Working Set Tuner (средство настройки рабочего набора). Эту программу Microsoft распространяла вместе с Platform SDK для... хм, для настройки рабочего набора. Однако в связи с этим вам могут прийти в голову и три других факта: во-первых, утилиту WST было очень сложно использовать, во-вторых, в ней было несколько ошибок, и в-третьих, она

больше не входит в состав Platform SDK. SWS — гораздо более простая в использовании замена WST.

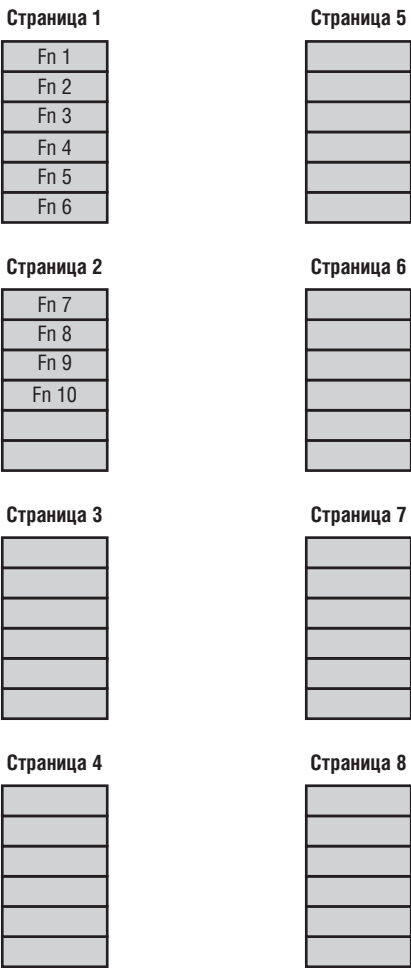


Рис. 19-2. Оптимизированная система

Если вам хочется больше узнать о настройке рабочего набора и получить прекрасное пособие по производительности Windows, попытайтесь найти четвертый том книги «Microsoft Windows NT 3.51 Resource Kit» под названием «Optimizing Windows NT» (оптимизация Windows NT) (Microsoft Press, 1995). Этот том, написанный Рассом Блейком (Russ Blake), представляет собой великолепное введение в указанную тему. В группе разработчиков Microsoft Windows NT Расс отвечал за настройку производительности системы. Эта книга раньше была в MSDN, но внезапно исчезла. Утилиту WST также разрабатывала группа Расса, и в своей книге он утверждает, что после использования программы оптимизации рабочего набора вы должны достигнуть его уменьшения на 35–50%. Каждый раз, когда ваше приложение может сбросить такой вес, эту возможность определенно стоит рассмотреть повнимательней!

Стандартный вопрос отладки

Могу ли я проверить процесс, выполняемый на главном сервере, на предмет утечки ресурсов без установки каких-либо программ?

Для выполнения быстрой проверки всегда подойдет PerfMon, однако нет ничего лучше, чем удивительный диспетчер задач. Открыв вкладку Processes, вы увидите столбец, отображающий различную информацию о производительности. Так как работа с ресурсами основана на дескрипторах, следует отслеживать число дескрипторов процесса. Откройте вкладку Processes, выберите в меню View (вид) пункт Select Columns (выбрать столбцы) и установите в диалоговом окне Select Columns (выбор столбцов) флажок Handle Count (счетчик дескрипторов). Если вы хотите следить за объектами GDI, установите также флажок GDI Objects (объекты GDI). Оба столбца, добавленных в диспетчер задач и отсортированных по дескрипторам, показаны на рис. 19-3.

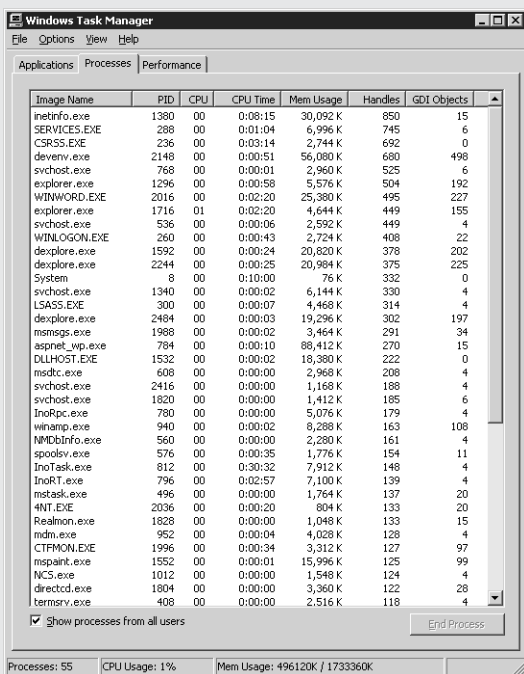


Рис. 19-3. Отображение дескрипторов и объектов GDI в диспетчере задач

По умолчанию диспетчер задач автоматически обновляет информацию каждые несколько секунд, но при этом вам нужно непрерывно следить за числом дескрипторов и объектов GDI, чтобы увидеть, когда оно начинает увеличиваться. Мне нравится приостанавливать обновление путем выбора пункта Paused (приостановить) в подменю Update Speed (скорость обновления) меню View. Благодаря этому я могу выполнить какую-либо опера-

см. след. стр.

цию, вернуться в диспетчер задач и нажать F5 для обновления экрана. Наблюдайте и за столбцом Mem Usage (память), потому что числу дескрипторов соответствует определенный объем памяти: если оба значения растут, утечка ресурсов в самом разгаре.

Прежде чем описать работу с SWS, я должен вернуть вас на землю. Во-первых, SWS — не панацея. Если до запуска SWS быстроедействие вашей программы было ужасным, таким оно и останется. Во-вторых, оптимизацию рабочего набора следует выполнять в самом конце работы над производительностью приложения, после улучшения его алгоритмов и исправления всех найденных проблем с быстроедствием. Наконец, использовать SWS следует только в конце цикла разработки. Максимальное преимущество SWS обеспечит, когда изменения кода приближаются к минимуму.

SWS может и не понадобиться. Разрабатывая небольшую программу, откомпилированные файлы которой занимают не более 2–3 Мб, вы можете вообще не заметить уменьшения рабочего набора. Максимальной выгоды от оптимизации рабочего набора вы добьетесь, работая над более крупными приложениями, потому что чем больше страниц требуется программе, тем больше возможностей для ее улучшения.

Работа с SWS

Работа с SWS состоит из трех этапов. На первом вы должны перекомпилировать свою программу для установки ловушки SWS, позволяющей накопить данные о вызовах функций. Второй этап заключается в выполнении с этой специальной откомпилированной версией программы ряда наиболее частых пользовательских сценариев. Для правильного использования SWS вам нужно определить эти сценарии, чтобы вы могли точно их воспроизвести. Случайное выполнение программы под управлением SWS может вообще не привести к сокращению рабочего набора. Третий этап включает генерирование для компоновщика файла порядка (что очень просто) и интеграцию этого файла в заключительную компоновку программы.

Настройка компилянтов SWS

Необходимость отдельной компиляции программы для использования SWS объясняется тем, что в основе SWS лежат те же принципы, что и в WST. Хотя я мог написать огромную, сложную и подверженную ошибкам программу для получения информации о функциях вашего приложения «на лету», гораздо проще предоставить установку функции-ловушки компилятору. Ключ `/Gh` (включить функцию-ловушку `_penter`) приказывает компилятору разместить в начале пролога всех генерируемых функций вызов функции `_penter`. В SWS функция `_penter` уже реализована; чтобы она могла продемонстрировать все свои магические возможности, вы должны скомпоновать с ней свою программу.

Скомпилировать программу для использования с SWS обычно просто — для этого надо только кое-что сделать.

1. Прежде всего убедитесь в правильности параметров заключительной компоновки своего проекта, так как эта конфигурация будет клонирована. Если вы

установили при помощи надстройки SettingsMaster из главы 9 ключи компилятора и компоновщика, которые я рекомендовал в главе 2, у вас все в порядке.

2. Клонировать конфигурацию, с которой вы хотите работать. Нажмите правой кнопкой название решения в окне Solution Explorer (проводник решений) и выберите в меню пункт Configuration Manager (менеджер конфигураций), после чего появится одноименное диалоговое окно. Новый конфигурационный параметр скрывается в списке Active Solution Configuration (активная конфигурация решения). Доступ к нужному нам полю <New...> показан на рис. 19-4 (я искал его довольно долго). Как правило, следует клонировать заключительные компоновки. Открыв диалоговое окно New Solution Configuration (конфигурация нового решения), дважды убедитесь в том, что вы выбрали нужное значение в списке Copy Settings From (копировать параметры из), потому что, если вы будете наследовать решение от <Default>, ваша конфигурация компоновки не будут перенесена. Называя новый проект, я предпочитаю указывать тип компоновки и буквы «-SWS», в результате чего получается имя Release-SWS. Диалоговое окно New Solution Configuration с этими новыми параметрами показано на рис. 19-5. Помните, что при создании новых решений каталог вывода и промежуточный каталог изменяются.

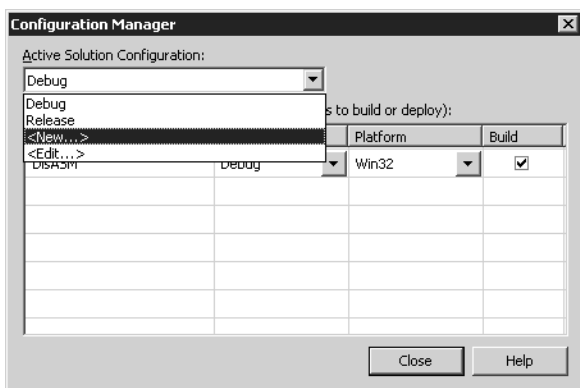


Рис. 19-4. Выбор новой конфигурации при помощи окна Configuration Manager

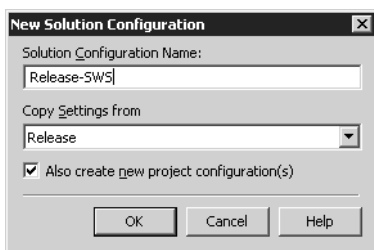


Рис. 19-5. Создание новой конфигурации Release-SWS

3. Если вы пользуетесь утилитой SettingsMaster (как и следует делать), настройка компилятора и компоновщика тривиальна. Нажмите кнопку SettingsMaster

Custom Project Update (обновление проекта) и, когда появится приглашение, выберите файл Release-SWS.XML. Это установит минимально необходимые параметры. Чтобы использовать SWS с отладочной компоновкой, выберите файл Debug-SWS.XML (оба файла находятся в каталоге SettingsMaster\SettingsMaster).

4. Если вы не используете SettingsMaster (позор!), вам нужно установить для проекта Release-SWS следующие параметры компилятора (CL.EXE) (см. по этому поводу главу 2):

- /Zi (формат отладочной информации);
- /Gy (включить компоновку на уровне функций);
- /Gh (включить функцию-ловушку _penter); стандартного способа установки этого ключа нет, поэтому вам придется выбрать в папке C/C++ пункт Command Line (командная строка) и ввести его вручную.

Кроме того, для проекта Release-SWS нужно задать ключи компоновщика (LINK.EXE):

- /DEBUG (генерировать отладочную информацию);
- /OPT:REF (оптимизация: удалять неиспользуемые функции).

5. Добавьте SWSDLL.LIB в список зависимостей (dependencies).

Чтобы сконфигурировать проект Debug-SWS, установите ключ /Gh для компилятора CL.EXE и добавьте зависимость от SWSDLL.LIB.

Выполнение приложений вместе с SWS

После настройки и компиляции приложения наступит самый трудный для вас этап работы с SWS — выполнение вашей программы. Вам придется самым тщательным образом обдумать, каковы наиболее частые сценарии использования вашего приложения. Если вы оптимизируете готовую программу, вам, вероятно, следует посетить своих клиентов, чтобы увидеть, что они чаще всего с ней делают. Если же вы работаете над новым приложением, обсудите это с сотрудниками службы маркетинга или представителями заказчика. Определив сценарии, их нужно будет выполнить на разных машинах с различной загрузкой. Скорее всего вам захочется сделать эти сценарии воспроизводимыми при помощи такой утилиты автоматизации, как Tester из главы 16.

Запустив программу вместе с SWS, вы обнаружите два новых файла в каталоге, в котором находится каждый двоичный файл, скомпилированный для SWS: <имя файла>.SWS и <имя файла>.1.SWS. Файл SWS без цифры — это базовый файл, содержащий адреса и размеры каждой функции модуля, а также пространство для счетчиков числа вызовов. При каждом запуске вашей программы этот файл копируется в новый файл, чтобы избежать повторного просмотра символов. Файлы с цифрой соответствуют запускам программы и содержат счетчики вызовов. При каждом выполнении вашего специального откомпилированного двоичного файла создается новый — <имя файла>.#.SWS.

Нужные компоненты SWS находятся в исполняемом файле SWS.EXE. Эта программа позволяет просматривать отдельные SWS-файлы, создавать новые файлы и выполнять финальную оптимизацию. Запустив SWS.EXE без всяких параметров или с ключом -?, вы увидите:

SWS (Smooth Working Set) 2.0

John Robbins - Debugging Applications for Microsoft .NET and Microsoft Windows

```
Usage: SWS [-t <module>][[-d <module>]][[-g <module>]][[-?] [-nologo]]
  -t <module> - Tune the module's working set
                  (run from directory with .SWS file)
  -d <module> - Dump the raw data for the module or #.SWS file
  -g <module> - Generate the initial SWS file for the module
  -?          - Show this help screen
  -nologo     - Do not display the program information
```

Чтобы увидеть, какие именно функции вызываются, вы можете просматривать все итоговые файлы .SWS; это позволяет гарантировать, что вы выполняете именно те функции, которые собирались. Вывод файла, соответствующего запуску вашей программы (<имя файла>#.SWS), имеет формат:

```
Link time   : 0x3E13849C
Entry count : 12
Image base  : 0x00400000
Image size  : 0x00007000
Module name : SimpleSWSTest.exe
```

Address	Count	Size	Name
-----	---	---	
0x00401050	2	22	?Bar@@YAXXZ
0x00401066	2	22	?Baz@@YAXXZ
0x0040107C	2	22	?Bop@@YAXXZ
0x00401092	4	10	?Foo@@YAXXZ
0x0040109C	1	49	_wmain
0x004010CD	2	10	_Ye01CFunc
0x004011E0	0	422	_wmainCRTStartup
0x00401C50	0	63	__onexit
0x00401C90	0	24	_atexit
0x00401DC0	0	23	__setdefaultprecision
0x00401DE0	0	7	__matherr
0x00401DF0	0	7	__wsetargv

При запуске вашей программы, использующей функцию `_penter` утилиты SWS, первоначальный файл .SWS генерируется автоматически. Однако это требует значительного объема работы с символьной машиной DBGHELP.DLL, поэтому вы сможете уменьшить время запуска, если будете предварительно генерировать SWS-файлы для модуля, указывая программе SWS.EXE ключ `-g`.

Генерирование и использование файла порядка

После завершения всех запусков вашей программы вам нужно оптимизировать ее и сгенерировать файл порядка, который будет передан компоновщику. Для этого SWS.EXE предоставляет ключ командной строки `-t`, после которого следует просто указать имя модуля оптимизируемого двоичного файла. В результате оптимизации создается действительный файл порядка с расширением .PRF; я выбрал это расширение потому, что такие файлы генерировала программа Working Set Tuner.

Если вам хочется увидеть, что происходит при создании файла порядка, т. е. сам процесс «упаковки» и упорядочения функций, укажите SWS в командной строке ключ `-v`. Понять подробный вывод легко:

```
Verbose output turned on
Action -> Tuning for : SimpleSWSTest
Initializing the symbol engine.
Loading the symbols
Processing : SimpleSWSTest.1.SWS
Processing : SimpleSWSTest.2.SWS
Processing : SimpleSWSTest.3.SWS
Processing : SimpleSWSTest.4.SWS
Processing : SimpleSWSTest.5.SWS
Order file output: SimpleSWSTest.PRF
Page Remaining (4086) (0010) : ( 20) ?Foo@@YAXXZ
Page Remaining (4064) (0022) : ( 10) ?Bop@@YAXXZ
Page Remaining (4042) (0022) : ( 10) ?Baz@@YAXXZ
Page Remaining (4032) (0010) : ( 10) _Ye01CFunc
Page Remaining (4010) (0022) : ( 10) ?Bar@@YAXXZ
Page Remaining (3961) (0049) : ( 5) _wmain
```

Числа, которые вы видите после слов «Page Remaining», указывают на оставшееся пространство страницы, размер функции и число ее вызовов.

При создании файла .PRF утилита SWS генерирует не просто текстовый листинг функций в порядке убывания числа вызовов, так как это не учитывало бы размера страницы ОС. Если функция пересекает границу страницы, то для ее выполнения потребуется загрузить в память две страницы. SWS гарантирует, что во всех случаях функция будет размещаться на одной странице, а также пытается обеспечить максимальную заполненность каждой страницы. Именно поэтому вы можете встретить в начале итогового файла порядка редко вызываемые функции. Избегая пустот на страницах, вы снизите объем оперативной памяти, нужный вашей программе.

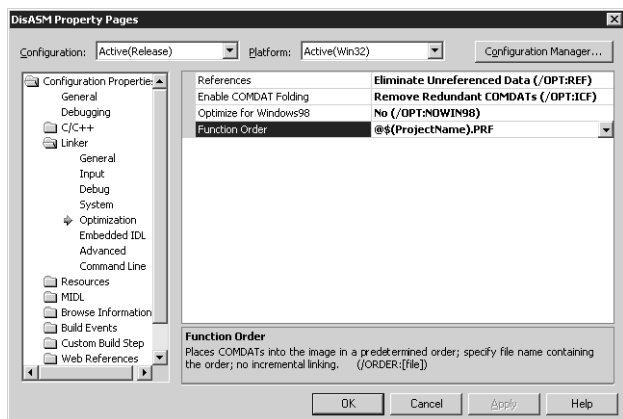


Рис. 19-6. Указание файла порядка для отладочной компоновки при помощи ключа `/ORDER`

После создания файла порядка нужно просто указать его компоновщику при помощи ключа `/ORDER`. Всегда сохраняйте файл порядка в том же каталоге, что и файл проекта и исходные файлы. Ключ `/ORDER` имеет одну особенность, которая часто вызывает замешательство: перед именем файла должен быть указан символ `@`. Если вы не используете `SettingsMaster`, вы можете указать файл порядка для двоичного файла своей заключительной компоновки, открыв окно `Property Pages` (страницы свойств), папку `Linker` (компоновщик), страницу `Optimization` и введя нужное выражение в поле `Function Order` (порядок функций) (рис. 19-6). При использовании `SettingsMaster` просто нажмите кнопку `SettingsMaster Custom Project Update` и выберите файл `ReleaseOrderFile.XML`.

Реализация SWS

Как это бывает со многими проектами, когда я впервые подумал о реализации SWS, это представлялось довольно сложным делом, однако при написании кода все оказалось проще. Я планировал сначала разобраться с функцией `_penter`, после этого — с двоичными файлами и перечислением символов и, наконец, — с периодом выполнения и алгоритмом оптимизации.

Функция `_penter`

Должен признать, что до того, как я решил использовать функцию `_penter` при помощи ключа компилятора `/Gh`, я хотел заставить SWS работать с немодифицированными двоичными файлами. Конечно, это возможно, но довольно затруднительно: во-первых, чтобы гарантировать включение кода ловушки в первую очередь, мне нужно было написать отладчик, а во-вторых, я должен был включить 5-байтовые переходы для всех функций, обнаруживаемых в таблице символов. У меня был опыт разработки похожего кода для коммерческих программ, так что я знал, насколько это сложно. В конце концов я счел специальную компоновку вполне приемлемой, особенно когда у нас есть такие средства, как `SettingsMaster`, благодаря которым добавить в конфигурацию компоновки SWS совсем легко.

Разработав установку ловушки, я приступил к рассмотрению действий, которые нужно выполнять при каждом вызове `_penter`. Моя функция `_penter` и ее вспомогательный код приведены в листинге 19-1. Как вы можете видеть, она использует соглашение вызова `naked`, поэтому я сам генерирую пролог и эпилог. Соглашение `naked` требуется в документации к `_penter`; кроме того, это облегчает получение адреса возврата из функции. К счастью, когда компилятор обещает, что он сгенерирует вызов `_penter` до всех остальных команд, он держит свое слово! Результат установки ключа `/Gh` показан в дизассемблированном фрагменте, из которого видно, что вызов `_penter` выполняется даже до команды `PUSH EBP`, одного из элементов стандартного пролога функции:

Bar:

00401050: E8B7000000	call	_penter
00401055: 55	push	ebp
00401056: 8BEC	mov	ebp, esp
00401058: E8A8FFFFFF	call	ILT+0(?Foo
0040105D: 3BEC	cmp	ebp, esp

```

0040105F: E8AE000000      call    _chkesp
00401064: 5D              pop     ebp
00401065: C3              ret

```

Немного пофантазировав, вы поймете, что ключ `/Gh` позволяет создать некоторые другие интересные утилиты. Первая, что пришла мне в голову, — это утилита контроля производительности. Благодаря тому, что Microsoft уже предлагает нам ключ `/Gh` (включить функцию-ловушку `_pexit`), использовать такое средство будет гораздо проще, так как вы будете знать момент окончания функции. Советую вам получше обдумать силу функций `_penter` и `_pexit`.

Стандартный вопрос отладки

Почему в окне Disassembly вызовы некоторых функций начинаются с «ILT»?

В отрывке, сгенерированном ключом `/Gh`, вы видели вызов функции `CALL ILT+0(?Foo`, что многих программистов приводит в недоумение. Подобные вызовы свидетельствуют о магической компоновке с приращением в действии. Аббревиатура ILT означает Incremental Link Table (таблица компоновки с приращением).

При создании отладочной компоновки компоновщик хочет скомпоновать двоичный файл как можно быстрее. Для этого он добавляет в каждую функцию довольно много пустого места, благодаря чему при изменении функции ему нужно только перезаписать функцию, не перемещая код в двоичном файле. Между прочим, это пустое пространство заполняют команды `INT 3`. Однако размер функции может превысить объем выделенного места. В этом случае компоновщик должен переместить функцию в другое место двоичного файла.

Если каждая функция, вызывающая перемещенную функцию, делала это при помощи ее действительного адреса, то при каждом перемещении функции в результате новой компоновки компоновщик должен был бы искать и обновлять все команды `CALL`. Поэтому для экономии времени компоновщик создает специальную таблицу компоновки с приращением, которую использует для всех вызовов. Теперь при изменении функции компоновщик должен обновить в двоичном файле только одно выражение. Таблица ILT показана в листинге 19-1.

```

@ILT+0(_wmain):
00401005  jmp     wmain (401070h)
@ILT+5(??_GCResString@@UAEPAXI@Z):
0040100A  jmp     CResString::'scalar deleting destructor' (401B40h)
@ILT+10(?ParseCommandLine@@YAHHQAPAGAAUtag_CMDOPTS@@@Z):
0040100F  jmp     ParseCommandLine (401439h)
@ILT+15(?ShowHelp@@YAXXZ):
00401014  jmp     ShowHelp (401644h)
@ILT+20(??1CResString@@UAE@XZ):
00401019  jmp     CResString::~CResString (401A00h)
@ILT+25(?LoadStringW@CResString@@QAEPBGI@Z):
0040101E  jmp     CResString::LoadStringW (401A30h)

```

```

@ILT+30(??2@YAPAXI@Z):
00401023 jmp      operator new (401B90h)
@ILT+35(??_GResString@@UAEPAIXI@Z):
00401028 jmp      CResString::'scalar deleting destructor' (401B40h)
@ILT+40(??0CResString@@QAE@PAUHINSTANCE_@@@Z):
0040102D jmp      CResString::CResString (401990h)
@ILT+45(??BCResString@@QBEPBGXZ):
00401032 jmp      CResString::operator unsigned short const * (401B20h)

```

Следовательно, команда `CALL @ILT+15(?ShowHelp@@YAXXZ)` на самом деле вызывает переход к метке `@ILT+15(?ShowHelp@@YAXXZ)`, которая представляет собой переход к реальной команде.

Листинг 19-1. PENTERHOOK.CPP

```

/*-----
Отладка приложений для Microsoft .NET и Microsoft Windows
Copyright © 1997-2003 John Robbins - All rights reserved.
-----*/
/*////////////////////
                Включение заголовочных файлов
////////////////////////////////////*/
#include "stdafx.h"
#include "SWSDDL.h"
#include "SymbolEngine.h"
#include "VerboseMacros.h"
#include "ModuleItemArray.h"

/*////////////////////////////////////
                Определения, константы, объявления typedef
////////////////////////////////////*/
// Описатель события, который функция _penter проверяет,
// чтобы узнать, не запрещена ли обработка.
static HANDLE g_hStartStopEvent = NULL ;
// Простой автоматический класс - я использую
// его для разного рода инициализации.
class CAutoMatic
{
public :
    CAutoMatic ( void )
    {
        g_hStartStopEvent = ::CreateEvent ( NULL
                                           ,
                                           TRUE
                                           ,
                                           FALSE
                                           ,
                                           SWS_STOPSTART_EVENT ) ;
        ASSERT ( NULL != g_hStartStopEvent ) ;
    }
    ~CAutoMatic ( void )
    {
        VERIFY ( ::CloseHandle ( g_hStartStopEvent ) ) ;
    }
}

```

см. след. стр.

```

    }
};

/*////////////////////////////////////
        Глобальные объекты с областью видимости файл
////////////////////////////////////*/
// Автоматический класс.
static CAutoMatic g_cAuto ;

// Массив модулей.
static CModuleItemArray g_cModArray ;

/*////////////////////////////////////
        Прототипы функций
////////////////////////////////////*/

/*////////////////////////////////////
////////////////////////////////////*/

extern "C" void SWSDDL_DLLINTERFACE __declspec(naked) _penter ( void )
{
    DWORD_PTR dwCallerFunc ;

    // Пролог функции.
    __asm
    {
        PUSH EBP                // Создание стандартного кадра стека.
        MOV  EBP , ESP

        PUSH EAX                // Сохранение EAX, так как он нужен
                                // мне до сохранения всех регистров.
        MOV  EAX , ESP          // Загрузка текущего указателя стека
                                // в регистр EAX.

        SUB  ESP , __LOCAL_SIZE // Резервирование пространства
                                // для локальных переменных.

        PUSHAD                   // Сохранение значений всех
                                // регистров общего назначения.

        // Теперь я могу вычислить адрес возврата.
        ADD  EAX , 04h + 04h     // Нужно учесть команды PUSH EBP
                                // и PUSH EAX.
        MOV  EAX , [EAX]         // Получение адреса возврата.
        SUB  EAX , 5              // Чтобы определить начало функции,
                                // надо вычесть 5-байтовый переход,
                                // использованный для вызова _penter.
        MOV  [dwCallerFunc] , EAX // Сохранение нового адреса возврата.
    }
}

```

```

}

// Если событие начала/завершения находится
// в сигнальном состоянии, ничего не делаем.
if ( WAIT_TIMEOUT == WaitForSingleObject ( g_hStartStopEvent , 0 ))
{
    // Выполняем всю работу.
    g_cModArray.IncrementFunctionEntry ( dwCallerFunc );
}
// Эпилог функции.
__asm
{
    POPAD                                // Восстановление всех регистров
                                        // общего назначения.

    ADD ESP , __LOCAL_SIZE              // Удаление пространства, выделенного
                                        // для локальных переменных.

    POP EAX                             // Восстановление регистра EAX.

    MOV ESP , EBP                       // Восстановление кадра стека.
    POP EBP

    RET                                 // Возврат в вызвавшую функцию.
}
}

```

Формат файла .SWS и перечисление символов

Как показывает листинг 19-1, в `_penter` ничего удивительного. Все становится интереснее, когда дело касается организации адресов функций. Так как мне нужно связать адрес с именем функции, то в некоторых местах программы я прибегаю к услугам своего старого друга — сервера символов `DBGHELP.DLL`. Однако просмотр символов при помощи сервера символов — не самая быстрая операция, а доступ к данным нужен при каждом вызове функции, поэтому я должен был найти компактный и быстрый способ его выполнения.

Размышляя об этом, я захотел упорядочить данные при помощи отсортированного массива всех адресов функций с соответствующими им счетчиками вызовов. В этом случае, получив адрес возврата в `_penter`, я мог бы просто выполнить для него быстрый двоичный поиск. Такое решение казалось относительно простым, потому что оно требовало только перечисления символов модулей и сортировки массива функций. Все данные для этого у меня имелись.

Я решил, что SWS подобно WST должна хранить счетчики вызовов для каждого запуска каждого модуля в отдельном файле данных. Я предпочел этот подход, потому что он позволяет удалить информацию о конкретном запуске приложения из объединенного набора данных, если она вам не нужна. WST использует для наименования файлов формат `<имя модуля>.<номер вызова>`, но я хотел, чтобы SWS поддерживала схему `<имя модуля>.<номер вызова>.SWS`, чтобы я мог в конеч-

ном счете написать графическую программу, облегчающую объединение данных обо всех запусках.

Выбрав способ обработки данных в период выполнения, я приступил к рассмотрению создания файла порядка. Как я уже говорил, мне нужен был способ объединения информации об отдельных запусках. Однако при размышлении над фактическим созданием файла порядка я понял, что у меня нет некоторых данных. Файл порядка должен содержать имена функций, а также их размеры, в то время как я планировал хранить только адреса функций. Хотя я опять мог бы использовать символьную машину при генерировании файла порядка, единственный способ получения размера символа — перечисление всех символов модуля. Так как я уже выполнял полное перечисление символов на первоначальных этапах генерирования данных, я решил, что мне следует просто добавить в файл размеры функций. Я не нуждаюсь в хранении имен функций, потому что их всегда можно узнать, загрузив PDB-файл для двоичного файла.

Если вы все же нашли книгу Расса Блейка «Optimizing Windows NT» и прочитали главу «Tuning the Working Set of Your Application» (настройка рабочего набора программы), вас, вероятно, интересует, почему я ничего не говорю о наборах битов и интервалах времени. Группа, работавшая над производительностью Windows NT, использовала при создании WST схему, в которой каждой функции соответствует один бит из набора. Каждые столько-то секунд WST регистрирует при помощи этого набора битов функции, выполненные за прошедший интервал времени. Меня часто удивляет, почему они реализовали WST именно так. На первый взгляд, набор битов позволяет экономить память, но при этом нужно помнить, что должен быть реализован некоторый способ отображения битов и адресов функций. Не думаю, что такая схема экономит намного больше памяти, чем мой метод. Мне кажется, что программисты, работавшие над производительностью Windows NT, использовали набор битов потому, что оптимизировали при помощи WST целую ОС. Я же, напротив, работаю с отдельными двоичными файлами, так что это вопрос масштаба.

Разрабатывая структуры данных, я был озабочен одним моментом: при вызове функции я просто хотел увеличивать счетчик. В многопоточных программах я должен защищать это значение, чтобы в каждый конкретный момент времени им мог манипулировать только один поток. Я хотел сделать SWS как можно более быстрой, поэтому увеличение счетчика вызовов функций лучше всего выполнять при помощи API-функции `InterlockedIncrement`. Так как она использует аппаратный механизм блокировки (префикс команды `LOCK`), то гарантирует согласованность данных в многопоточных приложениях. Однако в Microsoft Win32 наибольшим числом, которое можно передать в `InterlockedIncrement`, является 32-разрядное значение, в связи с чем возникает проблема с превышением 4 294 967 295 вызовов функций. Четыре миллиарда — много, но и этого может не хватить для некоторых циклов сообщений при долгом выполнении приложения.

Для решения этой проблемы программа WST в период выполнения только записывала вызовы функций, а общее их число подсчитывалось постфактум, когда проще обрабатывать возможное переполнение. При настройке ОС вероятность выполнения некоторых функций более 4 миллиардов раз довольно высока. Од-

нако, планируя оптимизировать программу при помощи SWS, я рассматриваю эту проблему в самом начале и провожу тестирование, чтобы определить, возможно ли переполнение счетчика вызовов функций. Вероятность переполнения счетчика функций в SWS можно уменьшить, потому что я оставил ловушки, позволяющие начинать и прекращать сбор данных. Так что при каждом запуске приложения вы можете генерировать данные, только когда они вам нужны.

В тех крайне редких случаях, когда функция выполняется более 4 миллиардов раз, у вас есть два варианта. Во-первых, можно реализовать переменную счетчика как 64-разрядное целое, защитив ее при помощи объекта синхронизации, отдельного для каждого модуля. Другой, более радикальный вариант — разработка схемы, подобной алгоритму WST. Конечно, есть и еще один вариант: так как проблема переполнения возможна только в Win32, можно реализовать SWS только для Microsoft Win64. Даже за всю свою жизнь вы не сможете выполнить 18 446 744 073 709 551 615 вызовов функции.

Изучив листинг 19-2, вы увидите, что формат SWS-файла очень прост. Я быстро понял, что для обработки всех манипуляций с файлами мне нужен общий базовый класс — CSWSFile, определенный в файлах SWSFILE.H и SWSFILE.CPP. По сути этот класс — не более чем оболочка для обычных действий с файлами Win32.

Листинг 19-2. FILEFORMAT.H

```
/*-----
Отладка приложений для Microsoft .NET и Microsoft Windows
Copyright © 1997-2003 John Robbins - All rights reserved.
-----*/

#ifndef _FILEFORMAT_H
#define _FILEFORMAT_H

/*//////////////////////
Директивы define и объявления структур
////////////////////////*/
// Сигнатура SWS-файла (SWS2).
#define SIG_SWSFILE '2SWS'
#define EXT_SWSFILE_T ( ".SWS" )

/*//////////////////////
Заголовок SWS-файла.
////////////////////////*/
typedef struct tag_SWSFILEHEADER
{
    // Сигнатура файла. См. определения SIG_*, указанные выше.
    DWORD   dwSignature ;
    // Время компоновки двоичного файла, ассоциированного с этим файлом.
    DWORD   dwLinkTime ;
    // Адрес загрузки двоичного файла.
    DWORD64 dwLoadAddr ;
    // Размер образа.
    DWORD   dwImageSize ;
}
```

см. след. стр.

```

// Число записей в этом файле.
DWORD   dwEntryCount ;
// Поле флагов.
DWORD   dwFlags ;
// Имя модуля для этого файла.
TCHAR   szModuleName[ MAX_PATH ] ;
DWORD   dwPadding ;
} SWSFILEHEADER , * LPSWSFILEHEADER ;

/*//////////////////////////////////////
Тип записи SWS-файла.
//////////////////////////////////////*/
typedef struct tag_SWSENTRY
{
    // Адрес функции.
    DWORD64 dwFnAddr ;
    // Размер функции.
    DWORD   dwSize ;
    // Счетчик вызовов.
    DWORD   dwExecCount ;
} SWSENTRY , * LPSWSENTRY ;

#endif // _FILEFORMAT_H

```

В связи с форматом SWS-файла я хочу обратить ваше внимание на то, что я храню в нем адрес загрузки двоичного файла. Сначала я хранил в нем только адреса функций, но потом вспомнил о возможности перемещения двоичного файла в памяти. В этой ситуации SWSDLL.DLL могла бы быть вызвана с каким-либо адресом, и у меня не было бы никакой записи для этого адреса в SWS-файлах, загруженных для модуля. Хотя всем нам следует модифицировать базовые адреса наших DLL, иногда мы про это забываем, и я хотел, чтобы SWS правильно обрабатывала такие случаи.

Некоторые проблемы у меня вызвало генерирование символов для первоначального модуля SWS. Из-за особенностей компоновки программ и генерирования символов многие символы модуля не имеют в себе вызовов `_penter`. Скажем, при компоновке программы со статической стандартной библиотекой C в вашем модуле будет содержаться множество стандартных функций C. Чтобы ускорить просмотр адресов утилитой SWS, я реализовал несколько способов сокращения числа символов.

Функция обратного вызова для перечисления символов и некоторые мои попытки ограничения их числа приведены в листинге 19-3. Прежде всего я реализовал проверку того, имеет ли символ соответствующую информацию о номере строки. Так как я полагал, что функции, содержащие в себе вызовы `_penter`, будут скомпилированы правильно, с соблюдением всех вышеописанных этапов, я смог безопасно избавиться от многих посторонних символов. Следующий шаг на пути к устранению символов заключался в проверке, являются ли частью символов специфические строки. Например, все символы, начинающиеся с `_imp__`, представляют собой функции, импортируемые из других DLL. Еще две проверки я оставил

вам в качестве упражнения. Во-первых, вы должны реализовать список для игнорирования символов из специфических файлов. Эта функция полезна в первую очередь тем, что позволит игнорировать исходные файлы стандартной библиотеки C. Во-вторых, не помешала бы возможность проверить, что интересующие вас адреса относятся только к разделу кода модуля. Возможно, эта проверка вам не нужна, но так вы смогли бы гарантировать, что используются только истинные символы кода.

Листинг 19-3. Перечисление символов SWS

```

/*-----
ФУНКЦИЯ:   SymEnumSyms
ОПИСАНИЕ:
    Функция обратного вызова для перечисления символов. Эта функция только
    добавляет данные в SWS-файлы и все.
ПАРАМЕТРЫ:
    szSymbolName      - имя символа.
    ulSymbolAddress   - адрес символа.
    ulSymbolSize      - размер символа в байтах.
    pUserContext      - файл SWS.

ВОЗВРАЩАЕМОЕ ЗНАЧЕНИЕ:
    TRUE  - все отлично.
    FALSE - при добавлении данных в файл возникла проблема.
-----*/
BOOL CALLBACK SymEnumSyms ( PSTR      szSymbolName      ,
                           DWORD64  ulSymbolAddress   ,
                           ULONG     ulSymbolSize      ,
                           PVOID     pUserContext      )
{
    LPENUMSYMCTX pCTX = (LPENUMSYMCTX)pUserContext ;

    CImageHlp_Line cLine ;
    DWORD dwDisp ;
    if ( FALSE == g_cSym.SymGetLineFromAddr ( ulSymbolAddress ,
                                              &dwDisp          ,
                                              &cLine            ) )
    {
        // Если для символа не было обнаружено исходного файла
        // и номера строки, игнорируем его.
        return ( TRUE ) ;
    }

    // Будущие улучшения для игнорирования конкретных символов:
    // 1. Реализуйте проверку того, не находится ли файл
    //    в списке игнорируемых файлов.
    // 2. Проверяйте, относится ли адрес к разделу кода модуля.
    // Это позволит избежать добавления в итоговые файлы символов IAT.

    // Есть ли этот символ в списке игнорируемых символов?

```

см. след. стр.

```

for ( int i = 0 ; i < IGNORE_CONTAINING_COUNT ; i++ )
{
    if ( NULL != strstr ( szSymbolName
                        , g_szIgnoreContaining[ i ] ) )
    {
        // Выход.
        return ( TRUE ) ;
    }
}

if ( NULL != pCTX->pfnVerboseOutput )
{
#ifdef _WIN64
    pCTX->pfnVerboseOutput(_T(" Adding Symbol : 0x%016I64X %S\n"),
#else
    pCTX->pfnVerboseOutput(_T(" Adding Symbol : 0x%08X %S\n" ) ,
#endif
                        (DWORD_PTR)ulSymbolAddress
                        , szSymbolName );
}
if ( FALSE == pCTX->pSWSFile->AddData ( ulSymbolAddress ,
                                        ulSymbolSize ,
                                        0 ) )
{
    ASSERT ( !"Adding to SWS file failed!" ) ;
    return ( FALSE ) ;
}
pCTX->iAddedCount++ ;
return ( TRUE ) ;
}

```

Период выполнения и оптимизация

Одна проблема с символами в период выполнения была связана с тем, что символьная машина не возвращает статические функции. Становясь подозрительным, если я не находил в модуле адрес, я, как обычно, включал в программу вызовы 6–7 диагностических информационных окон. Сначала я несколько смутился тем, что видел диагностические сообщения, так как в одной из моих тестовых программ никакая функция не была объявлена статической. Взглянув на стек в отладчике, я увидел символ с именем наподобие \$E127. В функции имелся вызов `_penter`, и все казалось правильным. Наконец я понял, что это функция, сгенерированная компилятором, такая как конструктор копий. Хотя мне по-настоящему нравится выполнять проверку ошибок в коде, я заметил, что в некоторых программах хватало этих сгенерированных компилятором функций, поэтому я мог только сообщить о проблеме в отладочных компоновках при помощи TRACE.

Последний интересный аспект SWS — оптимизация модуля. Функция `TuneModule` довольно объемна, поэтому в листинге 19-4 я привел только ее алгоритм. Как вы можете увидеть, на каждой странице кода я размещаю как можно больше функ-

ций, чтобы исключить пустое пространство. Наибольший интерес представляет поиск функции, лучше всего соответствующей странице. Я решил размещать на странице в первую очередь как можно больше функций с ненулевым числом вызовов. Если я не мог найти такую функцию, я располагал на странице функцию с нулевым числом вызовов. Мой первоначальный алгоритм работал великолепно. Однако при оптимизации определенных модулей он стал приводить к ошибкам.

Небольшое исследование показало, что в таких ситуациях страница была почти заполнена и при этом требовалось обработать функцию, размер которой превышал размер страницы. Да, я не ошибся: размер функции, сообщенный символьной машиной, был больше, чем размер страницы памяти. Изучив проблему тщательней, я заметил, что эти огромные функции появлялись, только когда они были последними символами в разделе кода. Очевидно, символьная машина считает всю информацию, расположенную после определенных символов, их частью, возвращая в результате этого ошибочное значение. В алгоритме оптимизации вы можете увидеть, что, когда размер символа превышает размер страницы, мне ничего не остается, кроме как записать символ в файл порядка. Это не лучшее решение, однако и подобная ситуация встречается не часто.

Листинг 19-4. Алгоритм настройки SWS

```
// Алгоритм функции TuneModule.  
BOOL TuneModule ( LPCTSTR szModule )  
{  
    Сгенерировать имя SWS-файла вывода.  
  
    Скопировать базовый SWS-файл во временный файл.  
  
    Открыть временный SWS-файл.  
  
    Для каждого файла szModule.#.SWS в этом каталоге  
    {  
        Проверить, что время компоновки этого файла #.SWS  
        соответствует времени компоновки временного SWS-файла.  
  
        Для каждого адреса в этом файле #.SWS  
        {  
            Прибавить значение счетчика вызовов для этого адреса  
            к аналогичному счетчику во временном файле.  
        }  
    }  
  
    Получить размер страницы этого компьютера.  
  
    Пока не готово.  
    {  
        Найти первую запись во временном SWS-файле,  
        для которой указан адрес.  
  
        Если я проверил все адреса, но такой записи не нашел.
```

см. след. стр.

```
{
    готово = TRUE
    Прервать цикл.
}

Если счетчик вызовов для этой записи равен 0.
{
    готово = TRUE
    Прервать цикл.
}

Если размер этой записи меньше, чем оставшееся
пространство страницы.
{
    Вывести имя функции этой записи в PRF-файл.
    Обнулить адрес, чтобы я не использовал его еще раз.
    Вычесть размер этой записи из размера оставшегося
    пространства страницы.
}
Иначе, если размер этой записи больше, чем размер страницы
{
    Просто записать адрес в PRF-файл,
    так как я ничего не могу сделать.
    Присвоить значению оставшегося объема страницы
    размер всей страницы.
}
Иначе.
{
    // Эта запись слишком велика для размещения
    // на странице, поэтому выполняется поиск функции,
    // лучше всего подходящей для этой страницы.
    Для каждого элемента временного SWS-файла.
    {
        Если адрес имеет ненулевое значение.
        {
            Если наилучшее соответствие не найдено.
            {
                Обозначить эту запись как наилучшее соответствие
                в целом.
                Обозначить эту запись как наилучшее соответствие
                по числу вызовов.
            }
            Иначе.
            {
                Если размер этой записи > размер наилучшего соотв-ия
                {
                    Обозначить эту запись как наилучшее
                    соответствие в целом.
                }
            }
        }
    }
}
```

```

        Если счетчик вызовов для этой записи не равен 0.
        {
            Если размер этой записи > размер наилучшей
            записи по числу вызовов.
            {
                Обозначить эту запись как наилучшее
                соответствие по числу вызовов.
            }
        }
    }

    Если наилучшее соответствие по числу вызовов не найдено.
    {
        Считать наилучшей записью по числу вызовов
        наилучшую запись в целом.
    }
    Вывести имя функции, наилучшей по числу вызовов в PRF-файл.
    Присвоить значению оставшегося объема страницы размер всей
    страницы.
}

}
Закреть все временные файлы
}

```

Что после SWS?

Как я уже говорил, SWS обеспечивает довольно хорошую возможность оптимизации программ. Если вам хочется сделать ее еще лучше, вот несколько полезных советов.

- Напишите программу начала и прекращения сбора данных. В функции `_penter` я выполняю проверку того, находится ли событие в сигнальном состоянии. Вы можете написать отдельную программу, которая будет генерировать событие, управляющее сбором данных утилитой SWS. Просто создайте событие с именем `SWS_Start_Stop_Event` и генерируйте его, когда хотите прекратить накопление данных.
- Реализуйте упомянутые мной возможности исключения символов, чтобы их число в ваших SWS-файлах было минимальным.
- Если вы по-настоящему честолобивы, создайте для просмотра данных и оптимизации программу с графическим интерфейсом. Работать с ней будет гораздо удобней, чем с утилитой, основанной на командной строке.

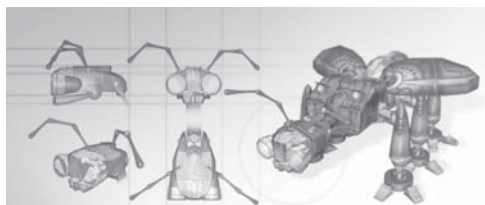
Резюме

Чрезмерный объем кода не имеет никаких оправданий, поэтому в качестве заключительной оптимизации приложения всегда следует выполнять упорядочение и максимальное уплотнение двоичных файлов. SWS позволяет посадить файлы на диету и уменьшить рабочий набор относительно безболезненно, особенно если использовать ее вместе с программой SettingsMaster, автоматизирующей управление конфигурацией проектов.

ЧАСТЬ V

ПРИЛОЖЕНИЯ





Чтение журналов Dr. Watson

Я надеюсь, что для облегчения отладки приложений вы будете включать в них возможность создания минидампов (см. главу 13) и вам не придется изучать журналы Dr. Watson. Однако, если у вас есть готовое приложение или ваши клиенты не могут высылать вам двоичные минидампы по электронной почте, Dr. Watson всегда сможет указать вам на время и место возникновения проблемы.

Пожалуй, доктора Ватсона следовало бы назвать доктором Джекилом и мистером Хайдом. В режиме доктора Джекила вы получаете информацию об ошибке на машине пользователя, легко находите место проблемы и быстро ее исправляете. В режиме мистера Хайда вы получаете лишь еще один ни о чем не говорящий набор чисел. В этом приложении я объясню работу с журналами Dr. Watson, что позволит вам реже встречаться с мистером Хайдом и чаще с доктором Джекилом.

На следующих страницах я рассмотрю полный журнал Dr. Watson, объясняя по ходу дела всю важную информацию (релевантные данные в конкретном разделе выделены полужирным начертанием). Этот журнал был создан в результате одной из ошибок ранней версии WDBG.EXE — отладчика, написанного мной для главы 4.

После знакомства с этой книгой ничто в журнале Dr. Watson не должно быть для вас незнакомым. Что до различий между журналами Dr. Watson в Microsoft Windows 2000, Windows XP и Windows Server 2003, то их немного. Однако, как вы увидите ниже, версии Dr. Watson в двух последних ОС несколько лучше.

Для получения журналов запустите Dr. Watson (DRWTSN32.EXE). В списке Application Errors (ошибки приложения) вы увидите недавние ошибки. Если этот список пуст, возможно, Dr. Watson не задан в качестве отладчика по умолчанию. Чтобы сделать Dr. Watson отладчиком по умолчанию, запустите его с ключом `-i`, т. е. введите выражение `drwtsn32 -i`. Для генерирования тестовой ошибки запустите

программу CrashTest.EXE из главы 13 и нажмите кнопку Crash Away (сгенерировать ошибку). Пользовательский интерфейс Dr. Watson изображен на рис. А-1.

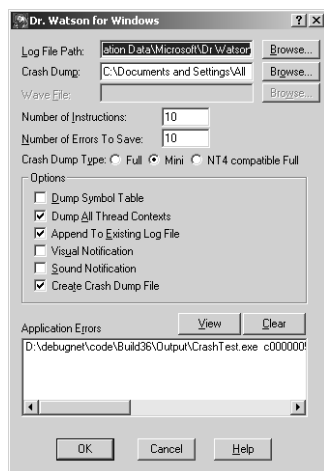


Рис. А-1. Пользовательский интерфейс Dr. Watson

Выберите в списке Application Errors интересующую вас ошибку и нажмите кнопку View (показать), после чего появится диалоговое окно Log File Viewer (просмотр журнала) (рис. А-2). В Windows 2000 в окне Application Errors вы увидите только номера ошибок приложений и их адреса. В Windows XP/Server 2003 вы увидите еще и имя процесса. Скопируйте из окна Log File Viewer описание конкретной ошибки. Если вам хочется получить минидамп последней ошибки, полный путь к нему указан в поле Crash Dump (аварийная копия памяти) окна Dr. Watson.

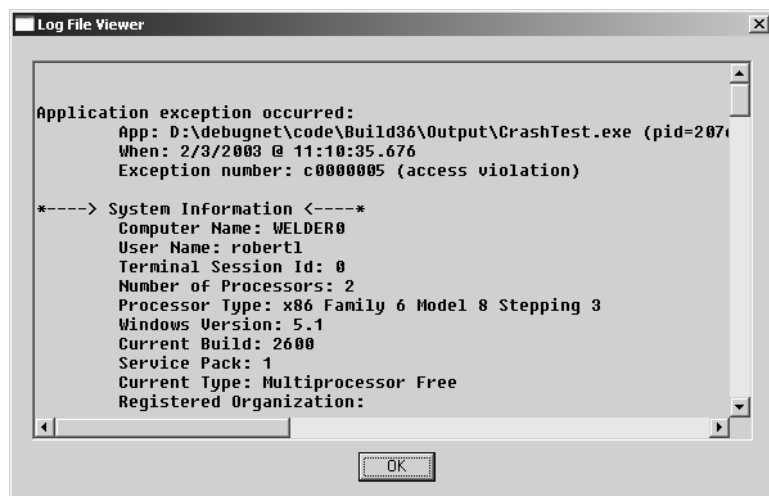


Рис. А-2. Диалоговое окно Log File View утилиты Dr. Watson для Windows XP

Журналы Dr. Watson

Первый раздел журнала Dr. Watson имеет вид:

Исключение в приложении:

Прил.: (pid=1796)

Время: 1/2/2003 @ 13:42:56.208

Номер: c0000005 (нарушение прав доступа)

Заголовок содержит информацию о причине ошибки: в моем примере это исключение в приложении. В случае некоторых ошибок номера исключений не всегда преобразуются в понятную людям форму, такую как «нарушение прав доступа» для исключения 0xC0000005. Все номера исключений вы можете узнать, отыскав в файле WINNT.H строки STATUS_. Коды ошибок указаны в документации как значения EXCEPTIОN_, возвращаемые функцией GetExceptionCode, однако реальные значения определяются в директивах #define STATUS_. После преобразования кода ошибки в значение EXCEPTIОN_ вы сможете просмотреть ее описание в документации к GetExceptionCode.

Раздел System Information (сведения о системе) в объяснении не нуждается:

--> Сведения о системе <--

Имя компьютера: HUME

Имя пользователя: john

Число процессоров: 2

Тип процессора: x86 Family 15 Model 0 Stepping 10

Версия Windows 2000: 5.0

Текущая сборка: 2195

Пакет обновления: 3

Текущий тип: Multiprocessor Free

Зарегистрированная организация: Wintellect

Зарегистрированный пользователь: John Robbins

Раздел Task List (список задач) выглядит так:

--> Список задач <--

0 Idle.exe

8 System.exe

132 smss.exe

160 csrss.exe

156 winlogon.exe

208 services.exe

220 lsass.exe

364 svchost.exe

424 svchost.exe

472 spoolsv.exe

504 MWMDMSVC.exe

528 MWSSW32.exe

576 regsvc.exe

592 MStask.exe

836 Explorer.exe

904 tp4mon.exe

912 tphkmgr.exe

```
920 4nt.exe
940 taskmgr.exe
956 tponscr.exe
268 msdev.exe
252 WDBG.exe
828 NOTEPAD.exe
416 drwtsn32.exe
0 _Total.exe
```

В разделе Task List выводится список процессов, выполнявшихся в момент ошибки. К сожалению, в нем нет информации об их версиях, поэтому вам придется узнать у пользователя версии файлов всех процессов. В левом столбце указаны десятичные идентификаторы процессов (PID), выполнявшихся в момент ошибки. После ошибки они совершенно бесполезны.

В разделе Module List (список модулей) указываются все модули, загруженные в момент ошибки в адресное пространство. Информация обо всех модулях имеет формат (адрес загрузки – максимальный адрес). В этом месте впервые появляются различия между журналами Dr. Watson в Windows 2000, Windows XP и Windows Server 2003. В журналах Windows 2000 вы увидите только диапазоны адресов модулей и больше ничего. Если ваше приложение откомпилировано в Microsoft Visual Studio 6 и для него доступны символы, то после диапазона адресов будут указаны загруженные символы. Так как DBGHELP.DLL, поставляемая с Windows 2000, ничего не знает о символах Microsoft Visual Studio .NET, вы никогда не увидите загруженные символы для двоичных файлов, откомпилированных в этой среде. Если в Windows 2000 список модулей имеет некоторые недостатки, то Dr. Watson в Windows XP/Server 2003 достаточно умен, чтобы указывать после каждого диапазона адресов имя соответствующей DLL.

Список модулей в Windows 2000

```
(00400000 - 00460000)
(77F80000 - 77FB0000)
(63000000 - 6301B000)
(77E10000 - 77E6F000)
(77E80000 - 77F31000)
```

Список модулей в Windows XP

```
(0000000000400000 - 0000000000460000: d:\Dev\BookTwo\Disk\Output\WDBG.exe
(00000000071c20000 - 00000000071c6e000: E:\WINDOWS\System32\NETAPI32.dll
(00000000075a70000 - 00000000075b15000: E:\WINDOWS\system32\USERENV.dll
(00000000075f40000 - 00000000075f5f000: E:\WINDOWS\system32\appHelp.dll
(000000000763b0000 - 000000000763f5000: E:\WINDOWS\system32\cmd1g32.dll
```

Если вы желаете узнать, какие модули были загружены в Windows 2000, вам остается только гадать. Однако, как я несколько раз говорил, чрезвычайно важно знать, в какие области адресного пространства процесса загружаются ваши DLL. Скорее всего вы сможете узнать свои DLL по адресам загрузки. Чтобы получить сведения о других DLL на компьютере пользователя, можно написать утилиту, которая просматривала бы их и сообщала их имена, адреса загрузки и размер.

Следующий фрагмент представляет собой начало раздела состояния потока, состоящего из трех частей (из-за размеров страницы мне пришлось удалить коды

операций, расположенные после адресов дизассемблированных команд, и перенести строки информации о регистрах).

--> State Dump for Thread Id 0xe14 (Копия памяти для потока 0xe14) <--

```

eax=00000000 ebx=00000000 ecx=011305d8 edx=00000a30 esi=00154b40
edi=0012fae4
eip=00410144 esp=0012faa8 ebp=0012faf0 iopl=0
nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000 efl=00000202

```

функция: WDBG!CWDBGProjDoc__HandleBreakpoint

```

0041012b push    esi
0041012c push    edi
0041012d push    ecx
0041012e lea     edi,[ebp-0x40]
00410131 mov     ecx,0xd
00410136 mov     eax,0xc0000000
0041013b rep     stosd
0041013d pop     ecx
0041013e mov     [ebp-0x10],ecx
00410141 mov     eax,[ebp+0xc]
СБОЙ -> 00410144 mov     ecx,[eax+0x4]      ds:0023:00000004=????????
00410147 cmp     dword ptr [ecx],0x80000003
0041014d jz      WDBG!CWDBGProjDoc__HandleBreakpoint+0x90 (004101a0)
0041014f mov     [ebp-0x14],esp
00410152 mov     [ebp-0x18],ebp
00410155 mov     esi,esp
00410157 push    0x456070
0041015c push    0x45606c
00410161 mov     edx,[ebp-0x18]
00410164 xor     eax,eax
00410166 push    eax

```

Dr. Watson отображает информацию о состоянии каждого потока, выполнявшегося в процессе в момент ошибки. Состояния потока содержат всю информацию, необходимую для обнаружения механизма и причин краха.

В разделе регистров указываются значения всех регистров в момент ошибки. Особое внимание следует уделить регистру EIP, указателю команд. Для моего примера из Windows XP у меня имелись символы, поэтому вы можете видеть, какую функцию выполнял этот поток в момент ошибки, но в большинстве журналов Dr. Watson информации о символах не будет. Конечно, если Dr. Watson не сообщает вам имя функции, это не проблема. Просто загрузите в программу CrashFinder из главы 12 проект CrashFinder вашего приложения, введите адрес в поле Hexadecimal Address(es) (шестнадцатеричные адреса) и нажмите кнопку Find (найти).

Поток из нашего фрагмента оказался потоком, вызвавшим ошибку. Об этом свидетельствует только указатель FAULT-> (СБОЙ->) в середине дизассемблированного листинга. Пару раз я видел журналы Dr. Watson, в которых не было указателя FAULT->. Если вы не можете найти в журнале этот указатель, изучите со-

стояние каждого потока и введите каждый адрес EIP в CrashFinder, чтобы узнать, какую команду выполнял поток в момент ошибки.

Если вы читали главу 7, дизассемблированный листинг должен быть вам понятен. Новыми элементами будут только значения, показанные после команд. Чтобы вы могли узнать, какие значения использовались командой, дизассемблер Dr. Watson пытается просмотреть эффективный адрес ссылки на память. Адреса, начинающиеся с букв *ss*, свидетельствуют о том, что происходил доступ к сегменту стека; *ds* — к сегменту данных. В Windows XP/Server 2003 эффективный адрес будет указан только после строки, на которую указывал регистр EIP в момент ошибки.

В журналах Dr. Watson из Windows 2000 эффективные адреса будут указаны после каждой ассемблерной команды. Однако при этом гарантируется правильность только того адреса, что указан в строке, на которой находился указатель команд. Другие адреса могут быть неверны, так как значения, используемые командой, могли измениться. Допустим, первая дизассемблированная команда в состоянии потока ссылалась на память при помощи регистра EBX. Если ошибка случилась после выполнения еще 10 команд, то одна из промежуточных команд легко могла изменить EBX. Однако, когда Dr. Watson в Windows 2000 дизассемблирует программу, для преобразования эффективного адреса он использует текущее значение EBX — то, которое имело место в момент ошибки. Поэтому эффективный адрес, показанный в дизассемблированном коде, может быть неверным. Итак, прежде чем поверить в значения эффективных адресов, убедитесь, что нужные регистры не были изменены никакой командой.

Благодаря недавно приобретенным навыкам работы с ассемблером, вы должны легко узнать, почему этот поток потерпел крах. Читая ассемблерный листинг Dr. Watson (или отладчика), большинство программистов допускает серьезную ошибку: они изучают его сверху вниз. Настоящая хитрость в том, чтобы начать исследование с места ошибки и постепенно подниматься вверх в поисках команды, присвоившей значения регистрам, использованным в команде, вызвавшей ошибку.

В нашем случае поток потерпел крах на команде `00410144 MOV ECX, [EAX+0x4]`, при которой регистр EAX имел значение 0. В Microsoft Windows все адреса, расположенные ниже 64 кб, отмечены как не имеющие доступа, поэтому попытка чтения памяти по адресу `0x00000004` — не лучшая идея. Итак, мы должны найти команду, заносщую 0 в EAX. Поднявшись на одну строку, вы увидите команду `MOV EAX, [EBP+0xC]`. Помните, что второй операнд, источник, помещается в первый операнд, приемник (иначе говоря, помните про правило «от источника к приемнику»). Это значит, что в EAX было скопировано значение, находившееся по адресу `[EBP+0xC]`. Следовательно, по адресу `[EBP+0xC]` располагался 0.

В этот момент вы должны вспомнить еще одну хитрость, которую я описал в главе 7: «параметры располагаются по положительным смещениям»! Параметры располагаются по положительным смещениям от регистра EBP, причем первый находится по адресу `[EBP+0x8]`, а каждый следующий отстоит от предыдущего на 4 байта. Так как `0xC` на 4 байта больше, чем `0x8`, я могу предположить, что ошибка была вызвана тем, что второй параметр этой функции был равен `NULL` (надеюсь, прочитав эти два абзаца, вы поняли, как важно знать ассемблер в достаточном объеме для чтения журналов Dr. Watson!).

--> Stack Back Trace (Обратная трассировка стека) <--

ChildEBP	RetAddr	Args	to Child
0012faf0	004100cd	00000a30	00000000 80000003 WDBG!CWDBGProjDoc__HandleBreakpoint +0x34
0012fb0c	004075f1	00000a30	0164f8fc 01130b68 WDBG!CWDBGProjDoc__HandleExceptionEvent+0x6d
0012fb20	7c3422b2	00000a30	0164f8fc 0000000d WDBG!CDocNotifyWnd__HandleExceptionEvent+0x21
0012fc28	7c341b2e	00000502	00000a30 0164f8fc MFC71UD!CWnd__OnWndMsg+0x752
0012fc48	7c33f2f0	00000502	00000a30 0164f8fc MFC71UD!CWnd__WindowProc+0x2e
0012fcc0	7c33f7ce	01130b68	002502ca 00000502 MFC71UD!AfxCallWndProc+0xe0
0012fce0	7c3b072a	002502ca	00000502 00000a30 MFC71UD!AfxWndProc+0x9e
0012fd10	77d67ad7	002502ca	00000502 00000a30 MFC71UD!AfxWndProcBase+0x4a
0012fd3c	77d6ccd4	7c3b06e0	002502ca 00000502 USER32!SetWindowPlacement+0x57
0012fda4	77d445bd	00000000	7c3b06e0 002502ca USER32!DefRawInputProc+0x284
0012fdf8	77d447d4	00593330	00000502 00000a30 USER32!TranslateMessageEx+0x78d
0012fe20	77fb4da6	0012fe30	00000018 00593330 USER32!DefWindowProcA+0x209
0012fe64	7c34e8e1	00154af8	00000000 00000000 ntdll!KiUserCallbackDispatcher+0x13
0012fe90	7c34fb4c	00455e30	0012feb8 7c34f407 MFC71UD!AfxInternalPumpMessage+0x21
0012fe9c	7c34f407	00000001	00455e30 00154ac8 MFC71UD!CWinThread__PumpMessage+0xc
0012feb8	7c34fe87	00455e30	00434ffa 0040a66b MFC71UD!CWinThread__Run+0x87
0012fecc	7c34865a	1020c034	102682d0 ffffffff MFC71UD!CWinApp__Run+0x57
0012fef0	00430008	00400000	00000000 00020c22 MFC71UD!AfxWinMain+0xda
0012ff08	004284b8	00400000	00000000 00020c22 WDBG!wWinMain+0x18
0012ffc0	77e814c7	00140000	01f88550 7ffdf000 WDBG!wWinMainCRTStartup+0x1f8
0012fff0	00000000	004282c0	00000000 78746341 kernel32!GetCurrentDirectoryW+0x44

В моем примере журнала Dr. Watson из Windows XP имеются символы, но в журналах ваших пользователей их скорее всего не будет. Однако в столбце RetAddr указаны адреса возврата из функций в стеке вызовов. Если в журнале вашего клиента нет символов, то, чтобы узнать приведшую к ошибке последовательность вызовов функций, нужно только загрузить каждый адрес из столбца RetAddr в CrashFinder.

В столбцах Args To Child выводятся три первых параметра функции в стеке. В случае оптимизированных заключительных компоновок при отсутствии символов эти значения, вероятно, будут ошибочными. Однако вы все же можете использовать их в качестве отправной точки для изучения своего кода «вручную».

В Windows 2000 значения символов утилитой Dr. Watson не обрабатываются, поэтому ваши журналы будут выглядеть так:

--> Обратная трассировка стека <--

FramePtr	ReturnAd	Param#1	Param#2	Param#3	Param#4	Function Name
0012FBA0	004100CD	00000714	00000000	80000003	CCCCCCCC	!<nosymbols>
0012FBBC	004075F1	00000714	018EF8FC	016705F8	0012FCD8	!<nosymbols>
0012FBD0	7C3422B2	00000714	018EF8FC	0000000D	016705F8	!<nosymbols>
0012FCD8	7C341B2E	00000502	00000714	018EF8FC	0012FCF4	!Ordinal6841
0012FCF8	7C33F2F0	00000502	00000714	018EF8FC	0013BD01	!Ordinal8666
0012FD70	7C33F7CE	016705F8	00090500	00000502	00000714	!Ordinal1363
0012FD90	7C3B072A	00090500	00000502	00000714	018EF8FC	!Ordinal1580
0012FDC0	77E3A244	00090500	00000502	00000714	018EF8FC	!Ordinal1581
0012FDE0	77E14730	7C3B06E0	00090500	00000502	00000714	
						user32!SetWindowPlacement
0012FDFC	77E1558A	00517E40	00000502	00000714	018EF8FC	
						user32!TranslateMessageEx
0012FE24	77FA02FF	0012FE34	00000018	00517E40	00000502	user32!DefWindowProcA
0012FE64	7C34E8E1	00136BC0	00000000	00000000	00000000	
						ntdll!KiUserCallbackDispatcher
0012FE90	7C34FB4C	00455E30	0012FEB8	7C34F407	00000001	!Ordinal1462
0012FE9C	7C34F407	00000001	00455E30	00136B90	00000002	!Ordinal7046
0012FEB8	7C34FE87	00455E30	00434FFA	0040A66B	0012FEF0	!Ordinal7554
0012FECC	7C34865A	1020C034	102682D0	FFFFFFFF	0012FEF0	!Ordinal7553
0012FEF0	00430008	00400000	00000000	000209D8	00000005	!Ordinal1578
0012FF08	004284B8	00400000	00000000	000209D8	00000005	!<nosymbols>
0012FFC0	77EA847C	0013BD01	0013BD01	7FFDF000	C0000005	!<nosymbols>
0012FFF0	00000000	004282C0	00000000	000000C8	00000100	
						kernel32!ProcessIdToSessionId

Имена функций выводятся в формате <модуль>!<функция>. Функции, показанные как Ordinal#, — это функции, экспортируемые по ординалу. Если у вас нет исходного кода DLL, функции которой экспортируются по ординалу, вам не повезло. Однако у вас есть исходный код библиотеки Microsoft Foundation Class (MFC), поэтому вы можете просмотреть значения ординалов MFC. Например, я знаю, что в программе WDBG по адресу 0x7C250000 загружается библиотека MFC71UD.DLL, благодаря чему я могу просмотреть ординалы, так как все функции MFC экспортируются по ординалам посредством файла определений компоновщика (DEF-файла).

Единственное условие успешного преобразования значений ординалов MFC в имена функций состоит в том, что вы должны знать версию DLL библиотеки MFC на компьютере, на котором произошла ошибка. В моей системе — \\HUME — установлена MFC71UD.DLL из Visual Studio .NET 2003. Если вы не уверены в версии MFC, установленной на машине пользователя, спросите его об этом или просто потребуйте нужную информацию — как видите, у вас есть выбор.

Чтобы преобразовать ординалы в имена функций, выполните следующие простые действия.

1. Откройте подкаталог <каталог установки Visual Studio .NET > \VC7\ATLMFC\SRC\MFC\Intel.
2. Выберите DEF-файл, соответствующий нужному вам файлу MFC. Например, файлу MFC71UD.DLL соответствует файл MFC71UD.DEF.
3. Отыщите номер ординала. Для нахождения Ordinal6841 из предыдущего стека я поискал бы в файле MFC71UD.DEF значение 6841. Значение 6841 содержится в строке «?OnWndMsg@CWnd@@MAEHIIJPAJ@Z @ 6841 NONAME».
4. Слева от выражения «@6841 NONAME» указано расширенное имя функции, экспортируемой по данному ординалу. Для приведения имени функции в нормальный вид используйте программу UNDNAM.EXE из состава Visual Studio .NET. В нашем случае это функция CWnd::OnWndMsg.
5. Третья, заключительная часть состояния потока указана в разделе Raw Stack Dump (копия необработанного стека):

```
*--> Копия необработанного стека <--*
0012faa8 a8 fc 12 00 40 4b 15 00 - cc cc cc cc cc cc cc cc
                                ....@K.....
0012fab8 cc cc cc cc cc cc cc cc - cc cc cc cc cc cc cc cc
                                .....
0012fac8 cc cc cc cc cc cc cc cc - cc cc cc cc cc cc cc cc
                                .....
0012fad8 cc cc cc cc cc cc cc cc - d8 05 13 01 1c fc 12 00
                                .....
0012fae8 af 55 43 00 ff ff ff ff - 0c fb 12 00 cd 00 41 00
                                .UC.....A.
0012faf8 30 0a 00 00 00 00 00 00 - 03 00 00 80 cc cc cc cc
                                0.....
0012fb08 d8 05 13 01 20 fb 12 00 - f1 75 40 00 30 0a 00 00
                                .... ..u@.0...
0012fb18 fc f8 64 01 68 0b 13 01 - 28 fc 12 00 b2 22 34 7c
                                ..d.h... (...)"4|
0012fb28 30 0a 00 00 fc f8 64 01 - 0d 00 00 00 68 0b 13 01
                                0.....d....h...
0012fb38 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
                                .....
0012fb48 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00
                                .....
0012fb58 b2 98 d4 77 85 70 9b c2 - 00 00 00 00 cc a5 30 7c
                                ...w.p.....0|
0012fb68 8c fb 12 00 c7 a7 28 7c - 01 00 00 00 4c f5 25 7c
```

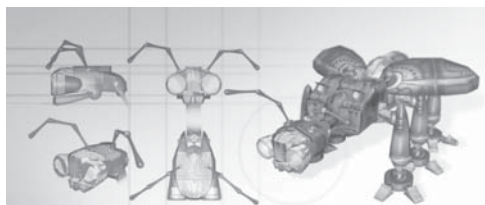


```

.....(|....L.%|
0012fb78 5c 01 00 00 03 00 00 00 - 04 00 00 00 e0 f4 25 7c
\.....%|
0012fb88 cc fb 12 00 b4 fb 12 00 - 91 a7 28 7c 4c f2 43 7c
.....(|L.C|
0012fb98 4c f5 25 7c 5c 01 00 00 - 03 00 00 00 04 00 00 00
L.%|\.....
0012fba8 e0 f4 25 7c cc fb 12 00 - 00 00 00 00 4c fc 12 00
..%|.....L...
0012fbb8 b7 b4 39 7c e8 fb 12 00 - 03 00 00 00 04 00 00 00
..9|.....
0012fbc8 e0 f4 25 7c 68 4f 25 7c - ca 02 25 00 f4 fb 12 00
..%|h0%|..%....
0012fbd8 30 0a 00 00 32 b5 39 7c - 00 00 00 00 00 00 00 00
0...2.9|.....

```

Я редко использую эту информацию. Однако, попадая в тупик, я иногда обращаюсь к ней, пытаясь определить значения локальных переменных. Три адреса возврата, которые я могу сопоставить с описанным ранее анализом стека, выделены курсивом.



Ресурсы для разработчиков приложений .NET и Windows

Как я говорил в главе 1, успешно отлаживать программы может только высококвалифицированный программист. Лучшие специалисты по отладке приложений обладают рядом навыков и, что наиболее важно, широким диапазоном знаний. От понимания вами языков, технологий, архитектуры ОС и механизма работы процессора очень часто будет зависеть, как скоро вы решите проблему: в течение нескольких минут или проведете несколько дней за отладчиком, только пытаясь выяснить, что же происходит.

Я привел этот список ресурсов потому, что меня постоянно спрашивают, какие источники я использую для изучения разработки приложений Microsoft .NET и Microsoft Windows. Помните, что он ни в коем случае не является исчерпывающим и вы можете не согласиться с некоторыми моими советами. Все мои предложения носят рекомендательный характер, кроме одного. Если вы разрабатываете программы для Windows, вам абсолютно необходимо подписаться на компакт-диски Microsoft Developer Network (MSDN), иначе даже не мечтайте о профессиональной разработке. Конечно, Microsoft поддерживает проект MSDN Online, однако кое-какую информацию можно найти только на CD. Чтобы узнать условия подписки на MSDN, посетите сайт msdn.microsoft.com/subscriptions.

Хочу обратить ваше внимание, что я участвовал в создании некоторых рекомендуемых мной программ и сотрудничал с авторами ряда указанных книг. Думаю, с моей стороны было бы некрасиво скрывать это.

Книги

Во время разработки и отладки приложений я часто обращаюсь к следующим книгам. Настоятельно рекомендую их вам.

Разработка ПО

- Steve McConnell. Code Complete. — Microsoft Press, 1993
Лучшая книга по разработке ПО из тех, что когда-либо попадались мне в руки. Каждый разработчик должен иметь ее и перечитывать каждый год. Я ежегодно перечитываю ее на протяжении вот уже девяти лет, и все еще узнаю много нового!
- Steve McConnell. Rapid Development. — Microsoft Press, 1996
Эта книга научила меня управлять группами и планировать проекты.
- Steve Maguire. Debugging the Development Process. — Microsoft Press, 1994
Отличное введение в методику разработки ПО, используемую в Microsoft. Microsoft — самая успешная компания по разработке ПО в мире, значит, они уж точно в этом что-то понимают. Из этой книги можно многому научиться.
- Jim McCarthy. Dynamics of Software Development. — Microsoft Press, 1995
В этой книге описаны интересные взгляды на разработку ПО с точки зрения менеджера, имеющего богатый опыт создания прекрасных программ. Джим приводит ряд полезных правил, подкрепленных реальным опытом, а не только академическими рассуждениями.
- Ed Sullivan. Under Pressure and On Time. — Microsoft Press, 2001¹
Эд — очень успешный руководитель проектов, у которого я позаимствовал большинство идей по поводу создания ПО. Эд рассматривает управление проектами в реальном мире и рассказывает, как ему удастся создавать слаженные команды, которые разрабатывают неизменно великолепные продукты в установленные сроки. Хочу также отметить, что предисловие к этой книге написал один из самых сексуальных разработчиков² современности.
- Christopher Duncan. The Career Programmer: Guerilla Tactics for an Imperfect World. — APress, 2002
Во всех книгах подобного рода разработка ПО рассматривается только с точки зрения руководителя. Наконец хоть кто-то позаботился о рядовых программистах! Крис описывает тактику и методы, позволяющие довести работу до успешного завершения, даже когда ваш начальник болван.
- Michael Howard and David LeBlanc. Writing Secure Code, Second Edition. — Microsoft Press, 2002³
Нет ни одного человека, который не придавал бы огромного значения собственной безопасности, поэтому все разработчики должны иметь эту книгу в своей коллекции. В ней приводится не только фантастическое введение в сам предмет, но и огромное число реальных рекомендаций и примеров того, что нужно делать при создании программ.
- Steve Krug. Don't Make Me Think: A Common Sense Approach to Web Usability. — New Riders, 2000
Все считают себя экспертами по пользовательскому интерфейсу; особенно это заметно при перемещении вверх по служебной лестнице. В этой великолеп-

¹ Салливан Э. Время — деньги. — М.: Русская Редакция, 2002. — *Прим. перев.*

² Предисловие написано самим Джоном Роббинсом. — *Прим. перев.*

³ Ховард М., Лебланк Д. Защищенный код. — М.: Русская Редакция, 2003. — *Прим. перев.*

ной книге обсуждается разработка реального web-сайта и описываются методы, позволяющие гарантировать, что созданный вами интерфейс не вызовет недоумения.

- Philip Greenspun. *Philip and Alex's Guide to Web Publishing*. — Morgan Kaufmann, 1999

Если вы проектируете web-сайт, то просто обязаны прочитать эту книгу. Вероятно, не все в ней вызовет у вас согласие (в моем случае это определенно так), но в ней приводится одно из наиболее основательных обсуждений того, как оставить у пользователей благоприятное впечатление и вызвать у них желание посетить ваш сайт еще раз. Кроме того, в книге прекрасные фотографии. Мой отец, художник с мировым именем, никак не может понять, почему мои книги не выглядят так же хорошо.

- Chris Loosley and Frank Douglas. *High-Performance Client/Server*. — Wiley, 1998
Отличная книга по разработке высокопроизводительных приложений. Особо стоит отметить великолепное обсуждение архитектур высокоскоростных систем.

- Dr. International. *Developing International Software, Second Edition*. — Microsoft Press, 2003

Слышали старую шутку? Человека, знающего три языка, называют трехязычным, два — двуязычным, а если кто-то знает только один язык, то это американец. От продаж своих программ вне США Microsoft получает 60% дохода, и вы тоже можете достигнуть этого, если выполните необходимые действия. В данной книге описана разработка интернациональных приложений с учетом всех технологий Microsoft.

Отладка и тестирование

- Brian W. Kernighan and Rob Pike. *The Practice of Programming*. — Addison-Wesley, 1999

Прекрасное обсуждение разработки, отладки и тестирования программ.

- Steve Maguire. *Writing Solid Code*. — Microsoft Press, 1993)

Эта книга посвящена преимущественно программированию на C, но в ней можно найти полезные советы по определению интерфейсов и избеганию ряда неприятных проблем языка C.

- Rex Black. *Managing the Testing Process*. — Microsoft Press, 1999

Чтобы эффективней разрабатывать программы, нужно уметь правильно их тестировать. Прочитав эту великолепную книгу, вы измените взгляды на разработку программ и взаимодействие с группами контроля качества.

- Jonathan B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. — John Wiley & Sons, 1996

Отличное введение в работу отладчиков и некоторые вопросы, которые нужно рассматривать при создании отладчика.

- Everett N. McKay and Mike Woodring. *Debugging Windows Programs: Strategies, Tools, and Techniques for Visual C++ Programmers*. — Addison Wesley, 2000

Сразу же после первого издания книги, которую вы держите в руках, вышел труд Эверетта и Майка. Фантастическая книга! Хотел бы я, чтобы некоторые

из содержащихся в ней идей, например, относительно размышлений об отладке, принадлежали мне.

Технологии .NET

- Jeffrey Richter. Applied Microsoft .NET Framework Programming. — Microsoft Press, 2002⁴
- Jeffrey Richter and Francesco Balena. Applied Microsoft .NET Framework Programming in Microsoft Visual Basic .NET. — Microsoft Press, 2002
Просто лучшее введение во внутренние механизмы .NET. В этих книгах вы узнаете все то, что обещают, но, к сожалению, не могут объяснить авторы тысяч книг типа «Введение в .NET».
- Jeff Prosise. Programming Microsoft .NET. — Microsoft Press, 2002⁵
После книги Джеффри Рихтера было бы просто прекрасно прочитать и этот труд. Каким-то образом Джеффу удалось угадать все вопросы, которые у меня когда-либо возникали по поводу Microsoft ASP.NET, и ответить на них в своей великолепной книге.
- Adam Nathan. .NET and COM: The Complete Interoperability Guide. — Sams, 2002
Если ваше приложение .NET должно выходить за пределы CLR и взаимодействовать с суровым большим миром приложений Windows или COM, в этой книге вы найдете все, что нужно для правильной реализации такого взаимодействия.
- Dino Esposito. Applied XML Programming for Microsoft .NET. — Microsoft Press, 2002
XML сегодня повсюду, и книга Дино — отличное руководство по использованию XML в Microsoft .NET.
- Jimmy Nilsson. .NET Enterprise Design with Visual Basic .NET and SQL Server 2000. — Sams, 2002
Книга Джимми относится к тем немногим изданиям, которые посвящены вопросам разработки реальных программ. Читать ее и узнавать о средствах решения проблем — одно удовольствие.
- Microsoft Application Consulting and Engineering (ACE) Team. Performance Testing Microsoft .NET Web Applications. — Microsoft Press, 2003
Выполнение многих задач программирования платформа .NET берет на себя, но за быстроедействие программы отвечаете все же вы. Эта книга — отличное введение в тестирование производительности.
- Serge Lidin. Inside Microsoft .NET IL Assembler. — Microsoft Press, 2002
Вероятно, вам никогда не придется программировать на IL на профессиональном уровне, однако осознавать, что вы это умеете, очень приятно.

⁴ Рихтер Дж. Программирование на платформе Microsoft .NET Framework. — М.: Русская Редакция, 2002. — *Прим. перев.*

⁵ Просиз Дж. Программирование для Microsoft .NET. — М.: Русская Редакция, 2003. — *Прим. перев.*

Языки C/C++

- Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language, Second Edition. — Prentice Hall, 1988
Классическое руководство по программированию на C. Кроме того, я считаю эту книгу образцом того, как надо писать книги по программированию.
- Neill Graham. Learning C++. — McGraw-Hill, 1991
По этой книге я изучал C++. В отличие от других книг по C++ для начинающих «Learning C++» лаконична, понятна и попадает точно в цель.
- Mark Nelson. C++ Programmer's Guide to the Standard Template Library. — IDG Books, 1995
Standard Template Library вызывает у меня очень противоречивые чувства. К счастью, в своей книге Марк объясняет и иллюстрирует на примерах лучшие способы использования этой библиотеки.

ОС Windows и технологии Windows

- Charles Petzold. Programming Windows, Fifth Edition. — Microsoft Press, 1999
В одной этой книге вы найдете все, что нужно знать о самых фундаментальных аспектах работы приложений Windows. Люди, которые обращаются ко мне за помощью, могли бы решить много проблем самостоятельно, если бы лучше понимали, как обработка сообщений, интерфейс графических устройств (GDI) и другие компоненты работают на уровне Microsoft Win32.
- Jeffrey Richter. Programming Applications for Microsoft Windows, Fourth Edition. — Microsoft Press, 1999
В данной книге рассматриваются все уникальные аспекты разработки программ Win32. Главы, посвященные DLL, потокам, синхронизации и структурной обработке исключений, — самые лучшие из известных мне описаний соответствующих вопросов; вся эта информация непосредственно связана с быстротой и эффективностью отладки.
- Jeff Prosise. Programming Windows with MFC, Second Edition. — Microsoft Press, 1999
Это подробный курс программирования с использованием библиотеки Microsoft Foundation Class (MFC). Если вы работаете с MFC, то обязательно должны иметь эту книгу.
- Paul Dilascia. Windows++: Writing Reusable Windows Code in C++. — Addison-Wesley, 1992
В отличие от других книг по программированию, основанных на детских примерах, здесь разрабатывается нетривиальная и вполне жизнеспособная библиотека классов C++ для Windows. Я сам эффективнее всего учусь на примерах, и эта книга научила меня думать в терминах объектов.
- Don Box. Essential COM. — Addison-Wesley, 1998
В окне Disassembly («недвусмысленный режим») интерфейс COM выглядит как указатель на массив указателей, поэтому модель COM кажется довольно простой. Однако на самом деле COM настолько сложнее, чем обычный указатель, и в то же время настолько фундаментальна, что, если вы рассчитываете пра-

вильно разрабатывать и отлаживать свой код, вам обязательно нужно понять ее работу. Книга Дона — прекрасный старт на пути к COM-просветлению.

- David A. Solomon and Mark E. Russinovich. Inside Microsoft Windows 2000, Third Edition. — Microsoft Press, 2000⁶.

Эта книга является официальной «общей картиной» ядра Microsoft Windows 2000. Она адресована скорее разработчикам драйверов устройств, однако описание совместной работы различных компонентов Windows 2000 будет полезным любому программисту.

- Matt Pietrek. Windows 95 System Programming Secrets. — IDG Books, 1995
Книга Мэтта уже давно продана, но вам по-настоящему следует попытаться найти ее. В Microsoft Windows 95/98 отладка чего-либо более сложного, чем простое нарушение доступа, часто связана с очень неприятными впечатлениями; в первую очередь это объясняется тем, что Windows 95 и Windows 98 — гибридные 16/32-разрядные ОС. Значительная часть материала этой книги, например, глава, посвященная формату файлов Portable Executable (PE), в равной степени применима и к Windows 2000.
- Brent Rector and Chris Sells. ATL Internals. — Addison-Wesley, 1999
Библиотека Active Template Library (ATL) позволяет создавать самые компактные и быстрые COM-объекты. Данная книга поможет вам задействовать все преимущества ATL.
- Keith Brown. Programming Windows Security. — Addison-Wesley, 2000
Просто лучшая книга по программированию защищенных программ для Windows.
- Jeffrey Richter and Jason D. Clark. Programming Server-Side Application for Microsoft Windows 2000. — Microsoft Press, 1999
Если вы работаете над неуправляемыми серверными приложениями, обязательно купите эту книгу. В разделе, посвященном масштабируемому вводу/выводу, приводится самое лучшее описание этого вопроса.

Процессоры Intel и аппаратные средства ПК

- Andrew S. Tanenbaum. Structured Computer Organization, Fourth Edition. — Prentice-Hall, 1998
Великолепное введение в архитектуру компьютеров. При отладке проблем я очень часто обращаюсь к этой книге. В ней есть ряд опечаток и технических ошибок, но если вас это не смущает, вы будете рады, что прочитали ее.
- Документация по процессорам Intel
Intel предоставляет бесплатный доступ к документации по своим процессорам. Если вы всерьез занимаетесь отладкой, то найдете приведенные в ней сведения очень полезными. Загрузить руководства в формате PDF можно с сайта Intel по адресу developer.intel.com/design/litcentr/index.htm. Кроме того, вы можете заказать у Intel эти же руководства в бумажной форме.

⁶ Соломон Д., Русинович М. Внутреннее устройство Microsoft Windows 2000. — М.: Русская Редакция, 2001. — *Прим. перев.*

- Hans-Peter Messmer. The Indispensable PC Hardware Book, Fourth Edition. — Addison-Wesley, 2001
Самое лучшее описание аппаратных средств ПК. Если вам нужно работать с аппаратурой на низком уровне, эта книга просто не имеет цены.

Программные средства

- Visual Assist компании Whole Tomato (www.wholetomato.com)
Отличная надстройка для Microsoft Visual Studio .NET, которая расширяет функции редактора и включает настоящую технологию IntelliSense и другие усовершенствованные возможности редактирования.
- Source Insight компании Source Dynamics (www.sourceinsight.com)
Если вам нужно видеть совместную работу компонентов крупных программ C++, C# или Java, вы можете использовать это средство навигации по исходному коду.
- DevPartner компании Compuware (www.compuware.com/products/numega)
Этот пакет включает в себя утилиты BoundsChecker (обнаружение ошибок), TrueTime (профилирование), TrueCoverage (определение покрытия кода тестами), CodeReview (статический анализ) и Distributed Analyzer (кросс-машинный анализ). Все утилиты поддерживают работу как с неуправляемым кодом, так и с программами .NET.
- C-Cover компании Bullseye Testing Technology (www.bullseye.com)
Фантастическое средство для определения покрытия тестами неуправляемого кода C++.
- 4NT компании JPSoft (www.jpsoft.com)
Прекрасная командная оболочка для ОС Windows. В ней даже есть отладчик командных файлов! При установке новой ОС я всегда устанавливаю эту программу в первую очередь.
- Anakrino Джея Фримена (Jay Freeman) (www.saurik.com/net/exemplar)
Самый простой способ лучше понять .NET — декомпилировать ее!
- Reflector Лутца Родера (Lutz Roeder) (www.aisto.com/roeder/DotNet)
Улучшенный вариант ILDASM.
- Demeanor for .NET компании WiseOwl (www.wiseowl.com)
Великолепный обфускатор кода, написанный Брентом Ректором (Brent Rector).
- SOAPscope компании MindReef (www.mindreef.com)
Если вы хотите знать все, что происходит с вашими web-сервисами, нет ничего лучше, чем SOAPscope.
- VMWare (www.vmware.com)
Вместо того чтобы устанавливать в тестовом отделе 300 компьютеров, купите несколько мощных серверов и запустите на них VMWare. Я знаю несколько организаций, которые благодаря VMWare сэкономили огромные деньги на покупке оборудования и при этом значительно повысили покрытие кода своих программ тестами.

Web-сайты

■ MSDN Online (msdn.microsoft.com)

Поиск любой информации о .NET и Windows следует начинать с MSDN. Доступ к MSDN можно получить как по указанному мной адресу, так и через подерживаемый Microsoft журнал «MSDN Magazine». Помните, что MSDN — частично коммерческая организация, поэтому предоставляемая ею информация иногда носит несколько рекламный характер.

■ ASP.NET Web (www.asp.net)

Официальный сайт группы, отвечающей в Microsoft за ASP.NET. Никак не могу понять, почему Microsoft не накрыла этот сайт зонтиком MSDN.

■ Раздел Microsoft в поисковой системе Google (www.google.com/microsoft.html)

Хотя для изучения технологий Microsoft у нас есть сайт MSDN, его возможности поиска оставляют желать лучшего. К нашей радости, в системе Google есть специальный сайт, предназначенный для поиска информации, связанной только с технологиями Microsoft. Если вы хотите использовать хваленую систему Google для поиска информации только на сайте MSDN, введите в строке поиска выражение `site:msdn.microsoft.com`.

■ Sysinternals (www.sysinternals.com)

Марк Руссинович (Mark Russinovich) и Брайс Когсвелл (Bryce Cogswell) разработали некоторые из самых лучших утилит для отладки неуправляемого кода: Regmon, Filemon, DebugView, Process Explorer и многие другие. Ко многим утилитам прилагается полный исходный код, и все они бесплатны! Чтобы не отстать от остального мира, я посещаю сайт Марка и Брайса минимум раз в неделю.

■ SmidgeonSoft Расса Остерлунда (Russ Osterlund) (www.smidgeonsoft.com)

Расс собрал коллекцию фантастических бесплатных утилит, успешно конкурирующих с программами Sysinternals. На его сайте вы сможете найти отладчик PEBrowse Professional Interactive, который поддерживает не только управляемый код, но и код .NET. Если вы хотите увидеть внутренности ОС, Расс поможет вам в этом.

■ Code Project (www.codeproject.com)

Прекрасный сайт сообщества разработчиков неуправляемых программ и программ .NET.

■ CodeGuru (www.codeguru.com)

Это прародитель всех сайтов, посвященных программированию для Windows. Теперь он содержит информацию и о .NET!

■ VB2TheMax (www.vb2themax.com)

Информационный ресурс для программистов, работающих с Microsoft Visual Basic .NET.

■ Wintellect (www.wintellect.com)

Сайт компании Wintellect, на форумах которого мы отвечаем на разные вопросы (а иногда и задаем их!).

Предметный указатель

A

ACT (Application Compatibility Toolkit) 643
ActiveX 44
API 105, 516
ASP (Active Server Pages) 195
assembly dependency walker *см.* средство
обхода зависимостей сборок
assertion *см.* утверждение
assertion notifications *см.* утверждение,
уведомление
assertion suppression *см.* утверждение,
подавление
ATL (Active Template Library) 518

B

bit blitting *см.* битовый перенос
BoundsChecker 144
build *см.* компоновка

C

checkpoint *см.* контрольная точка
CLI (Common Language Infrastructure) 414
CLR (common language runtime) 144, 234, 420,
438
COFF (Common Object File Format) 183
CrashFinder 446, 447, 454, 455, 456, 457
CrashHandler 477
CRT 618
CRT library 609

D

data breakpoint *см.* точка прерывания,
по данным
DCRT 611, 613, 617, 618
DCRT library 609
DeadlockDetection 540–545, 553, 555, 567
debug thunk *см.* отладочный шлюз
debuggee *см.* отлаживаемая программа
debugger *см.* отладчик
defensive programming *см.* защитное
программирование
DIA (Debug Interface Access) *см.* интерфейс,
доступа к отладочным данным
dialog box folding *см.* свертывание
диалоговых окон

E

error detection tools *см.* инструменты
обнаружения ошибок
event mask *см.* маска событий
ExceptionMon 424, 425
expression *см.* выражение
extension commands *см.* команды,
расширения

F

FastTrace 655, 657, 659
FCL (.NET Framework class library) 430, 431, 436
final exception handler *см.* конечный
обработчик исключений
finalizer thread *см.* поток финализации
FlowTrace 437, 439, 441
FPO (Frame pointer omission) 56

G

GAC 91
GDI 635
GUI 142

H

handle *см.* описатель

I

IA32 18
IAT (import address table) 547
IDE 197
IIS 420, 515, 522, 528
ILDASM (Microsoft Intermediate Language
Disassembler) 215, 228, 229, 242

J

JIT compiler (just-in-time compiler)
см. компилятор, по требованию
JIT-компиляция 414

K

kernel handle *см.* описатель ядра

L

linear address *см.* линейный адрес
logging code *см.* код, регистрирующий

M

mail slot *см.* почтовый ящик
MAP-файл 446, 447
— адрес ошибки 450
— содержание 447
— создание 446, 452
— чтение 446
MASM (Microsoft Macro Assembler) 274
MDI (multiple-document interface) 456
memory-mapped files *см.* проецируемые
в память файлы
meta commands *см.* команды, мета
MFC (Microsoft Foundation Class) 104, 174, 457,
469, 476, 528
Microsoft Visual SourceSafe 30
minidump with handles *см.* минидамп,
с описателями
minidump with heap *см.* минидамп, с кучей
mixed mode debugging *см.* отладка,
в смешанном режиме
MKS Source Integrity 30
MSAA (Microsoft Active Accessibility) 581
MSIL (Microsoft Intermediate Language) 18, 228,
235
MSSCCI (Microsoft Source Code Control
Interface) 33
multithreaded corruption *см.* многопоточная
среда

N

name sampling profiler *см.* профилирование
с выборкой имен
named pipe *см.* именованный канал
native application *см.* приложение,
неуправляемое
NTSD (NT Symbolic Debugger) 144

- O**
 object lifetime tracking code *см.* код, слежения за временем жизни объекта
 OCX 44
- P**
 pass-through function *см.* функция, ловушка, сквозная
 PDB (Program Database) 447
 ProfilerLib 422, 423
 Profiling API 414, 421, 433, 437
 property get accessor *см.* аксессор чтения свойства
 public key tokens *см.* маркеры открытых ключей
 PVCS Version Manager 30
- R**
 reader interface *см.* интерфейс, чтения
 regular commands *см.* команды, стандартные
 reverse engineering *см.* восстановление алгоритма
- S**
 sampling *см.* выборка
 SCM (Service Control Manager) 516
 SEH (structured exception handling)
 см. обработка исключений, структурная
 semaphore handle *см.* описатель семафора
 stack chain *см.* стековая цепочка
 STL (Standard Template Library) 41, 43, 342, 439, 452
 stub function *см.* функция, заглушка
 suspend count *см.* счетчик, приостановок
 SWS (Smooth Working Set) 661, 664, 666, 668, 671
- T**
 TEB (Thread Environment Block) 337
 Tester 571, 576, 580, 583, 607
 TIB (Thread Information Block) 337
 token *см.* маркер
 top-level symbol *см.* символ, верхнего уровня
 Trap Flag *см.* флаг ловушки
- U**
 unit test *см.* тест, блочный
- V**
 version label *см.* метка версии
- W**
 WDBG 142, 173, 174, 176, 193
 WinDBG (Windows Debugger) *см.* отладчик, режима ядра, WinDBG
 Windows Forms 83, 131, 420
 writer interface *см.* интерфейс, записи
 WSH (Windows Scripting Host) 571
- X**
 XML 3
-
- A**
 аксессор чтения свойства 216
 анализ стека 183, 190
- Б**
 база данных программы *см.* PDB
 библиотека
 — классов .NET Framework *см.* FCL
 — стандартных шаблонов *см.* STL
 битовый перенос 582
 блок
 — информации о потоке *см.* TIB
 — переменных окружения потока *см.* TEB
- В**
 восстановление алгоритма 17
 выборка 414
 выражение 348
- Г**
 глобальный кэш сборок *см.* GAC
 графический интерфейс пользователя *см.* GUI
- Д**
 дизассемблер промежуточного языка
 Microsoft *см.* ILDASM
 диспетчер управления службами *см.* SCM
- Ж**
 журнал трассировки 658
- З**
 защитное программирование 72
- И**
 идентификатор функции 418
 именованный канал 517
 инструменты обнаружения ошибок 413
 интегрированная среда разработки *см.* IDE
 интерфейс
 — ICorDebug 426
 — ICorDebugChainEnum 430
 — ICorDebugGenericValue 428
 — ICorDebugHeapValue 428
 — ICorDebugObjectValue 428
 — ICorDebugReferenceValue 428
 — ICorDebugThread 426
 — ICorDebugValue 428
 — ICorDebugValue 426
 — ICorProfilerCallback 439
 — ICorProfilerCallback 415, 422, 423, 424, 439
 — ICorProfilerInfo 415, 418, 425
 — IDiaSymbol 185
 — IUnknown 415, 426
 — доступа к отладочным данным 184
 — записи 418
 — контроля над исходным кодом
 Microsoft *см.* MSSCCI
 — метаданных 418
 — отладочный 426
 — чтения 418
 исключение 350, 413, 430, 431
 — адрес 507
 — асинхронное 474
 — развертывание 466
 — фильтр 467, 500
- К**
 кадр стека с отсутствующим указателем
 см. FPO
 класс
 — ASPTraceListener 93, 95
 — AssertHttpApplication 93
 — AutoMatic 634
 — BugslayerEventLogTraceListener 86
 — BugslayerStackTrace 87, 138
 — BugslayerTextWriterTraceListener 86
 — CBaseProfilerCallback 423
 — CBinaryImage 462
 — CCrashFinderDoc 457
 — CException 469

- CFile 469
- COject 109, 617
- CSymbolEngine 457
- CUseCriticalSection 531, 532
- Debug 92
- Debugger 95
- EventLogTraceListener 86
- ExceptApp.Days 366
- Global 93
- MyThreadClass 199
- StackFrame 86
- StackTrace 86
- String 366
- StringBuilder 223
- System.Threading.Thread 220, 222
- System.Web.UI.Page 133
- TextWriterTraceListener 86, 91
- TNotify 581
- TraceListener 86, 89
- TraceSwitch 132

ключ

- /? 69
- /ADV 230
- /BASE 47
- /BASEADDRESS 48
- /c 69
- /C 50
- /checked+ 49
- /DEBUG 36, 37, 55, 232
- /DOC 40
- /EHa 475
- /EHs 475
- /EP 50
- /f File 68
- /Gh 434, 666
- /GH 434
- /GL 53, 55
- /GS 52, 653
- /GX 475
- /HEADERS 184
- /i 69
- /INCREMENTAL:NO 36
- /LTCG 53, 55
- /MAP 54, 446
- /MAPINFO:EXPORTS 54, 446
- /MAPINFO:LINEs 54, 446
- /noconfig 49
- /NODEFAULTLIB 54
- /o 69
- /O1 52
- /OPT:ICF 37
- /OPT:NOIN98 54
- /OPT:REF 37
- /ORDER 55, 663, 671
- /OUT 232
- /P 50
- /PDB 36, 37
- /PDBSTRIPPED 56
- /r 68
- /REFERENCE 50
- /RELEASE 55, 332
- /RTC 52
- /RTC1 648
- /RTCc 648
- /RTCu 648
- /s Store 68
- /showIncludes 53
- /SOURCE 232
- /t 69
- /t Product 69

- /v 69
- /v Version 69
- /VERBOSE:LIB 55
- /W3 41
- /W4 41, 44, 648
- /WARN 39
- /WARNASERROR+ 40
- /Wp64 51
- /WX 41, 44, 648
- /X 51
- /Zi 35
- /Zp 51
- add 68
- del 68
- /RTCs 647

код

- обзор 536
- прекомпилированный 425
- регистрирующий 519
- слежения за временем жизни объекта 474
- стартовый 523

команда

- записи метки 30
- мета 330, 352
- — запись файлов дампа 330
- — присоединение к процессам 330
- — создание файлов регистрации 330
- расширения 330, 354, 358
- — анализ аварийного завершения 330
- — анализ критических секций 330
- — вывод описателей 330
- стандартные 330
- — исполнение по шагам 330
- — просмотр памяти 330
- — проход по шагам 341
- — трассировка 330
- точка 330

комментирование кода 136

компилятор 39

- по требованию 34, 434

компоновка 30, 34

- заключительная 34
- отладочная 61

компоновщик 38

конечный обработчик исключений 270

контрольная точка 30

критическая секция 532

Л

линейный адрес 337

М

маркер 425

- открытых ключей 91

маска событий 436

метка 30

метка версии 30

метод

- Add 89
- AddNotification 576
- ASPTTraceListener.HandleOutput 94, 95
- ASPTTraceListener.IsRequestFromLocalMachine 95
- Assert 84
- AssertValid 109
- BeginInProcDebugging 426
- CheckNotification 575, 582, 583
- CheckVirtualResolution 578
- ClassDumper 627
- Debugger.Launch 95

- EndInProcDebugging 426
 - EnumLocalVariables 188
 - ExceptionThrown 426
 - FindTopTWindowByTitle 574
 - FuncEnter 435
 - GetClass 428
 - GetCurrentException 426
 - GetEventMask 416
 - GetInprocInspectionInterface 426
 - GetInprocInspectionIThisThread 426
 - GetModuleInfo 415
 - HandleOutput 94
 - ICorProfilerCallback 417
 - Init 92
 - Initialize 415, 416, 437
 - InitializeComponent 224
 - Launch 95
 - MyDataCheck () 211
 - PlayInput 574, 577, 604
 - Remove 89
 - SetEnterLeaveFunctionHooks 434
 - SetEventMask 434, 436
 - SetFocusTWindow 581
 - SetForegroundTWindow 577
 - SetFunctionIDMapper 437
 - SetSpecificFocus 581
 - Shutdown 415, 417
 - StackWalk 188
 - SymEnumSymbols 187
 - ThreadFunc 199
 - ToString 86
 - Warn 133
 - Write 131, 133
 - Writelf 131
 - WriteLine 131
 - WriteLineIf 131
 - встраивание 436
 - минидампы 323, 327, 331, 360, 361, 445, 465, 502, 503, 504, 505, 506, 535
 - с кучей 273
 - с описателями 274
 - многодокументный интерфейс *см.* MDI
 - многопоточная среда 415
 - многопоточность 528
- О**
- обработка исключений
 - C++ 465, 468, 469, 470, 474
 - векторная 468
 - синхронная 475
 - структурная 465
 - обработчик
 - исключений 464
 - ошибок 475
 - общезыковая
 - инфраструктура *см.* CLI
 - исполняющая среда *см.* CLR
 - общий формат объектных файлов *см.* COFF
 - объект
 - CBinaryImage 462
 - CUseCriticalSection 530
 - Debug 84, 131
 - DefaultTraceListener 85, 89
 - Page 94
 - System.Diagnostics.Debugger 95
 - TInput 574
 - TNotify 574, 575, 576, 583
 - Trace 84, 131
 - TraceListener 84, 89, 90, 91, 92
 - TraceSwitch 132
 - TSystem 574, 581
 - TWindow 574, 577, 581
 - TWindows 581
 - описатель 45
 - семафора 532
 - ядра 110
 - отладка 18
 - JIT (Just-In-Time) 148
 - базового кода 518
 - в смешанном режиме 225
 - внутрипроцессная 425
 - интенсивная 23
 - на главном сервере 226
 - планирование 14
 - сервер символов 63, 68
 - службы 518, 519
 - событие 156
 - стартового кода 523
 - удаленная 226, 265
 - отладочный шлюз 547
 - отладчик 25, 143
 - Borland Delphi 144
 - C++ Builder 144
 - CORDBG.EXE 223
 - MiniDBG 154
 - NTSD (Microsoft NT Symbolic Debugger) 324
 - SOS (Son of Strike) 324, 362
 - WinDBG 323, 325, 326, 328, 331, 340
 - — управление 352
 - автоматический запуск 152
 - пользовательского режима 143
 - режима ядра 143, 146
 - — SoftICE 147
 - — WinDBG 146
 - — ядра KD 146
 - отлаживаемая программа 143, 148, 175
 - ошибка 2, 3
 - адрес 444, 446, 450, 451, 456
 - воспроизведение 19
 - обработка 6
 - обработчик 464
 - описание 20
 - поиск 444

П

- поток финализации 438
- почтовый ящик 517
- предупреждение 38
- приложение
 - неуправляемое 245
 - профилируемое 414
- проецируемые в память файлы 517
- пространство имен
 - System.Diagnostic 84
 - System.Diagnostics 92
- профилирование 413, 414
 - с выборкой имен 414
- процесс
 - дочерний 335
 - присоединение 338
 - просмотр 336
 - создание 337

Р

- расширение 353
 - загрузка 353
 - команды 354, 358
 - управление 353
- расширяемый язык разметки *см.* XML

С

- сборка 58
 - идентификатор 415
- свертывание диалоговых окон 109
- свойство
 - Debugger.IsAttached 95
 - EventSource 93
 - HttpContext.Current.Handler 94
 - LaunchDebuggerOnAssert 93
 - ShowDebugLog 93
 - ShowOutputDebugString 93
 - Writer 93
- символ
 - верхнего уровня 246
 - корректная загрузка 331
 - отладки 34, 35
 - перечисление 675
 - сервер 69, 70, 183, 452
- система
 - отладки 33
 - отслеживания ошибок 29, 31
 - управления версиями 26, 27, 30, 32
- служба 515
 - идентификационные данные 520
 - отладка 519
 - подключение 520
- событие 350
- спин-блокировка 532, 533
- средство
 - обхода зависимостей сборок 414
 - профилирования 420
 - тестирования 28
- стандартная библиотека шаблонов *см.* STL
- стековая цепочка 426
- счетчик
 - выполнения 205
 - приостановок 337

Т

- таблица
 - адресов импортируемых функций *см.* IAT
 - символов 183
- тест
 - блочный 28, 73, 137
 - дымовой 59
- тестовое приложение 28
- точка прерывания 178, 182, 347
 - код окна 198
 - модификатор 205, 216
 - общая 347
 - по данным 246, 252
 - по обращению к памяти 349
 - подсказка 197
 - расширенная 196
 - синтаксис 246
 - усложненная 216, 245
 - установка 347
- трассировка 130, 131, 132, 133, 134, 135, 341, 433, 655

У

- управляемый модуль 48
- условное выражение 207, 216, 250
- утверждение 74, 79, 102, 130, 610
 - игнорирование 114
 - подавление 109
 - тип 106
- уведомление 519

Ф

- файл
 - дампа 359
 - — краткий 359
 - — открытие 360
 - — отладка 361
 - — полный 359
 - — создание 359
 - порядок 669
- флаг ловушки 181
- функция
 - _AfxActivationWndProc 637
 - _beginthread 541
 - _beginthreadex 533, 541
 - _CorDllMain 48, 49
 - _CrtCheckMemory 617
 - _CrtIsMemoryBlock 617
 - _CrtIsValidHeapPointer 617
 - _CrtIsValidPointer 617
 - _CrtMemDifference 617
 - _CrtMemDumpStatistics 617
 - _CrtSetDbgFlag 634
 - _CrtSetReportMode 107, 649
 - _exitthread 541
 - _exitthreadex 533, 534, 541
 - _penter 671
 - _ReturnAddress 507, 555
 - _RTC_GetErrDesc 652
 - _RTC_Initialize 652
 - _RTC_NumErrors 652
 - _RTC_SetErrorFunc 652
 - _RTC_SetErrorType 652
 - _set_se_translator 474
 - _set_security_error_handler 654
 - AccessLocalsAndParamsExample 297
 - AddCrashHandlerLimitModule 500
 - AddVectoredExceptionHandler 468
 - AllocateProfilerCallback 423
 - AttachThreadInput 581
 - calloc 610
 - CheckMyMem 247
 - CloseHandle 541
 - CommonSnapCurrentProcessMiniDump 507
 - ContinueDebugEvent 158, 159, 181
 - CreateCurrentProcessCrashDump 506
 - CreateEventA 541
 - CreateEventW 541
 - CreateMutexA 541
 - CreateMutexW 541
 - CreateProcess 154, 258, 315
 - CreateRemoteThread 182, 183
 - CreateSemaphoreA 541
 - CreateSemaphoreW 541
 - CreateThread 533, 541
 - DBG_ReadProcessMemory 176
 - DeadDetExtClose 554
 - DeadDetExtOpen 554
 - DeadDetProcessEvent 554
 - DebugBreak 152, 182, 521, 644
 - DeleteCriticalSection 541
 - DllMain 468, 543
 - DoSomethingMultithreaded 530
 - EnterCriticalSection 533, 541, 545
 - EnumLocalVariables 188
 - ExitProcess 541, 567
 - ExitThread 533, 541
 - FindFirstFile 502
 - FindNextFile 502
 - FlushFastTraceFiles 658
 - FlushInstructionCache 158

- FreeLibrary 541
- FreeLibraryAndExitThread 541
- FunctionIDMapper 436
- GetDeadlockDetectionOptions 543
- GetExceptionCode 466, 467
- GetExceptionInformation 467
- GetFaultReason 502
- GetFirstStackTraceString 502, 503
- GetLastError 555
- GetLimitModuleCount 501
- GetLimitModulesArray 501
- GetNamedImportDescriptor 553
- GetNextStackTraceString 502, 503
- GetObjectType 102
- GetProcAddress 173, 505, 541, 546, 547, 567
- GetRealAddress 260
- GetRegisterString 502
- GetThreadContext 507
- GlobalLock 319
- HeapCreate 638
- HeapFree 638
- HookImportedFunctionsByName 547, 548, 553
- HookImportedFunctionsByNameA 548
- HookImportedFunctionsByNameW 548
- HookOrdinalExport 547
- InitializeCriticalSection 541
- InitializeCriticalSectionAndSpinCount 533, 541
- InitInstance 636
- InterlockedIncrement 300
- IsBadCodePtr 102
- IsBadReadPtr 102
- IsBadStringPtr 102
- IsBadWritePtr 102
- IsDebuggerPresent 144
- IsMiniDumpFunctionAvailable 506
- IsWindow 102
- keybd_event 581
- LeaveCriticalSection 541
- LoadLibrary 247, 543
- LoadLibraryA 541, 566
- LoadLibraryExA 541
- LoadLibraryExW 541
- LoadLibraryW 541, 566
- main 68
- malloc 610, 614
- MessageBox 637
- MiniDumpWriteDump 504, 505
- MsgWaitForMultipleObjects 541
- MsgWaitForMultipleObjectsEx 541
- new 610, 614
- NtWaitForSingleObject 531
- OnIdle 528
- OpenDeadlockDetection 543
- OpenEventA 541
- OpenEventW 541
- OpenMutexA 541
- OpenMutexW 541
- OpenSemaphoreA 541
- OpenSemaphoreW 541
- OutputDebugString 545, 546, 610
- PlayInput 581
- PopTheFancyAssertion 116
- PostMessage 541
- PrintDlg 319
- PulseEvent 541
- QueueUserWorkItem 528
- RaiseException 467, 471
- ReadDebuggeeMemoryEx 260
- ReadProcessMemory 158, 176
- RealSuperAssertion 116
- RegisterServiceCtrlHandlerEx 517
- ReleaseMutex 541
- ReleaseSemaphore 541
- RemoveCrashHandlerLimitModule 501
- ResetEvent 541
- ResumeDeadlockDetection 543
- ResumeThread 541
- SendInput 581, 607
- SendKeys 581
- SendMessage 541
- SetBreakpoint 178
- SetCrashHandlerFilter 500
- SetCriticalSectionSpinCount 533, 541
- SetDeadlockDetectionOptions 543
- SetEvent 541
- SetFastTraceOptions 657
- SetLastError 555
- SetServiceStatus 517
- SetSingleStep 181
- SetTimer 583
- SetUnhandledExceptionFilter 475, 476, 500
- SignalObjectAndWait 541
- SnapCurrentProcessMiniDump 506, 507, 535
- SnapFastTraceFiles 658
- StackWalk64 185, 190
- StartDebugging 80
- StartServiceCtrlDispatcher 516, 523
- SuperAssertion 115
- SuspendDeadlockDetection 543
- SuspendThread 541
- SymGetSymNext 452
- SymGetSymPrev 452
- SymInitialize 501
- SymSetOptions 457
- TerminateThread 541
- TryEnterCriticalSection 541, 545
- TuneModule 680
- ValidateAllBlocks 627, 629
- VirtualProtect 158
- VirtualProtectEx 176
- WaitForDebugEvent 105, 155, 158
- WaitForMultipleObjects 541
- WaitForMultipleObjectsEx 541
- WaitForSingleObject 531, 541
- WaitForSingleObjectEx 531, 541
- WaitMessage 542
- WriteProcessMemory 158
- wsprintf 501
- ZwWaitForSingleObject 531
- адрес 546
- добавление 15
- заглущка 137
- контрольная таблица 59
- ловушка 433, 434, 553, 555, 617
- — сквозная 555
- открытая 56, 447
- отладчика 18
- перехват 545, 567
- порядок 55
- преобразователь идентификаторов 436
- расползание 10
- шлюзовая 78

Х

- хранилище символов 62
- хронометраж кода 257

Джон Роббинс

Джон — один из основателей Wintellect (www.wintellect.com) — консалтинговой компании, занимающейся разработкой, отладкой и обучением. В Wintellect Джон руководит службой консалтинга и отладки, помогая отлаживать и настраивать приложения eBay, Microsoft, AutoDesk и многим другим корпоративным заказчикам. Он разъезжает по всему миру со своим учебным курсом «Отладка приложений .NET и Windows», и его слушатели учатся методикам, которые он использует для решения самых сложных проблем ПО. Один из наиболее признанных мировых авторитетов по отладке, он получает злорадное удовольствие, находя и исправляя совершенно невозможные ошибки в программах других авторов.

Джон живет в Нью-Гемпшире (США) с женой Пэм и самым лучшим в мире ловцом ошибок — кошкой Хлоей. Кроме того, что он написал эту книгу и «Debugging Applications» (Microsoft Press, 2000), Джон также постоянно ведет рубрику «Bugslayer» в журнале «MSDN Magazine». Он регулярно выступает на таких конференциях, как Tech-Ed, VSLive и DevWeek.

До основания Wintellect Джон был одним из первых сотрудников NuMega Technologies (ныне подразделение Compuware), где играл ключевую роль в проектировании, разработке, а также был менеджером таких проектов, как Bounds-Checker, TrueTime, TrueCoverage, SoftICE и TrueCoverage for Device Drivers.

Прежде чем вляпаться в разработку ПО, Джон, когда ему было около тридцати, служил десантником в армии США. Поскольку теперь у него нет возможности повышать адреналин, выпрыгивая из самолета среди ночи в темноту и неизвестность, ожидая кровопролитного боя, он довольствуется своим мотоциклом, гоняя на всей скорости — к великому неудовольствию своей жены.

ЛИЦЕНЗИОННОЕ СОГЛАШЕНИЕ MICROSOFT

(прилагаемый к книге компакт-диск)

ЭТО ВАЖНО — ПРОЧИТАЙТЕ ВНИМАТЕЛЬНО. Настоящее лицензионное соглашение (далее «Соглашение») является юридическим документом, оно заключается между Вами (физическим или юридическим лицом) и Microsoft Corporation (далее «корпорация Microsoft») на указанный выше продукт Microsoft, который включает программное обеспечение и может включать сопутствующие мультимедийные и печатные материалы, а также электронную документацию (далее «Программный Продукт»). Любой компонент, входящий в Программный Продукт, который сопровождается отдельным Соглашением, подпадает под действие именно того Соглашения, а не условий, изложенных ниже. Установка, копирование или иное использование данного Программного Продукта означает принятие Вами данного Соглашения. Если Вы не принимаете его условия, то не имеете права устанавливать, копировать или как-то иначе использовать этот Программный Продукт.

ЛИЦЕНЗИЯ НА ПРОГРАММНЫЙ ПРОДУКТ

Программный Продукт защищен законами Соединенных Штатов по авторскому праву и международными договорами по авторскому праву, а также другими законами и договорами по правам на интеллектуальную собственность.

1. ОБЪЕМ ЛИЦЕНЗИИ. Настоящее Соглашение дает Вам право:

а) **Программный продукт.** Вы можете установить и использовать одну копию Программного Продукта на одном компьютере. Основным пользователем компьютера, на котором установлен данный Программный Продукт, может сделать только для себя вторую копию и использовать ее на портативном компьютере.

б) **Хранение или использование в сети.** Вы можете также скопировать или установить экземпляр Программного Продукта на устройстве хранения, например на сетевом сервере, исключительно для установки или запуска данного Программного Продукта на других компьютерах в своей внутренней сети, но тогда Вы должны приобрести лицензии на каждый такой компьютер. Лицензию на данный Программный продукт нельзя использовать совместно или одновременно на других компьютерах.

с) **License Pak.** Если Вы купили эту лицензию в составе Microsoft License Pak, можете сделать ряд дополнительных копий программного обеспечения, входящего в данный Программный Продукт, и использовать каждую копию так, как было описано выше. Кроме того, Вы получите право сделать соответствующее число вторичных копий для портативного компьютера в целях, также оговоренных выше.

д) **Примеры кода.** Это относится исключительно к отдельным частям Программного Продукта, заявленным как примеры кода (далее «Примеры»), если таковые входят в состав Программного Продукта.

i) **Использование и модификация.** Microsoft дает Вам право использовать и модифицировать исходный код Примеров при условии соблюдения пункта (д)(iii) ниже. Вы не имеете права распространять в виде исходного кода ни Примеры, ни их модифицированную версию.

ii) **Распространяемые файлы.** При соблюдении пункта (д)(iii) Microsoft дает Вам право на свободное от отчислений копирование и распространение в виде объектного кода Примеров или их модифицированной версии, кроме тех частей (или их модифицированных версий), которые оговорены в файле Readme, относящемся к данному Программному Продукту, как не подлежащие распространению.

iii) **Требования к распространению файлов.** Вы можете распространять файлы, разрешенные к распространению, при условии, что: а) распространяете их в виде объектного кода только в сочетании со своим приложением и как его часть; б) не используете название, эмблему или товарные знаки Microsoft для продвижения своего приложения; в) включаете имеющуюся в Программном Продукте ссылку на авторские права в состав этикетки и заставки своего приложения; г) согласны освободить от ответственности и взять на себя защиту корпорации Microsoft от любых претензий или преследований по закону, включая судебные издержки, если таковые возникнут в результате использования или распространения Вашего приложения; и д) не допускаете дальнейшего распространения конечным пользователем своего приложения. По поводу отчислений и других условий лицензии применительно к иным видам использования или распространения распространяемых файлов обращайтесь в Microsoft.

2. ПРОЧИЕ ПРАВА И ОГРАНИЧЕНИЯ

- **Ограничения на реконструкцию, декомпиляцию и дизассемблирование.** Вы не имеете права реконструировать, декомпилировать или дизассемблировать данный Программный Продукт, кроме того случая, когда такая деятельность (только в той мере, которая необходима) явно разрешается соответствующим законом, несмотря на это ограничение.
- **Разделение компонентов.** Данный Программный Продукт лицензируется как единый продукт. Его компоненты нельзя отделять друг от друга для использования более чем на одном компьютере.
- **Аренда.** Данный Программный Продукт нельзя сдавать в прокат, передавать во временное пользование или уступать для использования в иных целях.
- **Услуги по технической поддержке.** Microsoft может (но не обязана) предоставить Вам услуги по технической поддержке данного Программного Продукта (далее «Услуги»). Предоставление Услуг регулируется соответствующими правилами и программами Microsoft, описанными в руководстве пользователя, электронной документации и/или других материалах, публикуемых Microsoft. Любой дополнительный программный код, предоставленный в рамках Услуг, следует считать частью данного Программного Продукта и подпадающим под действие настоящего Соглашения. Что касается технической информации, предоставляемой Вами корпорации Microsoft при использовании ее Услуг, то Microsoft может задействовать эту информацию в деловых целях, в том числе для технической поддержки продукта и разработки. Используя такую техническую информацию, Microsoft не будет ссылаться на Вас.
- **Передача прав на программное обеспечение.** Вы можете безвозвратно уступить все права, регулируемые настоящим Соглашением, при условии, что не оставите себе никаких копий, передадите все составные части данного Программного Продукта (включая компоненты, мультимедийные и печатные материалы, любые обновления, Соглашение и сертификат подлинности, если таковой имеется) и принимающая сторона согласится с условиями настоящего Соглашения.
- **Прекращение действия Соглашения.** Без ущерба для любых других прав Microsoft может прекратить действие настоящего Соглашения, если Вы нарушите его условия. В этом случае Вы должны будете уничтожить все копии данного Программного Продукта вместе со всеми его компонентами.

3. АВТОРСКОЕ ПРАВО.

Все авторские права и право собственности на Программный Продукт (в том числе любые изображения, фотографии, анимации, видео, аудио, музыку, текст, примеры кода, распространяемые файлы и апплеты, включенные в состав Программного Продукта) и любые его копии принадлежат корпорации Microsoft или ее поставщикам. Программный Продукт охраняется законодательством об авторских правах и положениями международных договоров. Таким образом, Вы должны обращаться с данным Программным Продуктом, как с любым другим материалом, охраняемым авторскими правами, **с тем исключением**, что Вы можете установить Программный Продукт на один компьютер при условии, что храните оригинал исключительно как резервную или архивную копию. Копирование печатных материалов, поставляемых вместе с Программным Продуктом, запрещается.

ОГРАНИЧЕНИЕ ГАРАНТИИ

ДАННЫЙ ПРОГРАММНЫЙ ПРОДУКТ (ВКЛЮЧАЯ ИНСТРУКЦИИ ПО ЕГО ИСПОЛЬЗОВАНИЮ) ПРЕДОСТАВЛЯЕТСЯ БЕЗ КАКОЙ-ЛИБО ГАРАНТИИ. КОРПОРАЦИЯ MICROSOFT СНИМАЕТ С СЕБЯ ЛЮБУЮ ВОЗМОЖНУЮ ОТВЕТСТВЕННОСТЬ, В ТОМ ЧИСЛЕ ОТВЕТСТВЕННОСТЬ ЗА КОММЕРЧЕСКУЮ ЦЕННОСТЬ ИЛИ СОТВЕТСТВИЕ ОПРЕДЕЛЕННЫМ ЦЕЛЯМ. ВСЕГДА РИСК ПО ИСПОЛЬЗОВАНИЮ ИЛИ РАБОТЕ С ПРОГРАММНЫМ ПРОДУКТОМ ЛОЖИТСЯ НА ВАС. НИ ПРИ КАКИХ ОБСТОЯТЕЛЬСТВАХ КОРПОРАЦИЯ MICROSOFT, ЕЕ РАЗРАБОТЧИКИ, А ТАКЖЕ ВСЕ, ЗАНЯТЫЕ В СОЗДАНИИ, ПРОИЗВОДСТВЕ И РАСПРОСТРАНЕНИИ ДАННОГО ПРОГРАММНОГО ПРОДУКТА, НЕ НЕСУТ ОТВЕТСТВЕННОСТИ ЗА КАКОЙ-ЛИБО УЩЕРБ (ВКЛЮЧАЯ ВСЕ, БЕЗ ИСКЛЮЧЕНИЯ, СЛУЧАИ УПУЩЕННОЙ ВЫГОДЫ, НАРУШЕНИЯ ХОЗЯЙСТВЕННОЙ ДЕЯТЕЛЬНОСТИ, ПОТЕРИ ИНФОРМАЦИИ ИЛИ ДРУГИХ УБЫТКОВ) ВСЛЕДСТВИЕ ИСПОЛЬЗОВАНИЯ ИЛИ НЕВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ДАННОГО ПРОГРАММНОГО ПРОДУКТА ИЛИ ДОКУМЕНТАЦИИ, ДАЖЕ ЕСЛИ КОРПОРАЦИЯ MICROSOFT БЫЛА ИЗВЕЩЕНА О ВОЗМОЖНОСТИ ТАКИХ ПОТЕРЬ, ТАК КАК В НЕКОТОРЫХ СТРАНАХ НЕ РАЗРЕШЕНО ИСКЛЮЧЕНИЕ ИЛИ ОГРАНИЧЕНИЕ ОТВЕТСТВЕННОСТИ ЗА НЕПРЕДНАМЕРЕННЫЙ УЩЕРБ, УКАЗАННОЕ ОГРАНИЧЕНИЕ МОЖЕТ ВАС НЕ КОСНУТЬСЯ.

РАЗНОЕ

Настоящее Соглашение регулируется законодательством штата Вашингтон (США), кроме случаев (и лишь в той мере, насколько это необходимо) исключительной юрисдикции того государства, на территории которого используется Программный Продукт.

Если у Вас возникли какие-либо вопросы, касающиеся настоящего Соглашения, или если Вы желаете связаться с Microsoft по любой другой причине, пожалуйста, обращайтесь в местное представительство Microsoft или пишите по адресу: Microsoft Sales Information Center, One Microsoft Way, Redmond, WA 98052-6399.