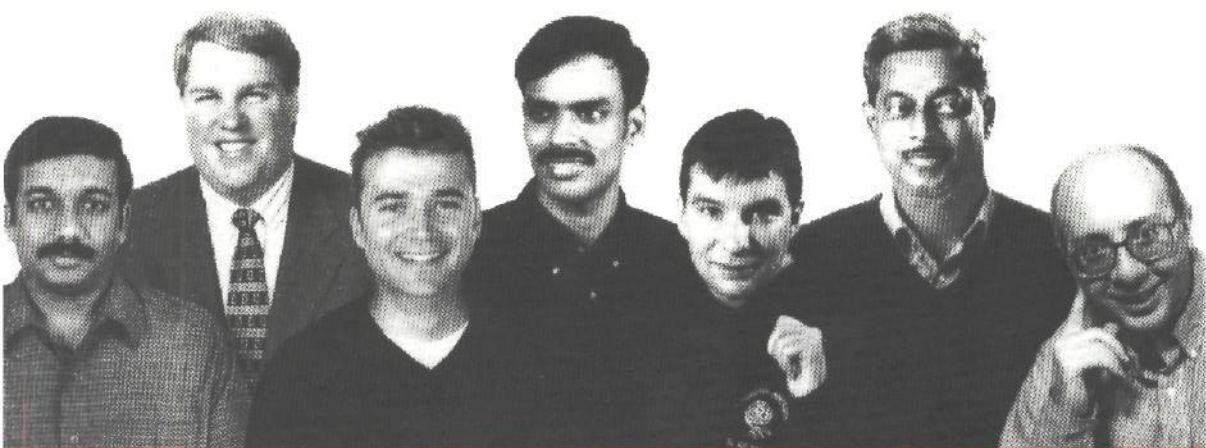




Visual C++ .NET

Пособие для разработчиков C++

Книга написана и протестирована для последнего выпуска

.NET v1.0

Аравинд Корера, Стивен Фрейзер, Скотт Маклин,
Ниранджан Кумар, Саймон Робинсон, д-р П.Г. Саранг,
Сэм Джентайл



PROGRAMMER TO PROGRAMMER™

Адрес группы технической поддержки издательства Wrox: support@wrox.com

Обновления и исходный код доступны на сайте: www.wrox.com

Поддержка разработчиков: p2p.wrox.com

Visual C++ .NET:

A primer for C++ developers

Aravind Corera
Stephen Fraser
Sam Gentile
Niranjan Kumar
Scott McLean
Simon Robinson
Dr P.G.Sarang

Wrox Press Ltd.

Visual C++ .NET:

Пособие для разработчиков C++

Аравинд Корера
Стивен Фрейзер
Сэм Джентайл
Ниранджан Кумар
Скотт Маклин
Саймон Робинсон
Д-р П.Г. Саранг

Издательство "Лори"

Aravind Corera, Stephen Fraser, Sam Gentile, Niranjana Kumar,
Scott McLean, Simon Robinson, Dr P.G.Sarang
Visual C++ .NET: A primer for C++ developers

© 2002 Wrox Press
All rights reserved

Published by Wrox Press Ltd, Arden House, 1102 Warwick Road, Acock 's Green,
Birmingham, B27 6BH, UK
Printed in the United States

ISBN 1-861005-96-2

Аравинд **Корера**, Стивен Фрейзер, Сэм **Джентайл**, Ниранджан Кумар,
Скотт **Маклин**, Саймон Робинсон, д-р П.Г. Саранг
Visual C++ .NET: Пособие для разработчиков C++

Переводчик В. Пучков
Научный редактор А. Вендров
Корректор Е. Пресс
Верстка Е. Самбу

© Издательство "ЛОРИ", 2003

Изд. N : **OAI(03)**
ЛР N :070612 30.09.97 г.
ISBN 5-85582-189-7

Подписано в печать 29.01.2003. Формат 70×100/16.
Печать офсетная
Печ.л. 26. Тираж 3200. Заказ N 71
Цена договорная

Издательство "Лори"
Москва, 123100, Шмитовский пр., д. 13/6, стр. 1 (пом. ТАРП ЦАО)
Телефон для оптовых покупателей: (095) 256-02-83
Размещение рекламы: (095) 259-01-62
www.lory-press.ru

Отпечатано в типографии ООО "Типография ИПО профсоюзов Профиздат"
109044, Москва, ул. Крутицкий вал д. 18

Об авторах

Аравинд Корера

Аравинд Корера — независимый программист. Он живет и работает в г. Ченнай, Индия. В настоящее время принимает участие в разработке проекта для компании, занимающейся автоматизацией документооборота. Аравинд Корера уже в течение 6 лет создает ПО на основе продуктов и технологий Microsoft. В частности, он использует в своей работе Microsoft COM. В сферу интересов Аравинда входят параллельное программирование, распределенные системы и мониторы обработки транзакций. Он имеет диплом инженера и специалиста по компьютерным технологиям института Coimbatore Institute of Technology. Ему можно послать сообщение по адресу: aravinda@netluminosity.com

Благодарю мою семью и друзей за неоценимую поддержку и за то, что они делили со мной все успехи и неудачи, пока я писал эту книгу.

Стивен Фрейзер

Стивен Фрейзер — главный менеджер компании Fraser Training, которая занимается обучением специалистов в области .NET-технологий. Более 15 лет Стивен проработал в сфере информационных технологий в различных консалтинговых компаниях, среди которых были как крупные фирмы (EDS и Anderson Consulting), так и небольшие компании, занимающиеся электронной коммерцией. Стивен имеет опыт разработки приложений, в том числе и для Web. Ему хорошо знакомы все этапы разработки и управления приложениями, начиная с исходной концепции и заканчивая внедрением.

В настоящее время Стивен живет со своей очаровательной женой Сарой и дочерью Шаиной в солнечном городе Tustin Ranch в Южной Калифорнии.

Сэм Джентайл

Сэм Джентайл — консультант в области .NET-технологий и главный инженер по разработке ПО в крупной производственной фирме, где используется большой программный комплекс, построенный на основе архитектуры COM. Он также занимается новыми разработками, среди которых создание вызываемых оболочек периода выполнения (runtime callable wrappers — RCW), обеспечение взаимодействия COM и традиционного кода C++, а также интеграция управляемого кода C++ и C# с Visual Studio. NET. Ранее Сэм был руководителем группы программистов и главным инженером по разработке ПО в компании NaviSite. Используя такие технологии .NET, как ASP.NET, Web-службы, ADO.NET, SOAP, C# и др., группа Сэма создала и внедрила два программных продукта на основе платформы .NET Beta 2. Сэм — участник программы Microsoft Early Adopter Program. В течение 17 лет он был разработчиком ПО и в основном использовал технологии Microsoft, что позволило ему стать специалистом по этим технологиям. Когда Сэм не занят манипуляциями с ПО, он любит читать научную фантастику. Можно посетить его Web-сайт (на котором представлен материал по .NET-технологиям) по адресу: www.project-inspiration.com/sgentile.

Кроме своего бесконечно терпеливого, замечательного редактора издательства Wrox Виктории Хадсон, Сэм хотел бы поблагодарить сотрудников корпорации Microsoft. Он признателен Саре Уильямс, которая способствовала привлечению его к участию в программе EAP. Сэм также выражает благодарность Марку Холлу из группы разработчиков компилятора C++ и Рональду Леремансу из группы разработки C++. Он благодарит Стэна Липпмана за его полезные послания по электронной почте и Адама Натана за то, что тот в

последний момент помог найти критическую ошибку, относящуюся к пространствам имен. Сэм особенно хотел бы поблагодарить за большую помощь Дона Бокса, Брента Ректора и Джона Лэма, которые предоставили поистине бесценную информацию о внутренних механизмах технологии *PInvoke*. Сэм считает, что без их помощи его текст многое бы потерял. Он уверен: на этих людей всегда можно положиться. Сэм признателен Томасу Рестрепо и Крису Селлсу не только за их рецензии, но и за то, что они помогли прояснять самые разные вопросы, а также за их дружеское отношение. Сэм благодарен своей прекрасной жене *Сьюзен* за ее любовь, поддержку и понимание и своему чудесному сыну *Джонатану* (у папы, теперь будет свободное время!). И наконец, но не в последнюю очередь, Сэм хотел бы поблагодарить своих родителей-тружеников, *Мафио* и *Катерину Джентайл*, которые эмигрировали из Италии. Они отправили Сэма учиться в университет, без чего сделать эту работу было бы совершенно невозможно.

Ниранджан Кумар

Ниранджан Кумар работает в компании Cognizant Technology Solutions (www.cognizant.com) в г. Ченнай, Индия. Он занимается информационными технологиями с середины 1997 г. Ниранджан Кумар начал свою карьеру как С-программист на платформе Unix, а затем перешел на языки C++ и Visual C++. Он является соавтором книги *Data-Centric .NET Programming with C#*. Ниранджан Кумар опубликовал большое количество статей на сайтах www.csharptoday.com и www.csharpcorner.com. В свободное от основной работы время Ниранджан занимается исследованиями технологий C# и .NET. Он любит плавать, нырять, играть на гитаре и разгадывать головоломки с картинками. Ниранджан с удовольствием путешествует по Европе. Его восхищает Италия и все итальянское. Ему можно направить сообщение по адресу: kniran-ja@chn.cognizant.com.

Прежде всего я благодарю сотрудников издательства Wrox, благодаря которым эта книга оказалась столь успешной. Я также признателен Джону Хиллу и всем рецензентам за их исправления и ценные указания.

Пользуюсь случаем, чтобы выразить благодарность всем моим близким друзьям в компаниях Sella Synergy India Ltd, Индия и Banca Sella, Италия (www.sella.it), где я научился основам программирования.

*Спасибо моей маме *Тирупурасундари* и моим друзьям за их поддержку на всех этапах этой работы.*

Скотт Маклин

Скотт Маклин начал заниматься программированием еще в школе на компьютере Atari 400 с 8-битовым микропроцессором 6502, оперативной памятью 16 Кб и тремя регистрами. Он самостоятельно изучил Atari Basic, а затем перешел на ассемблер 6502, когда родители выполнили его рождественское пожелание и подарили ему картридж для компьютера Atari, позволяющий писать программы на ассемблере. Это помогло Скотту глубже проникнуть в компьютерное железо. С тех пор прошло 20 лет.

За это время Скотт отслужил 6 лет в ВМФ США помощником машиниста на атомной подводной лодке. После выхода в отставку он получил в Университете Джорджии степень бакалавра в области компьютерных технологий и поступил на работу инженером-программистом в корпорацию Motorola, Inc (Чикаго). Скотт изучил технологии SS7, IS-41 и различные сетевые агенты, обеспечивающие работу сотовых телефонных сетей. Поскольку Скотт родом из Флориды, одной зимы в Чикаго ему оказалось вполне достаточно, чтобы решить перебраться в Атланту (Джорджия) с ее более теплым климатом. Здесь Скотт устроился на работу в корпорацию NCR, где он разрабатывал элементы управления ActiveX для периферийных

устройств пунктов продажи и терминалов, таких как сканеры штрих-кода, устройства чтения подписи, шелевые считыватели магнитных карт и др.

В настоящее время Скотт является разработчиком ПО в компании XcelleNet, Inc и занимается расширением коммуникационных функций сервера для продукта Afaria. В своей работе Скотт использует C++, COM, ATL, C# и практически все сопутствующие технологии. Он один из основателей сайта www.thinkdotnet.com — онлайн-ового ресурса, посвященного разработкам на платформе .NET. Читатель может просмотреть статьи Скотта на этом сайте или написать ему по адресу: scott@thinkdotnet.com.

Я благодарен моей замечательной жене Нэнси за то, что она неизменно поддерживает меня в моих поисках компьютерной нирваны.

Саймон Робинсон

Саймон Робинсон живет в Ланкастере (Великобритания), где снимает дом вместе с несколькими студентами. Он впервые занялся серьезным программированием во время работы над диссертацией по физике, которая была посвящена моделированию различных потусторонних сущностей, имеющих отношение к сверхпроводникам и квантовой механике. Опыт программирования Саймона оказался достаточным, чтобы на всю жизнь отвратить его от компьютеров (хотя, как ни странно, он выжил после всей этой квантовой механики), и он решил попробовать себя в качестве спортивного массажиста. Но затем он понял, насколько больше денег привлечено в компьютерные технологии по сравнению со спортивным массажем, и быстро устроился на работу программистом-исследователем C++. Саймон — щедрый, одухотворенный человек, который знает, в чем смысл жизни.

Карьера программиста в конечном итоге привела Саймона к написанию книг, и сейчас он в основном живет тем, что пишет замечательные книги для программистов. В процессе работы Саймон имел дело со многими аспектами программирования для Windows. Он регулярно пишет программы на C++, C# и VB. Саймон с энтузиазмом воспринял технологию .NET и твердо убежден, что она совершит революцию в программировании. В свободное время Саймон посещает уроки танцев (он любит исполнительское искусство и изучает его) или работает над своим любимым проектом: написанием компьютерной игры-стратегии.

Вы можете посетить Web-сайт Саймона на: www.simonrobinson.com.

Д-р П.Г. Саранг

Д-р Саранг имеет более чем 20-летний опыт работы в компьютерной сфере и занимается проектированием и разработкой решений на основе различных технологий. Он является генеральным директором компании ABCOM Information Systems Pvt, Ltd. Его деятельность связана с обучением специалистов и разработкой проектов с использованием Java, CORBA и платформ Microsoft. На протяжении своей многолетней карьеры д-р Саранг разработал ряд весьма успешных программных продуктов и осуществил несколько промышленных проектов. Он регулярно выступает с докладами на многих международных конференциях и постоянно пишет технические статьи в ведущие международные журналы. Д-р Саранг является соавтором нескольких книг издательства Wrox, посвященных Java, J2EE, электронной коммерции и технологии .NET.

Я особенно благодарен Чандану Парулкару и Рахулу Бируду за ценную помощь в написании кода.

Посвящаю написанные мною главы моей маме.

Содержание

Об авторах	v
<u>Введение</u>	<u>xiii</u>
Основные темы книги	xiv
Необходимые аппаратные средства	xiv
Соглашения, принятые в книге	xiv
Поддержка читателей и контакты с ними	xv
Исходный код и его обновления	xv
Опечатки и исправления	xvi
Техническая поддержка	xvi
Ресурс p2p.wrox.com	xvii
<u>Глава 1. Новые возможности Visual C++</u>	<u>1</u>
Платформа .NET Framework	2
Единая среда исполнения	2
Свойства управляемого кода	4
Библиотека классов .NET Framework	5
Новые особенности Visual Studio .NET	7
Новый вид "старых" мастеров	8
Новые мастера	14
Разработка и отладка приложений с использованием нескольких языков	16
Пример приложения на нескольких языках	16
Отладка	22
<u>Глава 2. Введение в управляемый код C++</u>	<u>24</u>
Интероперабельность языков	25
Метаданные	25

Система общих типов	26
Библиотека классов .NET Framework	26
Управляемая среда	30
Управляемый код и управляемые данные	31
Сборка мусора	32
Ссылочные типы и типы-значения	32
Создание управляемого кода	33
Использование управляемых расширений	36
Типы данных, удаляемых сборщиком мусора (<code>_gcotypes</code>)	37
Типы-значения (<code>_value types</code>)	52
Свойства	57
Делегаты	60
События	62
Исключительные ситуации	64
Ключевое слово <code>__identifier</code>	70
Сводка ключевых слов	70
 Глава 3. Сборки	 72
Что такое сборки	72
Структура сборки	73
Основные характеристики сборок	75
Создание сборок	76
Создание библиотеки классов	76
Доступность типов	78
Создание приложения	79
Просмотр сборки с помощью утилиты ILDasm	81
Декларация сборки	84
Файл <code>AssemblyInfo.cpp</code>	85
Закрытые и совместно используемые сборки	86
Создание совместно используемых сборок	87
Глобальный кэш сборок	91
Инсталляция совместно используемой сборки в GAC	93
Работа с совместно используемой сборкой	94
Замена ключа строгого имени на версию для распространения	94
Поддержка управления версиями	95
Работа с ресурсами	96
Создание файла ресурсов	96
Создание файлов ресурсов с помощью Visual Studio .NET	97
Локализация сборок	100
Доступ к файлам ресурсов	101
Динамические ресурсы	102
Развертывание сборок	104
 Глава 4. Атрибуты и отражение	 108
Атрибуты	109
Атрибуты C++	110
Атрибуты .NET	111
Написание пользовательских атрибутов	124
Отражение	127
Проект ListColors	127

Класс System::Type	131
Перечисление типов: пример с музыкальными инструментами	134
Комбинирование атрибутов и отражения	138
Глава 5. Классы-утилиты .NET Framework	142
Обработка текстовых данных	142
Класс StringBuilder	142
Класс Regex	145
Работа с файлами	148
Классы File и FileStream	148
Классы StreamReader и StreamWriter	150
Коллекции	151
Класс ArrayList	152
Класс SortedList	153
Интерфейс IComparer	155
Интерфейс Enumerator	159
Организация потоков	162
Классы управления потоками	163
Классы синхронизации потоков	168
Исключительные ситуации, генерируемые при работе потоков	183
Глава 6. Формы Windows	187
Формы Windows и библиотека MFC	187
Создание приложения Windows	
с помощью управляемого кода C++	188
Настройка форм и добавление событий	190
Добавление на форму подчиненных элементов управления	191
Работа с кнопочными элементами управления	192
Работа с текстовыми элементами управления	194
Элементы управления, предназначенные для выбора вариантов	198
Другие подчиненные элементы управления	202
Меню и формы с многодокументным интерфейсом	207
Элементы управления Windows Forms с расширенными возможностями	211
Разработка приложения Windows Explorer	212
Реализация операции drag-and-drop в формах Windows	220
Глава 7. Управляемый и неуправляемый код	225
Соединение управляемого и неуправляемого кода	226
Флаг /clr и механизм It Just Works (IJW)	227
Ключевое слово „pin	229
Использование управляемых типов в неуправляемом коде	230
Написание управляемых прокси-классов	233
Образец „прокси“	234
Основные этапы создания класса-оболочки	234
Неуправляемый связанный список	235
Построение класса-оболочки	237
Тестирование управляемого класса-оболочки	
с помощью клиента C#	242

Другие проблемы, возникающие при создании класса-оболочки	244
Маршалинг между управляемым и неуправляемым кодом	244
Процессы перехода между управляемым и неуправляемым кодом	245
Когда нужен маршалинг	246
Класс InteropServices::Marshal	247
Механизм Pinvoke: Вызов неуправляемых функций из управляемого кода	250
Вопросы производительности	258
Глава 8. Интероперабельность COM	260
Переход от COM к .NET	260
Необходимость взаимодействия	261
Использование COM-компонентов в управляемом коде C++	261
Вызов компонента COM из среды .NET	263
Передача в приложения .NET уведомлений о событиях компонента COM	272
Работа приложения .NET с классом коллекций COM	278
Применение элементов управления ActiveX в приложениях .NET	283
Применение в управляемом коде моделей повторного использования для COM-компонентов	289
Управляемые потоки и апартаменты COM	293
Работа COM-ориентированного кода C++ с управляемыми компонентами C++	294
Создание компонента .NET с помощью управляемого кода C++	295
Работа неуправляемого приложения с компонентами .NET	299
Использование управляемого компонента C++	301
Получение уведомлений о событиях, инициируемых управляемым компонентом C++	304
Размещение элементов управления форм Windows в неуправляемых контейнерах	312
Задание формы экспорта управляемого класса C++ в библиотеку типов COM	312
Задание формы экспорта управляемого интерфейса C++ в библиотеку типов COM	317
Глава 9. COM-программирование с помощью ATL	319
Что нового в ATL	320
Простой проект ATL 7.0	321
Добавление компонента в проект	326
Добавление методов	330
Создание проекта, использующего атрибуты	334
Преобразование DLL-файла в исполняемый файл	339
Новые классы ATL	341
Новые строковые классы	341
Классы преобразования строк	342
Пример проекта, использующего классы-утилиты	344
Глава 10. Введение в ATL Server	348
Архитектура	348

Разработка простого приложения ATL Server	349
Создание проекта	350
Генерируемый код	355
Компоновка, развертывание и запуск приложения	358
Модификация кода	359
Создание гостевой книги с помощью ATL Server	361
Модификация файла SRF	361
Реализация обработчиков	363
Компоновка и запуск проекта	367
Другие возможности ATL Server	367
Организация потоковых пулов	367
Кэширование	367
Мониторинг производительности	368
Глава 11. Создание Web-служб с помощью ATL Server	369
Преимущества Web-служб	370
Поиск информации о Web-службах	370
Web-службы и ATL Server	371
Создание Web-службы	372
Создание проекта	372
Файл Hello.h	373
Компоновка проекта	376
Запуск проекта	376
Простой клиент Web-службы	377
Проект службы биржевых котировок	380
Создание базы данных	380
Создание проекта	381
Написание кода	382
Компоновка и тестирование Web-службы	388
Разработка клиентского приложения	388
Работа с клиентским приложением	395

Введение

При всей шумихе вокруг Microsoft .NET многие программисты забыли поинтересоваться: а было ли время, затраченное на изучение C++, действительно удачным капиталовложением? Visual C++, флагман предыдущих версий Visual Studio (вспомните логотип "Created with Visual C++", которым сопровождалась версия 6.0), почти не упоминается в анонсах, где фигурируют ASP.NET, Visual Basic.NET и в особенности C#. Не означает ли это, что C++ не будет играть важной роли на новом этапе программирования для Windows? В настоящей книге мы покажем, что подобное утверждение сегодня не ближе к истине, чем 5 лет назад.

Во-первых, не следует забывать, что, когда речь идет о применении Visual C++, совсем не обязательно иметь в виду платформу .NET или даже программирование для Windows. Например, стандартное издание (Standard Edition) весьма популярно среди тех, кто хочет освоить "стандартное" программирование на C++ либо у себя дома, либо посещая учебные курсы. В Visual C++ .NET Microsoft реализовала ряд усовершенствований интегрированной среды разработки и продолжила свою политику следования стандарту ISO/ANSI для C++. В результате Visual C++ .NET достигает очень высокой степени соответствия стандартам.

Во-вторых, технологии COM и ATL подверглись существенному обновлению. В частности, продолжился процесс совершенствования ATL как библиотеки, применение которой не ограничивается COM-программированием (этот процесс начался с введения класса CWindow и соответствующей ему функциональности в версию 3.0). В распоряжении программистов теперь больше служебных классов, а также ATL Server — совершенно новый набор классов, основанный на существующей структуре ISAPI и предназначенный для создания оптимизированных Web-приложений и служб. Более того, COM-программирование получает мощный толчок благодаря введению атрибутов, которые избавляют разработчиков от необходимости манипулировать отдельными IDL-файлами и создают альтернативу написанию подробного повторяющегося кода.

Наконец, Microsoft включила в язык C++ расширения, благодаря которым этот язык получает все права на гражданство в .NET. Хотя, по-видимому, большая часть

кода для платформы .NET будет написана на Visual Basic или на C#, существует большой задел, который должен быть интегрирован в новые приложения в краткосрочной или среднесрочной перспективе. Только Visual C++ .NET может предложить механизм, обеспечивающий работу унаследованного кода совместно с новым кодом, и данный язык включает средства, специально предназначенные для этой цели.

Основные темы книги

По существу, настоящая книга преследует вполне конкретную цель. Она должна показать опытным программистам C++, что им необходимо знать о Visual C++ .NET. Для этого мы обсудим саму структуру .NET Framework, которая является лишь одной из основных тем. Из данной книги читатель также узнает о новых возможностях COM и ATL, а также о Web-службах и их реализации с помощью C++.

Первая половина книги целиком посвящена технологии .NET и последствиям ее внедрения в контексте программирования на C++. В главе 1 представлен краткий обзор тех особенностей .NET, которые следует учитывать при разработке приложений. Далее рассматриваются изменения языка C++, реализованные Microsoft для обеспечения доступа к указанным возможностям. Затем, вооруженные знаниями управляемого кода C++, мы обсудим некоторые ключевые особенности технологии .NET, имеющие отношение к программированию на C++. Мы начнем это обсуждение со сборок, атрибутов и метаданных и закончим анализом платформы .NET Framework как библиотеки классов, которая может использоваться совместно с MFC и даже с STL.

В начале второй половины книги обсуждается неуправляемый код C++ с точки зрения взаимодействия унаследованного кода с новым кодом, написанным для .NET Framework. Отметим, что подобное взаимодействие не поддерживается никакими другими языками программирования. Затем мы рассмотрим применение атрибутов в COM-программировании, новые особенности ATL (которая теперь представлена версией 7.0), ATL Server, а также Web-службы, созданные на основе ATL. Если вы хотите понять и оценить ту исключительно важную роль, которую язык C++ продолжает играть в создании приложений на платформах Microsoft, — эта книга для вас!

Необходимые аппаратные средства

Данная книга и содержащийся в ней код протестированы с помощью версии для распространения Microsoft Visual Studio.NET. Поэтому для работы с книгой необходимо выполнение таких же минимальных требований к аппаратным средствам, как и для указанного продукта: процессор PII-450, ОС Microsoft Windows NT 4.0 (или более поздняя версия), 128 Мб RAM и около 3 Гб свободного пространства на жестком диске. Код в большинстве примеров будет работать в среде стандартного издания (Standard Edition) Visual C++ .NET. Имея этот продукт, читатель сможет извлечь максимальную пользу из данной книги. В ней содержится также несколько демонстрационных примеров, в которых используются Visual Basic. NET и Visual C#.NET.

Соглашения, принятые в книге

Для того чтобы выделить разные типы информации, в книге используется несколько различных стилей форматирования текста и расположения материала. Так, код может быть представлен в различных видах. Если элементы кода

включены в текст (например, при обсуждении цикла `while`), этот код будет набран таким шрифтом. Если же код представляет собой блок, который должен быть набран в файле, то он будет иметь следующий вид:

```
• <?xml version 1.0?>
```

В других случаях возможен смешанный стиль:

```
<?xml version 1.0?>
<invoice>
  <part>
    <name>Widget</name>
    <price>$10.00</price>
  </part>
</invoice>
```

При этом на белом фоне располагается код, который уже встречался ранее либо не является важным для нашего обсуждения в данный момент. В противном случае код размещается на сером фоне.

Советы, пояснения и замечания представлены в этом стиле.

Важная информация помещается на прямоугольном поле, как в данном случае.

Маркеры располагаются после отступа. Мы используем следующие варианты набора текста:

- **Важные понятия и термины** выделяются жирным шрифтом.
- Текст, выводимый на экран или используемый в меню, например **File** или **Window**, набран тем же шрифтом, как на экране настольного компьютера с ОС Windows.
- Названия клавиш на клавиатуре набираются курсивом, например: *Ctrl* **ИЛИ** *Enter*.

Наконец, команды, которые должны вводиться в командной строке, помечаются пригласением ">" и набираются **жирным шрифтом**:

```
> something to type in the command line
```

Поддержка читателей и контакты с ними

Мы ценим отзывы, полученные от читателей, и хотим знать ваше мнение об этой книге: какие моменты вам понравились, а какие — нет и что следует улучшить в будущем. Вы можете прислать отзыв по электронной почте: feedback@wrox.com. Просим в ваших сообщениях указать название книги и ее номер ISBN.

Исходный код и его обновления

Весь исходный код для данной книги помещен на Web-сайт [wrox.com](http://www.wrox.com). Чтобы загрузить нужный файл, необходимо открыть страницу <http://www.wrox.com/> и найти заглавие книги с помощью системы поиска или одного из списков названий. Затем для получения исходного кода следует щелкнуть по ссылке **Download Code** на той странице, где дается детальная информация о книге.

Файлы на нашем сайте, предназначенные для загрузки на компьютеры пользователей, упакованы с помощью утилиты WinZip. После сохранения архива на вашем жестком диске следует извлечь файлы, хранящиеся в этом архиве, с помощью программы распаковки, такой как WinZip или PKUnzip. При распаковке нужно задать режим Use Folder Names (Использовать имена папок) либо эквивалентный ему режим в других программах.

Опечатки и исправления

Мы приложили все усилия, чтобы текст книги и приведенный в ней код не содержали ошибок. Однако никто не застрахован от опечаток и неточностей. Поэтому мы будем весьма признательны, если вы, обнаружив ошибку, сообщите о ней. Послав в издательство сообщения об ошибках, вы избавите других читателей от многих часов бесплодных усилий и поможете нам публиковать информацию более высокого качества. Направляйте сведения об обнаруженных вами неточностях по адресу: support@wrox.com. Ваша информация будет проверена, и в случае ее подтверждения мы опубликуем ее на странице, где представлены опечатки, или учтем в последующих изданиях книги.

Для просмотра опечаток следует зайти на сайт <http://www.wrox.com> и найти название книги с помощью системы поиска. Затем на странице, где дается детальная информация о книге, нужно щелкнуть на ссылке Book Errata. На открывшейся после этого странице будут представлены все ошибки, информация о которых получена и проверена издательством. Можно также щелкнуть мышью на ссылке Submit Errata, чтобы сообщить об обнаруженных вами ошибках.

Техническая поддержка

Если вы хотите обсудить проблему, относящуюся к материалу книги, со специалистом, знающим книгу в деталях, пошлите сообщение по адресу: support@wrox.com с указанием названия книги и последних четырех цифр номера ISBN в поле "Тема сообщения". Электронное послание должно иметь следующий вид:

- В поле "Тема сообщения" указываются **название книги, последние четыре цифры номера ISBN и номер страницы**, относящейся к вашей проблеме.
- О В основном тексте письма необходимо указать **ваше имя и контактную информацию**, а также изложить суть **проблемы**.

Мы *не будем* заваливать вас рекламными посланиями. Указанная информация нужна нам для того, чтобы сэкономить ваше и свое время. Полученное от вас сообщение может передаваться по следующей цепочке:

- **Поддержка читателей.** Ваше послание доставляется сотрудникам службы поддержки читателей, которые первыми знакомятся с ним. Они имеют перечень наиболее часто задаваемых вопросов и немедленно ответят на любые вопросы общего характера, относящиеся к материалу книги и Web-сайту.
- **Редакторы.** Более узкие вопросы передаются техническому редактору, ответственному за книгу. Редактор имеет опыт работы с интересующим вас продуктом и может ответить на технические вопросы по материалу книги. Найдя решение проблемы, редактор может опубликовать его на Web-сайте.
- **Авторы.** Наконец, в том маловероятном случае, если редактор не сможет решить вашу проблему, он направит запрос авторам. Они будут рады

ответить на действительно важные конкретные вопросы. Все авторы издательства Wrox участвуют в программе поддержки своих книг. Они ответят читателю и редактору, в результате чего выиграют все читатели.

Описанная процедура относится только к тем проблемам, которые непосредственно связаны с содержанием конкретной книги. Если суть вопроса выходит за рамки, установленные для обычной программы поддержки книг, то ответ можно получить с помощью сообщества программистов на нашем форуме по адресу: <http://p2p.wrox.com>.

Ресурс p2p.wrox.com

Для обсуждения проблем с авторами и коллегами вы можете добавить свой адрес в **списки рассылки сообщений P2P (Programmer to Programmer™** — Программист программисту). Наша уникальная система контактов между программистами, обеспечивающая обмен информацией с помощью списков рассылки, форумов и групп новостей, является *дополнением к* программе персональной поддержки читателей по электронной почте. Можете быть уверены, что ваша проблема будет проанализирована многими авторами издательства Wrox и другими специалистами, чьи адреса имеются в списках рассылки. На сайте p2p.wrox.com вы найдете несколько различных списков, которые помогут вам не только при работе над этой книгой, но и при создании ваших собственных приложений.

Чтобы включить свой адрес в список рассылки, выполните следующие действия:

1. Зайдите на страницу <http://p2p.wrox.com>.
2. Выберите нужную категорию в левой панели меню.
3. Щелкните на том списке рассылки, к которому вы хотите присоединиться.
4. Следуйте указаниям для подписчиков и введите ваш адрес электронной почты, а также пароль.
5. Ответьте на электронное сообщение, которое будет послано вам для подтверждения подписки.
6. Используйте диспетчер подписки для того, чтобы присоединиться к другим спискам рассылки и указать область своих интересов.

ГЛАВА 1



Новые возможности Visual C++

В течение многих лет C++ был наиболее популярным языком разработки приложений и компонентов с высоким уровнем оптимизации. Несмотря на большой успех Java в последнее время, в тех ситуациях, когда дело касается создания высокопроизводительного кода, языку C++ продолжают отдавать предпочтение миллионы программистов во всем мире.

Выход в свет новой версии Visual Studio был приурочен к выпуску платформы Microsoft .NET Framework, которая обеспечивает независимую от языка, устойчивую среду разработки приложений как для Web, так и для локальных компьютеров. В связи с появлением этой платформы был обновлен язык Visual Basic, теперь называемый Visual Basic .NET; технология ASP превратилась в ASP.NET; кроме того, появился новый язык C#. Когорты программистов C++ получили в свое распоряжение Visual C++ .NET.

Поскольку в течение многих лет на Visual C++ было написано огромное количество кода, Microsoft решила, что любые усовершенствования C++ не должны исключать использование предыдущих версий. В то же время Microsoft поставила задачу предоставить программистам C++ возможность использования новой платформы. В настоящей главе мы преследуем две цели: показать читателю .NET Framework "с умеренной высоты" и обсудить изменения в Visual Studio (теперь называемой Visual Studio .NET), которые оказывают влияние на процесс разработки приложений с помощью C++.

Маловероятно, что мы будем писать совершенно новые .NET-приложения на языке C++. Для этой цели существуют более подходящие варианты. В окружении .NET важная роль C++ заключается в обеспечении взаимодействия между проверенным и протестированным кодом, созданным до появления .NET, и новыми приложениями, использующими возможности .NET Framework. Чтобы добиться решения этой задачи, программистам необходимо четко понимать механизмы работы .NET, к обсуждению которых мы и приступим.

Не все нововведения в языке Visual C++ .NET, несмотря на его название, связаны с .NET Framework. Мы обсудим изменения и усовершенствования, относящиеся к COM- и ATL-программированию, а некоторых из этих усовершенствований коснемся в настоящей главе.

Данная глава распадается на три достаточно обособленные части, которые читатель может лишь бегло просмотреть, если он в достаточной степени знаком с .NET и Visual C++. Вначале мы дадим представление о .NET Framework, выделяя ключевые особенности этой платформы (которые мы более детально рассмотрим в последующих главах). Затем мы остановимся на изменениях среды Visual Studio IDE. Некоторые из этих изменений носят косметический характер, тогда как другие вытекают из новой функциональности. Наконец, мы рассмотрим простой пример применения некоторых возможностей .NET, который продемонстрирует средства отладки приложений, предоставляемые Visual Studio .NET.

Платформа .NET Framework

Microsoft называет .NET Framework "новой вычислительной платформой, которая упрощает разработку приложений в сильно распределенном окружении Интернета". Практически это означает, что любой компьютер с установленной на нем .NET Framework использует среду исполнения, совместимую с точки зрения производительности, интерфейсов и безопасности. Кроме того, благодаря различным механизмам, .NET Framework делает несущественным относительное расположение хранилища кода и процесса исполнения, а также упрощает сложные процедуры развертывания и управления версиями. Но и это еще не все. .NET Framework предоставляет также среду разработки, которая освобождает программистов от некоторых традиционных, но утомительных обязанностей.

Платформа .NET Framework состоит из двух различных, но связанных друг с другом частей: **единой среды исполнения (Common Language Runtime — CLR)** и **библиотеки классов**. В общих словах можно сказать, что .NET Framework — это "службы, предоставляемые ею", и "то, что она реализует". Большая часть общеизвестных возможностей .NET (сборка мусора, управление версиями, управление потоками и др.) обеспечивается средой CLR с помощью средств, к обсуждению которых мы скоро приступим. С другой стороны, когда программисты говорят, что .NET позволяет создавать приложения Windows Forms и Web-службы на основе XML, в этом случае речь идет о функциональности, предоставляемой библиотекой классов. Рассмотрим вначале среду CLR.

Единая среда исполнения

Когда мы компилируем исходный код (приложение, библиотеку, элемент управления и т.д.) для платформы .NET Framework, то генерируемый объектный код не ориентирован на набор команд CPU того компьютера, на котором мы работаем. Компиляция для .NET фактически представляет собой компиляцию для CLR, т.е. в результате компиляции генерируется код на новом языке, называемом **промежуточным языком Microsoft (Microsoft Intermediate Language - MSIL или просто IL)**. Язык MSIL определяет набор команд для "виртуального" процессора. Таким образом, код, компилируемый в IL, перед запуском на конкретном процессоре требует дальнейшей компиляции в набор команд, присущий этому процессору.

Среда CLR предоставляет такое средство, как **компилятор по требованию** или **оперативный компилятор (just-in-time compiler)**, который компилирует код на языке IL в собственный машинный код. Будучи однажды откомпилирован, код кэшируется. По завершении работы приложения этот код может быть удален из

системы. Более того, не обязательно компилируется сразу вся программа. Скорее всего, код будет компилироваться по частям там и тогда, когда это необходимо. При запуске программы весь код может и не потребоваться, поскольку пользователь не обязательно будет применять функциональность приложения в полном объеме.

Независимо от того, насколько хорошо оптимизирован описанный процесс, он вряд ли будет столь же эффективным, как в случае компиляции исходного кода в собственный машинный код. Поэтому должны быть веские причины, чтобы компилировать код для CLR. Естественно, такие причины существуют, причем их несколько (пару из них мы уже упомянули). Мы поговорим об этих причинах более подробно в последующих разделах.

Сборка мусора

Программистам C++ всегда досаждали утечки памяти, которые могут возникать в случае, когда ресурсы, более не используемые приложением, не освобождаются должным образом. Но проблема заключается не только в этом. Неприятности могут возникнуть и тогда, когда программист слишком торопится освободить ресурсы. В результате возникает риск появления в программе некорректного обращения к области памяти. Среда CLR платформы .NET Framework предлагает решение обеих проблем.

Среда CLR обеспечивает сборку мусора, а также позволяет управлять временем жизни объектов приложения. Она несет ответственность за учет действительных ссылок и за подсчет ссылок. CLR периодически проверяет все ссылки на объекты. Если ссылки на некоторый объект в данный момент отсутствуют, этот объект удаляется из памяти. Когда сборщик мусора завершает удаление объектов, он прибирает за собой, уплотняя кучу и обеспечивая тем самым непрерывность областей памяти, выделяемых новым объектам, что повышает эффективность приложения.

Переносимость кода

Способность CLR поддерживать переносимость кода естественным образом вытекает из тех особенностей этой среды, которые обсуждались выше. Поскольку исходный код .NET-приложения всегда компилируется в IL, то откомпилированный код IL может работать на любой платформе, обеспечивающей функционирование .NET CLR. На сегодняшний день среда CLR доступна только на платформах Windows. Однако это положение должно измениться: проект Mono Project (www.go-mono.com), призванный реализовать технологию .NET на платформе Linux, уже находится на стадии осуществления.

Интероперабельность языков

Каждый язык, который должен работать в среде .NET и поддерживать взаимодействие между приложениями, обязан соответствовать спецификациям Microsoft. Любой .NET-совместимый язык *должен* быть объектно-ориентированным и использовать стандартный набор типов данных, называемый **системой общих типов (common type system — CTS)**. В дополнение к тем изменениям, которые внесла Microsoft в свои собственные языки, многие сторонние организации также модифицировали языки для поддержки платформы .NET Framework, такие как COBOL, Eiffel, Perl, Python и SmallTalk.

С точки зрения CLR все языки оказываются одинаковыми. Если компилятор способен генерировать код IL, этого для CLR вполне достаточно. Как только код приложения или библиотеки откомпилирован в код IL, исходный язык программирования более не имеет значения. В результате можно вызывать любую

библиотеку, написанную на одном языке, из среды другого языка без каких бы то ни было сложностей: типы, используемые для передаваемых аргументов и возвращаемых значений, будут всегда совместимыми.

Другое преимущество, вытекающее из комбинирования IL и CTS, связано с облегчением отладки приложений. Можно не только использовать в одном и том же приложении код, написанный на различных .NET-совместимых языках, но и отлаживать многоязычный код в одном и том же сеансе, используя совместимый интерфейс.

Безопасность кода

CLR "доверяет" коду IL в различной степени, руководствуясь несколькими критериями, не последним из которых является место происхождения кода: локальная машина, LAN или Интернет. Исходя из степени доверия, могут накладываться ограничения на операции, выполняемые конкретным фрагментом кода. Здесь имеются в виду ограничения, связанные с доступом к ресурсам компьютера, где фактически исполняется приложение.

Поскольку код IL откомпилирован, CLR получает возможность проверить его, прежде чем он будет исполняться на реальном процессоре. Разрешение на исполнение этого кода, написанного в сторонней организации или другим членом группы, может быть предоставлено либо нет, в зависимости от привилегий текущего пользователя. Среда CLR обеспечивает безопасность на уровне приложения для выполняемой программы.

Доступ к библиотеке классов .NET Framework

Наконец, упомянем еще одно преимущество, предоставляемое средой CLR. При написании кода C++, предназначенного для работы в CLR, мы автоматически получаем быстрый и удобный доступ к библиотеке классов .NET Framework. Этот набор классов имеет столь богатую функциональность, что по сравнению с ней меркнет функциональность ATL и даже MFC. Указанная библиотека содержит типы, учитывающие все аспекты программирования для Windows, и мы скоро обсудим некоторые из них. Однако прежде коснемся вопроса о *сущности* кода, который используется средой CLR, благодаря чему мы и получаем упомянутые преимущества.

Свойства управляемого кода

Код, написанный для CLR и использующий ее возможности, называется **управляемым кодом (managed code)**. Нетрудно догадаться, что код, не использующий средства CLR (иными словами все то, что мы писали в течение многих лет), теперь называется **неуправляемым кодом (unmanaged code)**. Как уже было сказано, при компиляции кода, который должен работать в среде CLR, генерируется код на промежуточном языке (IL). Однако мы еще не говорили о том, каким образом этот код затем распределяется.

Когда мы компоуем приложение **Managed C++**, написанное на Visual C++ .NET, создается EXE-файл. В результате компоновки библиотеки Managed C++ генерируется DLL-файл. Однако на этом заканчивается сходство между тем, что мы при этом получаем, и тем, что имели раньше. Указанные файлы имеют переносимый **исполняемый формат (portable executable - PE)** и содержат не только откомпилированный код MSIL, но и метаданные, описывающие все, что относится к этому коду: типы, используемые кодом, классы, которые он содержит (в том числе их методы и свойства) и другие файлы, чьи типы также используются кодом. Такие PE-файлы называются **сборками** (подробнее о них см. в главе 3). Уже то немногое, что мы сказали о сборках, помогает понять, за счет чего среда CLR приобретает возможности, перечисленные выше, а также способствует решению двух других задач.

Развертывание

Применение платформы .NET Framework значительно упрощает развертывание приложений. Поскольку сборки несут в себе полное описание своей функциональности, нет необходимости регистрировать эту информацию где-либо в системе. Установка .NET-приложений часто столь же проста, как и перенос файлов из одного каталога в другой. Если одна из устанавливаемых библиотек несовместима с той, что уже установлена, это не создает проблемы.

Управление версиями

Одно из основных обоснований экономической целесообразности приобретения платформы .NET заключается в том, что эта платформа решает проблему, называемую "адом DLL" (DLL Hell). Данная проблема возникает, когда одна версия библиотеки заменяет другую (обычно старая версия заменяется новой, хотя и не всегда). Это приводит к тому, что многие установленные приложения не могут быть использованы. Сборки, создаваемые на платформе .NET, содержат номера своих версий, которые являются частью метаданных. Поэтому среда CLR имеет достаточно информации о версиях, чтобы в случае необходимости можно было одновременно загрузить в систему две версии одной и той же библиотеки для обслуживания как старых, так и новых клиентов.

Мы обсудим эти особенности управляемого кода более подробно в главах 2, 3 и 4. Теперь же обратимся ко второй составной части .NET Framework: библиотеке классов.

Библиотека классов .NET Framework

Библиотека классов .NET Framework содержит набор повторно используемых классов, позволяющих разрабатывать невероятно широкий спектр приложений. В частности, эти классы предоставляют стандартные сервисы, такие как доступ к базам данных, диагностика, обеспечение безопасности и др. Кроме того, указанные классы позволяют организовывать работу в сети и разрабатывать изолированные, распределенные и Web-ориентированные приложения. Классы, относящиеся к библиотеке .NET Framework, поддерживают также XML и WSDL (Web Service Description Language — язык описания Web-служб), что позволяет создавать Web-приложения и Web-службы на основе стандартных протоколов.

Этот набор классов совместно используется всеми .NET-приложениями. Все классы, относящиеся к библиотеке классов .NET, располагаются в определенном пространстве имен, что служит хорошим примером организации подобной системы. Так же, как библиотека STL для C++ организована с помощью std (и библиотека ATL теперь содержится в пространстве имен ATL), классы .NET Framework принадлежат к пространству имен System. Последнее, в свою очередь, содержит множество пространств имен второго и третьего уровня. Ниже перечислены некоторые из этих пространств и указана их функциональность:

- `System::Data` — определяет архитектуру объектов ADO.NET, которые используются для доступа к источникам данных и для управления ими.
- `System::Xml` — обеспечивает поддержку XML.
- `System::Diagnostics` — используется для отладки, трассировки, создания журналов регистрации и мониторинга производительности системы.
- `System::DirectoryServices` — содержит классы, обеспечивающие доступ к активным каталогам.
- `System::Net` — включает классы, предназначенные для организации работы в сети

- `System::Drawing` — обеспечивает доступ к графике GDI+.
- `System::Windows::Forms` — **определяет** классы, применяемые для создания приложений Windows.
- D `System::Security` — реализует систему безопасности CLR и содержит классы, обеспечивающие определение политики безопасности, управление доступом и перемещение по стеку.
- `System::Security::Cryptography` — **включает** классы, предназначенные для выполнения функций криптографии, таких, как кодирование/декодирование данных, хэширование, генерация случайных чисел, аутентификация сообщений и генерация цифровых подписей.
- a `System::Web` — **обеспечивает базовую инфраструктуру для ASP.NET** и поддержку форм Web.
- `System::Web::Services` — включает классы, необходимые для поддержки Web-служб, использующих протокол SOAP.

Этот список может создать впечатление, что мы будем обсуждать огромное количество приложений, разрабатываемых на платформе .NET Framework, не конкретизируя их. Чтобы сделать наше изложение более предметным, коснемся четырех типов приложений, которые являются наиболее вероятной целью разработок с использованием .NET Framework. В большинстве случаев мы не будем писать эти приложения на C++, который предлагает другие способы реализации той же самой функциональности. Однако с большой вероятностью читатель может обнаружить, что одному из этих типов приложений потребуется старый код, и именно в этом случае должен сыграть СБОЮ роль язык C++.

Приложения ASP.NET

Microsoft модифицировала свою модель ASP. ASP.NET может использоваться для разработки традиционных Web-приложений (создаваемых ранее с помощью ASP 2.0 и 3.0), для чего теперь служат формы Web. Кроме того, инфраструктура ASP.NET позволяет создавать Web-службы — новый тип приложений (см. ниже).

Приложение, разработанное на основе форм Web, является типичной конструкцией ASP, которая использует язык HTML для отображения данных на стороне клиента и протокол HTTP для связи между клиентом и сервером. Благодаря формам Web клиент приобретает внешний вид и возможности, аналогичные формам Visual Basic, а также может использовать компоненты ActiveX и фрагменты данных cookies для управления состоянием.

Приложения Windows

Технология .NET создавалась прежде всего для того, чтобы облегчить взаимодействие компонентов и приложений в сети. Однако эта технология, естественно, позволяет разрабатывать .NET-приложения, предназначенные для работы в Windows и представляющие собой отдельные исполняемые файлы, работающие на платформе .NET Framework под управлением CLR. Если не учитывать последнее обстоятельство, то можно сказать, что мы занимались разработкой этого типа приложений в течение многих лет.

Консольные приложения

Консольные приложения, будучи в достаточной степени старомодными, являются идеальным средством для тестирования базового кода. Поэтому они поддерживаются

библиотекой классов .NET Framework. Функциональность консольных приложений отнесена к пространству имен System; Console. Где бы мы были сейчас без них?

Web-службы

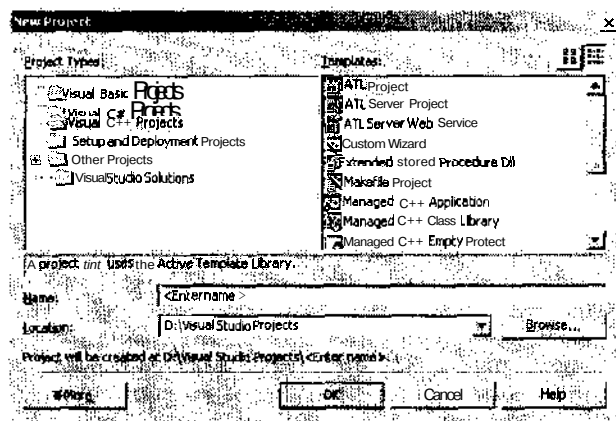
Web-службу можно определить как компонент программируемого приложения, который предоставляет сервисы и данные удаленным клиентам с помощью стандартных протоколов Web. Web-службы используют такие две технологии, основанные на XML, как WSDL (Web Service Description Language) и SOAP (Simple Object Access Protocol — Простой протокол доступа к объектам). Эти технологии соответственно служат для предоставления информации о действиях, выполняемых Web-службами, для передачи данных и вызова методов. Благодаря такой модели процедура доступа к Web-службам совершенно не зависит от их реализации.

Новые особенности Visual Studio .NET

рассмотрим некоторые более общие вопросы. Учитывая те изменения, которые реализованы в области программирования для Windows, естественно ожидать, что среда Visual Studio .NET также подверглась определенным изменениям. Конечно же, так оно и есть. В силу того, что все возможности и функции .NET Framework теперь доступны программистам, работающим с любым языком Microsoft, все эти языки совместно используют одну и ту же интегрированную среду разработки (IDE). Более не существует отдельной среды разработки для Visual Basic, как это было в версии 6.0.

Кроме соображений эстетики и удобства, существуют гораздо более веские причины для подобной интеграции. Напомним, что компоненты .NET-приложения могут создаваться с применением различных языков, и поэтому иногда бывает необходимо отлаживать приложение сразу на нескольких языках. Такая необходимость (вместе со многими другими) учтена в новой версии.

Нас интересуют в первую очередь те нововведения, которые оказывают влияние на процесс создания приложений с использованием языка Visual C++. Оставляя пока в стороне косметические изменения графического интерфейса пользователя (GUI), обратимся к команде меню File | New | Project.... Эта команда открывает диалоговое окно, содержащее больше различных типов проектов, чем когда-либо, в том числе новые типы проектов Visual Basic и C#. Если на панели Project Types (Типы проектов) выбрать папку Visual C++ Projects (Проекты Visual C++), диалоговое окно New Project (Новый проект) примет следующий вид:



- ATL Project (Проект ATL)
- ☐ ATL Server Project (Проект ATL Server)
 - ATL Server Web Service (Проект создания Web-службы с помощью ATL Server)
- a Custom Wizard (Мастер пользователя)
 - Extended Stored Procedure DLL (Проект создания библиотеки DLL для расширенной хранимой процедуры)
 - Makefile Project (Проект создания make-файла)
 - ManagedC++ Application (Управляемое приложение C++)
- ☐ Managed C++ Class Library (Библиотека управляемых классов C++)
 - Managed C++ Empty Project (Пустой проект для работы с управляемым кодом C++)
 - ManagedC++ Web Service (Создание Web-службы с помощью управляемого кода C++)
 - MFC ActiveX Control (Создание компонента ActiveX на базе MFC)
- D MFC Application (Приложение MFC)
 - ☐ MFC DLL (Создание динамической библиотеки на базе MFC)
 - ☐ MFC ISAPI Extension DLL (Создание на базе MFC динамической библиотеки для расширения ISAPI)
- O Win32 Project (Проект Win32)

Мы не будем подробно рассматривать каждый из перечисленных шаблонов, поскольку некоторые из них служат для решения задач, обсуждение которых выходит за рамки настоящей книги. Тем не менее ниже кратко описаны все указанные шаблоны, перечислены их новые возможности и показано, в какой степени эти шаблоны будут использоваться в проектах, создаваемых в последующих главах.

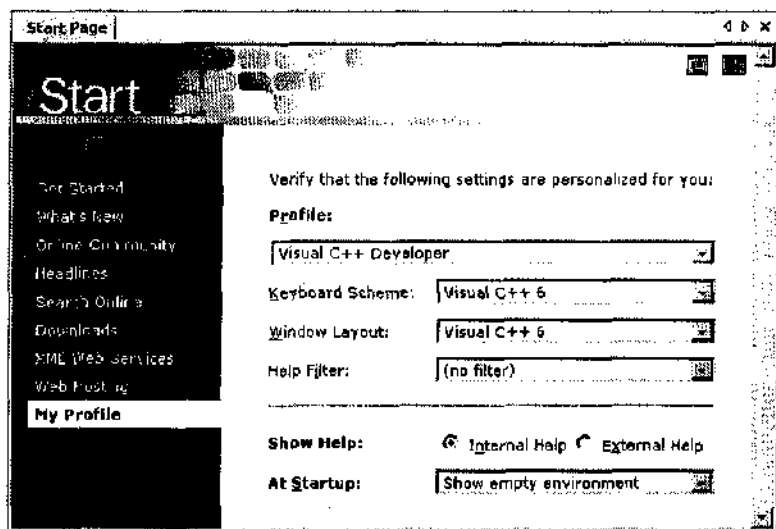
Новый вид "старых" мастеров

Прежде чем обратиться к новым типам проектов, рассмотрим изменения, которым подверглись мастера, имевшиеся в Visual C++ 6.0. Все мастера выглядят по-новому, и многие из них реализуют новые функции. Кроме того, изменились некоторые базовые технологии, на что мы также обратим внимание в процессе обсуждения. Наконец, анализ изменений внешнего вида мастеров Visual C++ помогает лучше проиллюстрировать изменения интегрированной среды разработки, и мы сделаем это на примере наиболее известного мастера.

MFC Application (Приложение MFC)

MFC Application Wizard (Мастер создания приложений MFC) использовался для разработки приложений MFC в течение многих лет. Этот мастер создает файлы, необходимые для начала разработки приложения на основе Microsoft Foundation Classes (Библиотека базовых классов Microsoft ~ MFC). Но, прежде чем приступить к изучению нововведений, стоит привести кое-что в порядок. Когда мы писали код на C++, окно Project Workspace (Рабочее пространство проекта), которое в Visual Studio .NET называется Solution Explorer, располагалось в левой части экрана. Однако в соответствии с установками, принятыми по умолчанию, это окно теперь размещается справа. К счастью, не нужно перебирать команды меню для поиска

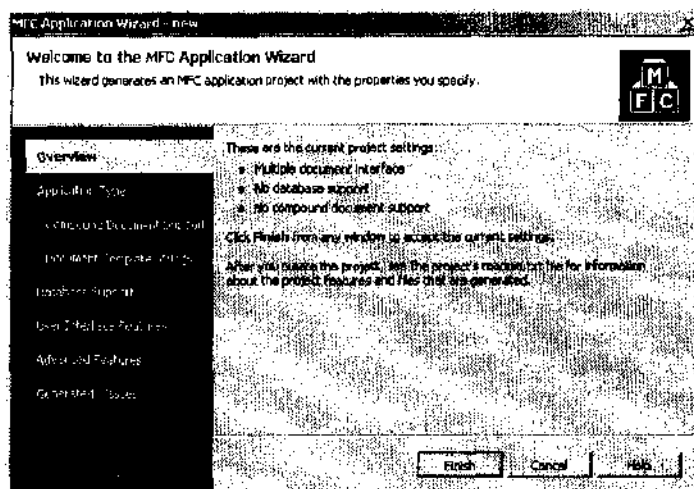
нужных окон. Вместо этого можно просто выбрать команду **Help | Show Start Page** (Справка | Показать начальную страницу):



Если теперь выбрать установки, показанные выше, то среда разработки примет более привычный вид. Нетрудно заметить несколько существенных усовершенствований этой среды.

Диалоговое окно мастера

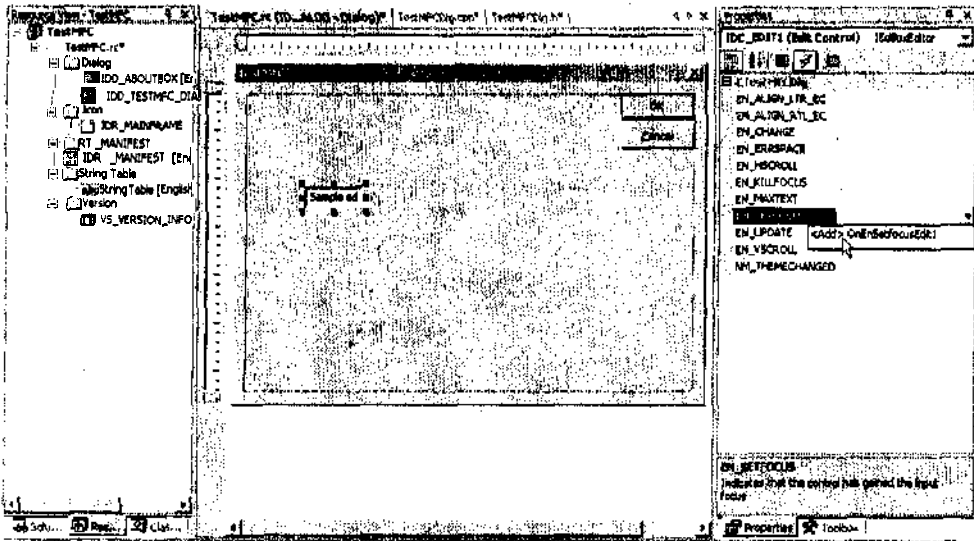
При вызове мастера MFC Application Wizard в Visual Studio .NET откроется следующее диалоговое окно:



Ясно, что этот пользовательский интерфейс существенно образом отличается от того, который применялся в Visual Studio 6.0. Вместо того, чтобы заставлять пользователя выполнять последовательность шагов, все возможные действия

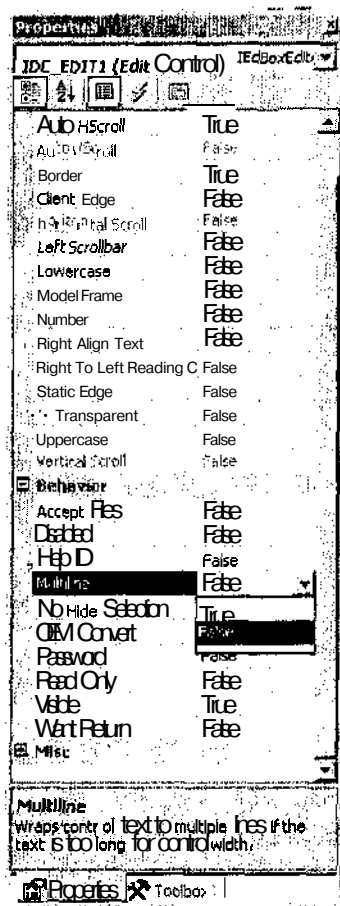
Элементы управления, свойства и события

Если окно Class View практически не изменилось, то этого нельзя сказать о мастере создания классов (ClassWizard), функциональность которого в значительной степени распределена между различными элементами среды разработки. Мастер же WizardBar полностью исключен. Не вдаваясь здесь в подробности, скажем лишь, что эти изменения, очевидно, продиктованы желанием разработчиков стандартизировать процедуру создания приложений для всех языков, поддерживаемых Visual Studio .NET. Так, вместо классических интерфейсов Visual C++ мы имеем окно Properties (Свойства) в стиле Visual Basic 6.0:



Здесь мы открыли окно Resource View (Просмотр ресурсов) и добавили текстовое поле в главное диалоговое окно приложения MFC. В окне Properties перечислены события, которые это текстовое поле может передавать родительскому окну. Выбор элемента списка, выделенного на рисунке, приведет к добавлению метода `OnEnSetFocusEdit1()` к основному классу диалогового приложения `CTestMFC`. Слева от кнопки со значком молнии, предназначенной для отображения событий, расположена кнопка, позволяющая выводить на экран свойства всех элементов проекта: исходных файлов, переменных, классов, элементов управления и т.д. В данном случае мы имеем пример удачного усовершенствования старых окон свойств, которые раньше надо было "закрепить", чтобы они не пропадали с экрана.

Мы не будем подробно рассматривать изменения интегрированной среды разработки Visual C++, однако уже это краткое обсуждение дает представление о сути изменений и показывает, почему они были необходимы. В целом можно сказать, что эти изменения представляют собой комбинацию прямых усовершенствований функциональности и модификации с целью достижения сходства с другими языками программирования. Имея это в виду, продолжим обзор "старых" мастеров.



Проект MFC DLL (Создание динамической библиотеки на базе MFC)

При разработке библиотеки DLL на основе MFC нужно, естественно, использовать вариант проекта MFC DLL. Как и в предыдущем случае, этот мастер изменился незначительно. Пользователь может задавать все установки проекта, выбрав категорию Application Settings (Установки приложения). При этом можно создать стандартный модуль DLL, который использует разделяемый DLL-модуль библиотеки MFC, стандартный DLL-модуль с постоянной привязкой к MFC или DLL-модуль расширения MFC. Можно также добавить поддержку автоматизации и интерфейса Windows Sockets.

Проект MFC ActiveX Control (Создание компонента ActiveX на базе MFC)

Этот проект используется, когда необходимо разработать элемент управления ActiveX с помощью библиотеки MFC. Мастер MFC ActiveX Control создает класс элемента управления на базе MFC и класс страницы свойств.

Окно Application Settings позволяет задать генерацию файла справки и включить поддержку лицензии для использования (run-time license). С помощью

категории Control Names (Имена элементов управления) можно задать имена элемента управления и классов страниц свойств, а также файлов, которые их реализуют. В окне Control Settings (Установки для элемента управления) можно придать дополнительные свойства элементу управления, такие как невидимость во время выполнения, активизация без эффекта мерцания, незаконная активизация и др.

Проект MFC ISAPI Extension DLL (Создание на базе MFC динамической библиотеки для расширения ISAPI)

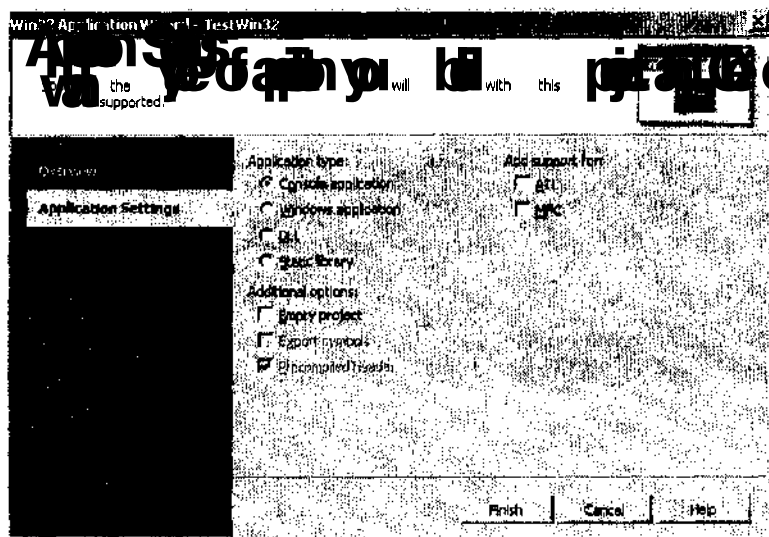
Функциональность сервера Internet Information Server (IIS) можно нарастить путем создания DLL-модулей расширения. Мастер MFC ISAPI Extension DLL представляет один из возможных способов разработки таких расширений. Категория Object Settings (Установки объекта) позволяет задать генерацию фильтра для расширения ISAPI, а также указать имя и описание для класса фильтра. Пользователь также может выбрать имя и описание для объекта расширения сервера. Опция Notifications (Уведомления) позволяет выбрать приоритет уведомлений, типы соединений и т.д. Мастер генерирует объект фильтра, уведомления объекта фильтра и класс расширения сервера.

Makefile Project (Проект Makefile)

Для тех пользователей, которые предпочитают компилировать и компоновать проекты из командной строки, Visual C++ .NET предоставляет такую возможность. Запустив мастер Makefile Application Wizard и открыв окно Application Settings, пользователь может задать командную строку Build со всеми ключами, именем выходного файла (Output file), любыми командами Clean и командной строкой Rebuild.

Win32 Project (Проект Win32)

В списке окна New Project можно заметить отсутствие одного из типов проектов — консольного приложения Win32, нашей излюбленной клиентской программы тестирования. К счастью, она по-прежнему с нами, хотя ей и не уделено места среди ведущих приложений. Мастер Win32 Application Wizard позволяет создавать как проект Win32, так и два других популярных типа проекта:



Как видно из рисунка, этот мастер служит для разработки приложений Windows, модулей DLL и статических библиотек с возможностью поддержки библиотекой ATL. Позже мы воспользуемся этой возможностью, например при рассмотрении программирования с применением ATL 7.0.

Custom Wizard (Мастер пользователя)

Шаблон Custom Wizard позволяет создать собственного мастера для разработки проектов в соответствии с определенными задачами. После указания того, сколько страниц должен содержать новый мастер пользователя, мастер Custom Wizard создаст страницу DHTML для каждой из запрашиваемых страниц. Все генерируемые страницы затем могут быть настроены так, чтобы они включали необходимые нам действия. Теперь, в связи с тем, что диалоги используют DHTML, мастера пользователя реализуются *на JavaScript*, а не на C++.

Extended Stored Procedure DLL (Проект создания библиотеки DLL для расширенной хранимой процедуры)

Последний из "старых" мастеров представлен шаблоном Extended Stored Procedure DLL, с помощью которого можно разрабатывать динамически подключаемую библиотеку для расширенной хранимой процедуры. Этот мастер создает один файл с расширением .cpp, содержащий именованную хранимую процедуру, и другой файл, определяющий точку входа для DLL.

Новые мастера

Обратимся теперь к тем мастерам, которые являются новыми для Visual C++ или, по крайней мере, существенно изменились в том или ином отношении. Последние служат для создания приложений тех типов, с какими мы в основном будем иметь дело на протяжении данной книги. Поэтому у читателя будет достаточно возможностей познакомиться с ними.

Managed C++ Application (Управляемое приложение C++)

Возможность создавать "скелет" приложения C++, которое будет использовать управляемый код, естественно является новой особенностью Visual C++ .NET. Мастер Managed C++ Application не требует задавать вручную какие-либо параметры проекта (кроме имени) и генерирует .cpp-файл, куда можно добавить управляемый код C++. Для тех читателей, кто уже проявляет любопытство, приведем пример этого файла:

```
// Это главный файл проекта для приложения VC++
// созданный мастером Application Wizard

#include "stdafx.h"

using <mscorlib.dll>
#include <tchar.h>
using namespace System;

// Это точка входа в приложение
int _tmain(void)
{
    // TODO: Замените приведенный ниже код своим собственным кодом.
    Console::WriteLine(S"Hello World");
    return 0;
}
```

Этот код выводит сообщение "Hello World" в окно консоли. Он имеет семантическое сходство с традиционным кодом Visual C++, но при этом использует библиотеки классов .NET Framework для реализации требуемой функциональности. Мы обсудим этот код более подробно в следующей главе и в дальнейшем приведем множество примеров использования управляемых расширений C++.

Managed C++ Class Library (Библиотека управляемых классов C++)

Для создания DLL, использующей управляемые расширения C++, служит мастер Managed C++ Class Library Wizard, генерирующий необходимые файлы с расширениями .cpp и .h.

Managed C++ Empty Project (Пустой проект для работы с управляемым кодом C++)

Как следует из названия, мастер Managed C++ Empty Project Wizard создает пустой проект, в который пользователь должен добавить свои собственные файлы, написанные по собственному усмотрению.

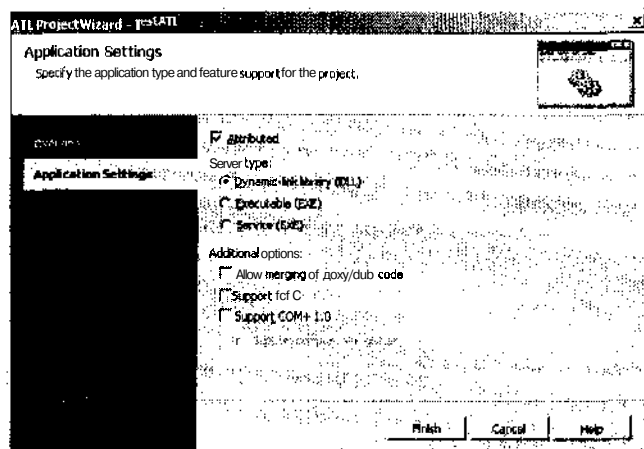
Managed C++ Web Service (Создание Web-службы с помощью управляемого кода C++)

Шаблон Managed C++ Web Service позволяет создавать Web-службы, используя управляемые расширения C++. В этом проекте генерируется файл обнаружения с расширением .vsdisco, описывающий Web-службу на языке WSDL. Замена принятых по умолчанию установок конфигурации Web-службы может быть выполнена с помощью файла Web config. Мастер Managed C++ Web Service Wizard генерирует также .asmx-файл, инкапсулирующий функциональность Web-службы.

Пока эта терминология может быть непонятна читателю. Мы рассмотрим Web-службы в главе 11, однако для их создания не будем применять управляемый код C++, а используем библиотеку ATL, которая приобрела перечисленные (а также некоторые другие) возможности в своей новой версии 7.0.

ATLProject (Проект ATL)

Кажущиеся на первый взгляд поверхностными изменения ATL Project Wizard могут ввести в заблуждение. Вместе с обычными вариантами выбора типа создаваемого сервера и дополнительных возможностей, которые мы хотим поддерживать, мастер содержит непритязательный флажок Attributed (Использование атрибутов):



Данный режим подробно обсуждается в главе 9. Вкратце скажем лишь, что это средство совершает революцию в COM- и ATL-программировании, позволяя добавлять к коду C++ атрибуты в стиле языка IDL. В результате обеспечивается включение в проект стандартных фрагментов кода непосредственно перед компиляцией, что неизмеримо облегчает процесс разработки.

ATLServerProject(ПроектATLServer)

ATL Server (Сервер ATL) представляет собой набор классов ATL, которые облегчают разработку высокопроизводительных Web-приложений, непосредственно связанных с технологиями IIS и ISAPI. Мастер ATL Server Project Wizard служит основой для создания таких приложений, а также Web-служб на базе XML. В числе других мастер генерирует файлы с расширениями .src и .h. Первый файл содержит инициализирующую программу для Web-приложения, а второй определяет класс обработчика запросов сервера ATL, содержимое которого зависит от установок мастера (подробнее о работе мастера ATL Server Project Wizard и сервера ATL см. в главе 10).

ATL Server Web Service (Проект созданияWeb-службы с помощьюATLServer)

Последний мастер, который имеет не столь важное значение, — это ATL Server Web Service Wizard. И дело здесь, разумеется, не в том, что Web-службы (см. главу 11) мало полезны. Этот мастер не предоставляет никаких новых возможностей по сравнению с предыдущим мастером и потому является избыточным, и не будет предметом нашего обсуждения.

Разработка и отладка приложений с использованием нескольких языков

Как уже отмечалось, основной особенностью платформы .NET является ее нейтральность по отношению к языкам программирования. Программисты могут разрабатывать компоненты на любом языке, который им нравится, будучи уверенными, что все, созданное ими, будет совместимо с компонентами, разработанными другими программистами, даже если последние используют иные языки программирования. Очевидно, что это существенный прогресс по сравнению с ситуацией до прихода .NET, когда разработка приложения на нескольких языках хотя и была возможна, но сталкивалась с общеизвестными сложностями, поскольку разрабатываемый код должен был удовлетворять весьма жестким требованиям. Многие разработчики, писавшие COM-компоненты на C++ для использования в Visual Basic, услышав слово "вариант", покрываются холодным потом.

В последующих главах мы подробно обсудим сборки, язык MSIL, систему типов CTS, обработку исключений и т.д. Все эти средства обеспечивают поведение управляемого кода и платформы .NET Framework. Завершая эту главу, мы позволим себе небольшое развлечение и разберем пример, подтверждающий сказанное выше. Мы создадим приложение .NET, реализовав его на трех языках, и покажем, каким образом средства отладки Visual Studio .NET позволяют легко и свободно переходить от одного языка к другому.

Пример приложения на нескольких языках

Ниже рассмотрен пример, демонстрирующий многоязыковое наследование. Соответствующая программа имеет следующие основные характеристики:

- Консольное приложение C# инициализирует класс Visual Basic.NET и затем вызывает его метод.
- Класс Visual Basic.NET наследует от класса C++.

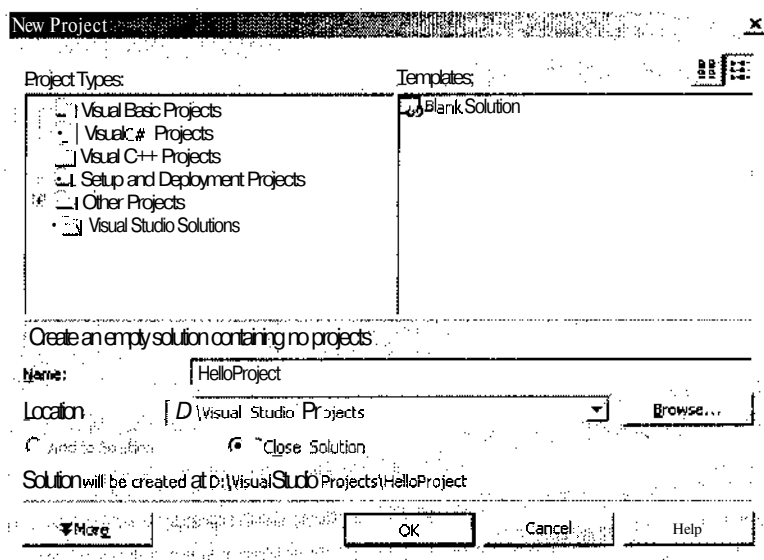
Для подтверждения того, что приложение работает согласно этому плану, мы должны получить на выходе следующую информацию:



```
D:\Visual Studio Projects\HelloProject\HelloSharp\bin\Debug>
Hello from Managed C++!
Hello from VB.NET!
Hello from C#!
Press any key to continue
```

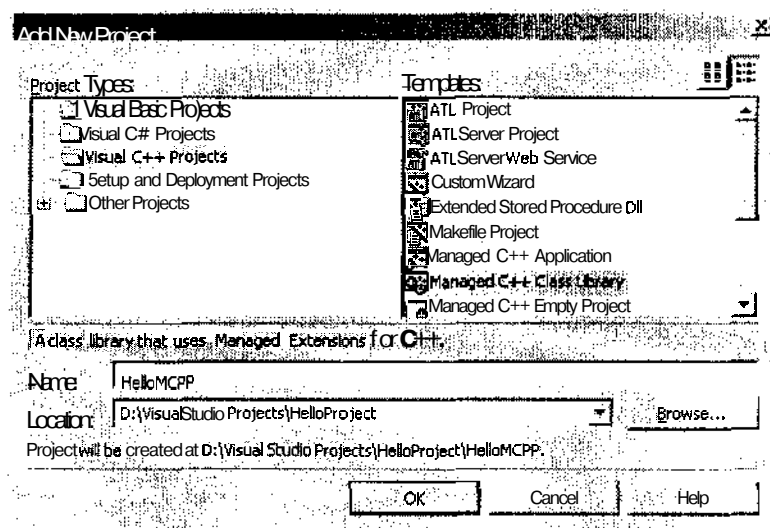
Этот пример может показаться искусственным, тем не менее он в полной мере подтверждает тот факт, что выбор языка для написания кода в среде .NET не имеет значения. Мы специально сделали каждый из элементов приложения как можно более простым, однако принципы остаются неизменными, независимо от сложности реального приложения. Важно лишь, чтобы при создании сборки учитывались общие требования к языкам для всех открытых переменных, методов или свойств.

Вначале обратимся к Visual Studio .NET и создадим пустой шаблон (Blank Solution) с именем HelloProject. Таким образом, мы с самого начала разрабатываем приложение, нейтральное по отношению к языку:



Написание класса на C++

Вначале добавим в решение HelloProject библиотеку управляемых классов C++ (Managed C++ Class Library) с именем HelloMCPPL (пользователь может выбрать любое другое название):



Не будем пока обращать внимание на некоторые новые элементы. Отметим лишь, что мастер создал класс `Class1` в пространстве имен `HelloMCP`. Единственное изменение, которое мы внесем в первоначальные установки, заключается в присвоении классу нового имени `HelloMCP`, совпадающего с именем пространства имен. Поскольку реализация приложения весьма проста, поместим наш код прямо в заголовочный файл `HelloMCP.h`:

```
// Файл HelloMCP.h

#pragma once

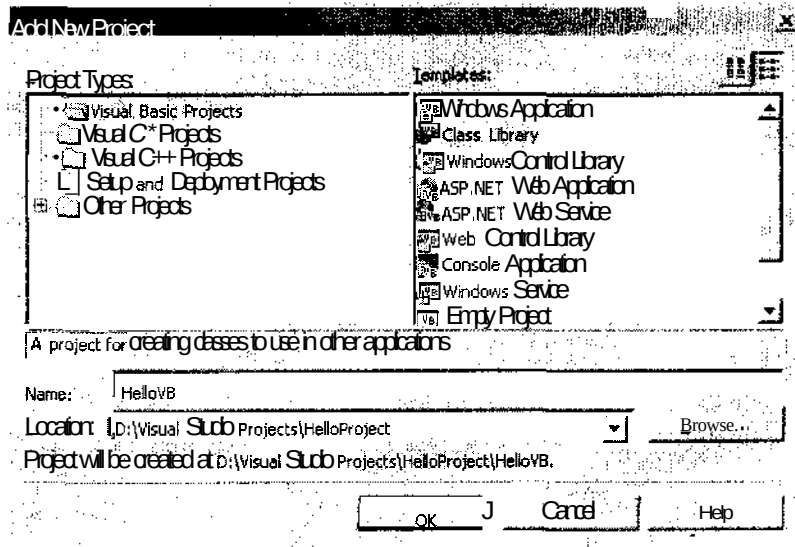
using namespace System;

namespace HelloMCP
{
    public __gc class HelloMCP
    {
    public:
        virtual void Hello()
        {
            Console.WriteLine(S"Hello from Managed C++!");
        }
    };
}
```

Здесь метод `HelloMCP::Hello()` использует функцию `System::Console::WriteLine()` из библиотеки классов `.NET Framework` для вывода в окно консоли приветствия от управляемого кода C++. Это все, что сейчас нужно, поэтому, откомпилировав проект, перейдем к следующему этапу.

Написание класса на Visual Basic .NET

Второй этап заключается в написании кода на языке Visual Basic .NET, и можно включить этот язык в дело, создав еще один новый проект. На этот раз сгенерируем с помощью Visual Studio .NET проект Visual Basic Class Library (Библиотека классов Visual Basic) с именем `HelloVB`:



Даже читателям, не знакомым с Visual Basic, будет нетрудно понять простой код, добавленный в проект. Этот код определяет класс HelloVB, наследующий от управляемого класса C++ HelloMCP. Класс HelloVB подменяет виртуальный метод Hello() своей собственной версией, которая вначале вызывает версию указанного метода из родительского класса, а затем выводит приветствие **Hello from VB.NET!** в окно консоли.

```

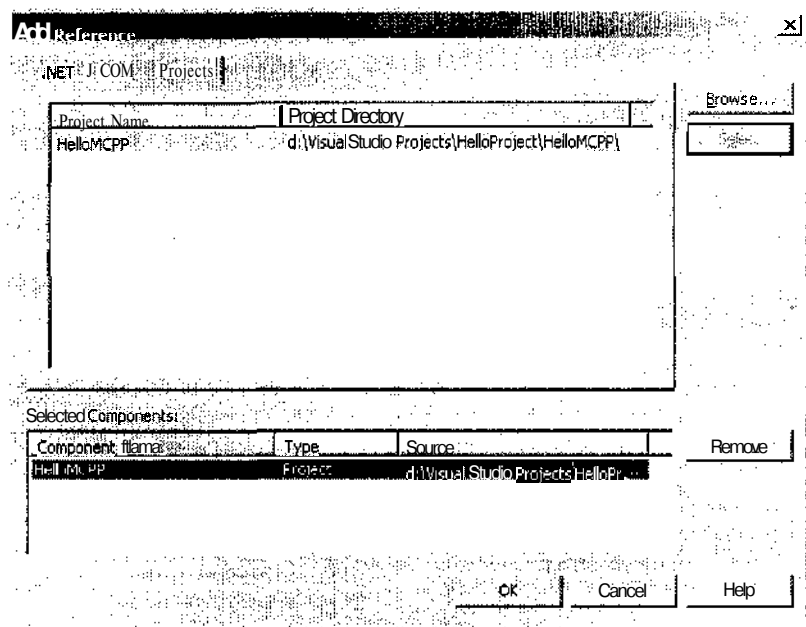
B Public Class HelloVB
    Inherits HelloMCP>HelloMCP

    Public Overrides Sub Hello()
        MyBase.Hello()
        System.Console.WriteLine("Hello from VB.NET!")
    End Sub
End Class
    
```

Подчеркивание волнистой синей линией на снимке с экрана означает, что Visual Basic не распознает некоторые имена, но это и не должно удивлять. Если поместить курсор над строкой Inherits, всплывающая подсказка точно определит проблему:

Type 'HelloMCP>HelloMCP' is not defined (Тип HelloMCP>HelloMCP не определен).

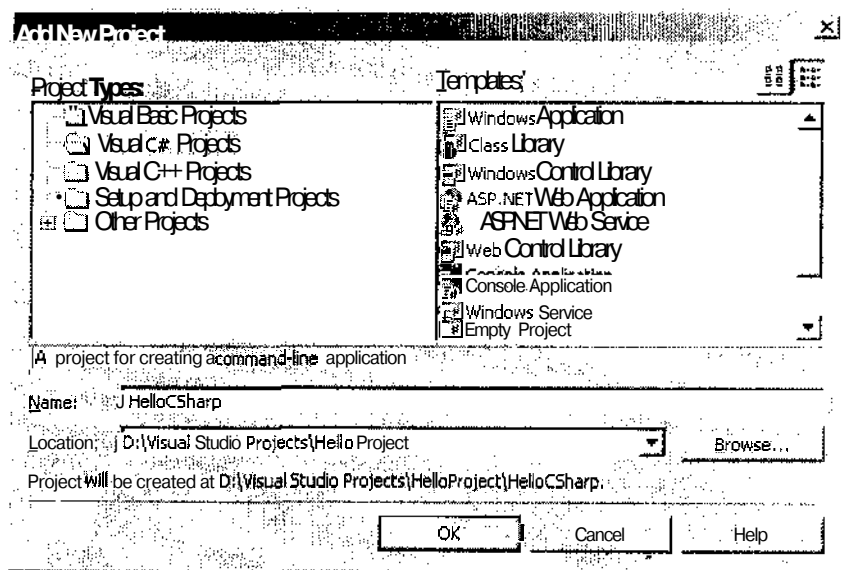
Необходимо сообщить язык) Visual Basic, где находится сборка, определяющая данный тип. Это совсем простая задача. Выберем команду меню **Project | Add Reference...** (Проект | Добавить ссылку...), затем щелкнем на вкладке **Project** появившегося диалогового окна, после чего найдем путь к проекту HelloMCP, как показано на рисунке:



По завершении этой операции раздражающие синие линии исчезнут. Скомпоновав эту часть проекта, перейдем к третьему этапу.

Написание класса на C#

Приступим к написанию кода на C#. На этот раз мы создадим консольное приложение Visual C# (Visual C# Console Application), которое назовем HelloCSharp:



Разобраться в коде C#, приведенном ниже, довольно легко, поскольку он во многом сходен с кодом C++. В нашем примере определяется класс HelloCSharp, имеющий статический метод Main(), который является точкой входа в приложение. Метод Main() создает новый объект класса HelloVB и вызывает его метод Hello(), прежде чем вывести свой собственный текст в окне консоли: Hello from C#!

```
using System;
```

```
namespace HelloCSharp
```

```
{
```

```
    /// <summary>
```

```
    /// Описание класса HelloCSharp.
```

```
    /// </summary>
```

```
    class HelloCSharp
```

```
    {
```

```
        /// <summary>
```

```
        /// Глазная точка входа в приложение.
```

```
        /// </summary>
```

```
        [STAThread]
```

```
        static void Main(string[] args)
```

```
        {
```

```
            HelloVB.HelloVB hello = new HelloVB.HelloVB();
```

```
            hello.HelloO;
```

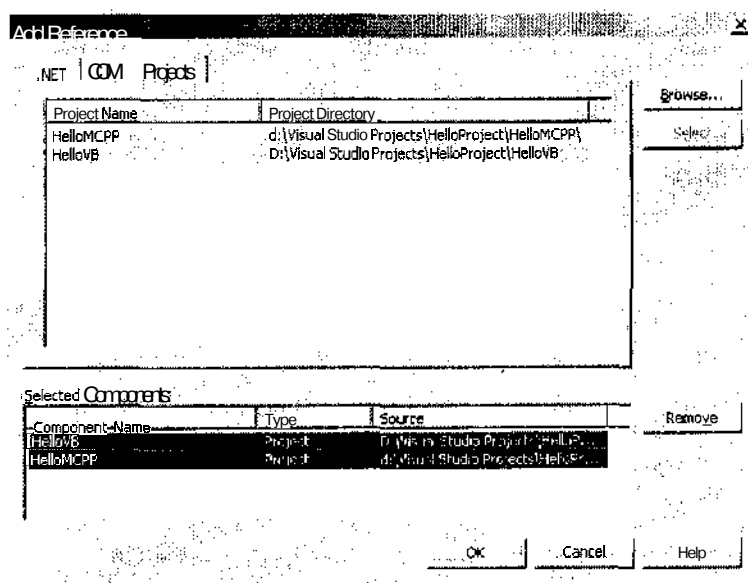
```
            Console.WriteLine("Hello from C#!");
```

```
        }
```

```
    }
```

```
}
```

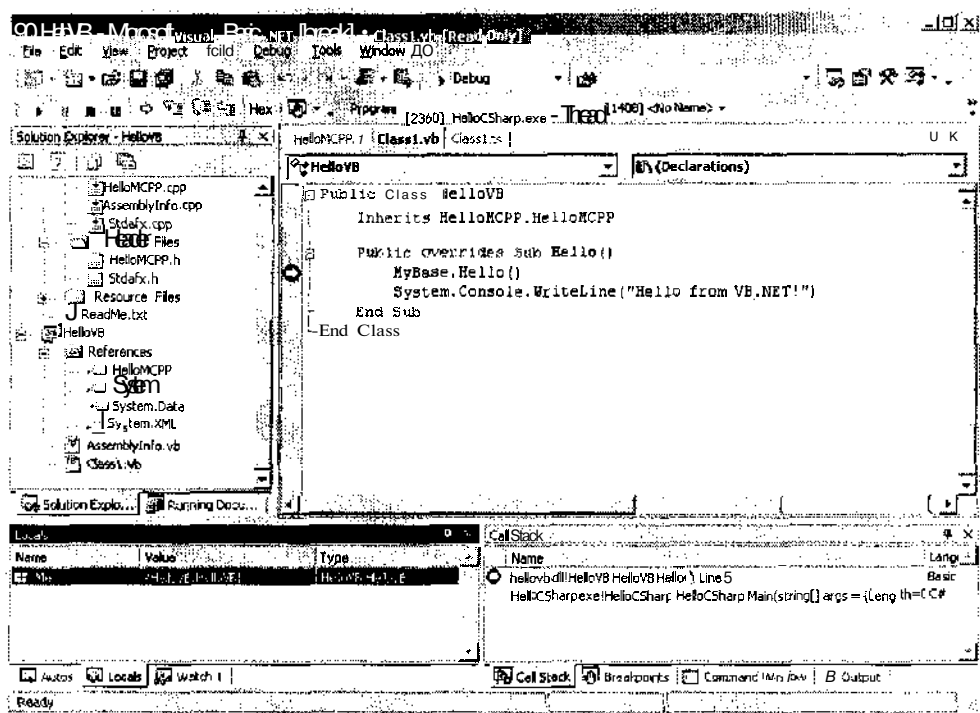
Однако, как и предыдущий проект, написанный на Visual Basic, данный проект вначале не будет откомпилирован, поскольку приведенному выше коду не известно, что обозначает идентификатор HelloVB.HelloVB. На этот раз следует добавить в проект C# ссылку на оба проекта: HelloVB и HelloMCP.



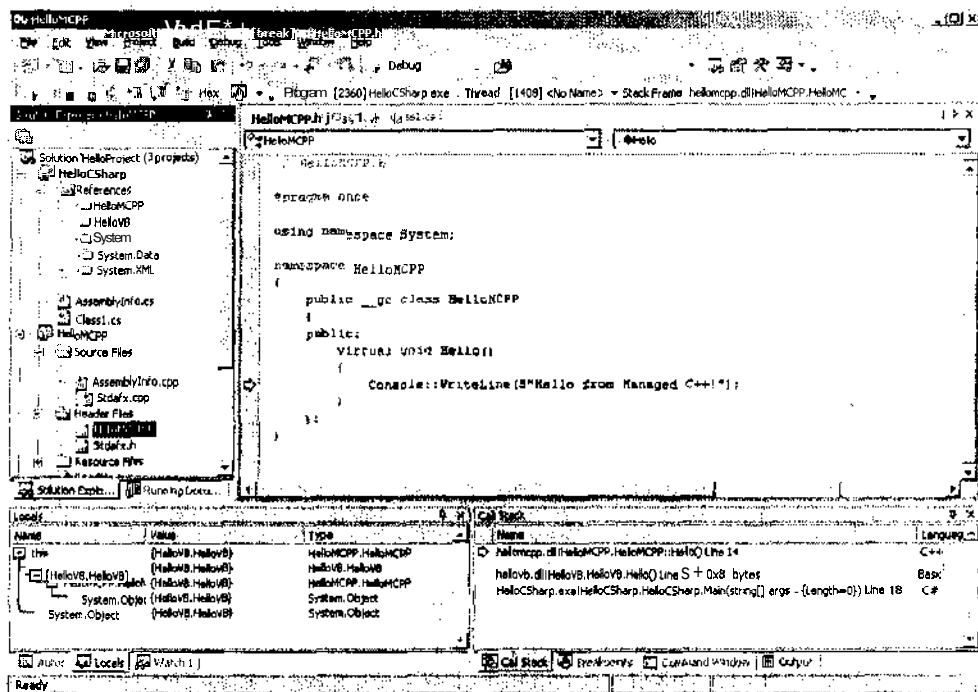
Наконец, нужно сделать проект C# "стартовым" проектом нашего решения. Для этого следует щелкнуть правой кнопкой мыши на имени проекта в Solution Explorer и выбрать пункт меню **Set as Startup Project** (Установить как первый загрузочный проект). Затем, откомпилировав и скомпоновав все решение, запустим его. При том количестве усилий, которое было затрачено, нельзя сказать, что реализация данного примера — это такое уж трудное занятие. Однако пусть кто-нибудь попробует сделать то же самое с COM!

Отладка

Когда мы создаем программы для .NET Framework, отладка приложений, написанных на нескольких языках, столь же проста, как и их разработка. Среда Visual Studio .NET безразлична к тому, какой язык является "ответственным" за отлаживаемый код, коль скоро это язык совместим с .NET. Зададим точку останова и будем перемещаться по приложению:



Вначале поместим точку останова в файл `Class1.vb` там, где вызывается метод `MyBase.Hello()`. При запуске приложения в режиме отладки выполнение программы приостанавливается в том месте, где это указано. Обратите внимание на окно Call Stack (Стек вызовов), которое правильно показывает вложение метода в метод `Main()` консольного приложения C#. Уже из этого видно, что Visual Studio может работать с точками останова, находящимися за пределами исходного языка приложения.



Далее откроем метод `MyBase.Hello()`, для чего следует нажать клавишу *F11* или выбрать команду `Step Into` в меню `Debug` (Отладка). И действительно, на экране мы увидим наш первоначальный класс `HelloMCP`, написанный на Visual C++, причем окно `Call Stack` вновь точно отобразит происходящее.

Даже из этого простого примера ясно, что выбор языка разработки не имеет значения для инструментов отладки Visual Studio .NET. Если читатель еще поэкспериментирует с отладчиком, то убедится, что вид и функционирование средств отладки останутся неизменными. Многоязыковая разработка наконец стала реальностью.

ГЛАВА 2



Введение в управляемый код C++

В предыдущей главе мы уже познакомились с единой средой исполнения (CLR) и рассмотрели пример, демонстрирующий совместимость различных языков. Мы видели, что применение управляемого кода позволяет, в частности, использовать такие средства, как сборщик мусора, библиотека классов .Net Framework, а также механизмы обеспечения безопасности кода.

Обратимся теперь к вопросу, каким образом можно использовать C++ в управляемом окружении .NET. Поскольку язык C++ существует уже длительное время и разрабатывался еще тогда, когда никто и не помышлял о создании .NET, ясно, что этот язык не ориентирован на работу в среде .NET. Поэтому Microsoft включила в C++ некоторые дополнения в форме нескольких новых ключевых слов, называемых **управляемыми расширениями** {Managed Extensions} языка Visual C++. Эти ключевые слова позволяют использовать возможности .NET Framework в существующих и в будущих проектах C++. Управляемые расширения следует применять в тех случаях, когда необходимо:

- Обеспечить миграцию существующего кода C++ в управляемую среду
- Использовать классы .NET Framework в коде C++
- Предоставить .NET-совместимым языкам возможность использования существующего кода C++

Вначале мы обсудим основные факторы, которые необходимо учитывать при написании на C++ кода, предназначенного для работы в среде CLR. В частности, следует подумать над тем, какие *встроенные* типы данных могут быть использованы в этом случае, как будут создаваться пользовательские типы и как нужно генерировать необходимые метаданные. Однако основное внимание мы уделим подробному рассмотрению управляемых расширений C++. Изучив эту главу, читатель будет иметь достаточно информации по данному вопросу и сможет приступить к разработке управляемых приложений C++.

Интероперабельность языков

Одна из основных целей, ради которых создавалась .NET Framework, заключается в обеспечении бесшовной интеграции компонентов, созданных с применением различных языков. Платформа .NET позволяет компонентам, написанным на разных языках, вызывать методы друг друга, обмениваться данными и даже наследовать друг от друга. Это достигается за счет использования двух средств:

- О Метаданные

- Система общих типов

Мы покажем, как применение метаданных и системы общих типов (CTS) обеспечивает взаимодействие языков и в чем заключаются особенности написания кода C++, который должен работать в среде .NET. На данном этапе мы не будем подробно рассматривать управляемые расширения, однако продемонстрируем, почему они необходимы для создания управляемых, интероперабельных компонентов.

Метаданные

Читатели, имеющие опыт COM-программирования, должны быть знакомы с файлами IDL и теми библиотеками типов, которые эти файлы определяют. Метаданные реализуют аналогичную концепцию. Они полностью описывают тип в форме, нейтральной по отношению к языку. Любой язык, совместимый с CLR, может использовать такой тип. Это значит, что класс, созданный с применением управляемого кода C++, может быть использован в качестве базового класса для типа VB.NET или C# и наоборот. Метаданные обеспечивают не только взаимодействие компонентов, написанных на разных языках, но и многоязыковое наследование.

В отличие от реализации библиотеки типов в COM метаданные встраиваются в переносимые исполняемые файлы (PE) вместе с кодом, реализующим типы, описанные этими метаданными. Благодаря применению платформы .NET отпадает необходимость разделения библиотек типов и файлов реализации, что уменьшает число проблем, с которыми сталкиваются разработчики. Поскольку метаданные связаны с кодом, который от них зависит, исключена возможность нарушения синхронизации между метаданными и кодом.

Третьим аспектом метаданных является их расширяемость через атрибуты. Разработчик может создавать свои собственные атрибуты, позволяющие генерировать пользовательские метаданные, предоставляющие дополнительную информацию о коде (подробнее см. в главе 4).

Основной предмет обсуждения в настоящей главе — это набор ключевых слов, известных как управляемые расширения C++. Борцы за чистоту C++ могут посчитать эти расширения ненужными и неизящными. Однако такие расширения абсолютно необходимы. Они позволяют указать класс или структуру, которые должны работать в среде CLR, а также сделать класс или структуру доступными для кода, написанного на других .NET-совместимых языках.

Благодаря применению ключевых слов, относящихся к управляемым расширениям C++, генерируются метаданные, полностью описывающие код в понятной для CLR форме. Это позволяет коду взаимодействовать с другим кодом, написанным на любом другом .NET-совместимом языке.

Система общих типов

Все языки, ориентированные на работу в среде CLR, должны использовать типы, соответствующие системе общих типов {Common Type System, CTS), что необходимо для обеспечения совместимости языков между собой. Такая совместно используемая система типов позволяет языкам легко обмениваться данными друг с другом. Если требуется передавать величины типа Double либо String (или даже более сложного типа) между управляемым кодом C++ и кодом Visual Basic .NET, можно делать это совершенно свободно, без каких бы то ни было издержек, связанных с маршалингом между типами, написанными на разных языках.

Ключевые слова, которые определяются как управляемые расширения C++, позволяют разработчикам создавать собственные типы, соответствующие системе CTS, и потому способные взаимодействовать с другими языками, работающими на платформе .NET.

Библиотека классов .NET Framework

При написании управляемого кода нужно использовать набор классов, определяемых библиотекой классов .NET Framework. Мы рассмотрим эту библиотеку более подробно в главе 5, когда речь пойдет об использовании наиболее распространенных служебных классов. Однако, прежде чем приступить к созданию управляемого кода, полезно обсудить некоторые базовые типы данных. К таковым относится, например, вездесущий строковый тип, который приходится использовать, как только мы начинаем писать любой управляемый код.

В приведенной ниже таблице перечислены базовые типы данных .NET Framework, соответствующие типам-примитивам C++, с которыми читатель, несомненно, знаком:

Тип-примитив C++	Тип, принадлежащий к библиотеке классов .NET
Char	SByte
Signed char	SByte
short	Int16
int	Int32
long	Int32
__int64	Int64
Unsigned char	Byte
Unsigned short	UInt16
Unsigned int	UInt32
Unsigned long	UInt32
Unsigned __int64	UInt64
float	Single
double	Double
void	Void

boolbool

Хорошая новость для разработчиков заключается в том, что при написании управляемого кода C++ можно просто использовать версии примитивов языка C++. Последние совместимы с CTS, так как эти версии и соответствующие им типы .NET Framework являются взаимозаменяемыми. Однако следует проявлять осторожность в случае перегрузки функции с двумя различными типами C++, которые соответствуют одному и тому же типу .NET. Отметим, что все эти типы определены в пространстве имен System.

Строки

Обработка строк встречается в большинстве приложений. Строки передаются функциям, возвращаются функциями, выводятся в файлы и потоки и т.д. Исторически сложилось так, что работа со строками на платформе Windows усложняла жизнь программистам из-за того огромного количества строковых типов, которое появилось за долгие годы. Кроме стандартных типов C++, имеются LPCTSTR, CString и CComBSTR, но и этим список не ограничивается. Однако в управляемом коде существует только один строковый тип: System::String.

В данном разделе мы используем строковые константы, однако мы должны представлять, как выполняется взаимное преобразование System::String и неуправляемых типов данных, с которыми мы знакомы. В главе 7 более подробно будет рассмотрен маршalling между типом String и неуправляемыми строковыми типами.

Преобразование строковых констант C++ в System::String*

В C++ имеется две формы строковых констант:

- Строки с однобайтовыми символами, определяемые с помощью двойных кавычек, которые служат ограничителями
- Строки с двухбайтовыми символами, определяемые аналогичным образом, с добавлением в начало записи символа L

В управляемом коде C++ строка String* автоматически принимает оба вида строковых констант C++:

```
String s1 = "Hello World!";
String* s2 = L"Hello World!";

Console.WriteLine("s1 = {0}", s1);
Console.WriteLine("s2 = {0}", s2);

if(s1 == s2)
    Console.WriteLine("s1 and s2 point to a single String instance");
else
    Console.WriteLine("s1 and s2 point to different String instances");
```

В результате выполнения этого кода будет выдана следующая информация:

```
s1 = Hello World!
s2 = Hello World!
s1 and s2 point to different String instances
(s1 и s2 указывают на разные экземпляры строк)
```

Представленный здесь код взят из управляемого проекта C++ под названием Strings, который можно загрузить с Web-сайта издательства Wrox, как и весь остальной код, обсуждаемый в настоящей главе.

Строковые константы типа S

В управляемый код C++ включена третья форма строковых констант, которые называются строковыми литералами типа S (S string literals). Использование данной формы в управляемом коде C++ является более предпочтительным и эффективным, чем применение упомянутых выше строковых литералов с одно- и двухбайтовыми символами. Причина в том, что все экземпляры типа String, создаваемые с помощью идентичных строковых литералов вида S, будут указывать на один и тот же экземпляр. Более того, все экземпляры типа String являются строками, состоящими из двухбайтовых символов Unicode. Поэтому в случае их инициализации с применением строковой константы, использующей однобайтовое представление символов, конструктор типа String будет выполнять преобразование в формат Unicode, а затем копировать строку в новую область памяти и поддерживать ссылку на эту область. В противоположность этому при инициализации строки с помощью строкового литерала типа S конструктор не вызывается и потому не происходит преобразования или копирования. Ниже приведен пример, иллюстрирующий использование строковых литералов типа S:

```
String* s3 = S"Hello World!";
String* s4 = S"Hello World!";

Console.WriteLine($"s3 = {0}", s3);
Console.WriteLine($"s4 = {0}", s4);

if(s3 == s4)
    Console.WriteLine($"s3 and s4 point to a single String instance");
else
    Console.WriteLine($"s3 and s4 point to different String instances");
```

При выполнении этого кода на экран будет выдана такая информация:

```
s3 = Hello World!
s4 = Hello World!
s3 and s4 point to a single String instance
(s3 и s4 указывают на один и тот же экземпляр типа String)
```

Сравнение строк

Класс String подменяет метод Object.Equals() для того, чтобы сравнивались значения строк, а не проверялось, являются ли два экземпляра одинаковыми:

```
if(String.Equals(s1,s2))
    Console.WriteLine($"Using Equals(), s1 and s2 string values identical");
else
    Console.WriteLine($"Using EqualsO, s1 and s2 string values different");
```

При этом мы получим следующий результат:

```
Using Equals(), s1 and s2 string values identical
(Использование метода Equals: значения строк s1 и s2 одинаковы)
```

Аналогично, если нужно сравнить значения для создаваемых типов, следует подменить метод Object.Equals().

Другой способ сравнения строк заключается в использовании интерфейса IComparable, который реализуется типом String. Этот интерфейс имеет единственный метод CompareTo():

```
if(0 == s3.CompareTo(s4))
    Console.WriteLine($"Using CompareTo(), s3 and s4 string values identical");
else
    Console.WriteLine($"Using CompareTo(), s3 and s4 string values different");
```

В результате выполнения данного кода получим:

Using CompareTo(), s3 and s4 string values identical
(Использование метода CompareTo(): значения строк s3 и s4 одинаковы)

IComparable представляет собой стандартный интерфейс, который должен быть реализован типами, если необходимо, чтобы они поддерживали упорядочение. Метод CompareTo() возвращает соответственно положительное или отрицательное целое число, в зависимости от того, является ли экземпляр больше или меньше величины, передаваемой для сравнения. Если две величины одинаковы, возвращается значение 0.

Тип String также реализует интерфейсы ICloneable, IEnumerable и IConvertible. Мы скоро обсудим интерфейс ICloneable, но для получения детальной информации обо всех этих интерфейсах необходимо обратиться к документации.

Чаще всего со строками выполняются такие стандартные операции, как слияние, копирование и клонирование. В последующих разделах описываются методы, поддерживающие эти операции, а также приводятся примеры использования указанных методов. Однако следует иметь в виду, что мы продемонстрируем лишь образец тех возможностей, которыми располагает тип String для поддержки обычных операций со строками. К последним относятся: удаление определенных символов в начале или в конце строки (trimming), заполнение начальной или конечной части строки нужными символами (padding), разбиение строк (splitting), их форматирование (formatting) и т.д.

Слияние строк

Тип String поддерживает слияние строк с помощью метода Concat():

```
String* s5 = S"Hello";
s5 = String::Concat(s5, S" World", S"! ");
Console::WriteLine(S"s5 = {0}", s5);
```

Данный метод является статическим и имеет несколько перегруженных версий, позволяющих выполнять слияние экземпляров типа String, а также возвращать строковое представление экземпляров класса System::Object.

Копирование строк

Экземпляры типа String можно копировать с помощью другого статического метода COEYO. Данный метод принимает копируемую строку в качестве параметра и возвращает новый экземпляр строки с тем же значением:

```
Console::WriteLine(S"Copying...");
String* s6 = String::Copy(s5);
Console::WriteLine(S"s6 = {0}", s6);

if(s5 == s6)
    Console::WriteLine(S"s5 and s6 point to a single String instance");
else
    Console::WriteLine(S"s5 and s6 point to different String instances");

// Использование статического метода String::Compare()...
if(0 == String::Compare(s5, s6))
    Console::WriteLine(S"Using Compare(), s5 and s6 string values identical");
else
    Console::WriteLine(S"Using Compare(), s5 and s6 string values different");
```

Результат выполнения этого кода будет следующим:

```
s5 = Hello World!
Copying...
s6 = Hello World!
s5 and s6 point to different String instances
<s5 и s6 указывают на различные экземпляры типа String>
Using CompareO/ s5 and s6 string values identical
(Использование метода Compare(): значения строк s5 и s6 одинаковы)
```

Отметим, что, хотя экземпляр, возвращаемый методом Copy(), отличается от копируемого экземпляра, значения обоих экземпляров равны.

Клонирование строк

Тип System::String реализует интерфейс ICloneable. Данный интерфейс имеет единственный метод Clone(), который не принимает параметров и возвращает величину типа Object-, представляющую собой ссылку на клонированный объект:

```
Console.WriteLine("Cloning...");
String* s7 = __try_cast<String*>(s6->Clone());
Console.WriteLine("s7 = {0}", s7);

if(s7 != s6)
    Console.WriteLine("s6 and s7 point to a single String instance");
else
    Console.WriteLine("s6 and s7 point to different String instances");

// Использование статического метода String::Compare()...
if(0 == String::Compare(s7, s6))
    Console.WriteLine("Using CompareO, s6 and s7 string values identical");
else
    Console.WriteLine("Using Compare(), s6 and s7 string values different");
```

Выполнение данного кода приведет к следующему результату:

```
Cloning...
s7 = Hello World!
s6 and s7 point to a single String instance
(s6 и s7 указывают на один экземпляр типа String)
Using Compare(), s6 and s7 string values identical
(Использование метода Using Compare(), значения строк s6 и s7 одинаковы)
```

В отличие от предыдущего примера, где демонстрировался метод Copy(), при использовании метода Clone() возвращается тот же самый экземпляр.

Нам приходится преобразовывать ссылку на клон в строку String, поскольку метод Clone() возвращает тип Object'. Оператор __try_cast<>() представляет собой управляемую версию известного оператора C++ dynamic_cast<>(). Мы подробнее рассмотрим оператор __try_cast<> () ниже в данной главе.*

Управляемая среда

В обычном языке C++ имеется две области памяти, где размещаются объекты: стек времени выполнения и куча. Локальные переменные хранятся в стеке, тогда как экземпляры типов, создаваемые с помощью оператора new языка C++, помещаются в кучу. Причем программист должен сам динамически удалять экземпляры, которым выделена память, когда те больше не нужны программе. В случае СОМ-программирования ситуация оказывается еще более сложной из-за необходимости реализовывать соответствующий подсчет ссылок. Подобное управление

временем жизни объекта может создать различные проблемы для разработчика, такие как утечки памяти и нарушение доступа.

Ситуация коренным образом меняется при переходе к платформе .NET. Среда CLR реализует сборку мусора — механизм, который автоматически отслеживает ссылки на объект со стороны приложения. Если приложение содержит ссылки на объект, последний будет существовать. Когда же приложение перестает ссылаться на данный объект, он будет уничтожен, хотя, возможно, и не сразу. Среда исполнения реализует такой механизм благодаря наличию дополнительной области памяти, предназначенной для размещения объектов, которая называется **управляемой кучей (managed heap)**. Управляемая куча похожа на кучу C++, поскольку также представляет собой большую область памяти, где создаются экземпляры типов, однако в отличие от последней она управляется сборщиком мусора. В управляемой куче могут существовать лишь **управляемые типы (managed types)**.

Управляемый код и управляемые данные

Важно различать **управляемые данные (managed data)** и **управляемый код (managed code)**. Управляемый код может быть написан на нескольких языках, таких как Visual Basic .NET, C# или C++. Этот код компилируется в промежуточный язык IL (Intermediate Language) и выполняется в среде CLR. Среда CLR использует метаданные для реализации важных служб, таких как обработка исключений, обеспечение безопасности, автоматическое управление временем жизни и т.д. Управляемый код C++ является аннотированным подмножеством C++ и компилируется для исполнения средой CLR с целью иметь доступ к вышеперечисленным средствам. Неуправляемый код исполняется процессором x86 и является традиционным кодом, который встречается в приложениях, написанных до появления .NET, а также в библиотеках ОС Win32. Неуправляемый код включает собственно код C/C++, а также компоненты COM. В этот код не заложено представление о среде исполнения .NET, и для его выполнения среда CLR не нужна. Неуправляемый код не может непосредственно использовать сервисы, предоставляемые платформой .NET.

Управляемые данные контролируются сборщиком мусора. В управляемой среде вся динамическая память, выделяемая в управляемой куче, есть не что иное, как управляемые данные. Все объекты обязаны поддерживать самоописание. Сборщик мусора должен проверить метаданные в период выполнения и установить точный тип объекта, а также убедиться в том, что никакая из операций, выполняемых с данными, не нарушает доступ и не разрушает их. Размещение управляемых данных автоматически обеспечивается средой CLR. Однако оно может быть изменено с помощью метаданных (для этого, например, служит атрибут `StructLayout`; см. главу 4). Время жизни управляемых данных также регулируется средой CLR.

Выделение памяти в управляемой куче происходит быстрее, чем в куче C++, и почти так же быстро, как в стеке периода выполнения. Это объясняется структурой управляемой кучи и размещением объектов в ней. Управляемая куча фактически представляет собой непрерывную область памяти с указателем базы и указателем следующего свободного адреса, который будет использован при выделении памяти новому объекту. Всякий раз, когда объекту выделяется память, указатель соответственно получает приращение, так что он вновь будет обозначать следующий свободный адрес. В подобном случае в неуправляемой куче должен выполняться поиск свободной области памяти подходящего размера. Не стоит и говорить, что выделение памяти будет успешным лишь в том случае если в управляемой куче имеется достаточно свободной памяти. Механизм сборки мусора используется для сжатия рабочей области памяти и позволяет максимизировать доступную

память. Данный механизм должен работать незаметно для программиста, однако имеет смысл более подробно обсудить его, что позволит лучше понять принцип работы управляемого окружения.

Сборка мусора

Когда в управляемой куче становится недостаточно места для выделения требуемого объема памяти новому объекту, активизируется механизм сборки мусора. Каждая ссылка со стороны приложения на экземпляр типа в управляемой куче считается корнем. Среда исполнения находит все корпи для каждого типа. При запуске механизма сборки мусора среда исполнения строит граф всех размещенных в куче объектов, достижимых через корневые ссылки приложения (под *достижимым* мы понимаем объект, на который где-либо в приложении имеется ссылка). Если сборщик мусора находит недостижимый объект, этот объект становится кандидатом на удаление. Когда сборщик мусора обнаруживает достаточное количество недостижимых объектов, каждому из достижимых объектов в процессе их уплотнения заново выделяется память в управляемой куче. По завершении процедуры уплотнения обновляются корни приложения, чтобы они адекватно отражали произведенное перемещение объектов в куче. Отсюда следует, что нельзя использовать обычные указатели C++ для ссылок на данные в управляемой куче, поскольку адреса памяти изменяются. Вместо этого нужно работать с **управляемыми указателями (managed pointers)**, адреса которых обновляются автоматически (подробнее об управляемых указателях см. ниже).

Алгоритм сборки мусора использует так называемый метод поколений, основанный на распределении всех объектов по отдельным областям управляемой кучи, исходя из того, какое количество операций сборки мусора пережил каждый объект. Всякий раз, когда объект сохраняется в куче после очередной сборки мусора, он переходит в следующее поколение. Алгоритм сборки мусора использует три поколения, которым присваиваются номера 0, 1 и 2, причем вновь размещенные в куче объекты относятся к поколению 0. Все уцелевшие объекты из поколения 0 после сборки мусора будут относиться к поколению 1. Самые долгоживущие объекты в конечном итоге переходят в поколение 2 и существуют там до тех пор, пока не перестанут быть достижимыми через корневые ссылки приложения. Распределение объектов по поколениям повышает производительность сборщика мусора, поскольку в процессе сборки может сжиматься только область одного поколения, а не вся куча.

Ссылочные типы и типы-значения

ЕСЛИ МЫ ХОТИМ создать собственный тип, которым будет автоматически управлять сборщик мусора, то для этого необходимо написать код C++, предназначенный для исполнения средой CLR.

Ключевые слова, относящиеся к управляемым расширениям C++, позволяют создавать управляемые типы, для которых может выделяться память в управляемой куче.

Все типы .NET, участвующие в процедуре сборки мусора, существуют в управляемой куче независимо от того, созданы ли они пользователем или принадлежат к библиотеке классов .NET Framework. Такие типы называются **ссылочными типами (reference types)**, и к ним получают доступ через управляемые указатели.

ЕСЛИ ТИП данных занимает мало памяти и является короткоживущим, нет смысла выделять для него место в управляемой куче, поскольку тогда операция сборки мусора будет слишком дорогостоящей. В этом случае данные могут быть помещены в стек периода выполнения. Такой тип называется **типом-значением (value type)** или **размерным типом**.

В категорию типов-значений попадают классы .NET Framework, представляющие собой примитивы C++. С другой стороны, тип String является значительно более сложным и определяется как ссылочный тип. Существуют специальные ключевые слова, которые позволяют указать, будет ли код определять тип, участвующий в процедуре сборки мусора, или же тип-значение (см. ниже).

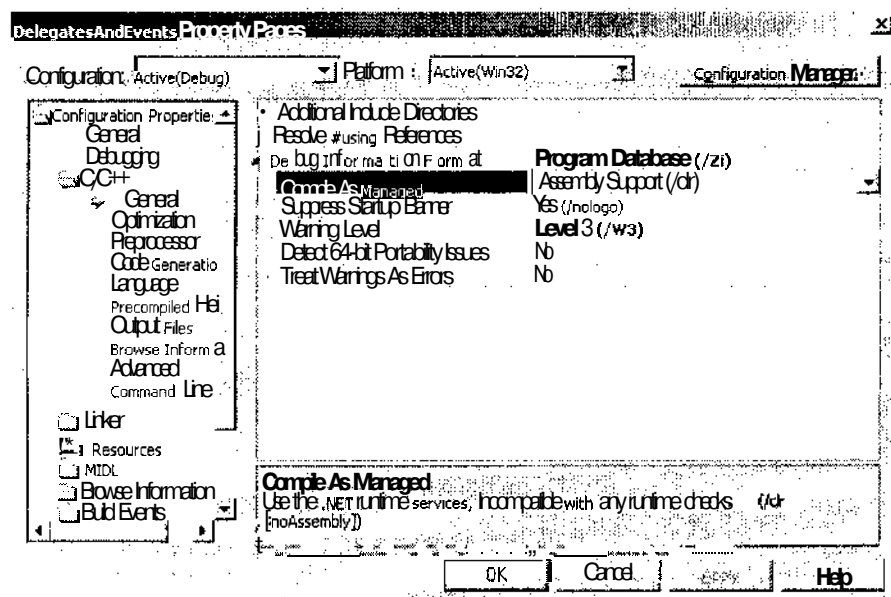
Создание управляемого кода

ДЛЯ ТОГО чтобы создаваемый код работал в среде CLR и получал доступ к классам .NET Framework, необходимо использовать некоторые новые опции компилятора, которые сами составляют подмножество управляемых расширений.

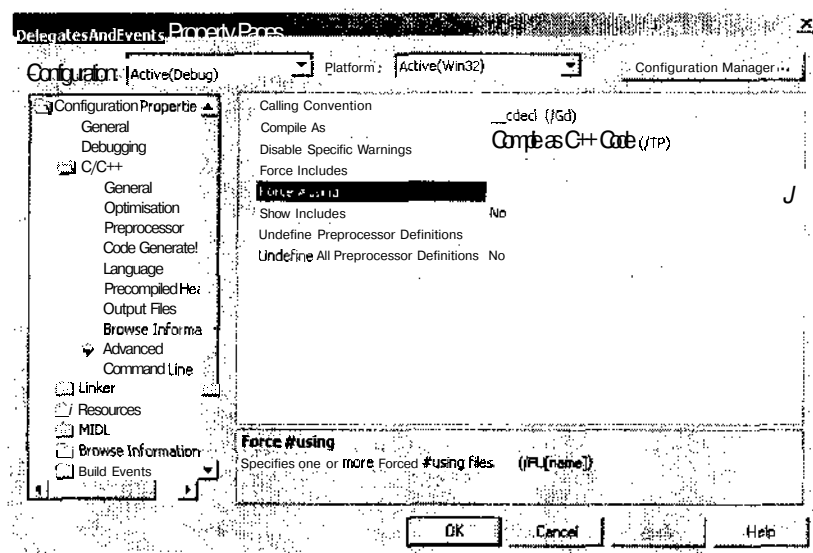
Опции компилятора

Для указания того, что код C++ должен компилироваться в язык **MSIL**, а не в машинный исполняемый код, следует добавить опцию `/clr` в список используемых режимов компилятора. В результате код будет компилироваться в сборку (подробнее о сборках см. в следующей главе). Сборки можно рассматривать как файлы DLL, которые вместе с откомпилированным кодом содержат метаданные. Сборка предполагает наличие в период выполнения среды CLR, однако это автоматически не означает, что сборка будет содержать управляемые данные. Управляемые типы должны указываться явно.

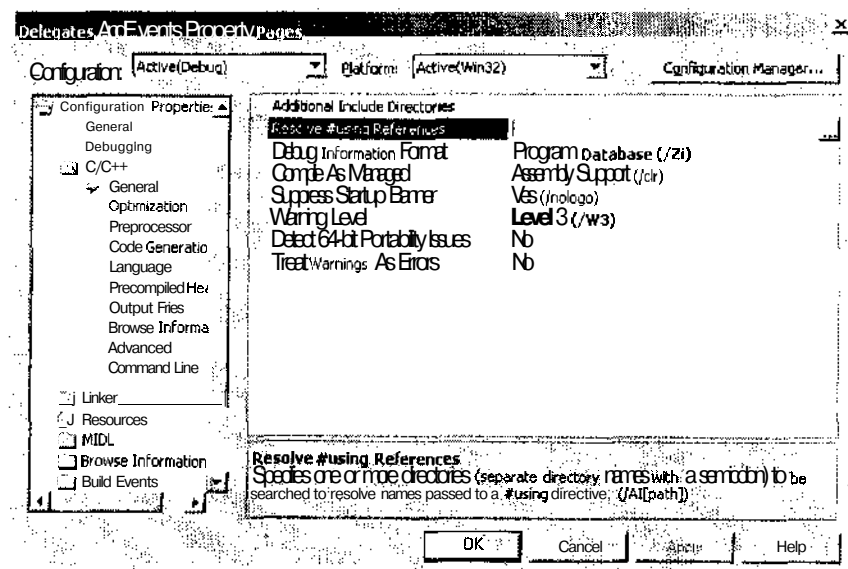
Приведенный ниже снимок экрана демонстрирует выбор режима компилятора `/clr` в диалоговом окне **Project | Properties**. Если для создания проекта используется один из мастеров, этот режим будет установлен автоматически:



Другая новая опция компилятора, `/Fu`, может быть использована в качестве замены директивы препроцессора `fusing` (см. ниже). Эта опция заставляет компилятор ссылаться на метаданные, содержащиеся в том файле, который мы указываем. Если необходимо, чтобы компилятор ссылался на несколько файлов, следует отдельно использовать данную опцию для каждого файла:



Опция `/AI` сообщает компилятору, в каком каталоге он должен искать файлы, указанные в случае выбора опции `/Fu` (или директивы (`fusing`)). Как и в предыдущем случае, опция `/AI` позволяет указать лишь один путь. Если же путей должно быть несколько, то эту опцию необходимо использовать отдельно для каждого из них:



Директива препроцессора #using

В течение многих лет разработчики C++ используют в своих программах директиву `((include`. А программисты, создающие объекты COM на основе C++, столь же хорошо знакомы с директивой `((import`. Управляемые расширения C++ включают новую директиву препроцессора — `#using`, которая применяется для того, чтобы сделать типы, определяемые в других сборках, доступными текущей программе:

```
#using <mscorlib.dll>
```

Эта строка кода демонстрирует наиболее распространенный случай применения директивы `Busing` для импорта метаданных из файла `mscorlib.dll`. Данный файл определяет ключевые типы, используемые управляемыми приложениями, и ссылку на него должен содержать любой, достаточно нетривиальный управляемый код C++.

Если для создания управляемого проекта C++ используется мастер Visual C++ .NET, то в этом случае приведенная выше строка автоматически добавляется в главный файл приложения с расширением `.cs`.

Простой пример

Продемонстрируем теперь все вышесказанное на конкретном простом примере, в котором создается неуправляемое приложение C++, а затем преобразуется в управляемое приложение. Читатель без труда разберется, что делает приведенный ниже код:

```
#include <iostream>

int main (void)
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Мы предполагаем, что этот код представляет собой неуправляемое консольное приложение, созданное в Visual Studio.NET. Его можно легко изменить так, чтобы оно выполнялось как управляемое приложение C++. Для этого можно создать пустой управляемый проект C++ и добавить в него приведенный выше `.cs`-файл, после чего проект должен компилироваться и выполняться как управляемый код.

Другой способ преобразования неуправляемого проекта в управляемый состоит в добавлении флага `/clr` (см. выше). Однако в этом случае могут возникнуть проблемы совместимости, если использовать уже существующие опции компилятора одновременно с флагом `/clr`. Если предварительно задана любая из опций, перечисленных в приведенной ниже таблице, необходимо отменить или изменить ее для того, чтобы приложение могло быть откомпилировано в управляемый код:

Опции компилятора	Проблемы совместимости с .NET
<code>/RTC</code> , <code>/RTCs</code> , <code>/RTCs</code> , <code>/RTCu</code> , <code>/RTC1</code>	Задать динамическую проверку в управляемом коде не нужно, поскольку среда CLR сама делает проверку. Компилятор в настоящее время не поддерживает указанные опции для некоторых неуправляемых функций в управляемом модуле.
<code>/Gm</code>	Разрешает минимальную перекомпоновку и заставляет компилятор отслеживать зависимости. Однако этот режим компилятора несовместим с директивой <code>«using»</code> .

Опции компилятора	Проблемы совместимости с .NET
/Zl	В этом режиме компилятора активизируется средство отладки "edit and continue" (исправить и продолжить), которое не поддерживается средой CLR.
/Mld, /ML	Указывают компилятору, что он должен выполнять компоновку с использованием однопоточной версии библиотеки периода выполнения C. Их необходимо заменять на опции /Mtd, /Mdd, /MT, /MD, соответствующие многопоточной версии, поскольку все приложения .NET существенно многопоточны.
/GL	Требует от компоновщика дальнейшей оптимизации кода путем встраивания фрагментов кода в модули (файлы с расширением .obj). В среде .NET такая функция выполняется процессором JIT.
/Zd	В этом режиме выдается отладочная информация в формате типа COFF. Такая функция не поддерживается средой исполнения .NET.

После разрешения проблем несовместимости опций компилятора можно просто перекомпоновать проект, который в результате будет откомпилирован в сборку, содержащую код IL и метаданные. Данный проект сможет выполняться в среде CLR, хотя при этом никакие изменения в код не вносятся. Таким образом проект становится управляемым.

Заменяем теперь оператор `std::cout` на соответствующий ему эквивалент .NET, который представляет собой метод `System.Console.WriteLine()`. Все управляемые приложения, использующие управляемые типы, должны ссылаться на файл `mscorlib.dll`. Добавим такую ссылку в наш код с помощью директивы `#using`:

```
#using <mscorlib.dll>

int main (void)
{
    System.Console.WriteLine("Hello World!");
    return 0;
}
```

Теперь приложение использует директиву препроцессора `#using` для импорта типов, включаемых в модуль `mscorlib.dll`, и может быть выполнено в среде CLR. Однако приведенный пример недостаточно впечатляюще демонстрирует возможности управляемого кода. Чтобы приступить к написанию более сложного управляемого приложения на C++, необходимо более подробно познакомиться с управляемыми расширениями C++.

Использование управляемых расширений

Здесь мы рассмотрим новые ключевые слова, относящиеся к управляемым расширениям C++. Их легко распознать, ПОСКОЛЬКУ ОНИ начинаются с двойного символа подчеркивания. Эти ключевые слова применяются для объявления и использования различных ссылочных типов, включая классы, интерфейсы, массивы и указатели. Затем мы обсудим разнообразные типы-значения, а также метод упаковки (boxing), который применяется для передачи типов-значений вместо ссылочных типов. В конце данного раздела обсуждается обработка управляемых исключительных ситуаций и использование управляемого кода C++ для работы со свойствами, делегатами и событиями.

Типы данных, удаляемых сборщиком мусора (`_gc types`)

В начале главы мы подробно рассказали о том, что представляют собой управляемый код и управляемые данные, а также рассмотрели работу сборщика мусора. Мы продемонстрировали необходимость создания отдельной управляемой кучи для того, чтобы сборщик мусора мог выполнять свои функции, и обсудили требования, предъявляемые сборщиком мусора к управляемому коду. Рассмотрим теперь, каким образом код C++ позволяет создавать объекты в управляемой куче, т.е. те объекты, которыми вместо нас будет управлять среда CLR.

Управляемые расширения C++ включают типы, участвующие в процедуре сборки мусора (`_gc types`), или управляемые типы. Программист не должен заботиться об удалении данных этих типов из памяти, поскольку среда CLR сама управляет временем их жизни. Если требуется, чтобы создаваемые нами типы могли быть использованы другими CLR-совместимыми языками, необходимо объявлять эти типы с ключевым словом `_gc`.

Visual C++ .NET позволяет создавать четыре различные категории типов с помощью ключевого слова `_дс`:

- Классы
- Структуры
- Интерфейсы
- Указатели

Классы и структуры

Имеется несколько различий между неуправляемыми классами C++ и классами, объявляемыми с ключевым словом `_дс`. Основные два различия заключаются в том, что экземпляры последних классов удаляются сборщиком мусора и такие классы могут происходить лишь от одного класса. Множественное наследование не поддерживается управляемыми классами, хотя они могут реализовывать любое количество управляемых интерфейсов. Если специально не указан никакой базовый класс, управляемый класс или структура будут неявно наследовать от `System::Object`.

Класс `System::Object` является базовым для всех классов в .NET.

Отсюда следует, что указатель типа `Object`- может рассматриваться как управляемый эквивалент `void*`, поскольку указатель на любой управляемый тип приводится к типу `Object*` (см. выше пример работы со строками). Рассмотрим члены класса `System::Object`, поскольку все классы .NET наследуют эти методы.

Имя члена	Доступ	Описание
<code>Equals</code>	<code>public</code>	Виртуальный метод, выполняющий сравнение своего экземпляра с другим экземпляром. По умолчанию сравниваются ссылки, причем возвращается значение <code>true</code> , если оба параметра представляют собой один и тот же экземпляр. Метод может быть подменен версией, обеспечивающей сравнение значений. Также доступен как статический метод, который принимает два объектных параметра.

Имя члена	Доступ	Описание
GetHashCode	public	Возвращает целое значение, используемое в качестве хэш-кода, которыми классами из пространства имен System.Collection (подробнее см. в главе 5).
GetType	public	Возвращает тип экземпляра объекта, созданного в период выполнения. Этот метод обсуждается в главе 4 вместе с ключевым словом „typeof“.
ReferenceEquals	public	Статический метод, который принимает два объектных параметра и определяет, ссылаются ли они фактически на один и тот же экземпляр объекта. Данный метод приходится использовать, если метод Equals() подменен для сравнения значений.
ToString	public	Виртуальный метод, который может подменяться для того, чтобы он возвращал „дружественное“ по отношению к пользователю представление экземпляра объекта в виде строки текста.
Finalize	protected	Принимает участие в процессе сборки мусора (подробнее см. ниже).
MemberwiseClone	protected	Возвращает поверхностную (shallow) копию экземпляра (см. ниже).

Управляемые класс или структура могут быть объявлены соответственно с ключевыми словами `class` или `struct`, перед которыми располагается ключевое слово `__gc`:

```
__gc struct SomeStruct
{
};

__gc class SomeClass
{
};
```

Существует несколько ограничений относительно того, какие категории элементов данных может содержать управляемый класс. Эти ограничения не запрещают элементу данных быть указателем на неуправляемый тип, однако такой элемент не может быть экземпляром неуправляемого типа. Последнее замечание не относится к POD-типам, которые имеют эквиваленты, определяемые библиотекой классов .NET, как показано в приведенной ранее таблице.

Конструкторы

Объекты всех управляемых типов должны создаваться в управляемой куче с помощью оператора `__gc new`:

```
SomeStruct* somestruct = __gc new SomeStruct();
SomeClass* someclass = new SomeClass();
// Неявное использование "оператора __gc new":
```

Здесь вторая строка показывает, что нет необходимости использовать ключевое слово `__gc`, если инициализируемый тип является управляемым. В этом случае оператор `__gc new` будет автоматически вызван компилятором. Оператор `__gc new` не имеет версии, задающей "размещение" объекта. Это вполне оправдано, поскольку в данном случае память выделяется в управляемой куче и мы не должны знать точное местоположение выделяемой области памяти. С подобной информацией работает среда CLR.

Статические конструкторы класса

Как и любые другие классы C++, управляемые классы могут определять конструкторы, которые применительно к управляемому коду C++ называются **конструкторами экземпляра (instance constructors)**. Такое название связано с тем, что конструктор вызывают любые экземпляры класса при их инициализации. Новыми для программистов C++ являются **статические конструкторы класса (static class constructors)**. Они вызываются только один раз перед созданием первого экземпляра типа. Такая возможность бывает полезной в ситуациях, когда все экземпляры класса совместно используют некоторый ресурс, который нужно инициализировать лишь однажды (например, файл), причем до того, как будет создан любой экземпляр класса. Совместно используемый ресурс можно инициализировать в статическом конструкторе, после чего указанный ресурс будет пригоден для использования любым новым экземпляром. Именно в статическом конструкторе нужно инициализировать все статические члены класса:

```
__gc class SomeClass
{
public:
    static SomeClass()
    {
        Console.WriteLine("In static class constructor");
        m_lInstanceCount = 0;
    }

    SomeClass()
    {
        m_lInstanceCount++;
        m_lInstance = m_lInstanceCount;
        Console.WriteLine("In instance {0} constructor. Total instances = {1}", m_lInstance.ToString(), m_lInstanceCount.ToString());
    }

private:
    static long m_lInstanceCount;
    long m_lInstance;
};

int tmain(void)
{
    SomeClass* pA1 = new SomeClass;
    SomeClass* pA2 = new SomeClass;
    SomeClass* pA3 = new SomeClass;
    return 0;
}
```

Хотя в коде C++ объявляется статический член `m_lInstanceCount` типа `long`, переменная `m_lInstanceCount` поддерживает метод `ToString()`. Причина здесь в том, что управляемый код обращается с типом `long` как с `System::Int32`, и потому тип `long` наследует все методы класса `System::Object`.

Результат данного кода демонстрирует, что статический конструктор выполняется лишь один раз перед выполнением конструкторов экземпляра:

```
In static class constructor
In instance 1 constructor. Total instances = 1
In instance 2 constructor. Total instances = 2
In instance 3 constructor. Total instances = 3
```

Копирование объектов

Управляемые классы используют иную, чем традиционные классы C++, форму семантики копирования. Управляемый класс не может определять конструктор копирования. Причина заключается в том, что среда исполнения .NET не поддерживает категорию конструкторов копирования, и в случае их реализации наши классы нельзя было бы использовать ни в каких других языках, кроме управляемого кода C++. Вместо этого можно применить метод `MemberwiseClone()` класса `System::Object` для получения поверхностной копии объекта.

Поверхностная копия создается в результате побитового копирования всех элементов данных. Пусть, например, экземпляр A имеет элемент `m_p`, который указывает на экземпляр C. Если выполняется поверхностное копирование экземпляра A в экземпляр B, то элементы `m_p` обоих экземпляров будут указывать на C. В неуправляемом коде C++ конструктор копирования по умолчанию реализует семантику поверхностного копирования, и разработчик может создать конструктор копирования, возвращающий полную или детальную копию (deep copy). Последний вариант копирования в нашем примере означает, что элемент `m_p` экземпляра B указывает на копию экземпляра C.

В управляемом коде C++ происходит иное. Метод `MemberwiseClone()` не может быть подменен и является защищенным, т.е. его можно вызывать только из кода внутри самого класса `System::Object` или из производного класса. Поэтому клиентский код не может использовать данный метод для копирования объекта. Вместо этого платформа .NET Framework реализует другую парадигму копирования, которая отличается от концепции конструктора копирования C++. Если нужно копировать объекты, следует реализовать интерфейс `ICloneable()`, который определяет лишь один член, а именно: метод `Clone()`.

Вернемся к примеру класса `SomeClass`. Сделаем этот класс клонируемым, реализовав в нем интерфейс `ICloneable()`:

```

using System;

public class SomeClass : public ICloneable
{
    public:
        // Для краткости код опущен
        Object Clone()
        {
            Console.WriteLine($"Cloning instance {0}, Total instances = {1}",
                               m_Instance.ToString(), (++m_InstanceCount).ToString());
            return MemberwiseClone(); // Поверхностная копия
        }
        // Для краткости код опущен
};

```

Для тестирования этого метода `Clone()` необходимо подменить метод `Object: Equals()`:

```

bool Equals(Object o)
{
    return (this->m_Instance == static-cast<SomeClass*>(o)->m_Instance);
}

```

Результат можно проверить с помощью следующего кода:

```

SomeClass* pA4 = _try_cast<SomeClass*>(pA3->Clone());
if (pA3->Equals(pA4))
{
    Console.WriteLine($"Shallow copy successful");
}

```

Реализация метода `Clone()` в классе `SomeClass()` вызывает метод `MemberwiseClone()`, который создает поверхностную копию объекта и возвращает управляемый указатель на экземпляр клонированного объекта.

Предположим теперь, что имеется класс, для которого необходимо создавать полную копию. Можно реализовать семантику полного копирования в методе `Clone`, как показано ниже:

```
#using <mscorlib.dll>
using namespace System;
```

```
__gc class DeepCopy : public ICloneable
{
public
    Object Clone()
    {
        DeepCopy p = new DeepCopy();
        if ( m_p != NULL )
        {
            // полное копирование
            p->m_p = static_cast<DeepCopy*>(m_p->Clone());
        }
        return p;
    }
    DeepCopy* m_p;
};
```

Здесь класс `DeepCopy` реализует метод `ICloneable` и таким образом поддерживает клонирование, однако при реализации метода `Clone()` выполняется полное копирование путем клонирования экземпляра, на который указывает член `m_p`. Протестировать наш класс `DeepCopy` можно с помощью следующего кода:

```
int _tmain(int argc, _TCHAR* argv[])
{
    DeepCopy dcC = new DeepCopy();
    DeepCopy dcA = new DeepCopy();
    dcA->m_p = dcC;

    DeepCopy dcB = static_cast<DeepCopy*>(dcA->Clone());

    if ( ! Object::ReferenceEquals(dcB->m_p, dcC) )
    {
        Console::WriteLine(S"Deep copy successful");
    }
    return 0;
}
```

Деструкторы

Как и в случае неуправляемого кода, в управляемых классах C++ можно объявить деструктор. Возникает вопрос: зачем нужно писать деструктор, если среда исполнения сама очищает память? Ответ состоит в том, что реализация деструктора по-прежнему необходима для выполнения тех операций по очистке ресурсов, которые не делает сборщик мусора. К подобным операциям относится закрытие соединений с файлами и др.

Однако, в отличие от неуправляемого кода C++, деструктор вызывается во время сборки мусора, и потому невозможно предсказать, когда это произойдет. Программист может вызвать метод деструктора с помощью оператора `delete`, однако

при этом память все равно не будет освобождена до сборки мусора. Такая методика может оказаться полезной, если необходимо, чтобы код деструктора выполнялся в определенный момент времени, не ожидая, пока придет черед сборки мусора. Если требуется вызвать метод `delete` для экземпляра управляемого объекта, обязательно нужно определить деструктор, иначе компилятор выдаст ошибку.

Когда управляемый объект удаляется сборщиком мусора, вызова деструктора на самом деле не происходит. Вместо этого сборщик мусора вызывает метод `Finalize()`. Однако в случае Visual C++ .NET метод `Finalize()` генерируется компилятором и выполняет все те действия, которые указаны в реальном деструкторе. Компилятор также добавляет в программу код, вызывающий метод `Finalize()` для любого базового объекта. Важно подчеркнуть, что мы сами не можем определить метод `Finalize()` — он, как уже отмечалось, генерируется компилятором. Разработчик должен написать любую необходимую реализации в форме деструктора C++.

Приведенный ниже фрагмент кода продолжает предыдущий пример и показывает, как вызываются деструкторы с помощью метода `delete` и сборщика мусора:

```
__gc class SomeClass : public ICloneable
{
public:
    // Для краткости код опущен

    ~SomeClass()
    {
        m_lInstanceCount--;
        Console::WriteLine(S"In instance {0} destructor. Total instances {1}", m_lInstance.ToString(), m_lInstanceCount.ToString());
    }
    // Для краткости код опущен
};

int _tmain(void)
{
    // Для краткости код опущен

    Console::WriteLine(S"Removing reference to instance 1");
    pA1 = 0;

    Console::WriteLine(S"Forcing Garbage Collection");
    GC::Collect();

    Console::WriteLine(S"Deleting remaining instances");
    delete pA2;
    delete pA3;
    delete pA4;

    return 0;
}
```

Здесь мы удалили ссылку на первый экземпляр, присвоив указателю нулевое значение. Поэтому первый экземпляр теперь может быть уничтожен сборщиком мусора. Затем мы инициируем сборку мусора, для чего вызываем метод `System::GC::Collect()`. После этого удаляются оставшиеся экземпляры.

Явный вызов сборщика мусора - потенциально дорогостоящая операция, к которой необходимо прибегать лишь изредка. В рассматриваемом примере это сделано для демонстрации принципа работы деструкторов применительно к управляемым типам.

Выполнение кода в нашем примере приведет к следующему результату:

```
In static class constructor (Статический конструктор класса)
In instance 1 constructor. Total instances = 1
(Конструктор экземпляра 1. Полное число экземпляров равно 1)
In instance 2 constructor. Total instances = 2
(Конструктор экземпляра 2. Полное число экземпляров равно 2)
In instance 3 constructor. Total instances = 3
(Конструктор экземпляра 3. Полное число экземпляров равно 3)
Cloning instance 3. Total instances = 4
(Клонирование экземпляра 3. Полное число экземпляров равно 4)
Shallow copy successful (Поверхностное копирование выполнено успешно)
Deep copy successful (Полное копирование выполнено успешно)
Removing reference to instance 1 (Удаление ссылки на экземпляр 1)
Forcing Garbage Collection (Вызов сборщика мусора)
Deleting remaining instances (Удаление оставшихся ссылок)
In instance 2 destructor. Total instances = 2
(Деструктор экземпляра 2. Полное число экземпляров равно 2)
In instance 3 destructor. Total instances = 1
(Деструктор экземпляра 3. Полное число экземпляров равно 1)
In instance 3 destructor. Total instances = 0
(Деструктор экземпляра 3. Полное число экземпляров равно 0)
In instance 1 destructor. Total instances = 3
(Деструктор экземпляра 1. Полное число экземпляров равно 3)
```

Наибольший интерес здесь представляет последовательность, в которой выводятся строки, относящиеся к работе деструкторов. Кажется, что порядок их вывода совсем не тот, какого можно было ожидать. Сообщение, выдаваемое деструктором экземпляра 1, появляется на экране в последнюю очередь несмотря на то, что мы вызвали сборщик мусора перед удалением остальных ссылок. Причина в том, что сборщик мусора выполняет завершение в отдельном потоке, который отличается от основного потока выполнения, и потому порядок вывода информации о работе деструктора экземпляра 1 может меняться.

Абстрактные классы

Создание абстрактных классов в неуправляемом коде C++ связано с объявлением чисто виртуальных методов, которые не позволяют инициализировать класс. Абстрактные классы могут использоваться только как базовые. В управляемом коде C++ подобное неявное создание базовых классов заменяется явным с помощью ключевого слова `__abstract`. Это ключевое слово прямо указывает, что объявляемые вместе с ним класс или структура не могут быть инициализированы.

Использование ключевого слова `__abstract` не запрещает объявлять виртуальные и чисто виртуальные методы в классе, а также не означает, что класс будет участвовать в процессе сборки мусора. Ниже приведен простой пример применения ключевого слова `__abstract`:

```
__abstract class MyAbstractClass
{
public:
    virtual void Foo()
    {
        Console.WriteLine("In MyAbstractClass.Foo()");
    }
};

class MyConcreteClass : MyAbstractClass
{
public:
```

```

void Foo()
{
    Console.WriteLine($"In MyConcreteClass::Foo()");
}
}

```

В этом коде объявляется класс `MyAbstractClass`, который определяет виртуальную функцию-член `Foo()`. Затем объявляется второй класс `MyConcreteClass`, который является производным от `MyAbstractClass` и подменяет виртуальный метод. Здесь мы имеем пример обычного поведения C++, и если бы не было нового ключевого слова, то можно было бы инициализировать оба класса.

Однако, поскольку класс `MyAbstractClass` объявлен с ключевым словом `__abstract`, он не может быть инициализирован напрямую. Любая попытка сделать это приведет к ошибке компиляции. Объявление абстрактного класса в описанной форме позволяет написать реализации по умолчанию для всех методов данного класса. Такая возможность отсутствует, если используется синтаксис чисто виртуальных функций в неуправляемом коде C++. Ключевое слово `__abstract` может использоваться только для управляемого класса или интерфейса.

Изолированные классы

В то время как абстрактный класс не может быть инициализирован (и, следовательно, должен всегда быть базовым для других классов), изолированный ("запечатанный") класс, объявляемый с ключевым словом `__sealed`, нельзя использовать в качестве базового класса. Таким образом, ключевые слова `__sealed` и `__abstract` имеют противоположный смысл. Ниже приведен простой пример использования ключевого слова `__sealed`:

```

__sealed __gc class MyNonDerivableClass
{
}

```

Класс `MyNonDerivableClass` не может быть базовым ни для какого класса. Рассмотрим, например, следующий код:

```

class Derived1 : MyNonDerivableClass
{
}

```

Этот код вызовет ошибку компиляции, сопровождаемую таким сообщением:

```

error C3246: 'Derived1': cannot inherit from 'MyNonDerivableClass' as it has
been declared as '__sealed'

```

(Ошибка C3246: 'Derived1' не может наследовать от класса 'MyNonDerivableClass', объявленного с ключевым словом '__sealed').

Ключевое слово `__sealed` может также использоваться для того, чтобы исключить подмену производными классами тех методов управляемых классов, которые не являются чисто виртуальными. Это оказывается полезным, когда производный класс создается на основе базового класса, содержащего один или несколько виртуальных методов, которые мы хотим подменить, однако при этом нужно запретить последующее создание производных классов, чтобы не допустить новой подмены методов.

Интерфейсы

Ключевое слово `__interface` позволяет объявить **интерфейс** — структуру, включающую только чисто виртуальные функции, которые могут быть реализованы производными классами. Для чего же нужно это новое ключевое слово? Обычный

стиль неуправляемого кода C++ состоит в объявлении интерфейса путем создания класса, содержащего только чисто виртуальные функции. Как и в случае большинства новых ключевых слов, причина появления ключевого слова `__interface` отнюдь не ограничивается стремлением разработчиков придать стилю языка C++ явно выраженную форму. Когда **интерфейс** объявляется с ключевым словом `__interface`, компилятор генерирует метаданные, описывающие этот интерфейс. В результате другие .NET-совместимые языки и код могут использовать такой интерфейс.

Интерфейс объявляется следующим образом:

```
__interface IBasicIntegerMath
{
    int Add(int x, int y);
    int Subtract(int x, int y);
    bool LessThan(int x, int y);
};
```

Этот фрагмент кода объявляет интерфейс `IBasicIntegerMath`, каждый метод которого неявно считается открытой, чисто виртуальной функцией. Нельзя объявлять члены интерфейса ни с каким другим спецификатором доступа, кроме `public`. В операторы объявления можно добавить конструкцию `"-0"`, однако в этом нет необходимости, поскольку здесь все методы являются чисто виртуальными по умолчанию. Также будет вполне законным использовать в приведенном коде ключевое слово `_abstract`, но само по себе это ничего не дает. Чтобы воспользоваться объявленным выше интерфейсом, необходимо лишь создать производный от него класс и реализовать чисто виртуальные функции:

```
class CIntegerCalculator : public IBasicIntegerMath
{
public:
    // Реализация интерфейса IBasicIntegerMath
    int Add(int x, int y) {return (x + y);}
    int Subtract(int x, int y) {return (x - y);}
    bool LessThan(int x, int y) {return (x < y);}
};
```

Интерфейсы не могут содержать объявления классов или структур, однако в них можно включать объявления вида `__value enum` (см. ниже в разделе о типах-значениях). Равным образом в интерфейсы нельзя включать элементы данных либо статические члены, но *можно* добавлять объявления *свойств* (подробнее см. ниже).

Если имеется класс с несколькими интерфейсами, может возникнуть ситуация, когда в разных интерфейсах этого класса объявляются методы с одинаковыми сигнатурами. В результате возникает неоднозначность, поэтому необходимо уточнять объявление/определение каждого члена, указывая идентификатор интерфейса. При этом в вызывающем коде следует приводить ссылку на объект к соответствующему интерфейсу, чтобы избежать неоднозначного вызова.

Указатели

Сталкиваясь с платформой .NET Framework, программисты C++ разделяются на два лагеря. Одни спрашивают: "Если этот хваленый сборщик мусора занимается вместо нас выделением и освобождением памяти, то зачем вообще нужны указатели?" Другие говорят: "Управляемая куча — это прекрасно... А как мы можем получить на нее указатель и поиграть с ним?"

К сожалению, и те и другие будут несколько разочарованы. Представители первого лагеря быстро убедятся в том, что при использовании управляемых

расширений доступ к *каждому* управляемому типу осуществляется через указатель. Кроме того, поскольку нам в конечном итоге придется реализовывать взаимодействие между управляемым и неуправляемым кодом, то обязательно в той или иной степени нужно будет пользоваться указателями.

Разработчики же, принадлежащие ко второму лагерю, должны иметь в виду тот факт, что управляемые расширения накладывают ограничения на указатели, относящиеся к управляемой куче, и четко определяют, как можно объявлять и использовать указатели на управляемые объекты.

Ключевое слово `__gc*`

До сих пор все объявления управляемых типов, а также доступ к этим типам реализовывались с помощью указателей. Фактически, такая ситуация имеет место для большинства управляемых типов, с которыми нам придется иметь дело. Однако следует отдавать себе отчет в том, что использовавшиеся нами указатели *не* являются обычными указателями C++. Поскольку управляемые объекты размещаются в управляемой куче, для работы с ними нужны **управляемые указатели (managed pointers — `__gc pointers`)**.

Как и в случае оператора `__gc new`, компилятор Visual C++ .NET автоматически определяет, когда создается указатель на управляемый тип, и реализует управляемый указатель тогда и там, когда и где это необходимо. Именно поэтому мы не видим отклонений от стандартного синтаксиса C++. Однако можно и напрямую объявлять, что мы собираемся использовать управляемый указатель:

```
CLivesOnManagedHeap __gc* pManagedHeapObject;
pManagedHeapObject = __gc new CLivesOnManagedHeap;
pManagedHeapObject->m_s = "This is fun!";
```

Первые две строки этого кода объявляют указатель на управляемый класс `CLivesOnManagedHeap` и устанавливают этот указатель равным адресу нового экземпляра `CLivesOnManagedHeap`, который создается оператором `__gc new`. После этого можно использовать переменную `pManagedHeapObject`, чтобы присвоить строковое значение члену экземпляра класса `CLivesOnManagedHeap`.

Компилятор автоматически предполагает, что указатель на управляемую кучу является управляемым указателем, а указатель на неуправляемую кучу есть, соответственно, неуправляемый указатель (`__nogc pointer`). Существует только один способ получить указатель вида `__nogc` на управляемый тип. Он заключается в **фиксации (pinning)** управляемого указателя (подробнее см. в главе 7).

Комбинация ключевых слов `__gc* __gc*`

Можно создать управляемый указатель на управляемый указатель. В этом случае указатель верхнего уровня будет указывать на другой указатель в управляемой куче. Последний обычно представляет собой подчиненный объект по отношению к управляемому объекту. Следующий пример демонстрирует подобную ситуацию:

```
// Объявляется управляемый указатель на управляемый указатель
CLivesOnManagedHeap __gc* __gc* p;

// Задаем указатель на подчиненный объект, представляемый управляемым
// указателем
p = &(pManagedHeapObject->m_pObj);

// Задаем такое значение указателя в управляемой куче, чтобы он указывал
// на сам объект в ней.
*p = pManagedHeapObject;
```

Здесь первая строка объявляет указатель `p` на другой указатель и **приравнивает** значение `p` к нулю. Пусть вас не беспокоит, что, как и в предыдущем примере,

указателю `p` не присвоено нулевое значение. Компилятор автоматически делает это, когда выполняет инициализацию управляемых указателей. Следующая строка задает значение `p` равным адресу члена `m_pObj` того объекта, на который указывает указатель `pManagedHeapObject`. Выполнение последнего оператора приводит к тому, что член `m_pObj` указывает на тот объект, на который указывает указатель `pManagedHeapObject`. В результате указатель `m_pObj` будет указывать на сам объект.

Операция взятия адреса

При взятии адреса экземпляра типа результирующий указатель *может* указывать на управляемую кучу, и в этом случае он будет управляемым указателем. Если же полученный адрес относится к экземпляру, находящемуся за пределами управляемой кучи (т.е. в стеке или в неуправляемой куче), то в этом случае указатель будет неуправляемым (`__nogc pointer`).

Неуправляемый указатель может автоматически преобразовываться в управляемый. Это позволяет управляемому указателю указывать на любой тип, где бы тот ни находился. Сборщик мусора игнорирует любые управляемые указатели, если они не содержат адресов управляемой кучи. Таким образом, ситуация, когда управляемый указатель указывает на неуправляемый экземпляр, не создает проблемы.

Преобразование типов

Стандартный язык C++ включает следующие операторы, которые позволяют преобразовывать один тип в другой:

```
D static_cast<>()
□ dynamic_cast<>()
□ reinterpret_cast<>()
• const_cast<>()
```

Семантика этих четырех видов преобразования не изменяется при переходе к управляемому коду C++. Однако в случае управляемого кода C++ при использовании оператора `reinterpret_cast<>()` возникают некоторые особенности (подробнее см. ниже). Кроме того, управляемые расширения C++ содержат еще один оператор преобразования типа `__try_cast<>()`.

Оператор `__try_cast<>()`

Оператор `__try_cast<>()` представляет собой альтернативу (возможно, даже замену) оператору `dynamic_cast<>()`. Как мы знаем, последний во время выполнения осуществляет проверку допустимости заданного преобразования. Если преобразование невозможно, оператор `dynamic_cast<>()` возвращает нулевое значение (нулевой указатель). Оператор `__try_cast<>()` имеет такую же базовую функциональность, однако с тем важным отличием, что в случае неудачи он выдает исключительную ситуацию `System::InvalidCastException`. Исключительные ситуации являются стандартным механизмом обработки ошибочных ситуаций в управляемом коде, поэтому имеет смысл использовать оператор `__try_cast<>()` вместо `dynamic_cast<>()`:

```
CLivesOnManagedHeap* pManagedHeapObject = new CLivesOnManagedHeap;
```

```
try
```

```
    /*Приведение к базовому типу
```

```
    Object* pObj = __try_cast<Object*>(pManagedHeapObject);
```

```
    /*Приведение к производному типу
```

```
    pManagedHeapObject = __try_cast<CLivesOnManagedHeap*>(pObj);
```

```
}
```

```
catch(InvalidCastException* e)
{
    Console.WriteLine(e->Message);
}
```

Этот код должен нормально работать, однако следующий пример содержит опасное преобразование типа, которое приводит к выдаче исключительной ситуации:

```
CLivesOnManagedHeap* pManagedObject = new CLivesOnManagedHeap;

try
{
    // Опасное приведение к производному типу запрещено, pObj не указывает
    // на тип CLivesOnManagedHeap
    Object* pObj = new Object;
    pManagedObject = __try_cast<CLivesOnManagedHeap*>(pObj);
    Console.WriteLine(S"__try_cast<CLivesOnManagedHeap*>(pObj) succeeded.");
}
catch(InvalidCastException* e)
{
    Console.WriteLine(e->Message);
}
```

При запуске этого кода на экран будет выведено следующее сообщение:

Specified cast is not valid (Указанное преобразование типа недопустимо)

Подобная обработка исключительных ситуаций избавляет от написания объемистых структур, содержащих оператор if и необходимых для выявления нулевых указателей в случае использования оператора преобразования `dynamic_cast<>()`.

Оператор `reinterpret_cast`

В неуправляемом коде C++ оператор `reinterpret_cast<>()` может использоваться для выполнения преобразования между совершенно разнородными типами, такими как целые числа и указатели. Данный оператор сохраняет свою функциональность в управляемом коде C++ с тем ограничением, что управляемый указатель обязательно должен остаться таковым. Это условие имеет место, даже если речь идет о преобразовании указателя в целый тип. Например, нельзя выполнить такую операцию:

```
// Объявляется указатель на величину типа Byte
Byte* pb;

// Преобразование управляемого указателя на объект типа Byte в целое число
UInt32 nn = reinterpret_cast<UInt32>(pb);
```

Здесь предпринята попытка преобразовать ссылочный тип в тип-значение, и компилятор выдаст следующее сообщение об ошибке:

```
error C2440: 'reinterpret_cast': cannot convert from 'unsigned char __gc*' to
'unsigned int'. Cannot Cast a __gc pointer to an integral type.
(Ошибка C2440. Оператор 'reinterpret_cast': Невозможно преобразование
типа 'unsigned char __gc*' в тип 'unsigned int'. Операция преобразования
управляемого указателя в целочисленный тип не разрешена.)
```

Ссылки

Управляемые расширения позволяют использовать ссылки на управляемые объекты. Иногда бывает удобнее работать со ссылками, чем с указателями, и каждый программист может сделать выбор между первыми и вторыми в зависимости от

конкретной ситуации. Ссылки на управляемые объекты подчиняются тем же правилам, что и стандартные ссылки языка C++. Как и следовало ожидать, результатом взятия адреса ссылки на управляемый объект является значение управляемого указателя.

Массивы

Управляемые расширения поддерживают такой полезный тип данных, как управляемые массивы (тип `Array`). Данный тип включают методы, позволяющие манипулировать информацией, содержащейся в массивах, и извлекать ее оттуда. Функциональность типа `Array` обеспечивает сортировку, поиск и копирование данных, а также получение информации о длине массива, типе его элементов, значениях границ и т.д. Ко всей этой информации легко получить доступ благодаря набору методов `GetXXX()`, что делает работу с массивами достаточно простой. Управляемые массивы могут также быть многомерными.

Теперь мы продемонстрируем, как объявляются обычные массивы C++ в управляемом коде. Для этого в операторе объявления используется ключевое слово `__gc`:

```
char ch_array __gc[5] = {'A', 'B', 'C', 'D', 'E'};
```

Для объявления управляемых массивов применяется синтаксис, похожий на синтаксис C++ (скобки массива располагаются после идентификатора). Однако размер управляемого массива указывается при вызове оператора `__gc new` либо определяется количеством элементов в инициализирующем списке. Следующая строка кода объявляет управляемый массив типа `System::Char`, причем начальные значения присваиваются элементам этого массива с помощью списка инициализации:

```
Char c = {'A', 'B', 'C', 'D', 'E'};
```

Хотя в предыдущих примерах использовались ключевые слова `__gc` и `__gc`, в этом нет необходимости, поскольку компилятор сам определит, является ли массив управляемым или неуправляемым, на основании типа элементов массива.

Пример использования массивов

Ниже приведен пример объявления массива указателей на строки:

```
String* s1[] = {"This", "is", "a", "string", "array\n"};
for(int i = 0; i < s1->length; i++)
{
    Console::Write(s1[i]);
}
```

Здесь мы не добавляли ключевое слово `__gc` в оператор объявления. Поскольку массив имеет тип `String*` (т.е. управляемый тип), данный массив становится управляемым по умолчанию. После инициализации массива мы выводим все элементы в окно консоли с помощью цикла `for` и метода `Write()`, а не `WriteLine()`, для того чтобы исключить нежелательные операции перевода строки.

Тип `System::Array`

Любой созданный нами управляемый массив есть объект, производный от класса `System::Array`. Этот класс предоставляет несколько методов для работы с массивами. Выше мы использовали свойство `Length` и оператор `[]` для перебора элементов массива. Спецификация индекса для управляемых и неуправляемых массивов выполняется по-разному. В случае неуправляемых массивов переменная массива может рассматриваться как указатель на его первый элемент. Поэтому можно применять арифметику указателей для получения доступа к элементам массива,

используя смещение указателей элементов относительно базового указателя. Такой способ доступа неприменим к управляемым массивам.

Класс `Array` содержит методы сортировки, поиска и извлечения информации о размерных характеристиках массива, таких как верхняя и нижняя граница, а также число элементов. Для получения полных сведений об этих методах необходимо обратиться к соответствующей документации. Одной из наиболее распространенных операций, используемых разработчиками, является копирование одного или нескольких элементов из одного массива в другой. Для этой цели класс `Array` предоставляет метод `Copy()`:

```
String* s2[] = {S"Wrox", S"Professional", S"Visual C++ .NET"};
String* s3[] = new String[s2->Length];
Array::Copy(s2, s3, 3);
```

Этот код объявляет и инициализирует массив указателей типа `String` с именем `s2`. Затем создается новый, пустой массив `s3` того же размера, что и `s2`, после чего используется метод `Copy` для копирования трех элементов из `s2` в `s3`. Результат копирования можно вывести на экран с помощью следующей функции:

```
void DumpArrayElements(Array* a)
{
    for(int i = 0; i < a->Length; i++)
    {
        Console.WriteLine(S"a[{0}] = {1}", i.ToString(), a->GetValue(i));
    }
}
```

Функция `DumpArrayElements()` принимает один аргумент типа `Array*`. Здесь для перебора элементов массива и вывода их на экран служит простой цикл `for`. Данная функция может работать только с одномерными массивами. Вместо использования обычных обозначений элементов массива для получения доступа к этим элементам нужно применить метод `GetValue()`. Причина заключается в невозможности манипулирования индексами с помощью арифметики указателей для перебора всех элементов массива, как это делается в неуправляемом коде. Передав массив `s3` в функцию `DumpArrayElements()`, получим следующий результат:

```
a[0] = Wrox
a[1] = Professional
a[2] = Visual C++ .NET
```

Класс `System: Array` реализует несколько интерфейсов, один из которых называется `IList` (см. главу 5).

Массивы массивов

`Visual C++ .NET` позволяет создавать управляемые массивы, элементы которых также являются массивами. В качестве массива верхнего уровня можно объявить массив типа `Object*` и присвоить каждому его элементу подчиненный массив. Ниже приведен пример такой методики:

```
Object* object_array[] = new System::Object*[2];

String* planets[] = {S"Mercury", S"Venus", S"Earth", S"Mars", S"Jupiter",
                    S"Saturn", S"Uranus", S"Neptune", S"Pluto", S"Wrox"};

Int32 moons[] = {0,0,1,2,16,18,21,8,1,100};

object_array[0] = planets;
object_array[1] = moons;
```

Этот код объявляет массив с двумя элементами типа `Object*`. Затем создается массив `olanets` типа `String*`, содержащий имена девяти планет Солнечной системы и недавно открытой "необычной планеты" под именем `Wrox`. После этого объявляется и инициализируется массив `moons`, имеющий тип `Int32`. Каждый элемент данного массива указывает количество лун, которое имеет планета с тем же индексом, что и сам элемент. Наконец, оба массива, `Planets` и `Moons`, сохраняются в одном массиве `Object_Array`.

Можно использовать определенную нами ранее функцию `DumpArrayElements()` для вывода элементов массива в окно консоли:

```
DumpArrayElements(object_array);
DumpArrayElements(__try_cast<Array*>(object_array[0]));
DumpArrayElements(__try_cast<Array*>(object_array[1]));
```

Результат выполнения кода показывает, что элементы массива `object_arrays` являются массивами, а также что метод `Array::ToString()` возвращает строковое представление типа массива:

```
a[C] = System.String[]
a[1]=System.Int32[]
```

```
a[0] = Mercury
a[1] = Venus
a[2] = Earth
a[3] = Mars
a[4] = Jupiter
a[5] = Saturn
a[6] = Uranus
a[7] = Neptune
a[E] = Pluto
a[&] = Wrox
```

```
a[C] = 0
a[1] = 0
a[2] = 1
a[3] = 2
a[4] = 16
a[5] = 18
a[6] = 21
a[7] = 8
a[6] = 1
a[9] = 100
```

Многомерные массивы

Управляемые расширения также поддерживают многомерные массивы:

```
String* state_abbr_names[] = {STL, s"Florida",
                              S"GA", S"Georgia",
                              S"NY", S"New York",
                              S"OH", S"Ohio",
                              S"CA", S"California"};
```

```
String* stateabbr2name[, ] = new String*[50, 2];
```

```
for(int j = 0; j < state_abbr_names->Length; j += 2)
{
    stateabbr2name[j / 2, 0] = state_abbr_names[j]; // Аббревиатура
    stateabbr2name[j / 2, 1] = state_abbr_names[j + 1]; // Название
}
```

Данный код объявляет и инициализирует одномерный массив указателей типа `String` с именем `state_abbr_names`. В этом массиве элементы с четными и нечетными индексами содержат, соответственно, аббревиатуры и названия штатов. Для краткости в массив включено только пять штатов.

Следующий массив, `stateabbr2name`, является двумерным. При объявлении многомерного массива число его измерений определяется количеством запятых в скобках, увеличенным на 1.

Затем в цикле `for` выполняется копирование аббревиатур и названий штатов из одномерного массива в двумерный. После этого можно вывести элементы полученного массива, несколько изменив функцию `DumpArrayElements()`, чтобы она работала еще с одним измерением массива:

```
void DumpArrayElements2(System::Array* a)
{
    for(int j=0; j < a->Rank; ++j)
    {
        for(int i=0; i < a->Length / a->Rank; ++i)
        {
            Console.WriteLine(S"a[{0},{1}] = {2}", i.ToString(),
                               j.ToString(), a->GetValue(i,j));
        }
    }
}
```

Передав массив `stateabbr2name` в функцию `DumpArrayElements2()`, получим следующий результат:

```
a[0,0] = FL
a[1,0] = GA
a[2,0] = NY
a[3,0] = OH
a[4,0] = CA
a[5,0] =
...
a[49,0] =

a[0,1] = Florida
a[1,1] = Georgia
a[2,1] = New York
a[3,1] = Ohio
a[4,1] = California
a[5,1] =
...
a[49,1] =
```

Те элементы массива, которые не получили строковых значений, содержат пустую строку, присвоенную им в процессе инициализации.

Типы-значения (`Value types`)

До сих пор мы рассматривали управляемые типы, размещаемые в той куче, где выполняется сборка мусора. Обратимся теперь к механизму управляемого кода C++, который позволяет определять **типы-значения** (`value types`). Типы-значения могут быть структурами или классами, определения которых начинаются с нового ключевого слова `__value` вместо `__ds`. Экземпляры этих типов помещаются в стеке, а не в куче, управляемой средой CLR. Можно также разместить их в *неуправляемой* куче C++ с помощью оператора `__nogc new`. Ключевое слово `__value` помогает

повышать производительность программы при работе с теми типами, экземпляры которых имеют небольшой размер и короткое время жизни. Для таких типов данных нецелесообразно использовать дорогостоящую операцию сборки мусора. Именно поэтому все классы .NET Framework, представляющие собой типы-примитивы, относятся к типам-значениям.

Как структуры, так и классы, которые представляют собой типы-значения, происходят от класса `System::ValueType`. Этот класс является корневым для всех типов-значений и наследует от `System::Object`. Класс `System::ValueType` подменяет следующие методы класса `System::Object`:

Имя члена	Описание
<code>Equals</code>	Метод подменяется для того, чтобы выполнять сравнение значений данного экземпляра и другого экземпляра того же типа.
<code>GetHashCode</code>	Возвращает 32-битовое целое число, используемое в качестве хэш-кода некоторыми классами пространства имен <code>System.Collection</code> .
<code>ToString</code>	Возвращает полностью квалифицированное строковое представление имени типа для данного экземпляра.

Если типы-значения вообще наследуют, то *только* от управляемых интерфейсов. Более того, нельзя создать класс, производный от типа-значения, поскольку такие типы неявно изолированы (sealed). Типы-значения могут подменять виртуальные методы класса `System::ValueType`, однако в них нельзя определять новые методы.

Объекты, относящиеся к типам-значениям, как и управляемые объекты, могут объявлять конструкторы экземпляра и статические конструкторы, а также определять свойства (см. ниже).

Объявление классов и структур с ключевым словом `__value`

Классы и структуры, относящиеся к типам-значениям, определяются с ключевым словом `__value`, которое должно стоять перед ключевыми словами `class` или `struct`:

```
__value struct PersonInfo
{
    char* m_FirstName;
    char* m_MiddleInitial;
    char* m_LastName;
};
```

Во время компиляции этого фрагмента компилятор сгенерирует код, создающий тип `PersonInfo`, который будет наследовать от класса `System::ValueType`. Поскольку структура `PersonInfo` не содержит управляемых указателей, она может быть размещена в неуправляемой куче или в стеке как обычный неуправляемый тип C++. Члены структуры `PersonInfo` являются неуправляемыми указателями. Эта структура абсолютно законно может также содержать экземпляры неуправляемых POD-типов.

Для размещения в стеке экземпляра структуры `PersonInfo` используется стандартный синтаксис C++:

```
PersonInfo person_info; // Выделение памяти в стеке
person_info.m_FirstName = "John";
person_info.m_MiddleInitial = "W";
person_info.m_LastName = "Smith";
```

Для того чтобы поместить экземпляр `PersonInfo` в неуправляемую кучу C++, нужно воспользоваться оператором `__nogc new`.

```

PersonInfo* pperson_info = __nogc new PersonInfo;
// Выделение памяти в куче C++
pperson_info->m_FirstName = "George";
pperson_info->m_MiddleInitial = "M";
pperson_info->m_LastName = "Jones";

```

Мы не можем создать автономный тип, объявленный с ключевым словом `__value`, в управляемой куче. Но это нам и не нужно, поскольку основной смысл типов-значений состоит в том, что это облегченные типы, которые помещаются в стек для повышения производительности программы во время выполнения.

Наша структура `PersonInfo` содержит только неуправляемые указатели. Объявим теперь тип-значение, который включает управляемые указатели:

```

__value struct PhoneNumber
{
    String* m_AreaCode;
    String* m_Prefix;
    String* m_LastFour;
};

```

Экземпляры такой структуры могут создаваться только в стеке:

```

// Выделение памяти в стеке
PhoneNumber phone_number;
phone_number.m_AreaCode = S"212";
phone_number.m_Prefix = S"555";
phone_number.m_LastFour = S"WROX";

```

Выше было показано, как создаются экземпляры типов-значений в стеке и в неуправляемой куче. Если нужно использовать для этой цели управляемую кучу, следует встроить тип-значение в управляемый тип, поскольку лишь последний может быть в нее помещен:

```

gc::class PhoneNumberObject
{
public:
    PhoneNumber m_PhoneNumber;
};

```

Этот код объявляет управляемый класс `PhoneNumberObject`, содержащий член `m_PhoneNumber`, который относится к типу-значению `PhoneNumber`. Поскольку экземплярам класса `PhoneNumberObject` память выделяется в управляемой куче, там же будет размещаться и встроенный экземпляр `m_PhoneNumber`, принадлежащий к типу-значению.

Упаковка и распаковка

Выше было продемонстрировано, как объявляются и размещаются типы-значения в стеке, неуправляемой куче и в управляемой куче CLR. В последнем случае тип-значение должен быть встроен в управляемый тип.

Необходимость заключать тип-значение в оболочку может создать проблему, если потребуется передать экземпляр типа-значения в функцию, которая принимает указатель на экземпляр `System::Object`. Это вполне обычная ситуация (напомним аналогию между указателями типа `Object*` и `void*`). Однако просто преобразовать тип-значение к ссылочному типу нельзя.

Вернемся к нашему примеру со структурой `PhoneNumber` и предположим, что нужно передать экземпляр типа `PhoneNumber` в следующую функцию:

```

void DumpObject(System::Object* o)
{
    System::Console::WriteLine(o->ToString());
}

```

Можно было бы создать экземпляр управляемого класса `PhoneNumberObject`, который помещает в оболочку структуру, объявленную с ключевым словом `__value`, а затем преобразовать указатель на эту структуру к типу `Object*` и передать данный указатель в метод `DumpObject()`. Такая процедура возможна, но она окажется весьма утомительной, если придется реализовывать ее всякий раз, когда нужно будет передать экземпляр типа-значения в метод, который принимает указатель на экземпляр типа `System::Object`.

Описанную процедуру можно значительно *упростить*, если использовать метод, называемый **упаковкой** (**boxing**).

Помещение типа-значения в оболочку

Управляемые расширения содержат ключевое слово `__box`, позволяющее выполнить необходимые действия для того, чтобы поместить любой тип-значение в оболочку, которая представляет собой управляемый класс. Доступ к членам подобного класса-оболочки осуществляется так же, как и в случае обычного типа-значения, однако при этом экземпляры класса-оболочки располагаются в управляемой куче, как и экземпляры любого другого управляемого класса.

Приведенный ниже код использует ключевое слово „`box`” для создания экземпляра упакованного типа `PhoneNumber`:

```
PhoneNumber phone_number;
phone_number.m_AreaCode = S"212";
phone_number.m_Prefix = S"555";
phone_number.m_LastFour = S"2122";

__box PhoneNumber* bxPN = __box(phone_number);
DumpObject(bxPN);
```

Обратите внимание на ключевое слово `__box` в операторе объявления указателя `bxPN` и присваивания ему значения. В результате компиляции первой строки кода компилятор реализует класс-оболочку, производный от `System::ValueType`, и генерирует код, создающий экземпляр класса-оболочки. После этого выполняется копирование экземпляра `phone_number` в член `PhoneNumber` класса-оболочки и возвращается указатель на экземпляр этого класса.

Извлечение типа-значения из класса-оболочки

Необходимо иметь в виду, что никакое изменение упакованного экземпляра не отразится на исходном экземпляре типа-значения. Причина заключается в различии этих экземпляров: один из них располагается в стеке, а другой — в управляемой куче как часть класса-оболочки:

```
bxPN->m_AreaCode = S"888";

Console::WriteLine(S"Stack instance area code = {0}", phone_number.m_AreaCode);
Console::WriteLine(S"Boxed instance area code = {0}", bxPN->m_AreaCode);
```

Результат выполнения этого кода будет следующим:

```
Stack instance area code = 212 (Междугородный код, сохраняемый в экземпляре,
который размещается в стеке, равен 212.)
Boxed instance area code = 888 (Междугородный код, сохраняемый в упакованном
экземпляре, равен 888.)
```

Для того чтобы экземпляр, хранящийся в стеке, отражал внесенные изменения, нужно распаковать упакованный экземпляр:

```
phone_number = *(__try_cast<__box PhoneNumber*>(bxPN));
Console::WriteLine(S"Stack instance area code = {0}", phone_number.m_AreaCode);
```

Здесь мы преобразовываем упакованный объект в ссылку на управляемый объект, находящийся в управляемой куче, и разыменовываем эту ссылку с целью получить экземпляр первоначального типа-значения.

Теперь на экран будет выведена такая информация:

Stack instance area code = 888 (Междугородный код, сохраняемый в экземпляре, который размещается в стеке, равен 888.)

При использовании управляемых расширений программист должен явно задать все операции упаковки, поскольку неявно такие операции не выполняются. Можно передавать в функции упакованные версии типов-значений в качестве параметров:

```
void DumpBoxedPhoneNumber(__box PhoneNumber* bxPN)
{
    DumpObject(bxPN);
}
```

Для вызова этого метода необходимо, чтобы вызывающий оператор упаковал тип-значение:

```
DumpBoxedPhoneNumber(__box phone_number);
```

Типы-значения, относящиеся к перечислениям

Управляемые расширения C++ позволяют определять с помощью ключевых слов `__value enum` типы-значения, представляющие собой перечисления. Эти типы могут использоваться как обычные перечисления C++, однако они предоставляют больше возможностей при разработке управляемых приложений. Одно из важных отличий новых типов заключается в том, что они позволяют программисту указывать базовые типы, которые применяются для представления перечислений:

```
__value enum Planets : Byte {Mercury, Venus, Earth, Mars,
                             Jupiter, Saturn, Uranus, Neptune, Pluto};
```

Здесь с помощью ключевых слов `__value enum` определяется перечисление `Planets` (Планеты), которое имеет базовый тип `System::Byte`, и члены: `Mercury`, `Venus` и т.д. Тип `Byte` соответствует типу `unsigned char` языка C++. Базовые типы для типов-значений, являющихся перечислениями, должны быть целыми числами C++ или любыми другими типами `System::xyz`, где `xyz` — целочисленный эквивалент C++.

Б языке C++ возникает проблема конфликта имен, когда в объявлении перечисления встречаются идентификаторы, совпадающие с другими идентификаторами из той же области видимости. К последним могут относиться, например, идентификаторы, входящие в другое перечисление. В приведенном ниже примере компилятор выдаст ошибки переопределения для каждого перечислимого идентификатора в объявлении типа `RomanGods` (Римские боги), поскольку такие идентификаторы уже определены в перечислении `GasGiants` (Газовые гиганты):

```
enum GasGiants {Jupiter, Saturn, Uranus, Neptune};
enum RomanGods {Jupiter, Saturn, Uranus, Neptune};
```

Объявление перечисления с ключевым словом `__value enum` решает эту проблему, заставляя программиста явно указывать имя перечислителя в случае неоднозначности идентификаторов. Если же объявляется идентификатор, не относящийся к перечислению, но с тем же именем, что и член перечисления, то компилятор предпочтет первый идентификатор второму:

```
__value enum InnerPlanets : Byte {Mercury, Venus, Earth, Mars};
__value enum Metals {Nickel, Gold, Mercury, Titanium};

int Gold = 0;
```

Приведенный код объявляет перечисления, каждое из которых включает элемент Mercury. Кроме того, объявляется переменная Gold, идентификатор которой совпадает с одним из идентификаторов в декларации типа-значения Metals. Этот управляемый код C++ будет откомпилирован без каких бы то ни было затруднений. Приведенный ниже фрагмент использует объявленные перечисления:

```
InnerPlanetsClosestToSun = InnerPlanets::Mercury;
MetalsLiquidAtRTP = Metal::Mercury;
int n = Gold;
```

Необходимо полностью квалифицировать имя Mercury, указав имена перечислений InnerPlanets и Metals. Поскольку идентификатор Gold объявлен в той же области действия, что и перечисление Metals, компилятор использует нумерованный идентификатор Gold в операторе присваивания значения целой переменной n.

Как и другие типы, объявленные с ключевым словом `__value`, перечисления могут быть упакованы. Упакованная версия будет наследовать от класса `System::Enum`. Типы-значения, представляющие собой перечисления, можно также объявлять в управляемых интерфейсах. Наконец, перечисления, создаваемые с применением управляемого кода C++, так же, как и все другие управляемые типы, будут доступны другим .NET-совместимым языкам.

Теперь обратимся к тем концепциям объектно-ориентированного программирования, которые до настоящего времени не были непосредственно реализованы в C++. Для того чтобы рассмотренные выше управляемые типы могли взаимодействовать с другими языками, они должны поддерживать свойства и события. Кроме того, необходимо, чтобы метаданные, описывающие управляемые типы C++, включали информацию о свойствах и событиях этих типов. Управляемые расширения содержат некоторые новые ключевые слова, обеспечивающие выполнение перечисленных требований.

Свойства

Свойства представляют собой методы, к которым клиенты могут получать доступ так же, как к переменным-членам. Иными словами, когда в коде клиента указывается некоторое свойство, то выполняется вызов метода в зависимости от того, каким образом осуществляется доступ к свойству. Методы позволяют клиентам взаимодействовать с объектами, используя четко определенные правила и упрощенные синтаксические конструкции.

Скалярные свойства

Ключевое слово `__property` позволяет определять два метода, реализующих механизм свойств. Скалярным является свойство, которое включает методы Get и Set в соответствии со следующими правилами сигнатур:

- Возвращаемый тип метода get скалярного свойства совпадает с типом свойства. Метод не принимает параметров.
- Возвращаемым типом метода set скалярного свойства является void. Метод set принимает один параметр, тип которого совпадает с типом свойства.

Ниже приведен пример свойства:

```
__gc class IceCream
{
public:
    __property String* get_Flavor(void);
};
```



```

        return m_Flavor;
    }

    [__property void set_Flavor(String* s)
    {
        m_Flavor = s;
    }

private:
    String* m_Flavor;
};

```

Здесь объявляется управляемый класс `IceCream` (Мороженое) и переменная-член, используемая для реализации методов свойства. Далее код объявляет свойство `Flavor` (Вкус) с помощью ключевого слова `__property`. Наличие метода `get` позволяет считывать значения свойства, а благодаря методу `set` можно присвоить свойству то или иное значение. Когда компилятор обрабатывает код, он использует текст, идущий после символа подчеркивания, как имя свойства. В нашем случае реализуется свойство с именем `Flavor`, принадлежащее к типу `String*`. Имя `Flavor` будет представлять созданное нами свойство в интегрированной среде разработки, и с помощью этого имени клиентский код получит доступ к свойству.

Метод `set_Flavor()` принимает единственный параметр типа `String*` и возвращает тип `void`. Для реализации данного метода используется член `m_Flavor`.

Клиентский код работает со свойством `Flavor` следующим образом:

```

IceCream* ic=new IceCream;
ic->Flavor = S"Chocolate";
Console.WriteLine(S"The flavor of the day is {0}", ic->Flavor);

```

Первый из операторов, которые осуществляют доступ к свойству, присваивает строку "Chocolate" (Шоколад) свойству `Flavor`. Отметим, что клиентский код реализует доступ к свойству как к члену класса, а не выполняет вызов метода. Далее мы выводим значение свойства в окно консоли. Клиентский код манипулирует свойством `Flavor`, словно оно является членом класса. При этом на экран будет выдан такой текст:

The flavor of the day is Chocolate (Вкус дня - шоколад)

Индексированные свойства

Иногда оказывается полезным сделать так, чтобы поведение свойства зависело от индекса, т.е. установить зависимость от некоторых критериев для поведения того объекта, к которому мы получаем доступ через свойство. В предыдущем примере можно создать свойство объекта `FlavorOfTheDay` (Вкус дня), которое зависит от задаваемого нами значения индекса, представляющего собой день недели. Тогда доступ к этому свойству будет осуществляться следующим образом:

```

IceCream* FlavorToday = obj->FlavorOfTheDay["Tuesday"];

```

В результате выполнения этой строки кода свойство `FlavorOfTheDay` возвратит указатель на экземпляр класса `IceCream`.

При определении индексированных свойств используются следующие правила для сигнатур:

- Метод `get` должен возвращать тип свойства и принимать любое число параметров, которые вместе составляют индекс.

- D Метод **set** должен возвращать тип `void` и принимать любое число параметров, составляющих индекс, после которых идет параметр, имеющий тип свойства. Этот параметр должен быть присвоен свойству в зависимости от параметров индекса.

Расширим предыдущий пример, добавив в него новый класс `IceCreamVendor` (Поставщик мороженого) и определив в этом классе **индексированное свойство** `FlavorOfTheDay`:

```
using namespace System::Collections;

__gc class IceCreamVendor
{
public:
    IceCreamVendor()
    { m_FlavorOfTheDay = new Hashtable(); }

    __property IceCream* get_FlavorOfTheDay(String* index)
    { return static_cast<IceCream*>(m_FlavorOfTheDay->Item[index]); }

    __property void set_FlavorOfTheDay(String* index, IceCream* value)
    { m_FlavorOfTheDay->Item[index] = value; }

private:
    Hashtable* m_FlavorOfTheDay;
};
```

Класс `IceCreamVendor` реализует свойство `FlavorOfTheDay`, которое использует класс `Hashtable`, принадлежащий к пространству имен `System::Collections`. При этом строке, обозначающей день недели, ставится в соответствие указатель на экземпляр класса `IceCream`.

Приведем пример клиентского кода, который использует созданное нами индексированное свойство:

```
IceCreamVendor* icv = new IceCreamVendor;

IceCream* icVanilla = new IceCream;
icVanilla->Flavor = S"Vanilla";

IceCream* ic2xChocChip = new IceCream;
ic2xChocChip->Flavor = S"Double Chocolate Chip";

IceCream* icStrawberry = new IceCream;
icStrawberry->Flavor = S"Strawberry";

icv->FlavorOfTheDay[S"Monday"] = icVanilla;
icv->FlavorOfTheDay[S"Tuesday"] = ic2xChocChip;
icv->FlavorOfTheDay[S"Wednesday"] = icStrawberry;

Console::WriteLine(S"Tuesday's flavor of the day is {0}",
    icv->FlavorOfTheDay[S"Tuesday"]->Flavor);
```

Этот код вначале создает экземпляр класса `IceCreamVendor`. Затем создается несколько экземпляров класса `IceCream`, и свойству `Flavor` каждого из них присваивается наименование одного из вкусных сортов мороженого. После этого мы присваиваем экземпляры класса `IceCream` свойству `FlavorOfTheDay` экземпляра класса `IceCreamVendor`, указывая в поле индекса день недели. В конце, используя индексированное свойство, мы выводим на экран вкус дня для четверга. Результат выполнения данного кода имеет вид:

Tuesday's flavor of the day is Double Chocolate Chip
(Вкус дня для четверга - двойная порция с шоколадной стружкой)

Следует иметь в виду, что методы `get` и `set` индексированного свойства могут содержать любое число параметров, составляющих индекс. Например, в качестве параметров можно было использовать не только день недели, но и еще одну величину, например цену, создав при этом пару индексов и реализовав двумерную индексную схему. Методы индексированных свойств также могут быть перегружены.

Делегаты

Внимательный читатель, скорее всего, отметил определенный пробел изложения в разделе, посвященном *указателям*. Мы имеем в виду указатели на функции. Причина состоит в том, что платформа .NET использует объектно-ориентированную версию указателей этого типа, которые в терминах технологии .NET называются **делегатами (delegates)**. Делегаты представляют собой полноценные объекты и содержат методы, позволяющие выполнять весьма полезные операции. Вполне естественно, что делегаты объявляются с ключевым словом `delegate`. При этом синтаксис очень похож на тот, что применяется при объявлении функций:

```
delegate void DelegateOne();
delegate int DelegateTwo(int x, String s);
```

Здесь делегат `DelegateOne` может указывать на любую функцию, которая возвращает тип `void` и не имеет параметров. С другой стороны, делегат `DelegateTwo` играет роль указателя на любую из функций, возвращающих тип `int` и принимающих два параметра, принадлежащих соответственно к типам `int` и `String*` в том порядке, как указано в операторе объявления. Приведенное ниже объявление класса включает методы, которые могут служить целевыми функциями для определенных нами делегатов:

```
public class Functions
{
    public Functions(String s)
    {
        m_s = s;
    }

    void Foo()
    {
        Console.WriteLine("Functions {0} - Foo()", m_s);
    }

    int Bar(int x, String s)
    {
        Console.WriteLine("Functions {0} - Bar({1}, {2})",
            m_s, x.ToString(), s);
        return x;
    }
}
```

```

static int Zoo(int x, String* s)
{
    Console.WriteLine(S"Functions: Zoo({0}, {1})".x.ToString(), s);
    return x;
}

private:
    String* ra.s;
};

```

Класс Functions содержит три функции-члена: Foo(), Bar() и Zoo(). Это очень простые функции. В нашем примере они лишь играют роль целевых функций для двух делегатов, объявленных выше, и выводят в окно консоли диагностический код, показывающий, что происходит с программой. Продемонстрируем, каким образом можно использовать класс Functions и определенные нами делегаты:

```

Functions* f1 = new Functions(S"Instance 1");
Functions* f2 = new Functions(S"Instance 2");
DelegateOne* d1 = new DelegateOne(f1, Functions::Foo);

```

Здесь создаются два экземпляра класса Functions, и каждому из них при инициализации передаются различные строковые значения. Более интересна строка, создающая экземпляр делегата DelegateOne. Конструктор делегата принимает два параметра, задающих **целевую** функцию делегата:

- Первый параметр представляет собой указатель на объект и определяет, к какому объекту будет относиться вызов функции (в случае статического метода данный параметр должен быть задан равным нулю).

D В качестве второго параметра указывается адрес вызываемой функции.

Целевой метод обязательно должен быть членом объекта, хотя он может быть как статическим методом, так и методом экземпляра. В приведенном выше фрагменте кода мы передаем делегату объект f1 и метод Functions::Foo в качестве первого и второго параметров соответственно. Это означает, что, обращаясь к делегату, мы будем вызывать функцию f1->Foo():

```

d1();

```

Внешне это похоже на обычный оператор вызова функции, однако на самом деле здесь мы обращаемся к целевому методу экземпляра делегата d1.

Использованиеэкземпляромделегата множественныхметодов(multicasting)

Если бы все возможности делегатов исчерпывались тем, о чем мы говорили выше, делегаты не заслуживали бы большого внимания. Однако они поддерживают весьма интересную **возможность (multicasting)**, которая позволяет одному экземпляру делегата работать с несколькими целевыми методами. Ниже показано, как дополнительный целевой метод присоединяется к экземпляру делегата d1.

```

DelegateOne* d1 = new DelegateOne(f1, Functions::Foo);
d1 +=
d1();

```

С помощью оператора `+=` мы добавляем новый экземпляр делегата `DelegateOne` к тому экземпляру, на который уже ссылается указатель `d1`. Новый целевой метод экземпляра делегата также представляет собой функцию `Functions : Foo`, однако при этом целевым объектом является экземпляр `f2`. При выполнении кода целевые методы вызываются в том порядке, в каком они присоединялись к делегату. В нашем случае вначале это будет метод `f1->Foo()`, а затем — `f2->Foo()`:

```
Functions Instance 1 - Foo() [Экземпляр 1 класса Functions - метод Foo()]
Functions Instance 2 - Foo() [Экземпляр 2 класса Functions - метод Foo()]
```

Как и следовало ожидать, оператор `-=` удаляет целевой метод из делегата. Метод делегата может быть как статическим, так и методом экземпляра:

```
DelegateTwo* d2 = new DelegateTwo(f1, Functions::Bar);
d2 += new DelegateTwo(0, Functions::Zoo);
d2 += new DelegateTwo(f2, Functions::Bar);
d2 += new DelegateTwo(0, Functions::Zoo);

d2(5, "Hello");
```

Метод `Functions::Zoo` является статическим, поэтому в качестве первого параметра в соответствующих строках кода конструктору делегата `DelegateTwo` передается значение 0. Функция `Functions::Bar` относится к методам экземпляра, и при работе с ней необходимо передавать указатель на объект в первый аргумент конструктора делегата `DelegateTwo`. Затем мы вызываем делегат `d2` и передаем ему два необходимых параметра. Результат выполнения приведенного кода будет следующим:

```
Functions Instance 1 - Bar(5, Hello)
Functions::Zoo(5, Hello)
Functions Instance 2 - Bar(5, Hello)
Functions::Zoo(5, Hello)
```

Мы видим, что целевые методы вызываются в том порядке, в каком они присоединялись к делегату. Кроме того, каждый метод получает одни и те же значения параметров, которые были переданы делегату при его вызове.

События

Делегаты составляют основу модели событий "публикация-подписка", используемой в технологии .NET. По существу, данная модель характеризуется наличием одного отправителя события и одного или нескольких его получателей. Такой механизм похож на событийную модель COM, где отправитель события является источником, а обработчик события — приемником. Объект может подписаться на событие, связав с ним обработчик событий. Когда событие публикуется, вызывается каждый связанный с ним обработчик событий. Объект, публикующий событие, уведомляет всех его получателей о том, что данное событие произошло.

Событие объявляется с ключевым словом `__event` и использует делегата, определенного где-либо в приложении. Приведенный ниже оператор объявляет делегата `WarningHandler`. Используя понятия архитектуры обработки событий, можно сказать, что мы задаем сигнатуру функций, которые необходимо написать для обработки событий, инициируемых в процессе выполнения программы:

```
__delegate void WarningHandler(Object* sender, WarningEventArgs* e);
```

Эта сигнатура подчиняется стандартным правилам для сигнатур обработчиков событий, которые определяются событийной моделью .NET. Отсюда не следует,

что нельзя определять события с помощью делегатов, имеющих другие сигнатуры, однако, поступая подобным образом, мы создадим трудности для тех разработчиков, которые привыкли иметь дело с общепринятой моделью событий. В модели событий .NET параметрами делегата являются указатель на объект, отправивший (или инициировавший) событие, и указатель на экземпляр класса System::EventArgs (либо класса, производного от него).

Класс System::EventArgs используется в тех случаях, когда передаваемое событие не содержит относящейся к нему дополнительной информации о состоянии, и простой факт наступления события несет все сведения, которые нужны получателю для работы. Однако, если обработчику события требуется дополнительная информация о состоянии, необходимо создать класс, производный от System::EventArgs, способный сохранять подобную информацию. В приведенном выше примере используется класс WarningEventArgs, производный от System::EventArgs, для сохранения сообщения с теми сведениями, которые относятся к событию:

```
__gc class WarningEventArgs : public System::EventArgs
{
public:
    WarningEventArgs(String* s) : m_WarningMessage(s){}
    String* m_WarningMessage;
};
```

Этот простой класс включает член с областью видимости public. Данный член будет содержать описание причины события, в связи с которым выдается предупреждение. Теперь можно определить событие OnWarning с помощью делегата WarningHandler:

```
__gc class SendWarningEvent
{
public:
    __event WarningHandler* OnWarning;
    void FireWarning(WarningEventArgs* e)
    {
        OnWarning(this, e);
    }
};
```

Код класса SendWarningEvent содержит ключевое слово __event, с помощью которого объявляется событие OnWarning. В модели событий указанный класс рассматривается как **публикующий событие (event publisher)**, а делегат WarningHandler представляет метод **обработчика события для подписчика (event subscriber)**. Метод FireWarning принимает указатель на экземпляр класса WarningEventArgs и передает его в качестве второго аргумента событию OnWarning. Возможно, все это может показаться несколько запутанным, однако следующий пример прояснит сделанные утверждения:

```
__gc class ReceivesWarningEvent
{
public:
    ReceivesWarningEvent(String* s) : m_s(s){}
    void Handler(Object* sender, WarningEventArgs* e)
    {
        Console::.WriteLine(S"Warning event rec'd by {0} - Warning message: {1}", m_s, e->m_WarningMessage);
    }
};
```

```
private:
    String* m_s
};
```

Здесь определен класс `ReceivesWarningEvent`, который содержит метод `Handler()` с требуемой сигнатурой для того, чтобы этот класс мог выступать в роли подписчика согласно событийной модели .NET. Задача метода `Handler()` заключается лишь в выводе на экран сообщения, содержащего строковый член экземпляра `m_s` и той текстовой информации, которую сохраняет экземпляр класса `WarningEventArgs`.

Далее нужно использовать предыдущий код для того, чтобы связать подписчиков с публикующим и инициировать само событие:

```
ReceivesWarningEvent* wh1 = new ReceivesWarningEvent(S"email");
ReceivesWarningEvent* wh2 = new ReceivesWarningEvent(S"console");

SendsWarningEvent* w = new SendsWarningEvent;
```

Вначале этот код инициализирует два экземпляра класса `ReceivesWarningEvent`. В первом случае конструктору передается строка "email", а во втором — "console". Это позволяет продемонстрировать, как два разных агента отслеживают наступление события `OnWarning`. Один агент (в реальном приложении) будет отправлять по электронной почте сообщение всякий раз, когда произойдет событие, а другой будет выводить сообщения на экран. Затем код инициализирует экземпляр класса `SendsWarningEvent`. Напомним, что этот класс содержит событие `OnWarning`.

Теперь используем два экземпляра класса `ReceivesWarningEvent`, чтобы подписаться на событие:

```
w->OnWarning += new WarningHandler(wh1, ReceivesWarningEvent::Handler);
w->OnWarning += new WarningHandler(wh2, ReceivesWarningEvent::Handler);
```

Здесь созданы два новых экземпляра делегата `WarningHandler` и для каждого из них задан целевой метод `Handler()`, соответственно принадлежащий к созданным ранее экземплярам "email" и "console" класса `ReceivesWarningEvent`. Оба экземпляра делегата добавляются к событию `OnWarning` таким же образом, как мы добавляли выше множественные целевые функции в экземпляр делегата.

Теперь осталось только инициировать событие, для чего следует вызвать метод `FireWarning()`:

```
w->FireWarning(new WarningEventArgs(S"System overload"));
```

В результате выполнения кода на экране появятся такие сообщения:

```
Warning event rec'd by email - Warning message: System overload!
(По электронной почте получено событие-предупреждение. Текст предупреждения:
"Система перегружена!")
Warning event rec'd by console - Warning message: System overload!
(На экран выведено событие-предупреждение. Текст предупреждения:
"Система перегружена!")
```

Исключительные ситуации

Платформа .NET Framework реализует такой механизм обработки ошибок, как исключительные ситуации. При этом тип, определяемый с помощью одного языка программирования, может передавать управляемые исключительные ситуации, которые будут улавливаться типом, созданным с использованием другого .Net-совместимого языка.

Ключевые слова `throw`, `try/catch/finally`

Управляемые расширения поддерживают механизм обработки исключительных ситуаций C++ и структурную обработку исключительных ситуаций. Однако способы стандартной обработки исключительных ситуаций C++ дополнены благодаря новому ключевому слову `finally`. Кроме многоязыковой обработки исключительных ситуаций, еще одно преимущество нового механизма работы с ошибками состоит в том, что теперь нет необходимости использовать значения `HRESULT`, и это, несомненно, хорошая новость для разработчиков.

Обычно код, который может выдавать исключительную ситуацию, помещается в блок `try` и один или несколько обработчиков исключительных ситуаций определяются в блоке `catch`:

```
try
{
    // Иницируем нехватку памяти.
    * __gc[1000000000];
}
catch (System::OutOfMemoryException)
{
    Console.WriteLine(e->Message);
    Console.WriteLine(e->StackTrace);
}
```

Здесь делается попытка выделить память для управляемого массива, состоящего из 1000000000 экземпляров класса `SomegcClass`. При вызове оператора `new` тестовая система выдает исключительную ситуацию `System::OutOfMemoryException` (нехватка памяти). Блок `catch` обрабатывает эту исключительную ситуацию и выдает о ней информацию на экран.

Документация MSDN содержит ценные справочные данные, где перечислены исключительные ситуации для конкретных методов. В этой документации указано также соответствие между значениями `HRESULT` и классами исключительных ситуаций .NET Framework. Данную информацию можно найти, воспользовавшись системой поиска и введя в качестве образца текст "HRESULTS and Exceptions".

Управляемые расширения содержат новое ключевое слово `finally`. Оно используется для объявления блока `finally`, который идет после блоков `try/catch`. Код внутри блока `finally` будет выполняться независимо от того, выдается ли исключительная ситуация (перехватываемая или неперехватываемая). Чтобы продемонстрировать результат действия ключевого слова `finally`, используем функцию, которая выдает исключительную ситуацию неуправляемого типа `noGCWroxException`:

```
void Foo()
{
    SomeClass noGCObj("Foo() on stack");
    SomeClass pObj = new SomeClass("Foo() on unmanaged heap");
    SomegcClass* pgcObj = new SomegcClass("Foo() on managed heap");
    try
    {
        Console.WriteLine("Throwing noGCWroxException");
        throw new noGCWroxException();
    }
    finally
    {
    }
```



```

        Console.WriteLine($"Foo() in __finally block");
        if(pObj)
        {
            delete pObj;
        }
        if(pgcObj)
        {
            // Вызываем деструктор и завершитель для управляемого объекта
            delete pgcObj;
        }
    }
}

```

Этот код выделяет память для двух экземпляров неуправляемого класса `SomeClass`, первый из которых размещается в стеке, а второй — в неуправляемой куче. Далее в управляемой куче выделяется память для экземпляра управляемого класса `SomegcClass`.

Классы `nogcWroxException` и `SomeClass` объявляются в другом месте программы, полный текст которой представлен на сайте издательства Wrox.

Приведенный код демонстрирует применение блока `__finally` для очистки всех тех ресурсов, которые класс все еще продолжает занимать после выдачи исключительной ситуации `nogcWroxException`. Поскольку предложение `catch` здесь отсутствует, исключительная ситуация будет передаваться по стеку вызовов обработчику более высокого уровня *после* выполнения блока `__finally`.

Ниже представлен код, вызывающий метод `Foo()` и содержащий блок `catch`:

```

// Перехват неуправляемой исключительной ситуации
try
{
    Console.WriteLine($"main() calling FooO");
    Foo();
}
catch(nogcWroxException*)
{
    Console.WriteLine($"main() in catch(nogcWroxException*) block!");
}
__finally
{
    Console.WriteLine($"main() in __finally block!");
}

```

В результате выполнения этого кода на экран будет выведена следующая информация:

```

main() calling Foo() [Подпрограмма main(). Вызов функции Foo()]
SomeClass() created by Foo() on stack
[Функцией Foo() создан экземпляр класса SomeClassO в стеке]
SomeClassO created by FooO on unmanaged heap
[Функцией FooO создан экземпляр класса SomeClassO в неуправляемой куче]
SomegcClass() created by FooO on managed heap
[Функцией FooO создан экземпляр класса SomegcClass() в управляемой куче]
FooO Throwing nogcWroxException!
[Передача исключительной ситуации nogcWroxException функцией FooO]
Foo() in __finally block
[Функция FooO. Блок __finally]
~SomeClass() created by FooO on unmanaged heap
[Деструктор экземпляра класса SomeClassO, созданного функцией FooO
в неуправляемой куче]
~SomegcClass() created by FooO on managed heap

```

```
[Деструктор экземпляра класса SomegcClass(), созданного функцией Foo()
в управляемой куче]
~SomeClass() created by Foo() on stack
[Деструктор экземпляра класса SomeClassO, созданного функцией Foo() в стеке]
main() in catch ( nogcWroxException* ) block!
[Процедура main(). Блок catch ( nogcWroxException* )]
main() in__finally block! [Процедура main(). Блок__finally]]
```

Обратите внимание на последовательность событий, происходящих после того, как функция Foo() передает исключительную ситуацию nogcWroxException. Вначале вызывается блок __finally в функции Foo(). Этот блок удаляет экземпляры классов как в управляемой, так и в неуправляемой куче. Затем автоматически удаляется экземпляр класса из стека вследствие работы механизма раскручивания стека (stack unwinding). Данный механизм разрушает все экземпляры, созданные в стеке между предложением catch, обрабатывающим исключительную ситуацию, и точкой выдачи этой исключительной ситуации. Поскольку управляемые объекты располагаются в управляемой куче, они не принимают участия в процессе раскручивания и их необходимо удалять явным образом (если, конечно, мы не хотим, чтобы об этих объектах позаботился сборщик мусора).

Блок __finally оказывается полезным, когда мы имеем дело с управляемыми объектами, занимающими системные ресурсы, и эти ресурсы должны своевременно освободиться. Пусть, например, в куче размещается объект, для которого открыт дескриптор файла. Тогда код программы можно структурировать таким образом, что в случае выдачи *неперехватываемого* исключения предложение __finally обеспечит закрытие указанного дескриптора. В результате система вернется в контролируемое состояние, даже если исключительная ситуация не будет перехвачена.

В предыдущем примере было продемонстрировано применение ключевого слова __finally для обработки неуправляемой исключительной ситуации C++. Обратимся теперь к другому примеру, в котором имеет место передача управляемого класса C++, после чего рассмотрим смешанный случай, когда обрабатываются как управляемая, так и неуправляемая исключительные ситуации:

```
void Bar()
{
    SomeClass nogcObj("Bar() on stack");
    SomeClass* pObj = new SomeClass("Bar() on unmanaged heap");
    SomegcClass* pgcObj = new SomegcClass("Bar() on managed heap");

    try
    {
        Console::WriteLine(S"Bar(), Throwing gcWroxException!");
        throw new gcWroxException();
    }
    __finally
    {
        Console::WriteLine(S"Bar() in__finally block");
        if (pObj)
        {
            delete pObj;
        }
        if (pgcObj)
        {
            // Вызываем деструктор и завершитель для управляемого объекта
            delete pgcObj;
        }
    }
}
```

Этот код определяет функцию Bar(), которая почти ничем не отличается от созданного нами ранее метода Foo() за исключением того, что функция Bar() передает экземпляр управляемого класса gcWroxException, производного от System::Exception:

```
__gc class gcWroxException : public System::Exception();
```

Возможно совместное использование блоков catch, обрабатывающих управляемые и неуправляемые исключительные ситуации. Ниже представлен код, вызывающий метод Bar():

```
try
{
    Console.WriteLine(S"main() calling Bar()");
    Bar();
}
catch(gcWroxException*)
{
    Console.WriteLine(S"main() in catch(gcWroxException*) block!");
}
catch(nogcWroxException*)
{
    Console.WriteLine(S"main() in catch(nogcWroxException*) block!");
}
finally
{
    Console.WriteLine(S"main() in „finally block!");
}
```

Результат выполнения данного кода будет таким:

```
main() calling Bar() [Подпрограмма main(). Вызов функции Bar()]
SomeClassO created by Bar () on stack
[Функцией Bar () создан экземпляр класса SomeClassO в стеке]
SomeClassO created by Bar() on unmanaged heap
[Функцией Bar() создан экземпляр класса SomeClass() s неуправляемой куче]
SomegcClass() created by Bar() on managed heap
[Функцией Bar() создан экземпляр класса SomegcClass() в управляемой куче]
BarO Throwing gcWroxException!
[Передача исключительной ситуации gcWroxException функцией Bar()]
Bar() in __finally block [Функция Bar(). Блок __finally]
~SomeClass() created by Barf) on unmanaged heap
[Деструктор экземпляра класса SomeClassO, созданного функцией Bar()
в неуправляемой куче]
~SomegcClass() created by Bar () on managed heap
[Деструктор экземпляра класса SomegcClassO, созданного функцией Bar()
в управляемой куче]
~SomeClassO created by Bar() on stack
[Деструктор экземпляра класса SomeClass(), созданного функцией Bar() в стеке]
main() in catch ( gcWroxException* ) block!
[Процедура main(). Блок catch ( gcWroxException- )]
main() in „finally block!
[Процедура main(). Блок „finally)]
```

Данный код выполняется в основном так же, как и код в предыдущем примере. Экземпляры объектов создаются в стеке, а также в управляемой и в неуправляемой куче. Затем выдается исключительная ситуация, выполняется блок __finally в функции Bar(), происходит раскручивание стека, и исключительная ситуация улавливается в подпрограмме main(). После этого в процедуре main() выполняется блок __finally.

Из приведенных примеров видно, что управляемый код позволяет передавать как управляемые, так и неуправляемые исключительные ситуации C++. Когда передается неуправляемый экземпляр C++, среда исполнения .NET помещает его в оболочку, представляющую собой экземпляр класса `System::Runtime::InteropServices::SEHException`. Затем ищется обработчик, и, если блок `catch` задает неуправляемый класс, оболочка удаляется, чтобы открыть первоначальный неуправляемый класс, который затем сравнивается с типом, задаваемым в блоке `catch`, и перехватывается в случае совпадения. Если же в блоке `catch` указывается тип `SEHException` или один из его родительских классов, то передаваемая исключительная ситуация не распаковывается и обработчик улавливает исключительную ситуацию как `SEHException` (или какой-либо из его базовых классов). Поэтому важно, чтобы все предложения `catch` для неуправляемых классов C++ располагались перед блоками, где указывается класс `SEHException` или один из его базовых классов.

Следующий пример показывает, что происходит, когда передается неуправляемый класс, а блок `catch`, улавливающий тип `SEHException`, находится перед блоком, в котором задан неуправляемый класс:

```
try
{
    Console.WriteLine("main() calling FooO");
    FooO;
}
catch(System.Runtime.InteropServices.SEHException*)
{
}

} Console.WriteLine("main() in catch(SEHException*) block!");
catch(gcWroxException*)
{
} Console.WriteLine("main() in catch(gcWroxException*) block!");
catch(nogcWroxException*)
{
} Console.WriteLine("main() in catch(nogcWroxException*) block!");
finally
{
    Console.WriteLine("main() in __finally block!");
}
```

После запуска этого кода на экран выводятся следующие сообщения:

```
main() calling FooO [Подпрограмма main(). Вызов функции FooO]
SomeClassO created by FooO on stack
[Функцией FooO создан экземпляр класса SomeClassO в стеке]
SomeClassO created by FooO on unmanaged heap
[Функцией FooO создан экземпляр класса SomeClassO в неуправляемой куче]
SomegcClassO created by FooO on managed heap
[Функцией FooO создан экземпляр класса SomegcClassO в управляемой куче]
Foo() Throwing nogcWroxException!
[Передача исключительной ситуации nogcWroxException функцией FooO]
Foo() in __finally block
[Функция FooO. Блок __finally]
"SomeClassO created by FooO on unmanaged heap
[Деструктор экземпляра класса SomeClassO, созданного функцией FooO
в неуправляемой куче]
~SomegcClass() created by Foo() on managed heap
[Деструктор экземпляра класса SomegcClassO, созданного функцией FooO
в управляемой куче]
```

```

~SomeClass() created by Foo() on stack
[Деструктор экземпляра класса SomeClass(), созданного функцией Foo() в стеке]
main() in catch ( SEHException* ) block!
[Процедура main(). Блок catch (SEHException* )]
main() in __finally block! [Процедура main(). Блок __finally]]

```

Здесь исключительная ситуация перехватывается блоком `catch(SEHException*)`. Причина заключается в том, что неуправляемый класс `noGwroxException` заключен в оболочку, созданную классом `SEHException`, пока исключительная ситуация передается по стеку, где система ищет подходящий обработчик. Если поменять местами блоки `catch`, чтобы блок, улавливающий тип `SEHException`, стоял на последнем месте, код будет вести себя так, как необходимо.

Ключевое слово `__identifier`

Следует упомянуть еще одно важное ключевое слово `__identifier`. Оно может потребоваться, если код будет импортировать тип (возможно, созданный на другом языке программирования), который имеет то же имя, что и ключевое слово Visual C++. Предположим, что используется сборка `friend.dll`, созданная с использованием C# и определяющая класс `friend` в пространстве имен `nsFriend`. Чтобы получить доступ к этому классу из управляемого кода C++, необходимо воспользоваться ключевым словом `__identifier`, как показано в приведенном ниже коде:

```

using "friend.dll"
using namespace nsFriend;

void Foo()
{
    __identifier(Friend)
}

```

Поскольку `friend` является ключевым словом C++, этот код не стал бы компилироваться, если бы мы написали его обычным образом. Расширение языка помогает реализовать обещанное преимущество платформы .NET Framework, позволяя обойти один из барьеров на пути многоязыковой разработки приложений.

Сводка ключевых слов

Управляемые расширения Visual C++ содержат 15 ключевых слов, большинство из которых мы обсудили в настоящей главе. В следующей таблице приведены все упомянутые ключевые слова:

Ключевое слово	Описание
<code>__abstract</code>	Используется в объявлении класса или структуры и указывает, что тип не может быть инициализирован.
<code>__box</code>	Применяется для создания управляемого объекта из объекта, принадлежащего к типу-значению.
<code>__delegate</code>	Служит для объявления "умного" указателя функции, называемого делегатом.
<code>__event</code>	Используется для объявления события различными способами в неуправляемом и управляемом коде, а также в программах COM.
<code>__finally</code>	Объявляет блок <code>__finally</code> , который принимает участие в обработке исключительных ситуаций.

Ключевое слово	Описание
<code>__gc</code>	Объявляет тип, удаляемый сборщиком мусора.
<code>__identifier</code>	Может применяться для работы с типами {объявленными с использованием других языков}, имена которых совпадают с ключевыми словами C++.
<code>__interface</code>	В случае использования при объявлении класса или структуры, указывает, что все члены объявленного типа будут чисто виртуальными функциями.
<code>__nogc</code>	Объявляет тип, который не будет удаляться сборщиком мусора (т.е. обычный тип C++).
<code>__pin</code>	Применяется для того, чтобы запретить сборщику мусора удалять объект или перемещать его.
<code>__property</code>	Используется для объявления метода, представляющего собой свойство.
<code>__sealed</code>	От класса или структуры, объявляемых с этим ключевым словом, не могут происходить другие классы.
<code>__try_cast</code>	Возможная замена оператору <code>dynamic_cast</code> . Если оператор не может быть выполнен, выдается исключительная ситуация.
<code>__typeof</code>	Оператор, возвращающий тип любого заданного управляемого объекта.
<code>__value</code>	Применяется для объявления типа-значения.

В этой таблице имеется два ключевых слова, которые мы не рассматривали: `__typeof` и `__pin`. Мы обсудим их в главах 4 и 7, когда будем изучать взаимодействие между управляемым и неуправляемым кодом.

ГЛАВА 3



Сборки

В настоящей главе подробно рассматриваются **сборки (assembly) .NET**. Мы детально изучим все основные вопросы, относящиеся к сборкам, начиная с обзора общих принципов этой технологии, а затем перейдем к механизмам внутренней реализации сборок, после чего обратимся к созданию сборок и их использованию. В частности, в данной главе освещены следующие темы:

- Сборки и их составляющие. Преимущества использования сборок, основные принципы их создания и применения.
- ❑ Просмотр сборок с помощью утилиты `ildasm.exe`
- ❑ Различие между закрытыми и совместно используемыми сборками. Создание совместно используемых сборок и их инсталляция в глобальном кэше сборок (GAC).
- Создание ресурсов и работа с ними в сборках
- ❑ Развертывание сборок

Что такое сборки

Выражаясь упрощенно, можно сказать, что сборки представляют собой основные строительные блоки, обеспечивающие развертывание и исполнение приложений .NET Framework. Каждый исполняемый файл, библиотека или файл ресурсов, созданный на платформе .NET, является сборкой — иного не дано. Сборки — не только фундаментальные компоненты в механизме развертывания приложений. Они представляют собой основную конструкцию, которая определяет систему управления версиями, область действия типов и ресурсов, зависимости, а также управление правами доступа. Применение сборок привнесло значительные изменения по сравнению с технологиями, использовавшимися до появления .NET, в которых исполняемые или DLL-файлы сохраняли лишь непосредственный код приложения или библиотеки, а метаданные, описывающие содержимое, помещались во внешних файлах. Фактически, внутренняя организация сборки .NET принципиально

отличается от той, которая используется в классическом случае исполняемого файла или DLL-библиотеки. Единственное очевидное сходство между двумя подходами заключается в тех целях, ради которых создаются файлы, а также в том, что эти файлы имеют расширения , exe или .dll.

Структура сборки

Сборка представляет собой набор элементов, вместе образующих сегмент приложения. Для формирования всего приложения требуется несколько сборок. Сборки могут¹ состоять из четырех различных типов управляемых элементов, таких как:

- **Декларация (manifest) сборки** — содержит метаданные, описывающие всю сборку. Сюда включается информация обо всех элементах, образующих сборку, и о связях этих элементов друг с другом. Декларация сборки также содержит данные, относящиеся к версиям сборки, безопасности и т.д.
- **Метаданные** — предоставляют информацию о типах, объявленных в сборке, т.е. об именах типов, их областях видимости, базовых классах и реализуемых интерфейсах. Наличие в сборке метаданных и декларации означает, что сборка является полностью самодокументированной.
- **Код приложения** — код, который компилируется в формат **промежуточного языка Microsoft (Microsoft Intermediate Language - MSIL)** и реализует типы, описываемые метаданными.

Д Другие ресурсы, такие как графические файлы, курсоры и статический текст.

Кроме того, сборка также может содержать неуправляемый код и неуправляемые данные. Существует множество способов группировки четырех перечисленных элементов в сборке. Единственным обязательным элементом сборки является декларация, однако без метаданных, кода приложения или ресурсов от сборки будет немного пользы. Сборка не обязана представлять собой один физический файл. Каждый из указанных нами элементов, кроме декларации, может быть распределен среди нескольких файлов. Однако чаще всего для этих целей используется единственный файл:

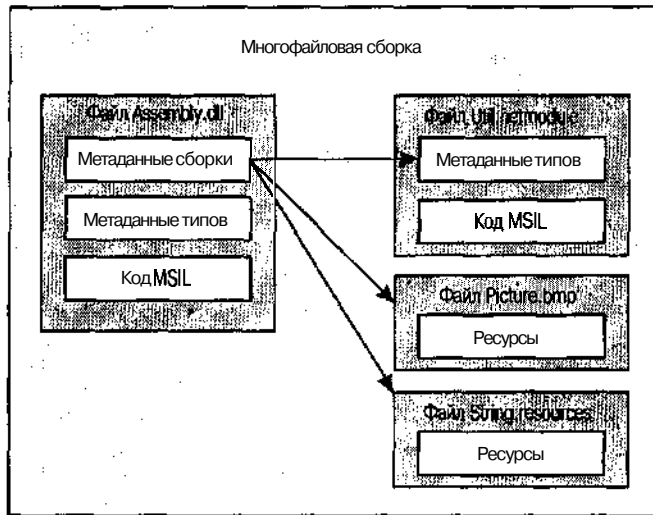


Такой файл называется **переносимым исполняемым файлом (Portable Executable - PE)**. Чаще всего сборки, состоящие из одного файла (т.е. те, которые мы создавали до сих пор), включают лишь декларацию, метаданные и код MSIL.

Если сборка содержит несколько файлов, ее элементы размещаются в отдельных модулях. Модули могут состоять из откомпилированного кода MSIL, файлов с расширениями .btr или .jred либо других файлов ресурсов. Модули, которые образуют сборку, состоящую из нескольких файлов, хранятся в файловой системе как отдельные файлы, физически никоим образом не связанные друг другом. Вместо

этого они связываются логически через декларацию сборки, где содержится детальная информация о файлах, составляющих сборку. При этом отдельные файлы "не знают" о том, что они являются частью сборки. Все файлы должны находиться в одном и том же физическом каталоге, и на их основе можно создать сборку с помощью утилиты Assembly Linker (AL.exe). Среда CLR работает с многофайловой сборкой как с единым объектом.

Лишь один из модулей сборки может содержать декларацию. При этом возможна ситуация, когда единственный модуль включает *только* декларацию. Но чаще всего декларация оказывается упакованной в один из связанных модулей. Ниже приведен пример многофайловой сборки, которая распределена среди четырех файлов. Декларация входит только в файл Assembly.exe:



Многофайловые сборки полезны при создании больших приложений, состоящих из компонентов, разрабатываемых на разных языках. В этом случае можно выделить для каждого конкретного языка свой собственный модуль и поручить разработку отдельных модулей различным разработчикам. Сборки, состоящие из многих файлов, удобно также применять для оптимизации процессов загрузки. В этом случае имеет смысл поместить редко используемые типы и методы в отдельные модули и загружать их только в случае необходимости.

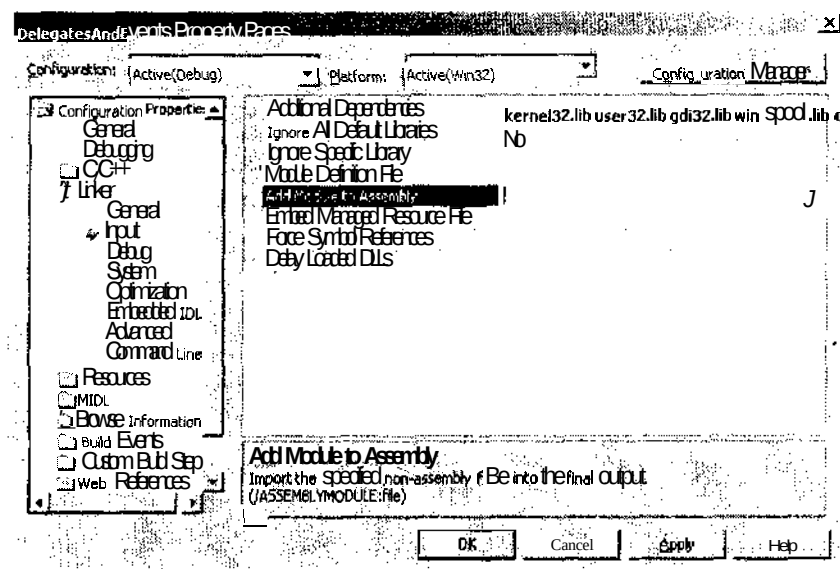
Опции компоновщика

При создании с помощью мастера управляемого приложения C++ по умолчанию создается однофайловая сборка. Однако можно задать компоновщику способ упаковки управляемого кода, используя новые опции, которые принадлежат к управляемым расширениям C++. Таким опциями являются:

- /NOASSEMBLY — В этом случае выходной файл не содержит информации о сборке, и создается модуль DLL, который может быть использован или упакован другими сборками.

D /ASSEMBLYMODULE — Позволяет импортировать в текущий проект модуль DLL (созданный с использованием опции /NOASSEMBLY либо компилятором другого .NET-совместимого языка). При этом типы, объявленные в модуле, будут недоступны импортирующей сборке.

Последняя опция может применяться для создания сборки из нескольких модульных файлов, скомпонованных с помощью опции /NOASSEMBLY. Каждый из этих модулей определяет подмножество некоторой библиотеки, так что все они в конечном итоге должны быть упакованы в одну сборку DLL. Опция /ASSEMBLYMODULE позволяет упаковать два модуля в единую сборку, предназначенную для поставки конечным пользователям. Представленный ниже снимок экрана иллюстрирует процесс добавления модуля в сборку с помощью окна свойств проекта:



Основные характеристики сборок

Ниже перечислены некоторые из основных характеристик сборок, облегчающие их использование, установку и распространение:

- Сборки определяют границы системы защиты для приложения. Они задают набор правил доступа, которые не должны нарушаться во время выполнения, чтобы приложение работало корректно.
- Благодаря сборкам предотвращаются конфликты типов, поскольку полное имя создаваемого разработчиком типа включает имя сборки, в которой этот тип определен.
- Декларация сборки содержит информацию, достаточную для работы с внешними ссылками, а также описывает все типы и ресурсы, предоставляемые сборкой.
- Сборка точно определяет, какие из версий других сборок, на которые она ссылается, ей необходимы.
- Сборки поддерживают самоописание. Вся информация, необходимая для исполнения сборки, содержится в ней самой.
- Благодаря возможностям управления версиями сборок две сборки в приложении могут ссылаться на различные версии третьей сборки, не вступая в конфликт друг с другом (позже мы продемонстрируем это на конкретном примере).

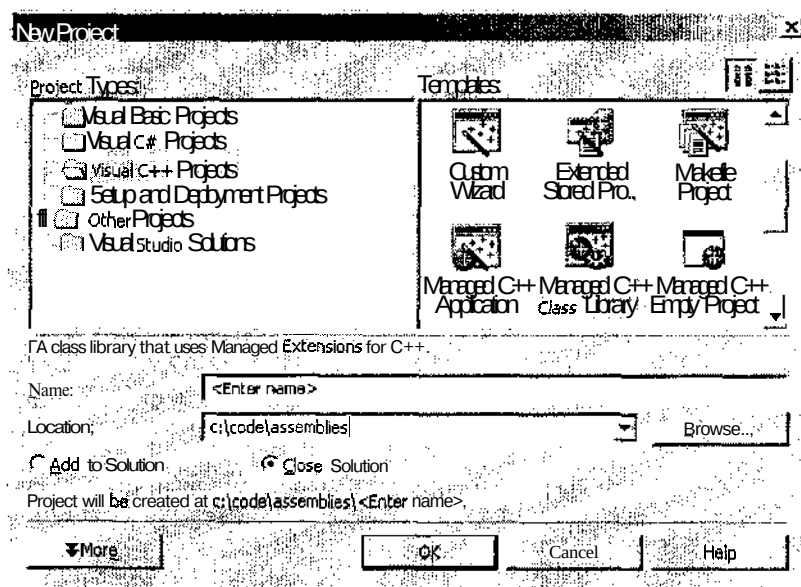
- а Преимуществом сборок является отсутствие проблем, связанных с инсталляцией и деинсталляцией. Инсталляция часто сводится к простому перетаскиванию мышью каталога приложения в выбранное нами место. Для автоматической инсталляции оказывается достаточно команды хсору (подробнее о разворачивании сборок см. в конце главы).

Создание сборки

Мы уже создавали сборки в предыдущих главах, поскольку все приложения .NET являются сборками. В настоящем разделе мы обсудим создание еще двух сборок, но теперь основное внимание обратим на сборки как таковые, а не на тексты программ. Вначале разработаем очень простую сборку, представляющую собой библиотеку, а затем создадим сборку для консольного приложения, вызывающего эту библиотеку. Как библиотека, так и приложение являются сборками. Единственное различие между ними состоит в том, что приложение имеет начальную точку входа, из которой запускается программа.

Создание библиотеки классов

Наша сборка будет включать класс Stopwatch, который реализует функциональность секундомера с остановом и располагается в пространстве имен MyClock. Создадим новый проект Managed C++ Class Library (Библиотека управляемых классов C++):



Когда мастер закончит свою работу, добавим следующий код в заголовочный файл MyClock.h, который будет сгенерирован:

```
#pragma once

using namespace System;
```

```

namespace MyClock
{
    public __gc class Stopwatch
    {
    public:
        Stopwatch();

        void Start();
        void Stop();
        __property int get_ElapsedTime();

    private:
        TimeSpan elapsedTime;
        DateTime startTime;
    };
}

```

V

Класс Stopwatch включает две закрытые переменные `elapsedTime` и `startTime`. Эти переменные имеют типы, непосредственно принадлежащие к пространству имен System, которое относится к библиотеке классов .NET Framework. Структура `TimeSpan` представляет собой интервал времени, а структура `DateTime` — определенный момент времени (обычно имеющий формат даты и времени суток).

Вместе с этими переменными класс Stopwatch определяет открытый конструктор и два открытых метода: `Start()` и `Stop()`. И, наконец, класс содержит свойство только для чтения `ElapsedTime`, возвращающее значение свойства `Total Milliseconds` структуры `TimeSpan`. Это значение представляет собой временной диапазон в миллисекундах.

Далее добавим в файл определений `MyClock.cpp` такой код:

```

#include "stdafx.h"

#include "MyClock.h"

MyClock::Stopwatch::Stopwatch()
{
    elapsedTime = TimeSpan::Zero;
    startTime = DateTime::MinValue;
}

void MyClock::Stopwatch::Start()
{
    startTime = DateTime::Now;
}

void MyClock::Stopwatch::Stop()
{
    if(startTime != DateTime::MinValue)
        elapsedTime = DateTime::Now - startTime;
}

int MyClock::Stopwatch::get_ElapsedTime()
{
    return Convert::ToInt32(elapsedTime.TotalMilliseconds);
}

```

В конструкторе мы инициализируем член `elapsedTime`, присваивая ему значение `TimeSpan::Zero`, чтобы свойство `ElapsedTime` возвращало нуль, если к нему обращается код до того, как вызван метод `Start()`. Кроме того, переменной `startTime`

присваивается начальное значение `DateTime.MinValue`, чтобы при вызове метода `Stop()` раньше `Start()` переменная `elapsedTime` не получила бы некорректное значение.

Метод `Start` просто задает значение переменной `startTime`, равное текущей дате и времени. Метод `Stop()` вначале проверяет, вызван ли метод `Start()`, а затем вычитает время, сохраняемое в переменной `startTime`, из текущего значения времени и передает результат переменной `elapsedTime`. Откомпилировав проект, мы получим в выходном каталоге библиотечную сборку `MyClock.dll`.

Доступность типов

Аналогично членам классов C++, типы, определяемые в сборках, характеризуются **доступностью (accessibility)**. По умолчанию все классы в сборке являются закрытыми (`private`), т.е. они могут использоваться только типами, определенными в той же самой сборке. Чтобы сделать *класс* видимым из *других* сборок, необходимо объявить его с ключевым словом `public` (открытый). Следующий фрагмент кода демонстрирует объявление открытого и закрытого классов:

```
public __gc class LookAtMe()
private __gc class CantTouchMe()
```

Поскольку открытые классы видимы из внешних сборок, должен существовать механизм, определяющий доступность членов класса для кода внутри сборки и кода, находящегося за пределами сборки. Управляемые расширения C++ реализуют такой механизм за счет использования двух спецификаторов доступа вместо одного. При этом спецификатор, накладывающий наибольшие ограничения, определяет область видимости со стороны внешних сборок, а спецификатор с меньшими ограничениями задает доступ из кода *внутри сборки*. Например:

```
public __gc class LookAtMeO
{
    private private:
        // Закрытый метод для внутреннего и внешнего кода
        void PrivatePrivateO {}

    private protected:
        // Защищенный метод для внутреннего кода и закрытый - для внешнего кода
        void PrivateProtectedO {}

    public private:
        // Открытый метод для внутреннего кода и закрытый - для внешнего кода
        void PrivatePublicO {}
};
```

Здесь метод `PrivatePrivate()` закрыт как для кода внутри сборки, так и для внешних сборок. Метод `PrivateProtected()` является защищенным (`protected`) для кода внутри сборки, но закрытым по отношению к другим сборкам. Это значит, что типы, определенные в той же самой сборке, что и класс `LookAtMe`, могут происходить от `LookAtMe` и получать доступ к защищенному члену данного класса. Однако типы, определяемые во внешней сборке, и производные от класса `LookAtMe` не имеют доступа к этому члену. Наконец, метод `PrivatePublicO` открыт для внутреннего кода и закрыт для внешних сборок. Порядок следования спецификаторов доступа не имеет значения. Важно то, что спецификатор с более сильными ограничениями всегда определяет доступ со стороны внешнего кода, а спецификатор с наименьшими ограничениями относится к доступу из *внутреннего* кода.

Спецификатор области видимости класса подменяет спецификатор доступа для члена, если первый определяет более ограниченную область видимости. Поэтому, даже если класс, объявленный с ключевым словом `private`, содержит члены, объявление которых задает область видимости `public` по отношению к внешним сборкам, эти члены после компиляции кода станут закрытыми.

И, наконец, необходимо иметь в виду, что объявление члена класса с одним спецификатором доступа эквивалентно использованию в операторе объявления пары таких же спецификаторов. Иными словами, метод `PrivatePrivate()` можно объявить только с одним ключевым словом `private` (вместо `private private`). Если же нужно создать метод, доступный для кода как внутри, так и за пределами сборки (и если это позволяет сделать спецификатор класса), то достаточно просто объявить данный метод как `public`.

Создание приложения

После этого небольшого отступления вернемся к созданному нами классу `Stopwatch`. Сам по себе этот класс бесполезен, поскольку он не содержит точки входа. Поэтому теперь следует написать приложение, использующее класс `Stopwatch`, которое позволило бы убедиться, что код работает должным образом. Создадим новое управляемое приложение C++ и назовем его `TestStopwatch`. Оно будет моделировать программу протоколирования, которая использует класс `Stopwatch` для определения того, сколько времени занимает выполнение отдельных разделов кода.

После создания нового пустого проекта изменим сгенерированный файл `TestStopwatch.cpp`, добавив в него следующий код:

```
#include "stdafx.h"

using <mscorlib.dll>
using "MyClock.dll".

ffinclude<tchar.h>

using namespace System;
using namespace MyClock;

// Это точка входа в приложение
int _tmain(void)
{
    Stopwatch Swatch = new Stopwatch; // Ссылка на объект Stopwatch
    for(__int64 i = 0; i < 100000000; i++)
        watch.Stop();
    Console.WriteLine($"Interval time in milliseconds: {0}",
        watch.ElapsedTime.ToString());

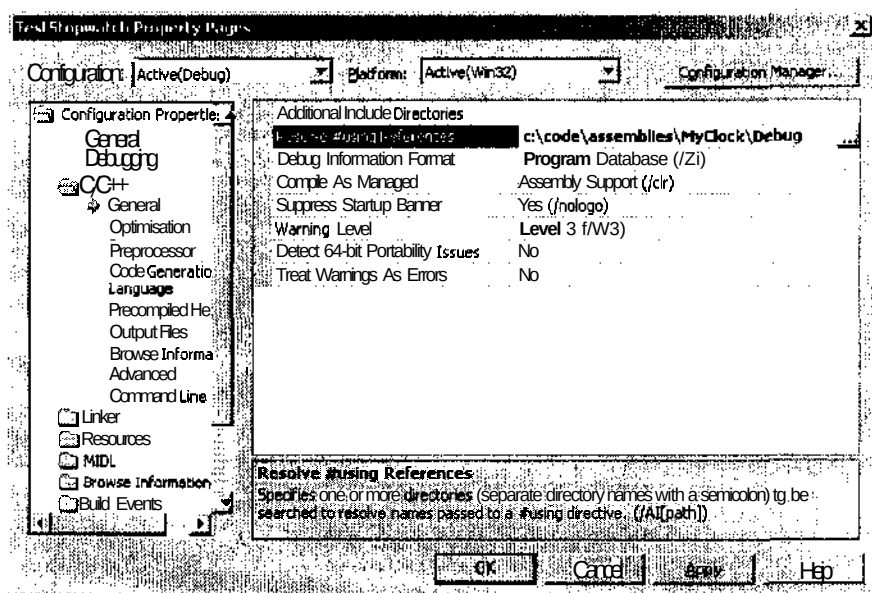
    for(__int64 i = 0; i < 100000000; i++)
        watch.Stop();
    Console.WriteLine($"End time in milliseconds: {0}",
        watch.ElapsedTime.ToString());

    watch.Start();

    for(__int64 i = 0; i < 100000000; i++)
        watch.Stop();
    Console.WriteLine($"Reset stopwatch elapsed time in milliseconds: {0}",
        watch.ElapsedTime.ToString());
    Console.WriteLine(watch.AssemblyVersion);
    return 0; }
```

Функция `main()` создает экземпляр класса `Stopwatch`, принадлежащего к пространству имен `MyClock`. Приложение вызывает метод `Start()` для запуска секундомера и выполняет некоторый код (в данном случае простой цикл `for`), прежде чем вызывается метод `Stop()` и результаты измерения выводятся в окно консоли. Затем, чтобы сделать пример более интересным, эта последовательность действий повторяется дважды.

Однако прежде, чем компилировать код, необходимо указать среде разработки Visual Studio, где следует искать сборку, на которую делается ссылка с помощью директивы `flusing` в начале файла. Щелкнем правой кнопкой мыши на имени проекта в `Solution Explorer` и выберем в контекстном меню пункт `Properties`. На левой панели открывшегося диалогового окна свойств проекта выберем папку `C/C++`, а затем щелкнем на узле `General` (Общие свойства). На правой панели появится соответствующая таблица свойств. Выберем здесь свойство `Resolve #using References` (Разрешение ссылок в директиве `#using`) и укажем путь к папке, содержащей сборку, на которую мы сослались в директиве `#using`:



С помощью этого диалога можно также добавить папки `Debug` и `Release`.

Теперь можно компоновать приложение. Если не добавить ссылку на нужную папку, то при попытке компоновки приложения будет выдана ошибка следующего вида:

fatal error C1107: could not find assembly 'MyClock.dll': please specify the assembly search path using `/AI` or by setting the `LIBPATH` environment variable (Фатальная ошибка C1107. Невозможно найти сборку 'MyClock.dll'. Укажите путь для поиска сборки с помощью ключа `/AI` или переменной окружения `LIBPATH`.)

Однако даже после указания нужного пути приложение все еще не будет работать. При попытке запустить его будет выдано исключение `System.IO.FileNotFoundException`, т.е. среда CLR не может найти файл `MyClock.dll`.

Среда исполнения **CLR** ищет сборки лишь в трех местах: в глобальном кэше сборок (GAC), в каталоге приложения или же в подкаталоге этого каталога.

Проблема заключается в том, что файл `MyClock.dll` не находится ни в одном из перечисленных мест, поэтому **CLR** и не может его найти. Добавив ссылку на этот файл, как было описано выше, мы указали путь к нему лишь для *Visual Studio.NET*. Чтобы **CLR** нашла файл, необходимо поместить его в одно из трех указанных выше мест, просматриваемых средой **CLR**. В настоящий момент проще всего переместить файл `MyClock.dll` в подкаталог `Debug` каталога `TestStopwatch` (о глобальном кэше сборок см. ниже).

Возникает вопрос, почему мы не столкнулись с этой проблемой в главе 1, когда создавали многоязыковое приложение. Причина в том, что при использовании Visual Basic и C# среда разработки Visual Studio .NET автоматически копирует в нужный каталог сборки, содержащую DLL, когда ссылка добавляется в проект.

Хотя в нашем простом примере нетрудно переместить файл с помощью `Windows Explorer`, очевидно, что в случае больших и сложных проектов это будет не самым хорошим решением. Обычно лучше добавить некоторые команды в событие, предшествующее компоновке проекта `TestStopwatch`, которое наступает всякий раз при перекомпиляции проекта. Для этого в окне свойств проекта `Property Pages` следует открыть папку `Build Events` (События компоновки) и выбрать пункт `Pre-Build Event` (Событие, предшествующее компоновке проекта).

После этого можно запустить приложение. В результате на экран будет выдана такая информация:

```
Interval time in milliseconds: 1432 (Интервал времени в миллисекундах - 1432)
End time in milliseconds: 2844 (Время завершения в миллисекундах - 2844)
Reset stopwatch elapsed time in milliseconds: 1722
(Время в миллисекундах, прошедшее с момента перезапуска секундомера - 1722)
```

Просмотр сборки с помощью утилиты **ILDasm**

Платформа `.NET Framework` включает дизассемблер **MSIL**, который называется **ILDasm** и позволяет просматривать содержимое сборки. С помощью этой утилиты можно выполнять разборку переносимых исполняемых файлов (PE), имеющих расширения `.exe`, `.dll`, `.obj` или `.lib`.

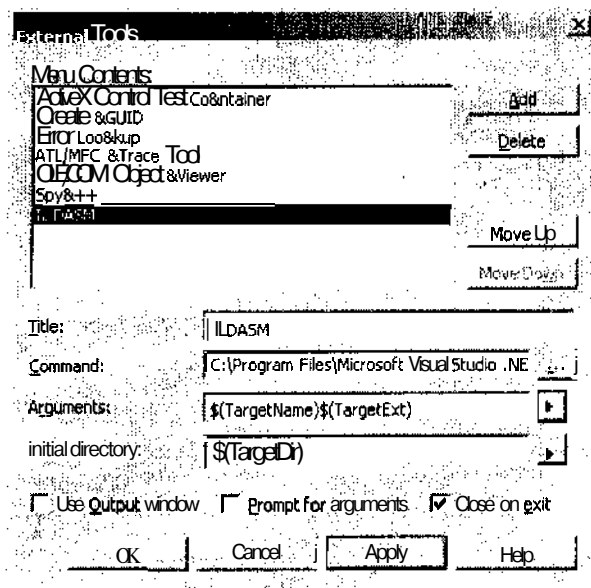
Утилита **ILDasm** может быть запущена из командной строки:

```
> ildasm [опции] [имя_PE-файла] [опции]
```

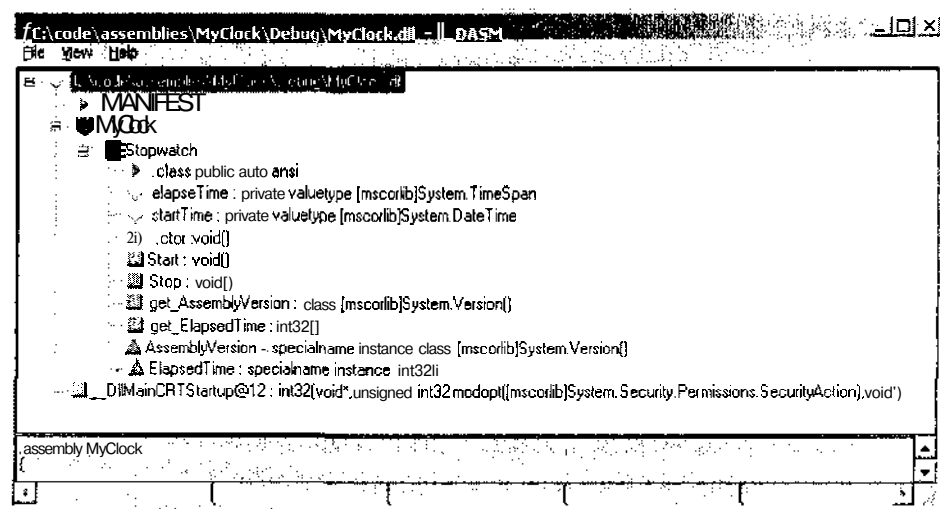
Вначале необходимо задать переменные окружения для `Visual Studio .NET`. Все параметры утилиты **ILDasm** являются необязательными, и в соответствии с обычной практикой использования командной строки можно получить список этих параметров, введя команду `"ildasm /?"`. Если опустить параметры, на экран будет выведено пустое окно, из которого можно получить доступ к нужному файлу, выбрав его с помощью команды меню `File | Open`.

Проще всего работать с командой **ILDasm**, если добавить ее в меню `Tools` (Инструменты) среды разработки `Visual Studio .NET`. Как и в предыдущих версиях `Visual Studio`, разработчик может включать в это меню любое количество своих собственных команд. Выберем пункт `External Tools...` (Внешние инструменты) в меню `Tools`,

щелчком кнопки Add (Добавить) и заполним поля диалогового окна как показано на рисунке:



Эти установки вызывают утилиту ILDasm, передают ей имя выходного файла текущего проекта и задают текущий каталог, в котором располагается этот файл. Если теперь открыть проект MyClock (см. выше в данной главе) и выбрать пункт IL DASM в меню Tools, то программе ILDasm.exe будут переданы имя файла MyClock.dll и исходный каталог Debug. В результате откроется окно:



Для того чтобы сделать результат дизассемблирования сборки более наглядным, программа ILDasm использует древовидную структуру, похожую на ту, что

появляется в окне Class View (Просмотр классов) среды разработки Visual Studio .NET. Элементы дерева отображают детали сборки, информацию о которых среда CLR будет использовать во время выполнения. При этом символы в изображении дерева имеют вполне конкретный смысл:

Символ	Значение
	Дополнительная информация о родительском классе
	Пространство имен
	Класс
	Интерфейс
	Тип-значение
	Перечисление
	Метод
	Статический метод
	Поле
	Статическое поле
	Событие
	Свойство

Если вспомнить код, который мы писали при создании сборки MyClock.dll, то можно заметить поразительное сходство между этим кодом и тем, что выдает на экран утилита ILDasm. Все детали сборки MyClock.dll здесь ясно представлены. Если дважды щелкнуть мышью по любому элементу дерева, можно увидеть дополнительные детали. Например, после щелчка на методе Start() откроется контекстное окно, содержащее IL-версию данного метода:

```
Stopwatch::Start: void()
.method public instance void Start() cil managed
{
    // Code size      22 (0x16)
    .maxstack 2
    .locals (valuetype [mscorlib]System.DateTime U_0,
            valuetype [mscorlib]System.DateTime U_1)
    IL_0000: call        valuetype [mscorlib]System.DateTime [mscorlib]System.DateTime::get_Now()
    IL_0005: stloc.0
    IL_0006: ldloc.0
    IL_0007: stloc.1
    IL_0008: ldarg.0
    IL_0009: ldflda     valuetype [mscorlib]System.DateTime MyClock.Stopwatch::startTime
    IL_000e: ldloc.s    U_1
    IL_0010: cpmovj     [mscorlib]System.DateTime
    IL_0015: ret
} // end of method Stopwatch::Start
```

Декларация сборки

В результате двойного щелчка мышью по элементу **MANIFEST** экран примет следующий вид;

```

<assembly extension="mscorlib"
  .publickeytoken = (B7 7A 5C 5* 19 34 E6 89 )
  .hash = (98 31 71 95 E9 91 51 E5 E2 00 25 55 57 E7 6E 9E 54 8C EF C9 )
  .ver 1:0:3300:0
  >
  <assembly extension="Microsoft.VisualBasic"
  .publickeytoken = (B1 3F 5F 7F 11 D5 0A 3A )
  .hash = (EC 47 1C 7B E2 10 BB EE 15 E2 3B 4E FB 55 44 9D E2 D0 4D B7 )
  .ver 7:0:3300:0
  >
  <assembly MyClock
  .custom instance void [mscorlib]System.Reflection.AssemblyKeyNameAttribute::ctor(string) = ( 01
  .custom instance void [mscorlib]System.Reflection.AssemblyCompanyAttribute::ctor(string) = ( 01
  .custom instance void [mscorlib]System.Reflection.AssemblyProductAttribute::ctor(string) = ( 01
  .custom instance void [mscorlib]System.Reflection.AssemblyDescriptionAttribute::ctor(string) = ( 01
  .custom instance void [mscorlib]System.Reflection.AssemblyConfigurationAttribute::ctor(string) = ( 01
  .custom instance void [mscorlib]System.Reflection.AssemblyCopyrightAttribute::ctor(string) = ( 01
  .custom instance void [mscorlib]System.Reflection.AssemblyTitleAttribute::ctor(string) = ( 01 01
  .custom instance void [mscorlib]System.Reflection.AssemblyTrademarkAttribute::ctor(string) = ( 01
  .custom instance void [mscorlib]System.Reflection.AssemblyDelaySignAttribute::ctor(bool) = ( 01
  >
  </assembly>
  </manifest>

```

Декларация содержит следующую информацию:

- ☐ Имя сборки, представляемое текстовой строкой. В нашем случае сборка носит имя **MyClock**.
- O Номер версии следующего формата:** <старшая_версия>:<младшая_версия>:<номер_компоновки>:<ревизия>.
- ☐ Культура (язык, валюта, форматирование чисел и т.д.), поддерживаемая данной сборкой.
 - Информация о строгом имени. Сюда входит предоставляемое автором сборки "причудливое" имя открытого ключа, которое является частью строгого имени (strong name) сборки. Строгое имя уникально и позволяет идентифицировать автора сборки (подробнее см. ниже).
 - Список файлов, составляющих сборку.
- ☐ Список сборок, на которые делаются ссылки. В него входят все имена, номера версий, а также строгие имена тех сборок, к которым данная сборка должна получать доступ.
- D** Информация о ссылках на типы. Она используется средой CLR для связывания ссылки на тот или иной тип с файлом, где этот тип объявляется и реализуется. Эти данные необходимы для типов, экспортируемых сборкой.
 - Список прав доступа (разрешений). Данный список регламентирует все права доступа к сборке, которые будет иметь среда CLR, чтобы обеспечить корректную работу сборки.

Рассматриваемая нами декларация сборки **MyClock** показывает, что сборка ссылается на такие совместно используемые сборки, как **mscorlib** и **Microsoft.VisualBasic**.

Чтобы гарантировать, что сборка обращается к *корректным* сборкам, декларация содержит следующие поля: `.publickeytoken` (идентифицирует сборку), `.hash` (проверяет, что сборка не была изменена) и `.ver` (содержит номер версии, необходимый для исполнения кода).

Далее в декларации указано имя сборки `MyClock`. Сборка имеет несколько атрибутов, содержащих информацию о ней. Доступ к этой информации можно получить программным образом, используя способ, называемый отражением (*reflection*). Для установки значений атрибутов декларации служит файл `AssemblyInfo.cpp`, генерируемый мастером.

В конце декларации (на рисунке это не отображено) помещается несколько директив ассемблера, используемых средой CLR для JIT-компиляции сборки. Поле `.module` указывает имена всех файлов, составляющих сборку. В нашем случае сборка состоит из одного файла `MyClock.dll`:

```
.module MyClock.dll
```

Файл `AssemblyInfo.cpp`

Детали, относящиеся к сборке, которые содержатся в декларации, непосредственно задаются в файле `AssemblyInfo.cpp`. Этот файл автоматически генерируется мастером проекта:

```
#include "stdafx.h"

using namespace System::Reflection;
using namespace System::Runtime::CompilerServices;

//
// Общая информация, относящаяся к сборке, задается нижеследующим
// набором атрибутов. Для того, чтобы модифицировать эту
// информацию, измените значения атрибутов.
//
[assembly: AssemblyTitle("")];
[assembly: AssemblyDescription("")];
[assembly: AssemblyConfigurationAttribute("")];
[assembly: AssemblyCompanyAttribute("")];
[assembly: AssemblyProductAttribute("")];
[assembly: AssemblyCopyrightAttribute("")];
[assembly: AssemblyTrademarkAttribute("")];
[assembly: AssemblyCultureAttribute("")];

// Информация о версии сборки состоит из следующих четырех величин
//
// Старшая версия
// Младшая версия
// Номер компоновки
// Ревизия
//
// Можно задать все четыре величины или использовать значения по
// умолчанию для ревизии и номера компоновки, используя символ
// *, как показано ниже:
[assembly: AssemblyVersion("1.0.*")];

//
// Чтобы подписать сборку, необходимо задать ключ. Для получения более
// подробной информации о создании подписи сборки, обратитесь
```

```
// к документации по Microsoft .NET Framework
//
// Используйте ниже следующие атрибуты, чтобы задать ключ,
// необходимый для подписи.
//
// Применения:
//
// <*) Если ключ не задан, сборка не будет подписана.
//
// (*) Атрибут KeyName задает ключ, который установлен
// в источнике криптографических служб Crypto Service Provider (CSP)
// на Вашей машине. Атрибут KeyFile указывает файл, содержащий
// ключ.
//
// (*) Если значения атрибутов KeyFile и KeyName заданы,
// то реализуется следующая процедура:
//
// (1) в случае, когда ключ, указанный атрибутом KeyName, найден
// в CSP, будет использован этот ключ.
//
// (2) Если ключ не найден, но файл KeyFile существует, то ключ
// в файле KeyFile устанавливается в CSP и будет использован.
//
// (*) Для создания файла KeyFile можно использовать утилиту
// sn.exe (строгое имя). Задавая KeyFile, необходимо указывать его
// расположение относительно каталога проекта.
//
// (*) Атрибут Delay Signing представляет собой расширенную опцию
// (см. документацию по Microsoft .NET Framework)
//
[assembly:AssemblyDelaySignAttribute(false)];
[assembly:AssemblyKeyFileAttribute("")]
[assembly:AssemblyKeyNameAttribute("")]
```

Этот файл будет компилироваться вместе с другими исходными файлами. Однако результатом компиляции AssemblyInfo. cpp будет не код MSIL (как в случае остальных файлов), а обновление декларации сборки. Если нужно обновить любой из атрибутов декларации, следует просто задать параметр в кавычках. Допустимые значения для атрибута Culture определены в документации RFC 1766 *Tags for the Identification of Languages* (Маркеры для идентификации языков). Атрибут AssemblyVersionAttribute позволяет задать конкретный номер версии. Это имеет очень большое значение для совместно используемых сборок (см. ниже). Каждый из последних трех атрибутов (AssemblyDelaySignAttribute, AssemblyKeyFileAttribute и AssemblyKeyNameAttribute) можно использовать для добавления в сборку строгого имени или открытого ключа. Следующий фрагмент кода иллюстрирует применение атрибутов:

```
[assembly:AssemblyTitleAttribute("ProfessionalC++")];
[assembly:AssemblyDescriptionAttribute("")]
[assembly:AssemblyConfigurationAttribute("RetailVersion")];
[assembly:AssemblyCompanyAttribute("Wrox Press")];
[assembly:AssemblyProductAttribute("Wrox Professional Series")];
[assembly:AssemblyCopyrightAttribute("Copyright (C) Wrox Press 2001")];
[assembly:AssemblyTrademarkAttribute("Wrox is trademark of Wrox Press Ltd")];
[assembly:AssemblyCultureAttribute("en-US")];
```

Закрытые и совместно используемые сборки

Сборки бывают двух видов: закрытые и совместно используемые. Эти широко трактуемые понятия достаточно хорошо описывают ситуацию. Оба типа сборок могут состоять из одного или нескольких модулей, но при этом каждый из них имеет лишь одну декларацию. Фактически и тот и другой тип предоставляют одинаковую функциональность. Единственное различие между ними состоит в том, для каких сборок они доступны.

Закрытые сборки (private assemblies) предназначены исключительно для приложения, которое их использует. Все сборки, созданные нами до сих пор, были закрытыми. Они должны находиться в каталоге приложения или в одном из его подкаталогов. Мы убедились в этом на примере копирования сборки MyClock в каталог приложения TestStopwatch.

Поскольку такие сборки могут использоваться лишь единственным приложением, их именование и задание информации о версиях не создают никаких проблем. Разработчик может не беспокоиться о конфликтах с другими приложениями. Если он придерживается своих собственных соглашений об именах и делает сборки уникальными в пределах приложения, трудностей не возникнет.

С совместно используемыми сборками (shared assemblies) могут одновременно работать несколько приложений. В отличие от закрытых сборок совместно используемые сборки размещаются в **глобальном кэше сборок (Global Assembly Cache - GAC)**. GAC представляет собой область дисковой памяти, доступную всем приложениям на машине и резервируемую только для сборок (об установке совместно используемых сборок см. ниже). Мы уже применяли совместно используемые сборки в нашем коде — все сборки, содержащие реализации классов .NET Framework, являются совместно используемыми и размещаются в GAC (например, `mscorlib.dll`).

Поскольку такие сборки могут использоваться несколькими приложениями, их именование и нумерация версий для них должны выполняться намного более аккуратно. Например, если два разных разработчика создадут сборки с одним и тем же именем, такая ситуация потенциально грозит конфликтом. Чтобы избежать этого, совместно используемые сборки должны также иметь строгое **имя (strong name)**, которое является гарантированно уникальным.

Вполне реален и такой случай, когда двум приложениям на одной и той же машине потребуются различные версии той же самой сборки. Применение совместно используемых сборок позволяет решить эту проблему. Однако, в отличие от COM, сборки не обязательно должны поддерживать обратную совместимость (хотя это и не помешает). Если приложение получит доступ к некорректной версии сборки, оно, скорее всего, не будет нормально работать.

Создание совместно используемых сборок

Для получения совместно используемой сборки вначале создается закрытая сборка, затем добавляется строгое имя, задается (не обязательно) конкретный номер версии, и сама сборка помещается в GAC. У нас уже есть закрытая сборка (Stopwatch). Далее мы преобразуем ее в совместно используемую сборку.

Имена совместно используемых сборок

Совместно используемая сборка должна удовлетворять двум требованиям. Во-первых, необходимо, чтобы ее имя было глобально уникальным. Во-вторых, никто посторонний не должен иметь возможность похитить имя сборки и использовать его (прием, известный как спуфинг). В старом мире COM уникальные имена реализовывались с помощью глобально уникального идентификатора (GUID), однако благодаря этому выполнялось только первое требование, предъявляемое к совместно используемым сборкам. В технологии .NET сборки используют **строгие имена**, которые состоят из трех частей: простого текстового имени, открытого ключа и цифровой подписи.

Технология открытого ключа обеспечивает высокий уровень защиты. Все, что зашифровано с помощью закрытого ключа, может быть расшифровано только с использованием соответствующего ему открытого ключа. До тех пор, пока закрытый ключ остается неизвестным другим разработчикам, никто из них не может создать код, который мог бы быть расшифрован с помощью открытого ключа.

Благодаря использованию ключа и цифровой подписи, мы будем знать, кто является автором сборки, и получим гарантию, что сборка не подвергалась несанкционированным манипуляциям. Вначале компилятор компонует сборку и помещает открытый ключ в декларацию для общего доступа. Затем компилятор создает хэш из всех файлов, принадлежащих сборке, и шифрует его с помощью закрытого ключа. Позже, во время выполнения, среда CLR загружает совместно используемую сборку и проверяет подпись хэша с помощью открытого ключа. Если результат оказывается некорректным, значит, содержимое сборки было изменено — только владелец закрытого ключа может создать корректный хэш.

Обычно для всей организации создается только одна пара, содержащая открытый и закрытый ключи, однако из-за этого возникают **некоторые** проблемы на стадии разработки. Если все программисты организации получают доступ к ключу для тестирования создаваемых ими совместно используемых сборок, уровень защиты снижается. Лучше использовать другой вариант, когда разработчики создают свои собственные ключи, а по окончании работы над сборкой она заново шифруется с ключом организации (см. ниже).

Генерация строгого имени

Создать строгое имя очень просто, хотя для этого нужно воспользоваться новой утилитой Microsoft (утилита SN — strong name). Этот инструмент имеет многочисленные опции командной строки, однако нас интересует только одна из них: опция -k, которая сообщает программе SN, что та должна создать файл ключа сборки, содержащий строгое имя. Можно активизировать утилиту sn.exe из командной строки или добавить пункт меню, запускающий эту утилиту, в меню Tools. В соответствии с соглашением об именах файл ключа, который содержит строгое имя, должен иметь расширение .snk. Поэтому для создания данного файла нужно просто набрать в командной строке такой текст:

```
> an -k StrongName.snk
```

На файл, генерируемый утилитой sn.exe, можно сослаться только из нашего проекта двумя способами: либо задав полное имя файла, включая путь, или же, указав местоположение этого файла относительно каталога проекта.

*При создании совместно используемой сборки с помощью C# следует проявлять осторожность, поскольку такая сборка ведет себя иначе, чем сборка C++. В случае C# SNK-файлы будут находиться в каталоге
%ProjectDirectory%\obj\configuration>.*

Создание подписи сборки

После создания пары ключей нам остается лишь соответствующим образом обновить файл AssemblyInfo.cpp. Для этого изменим атрибут AssemblyKeyFile, указав в его аргументе вместо пустой строки, заданной по умолчанию, имя файла StrongName.snk. Затем зададим значение аргумента атрибута AssemblyDelaySignAttribute, равное false. Это означает, что сборка будет полностью подписана при ее создании. Если указанному параметру присвоить значение true, в сборке будет зарезервировано место для подписи, которое можно заполнить позже с помощью утилиты создания подписи

sn.exe. Наконец, оставим пустым аргумент атрибута `AssemblyKeyNameAttribute` (причину мы скоро объясним). В результате указанные атрибуты примут следующий вид:

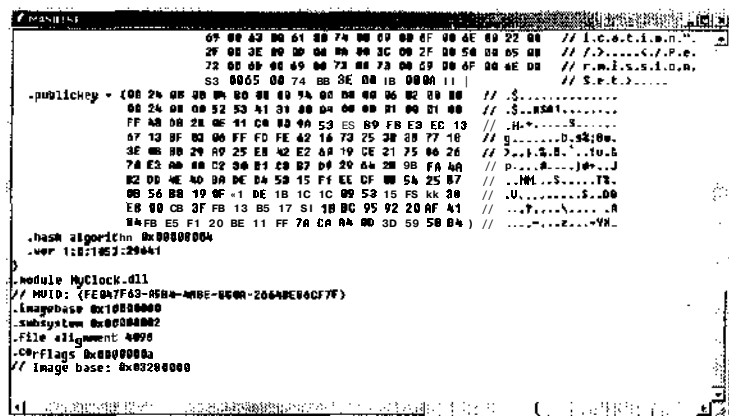
```
[assembly:AssemblyDelaySignAttribute(false)];
[assembly:AssemblyKeyFileAttribute("StrongName.snk")];
[assembly:AssemblyKeyNameAttribute("")];
```

`AssemblyKeyNameAttribute` позволяет получить пару ключей из источника криптографических служб (Cryptographic Service Provider - CSP). Этот атрибут задает имя контейнера ключа в CSP, содержащего пару ключей, которая используется для генерации строгого имени:

```
[assembly:AssemblyDelaySignAttribute(false)];
[assembly:AssemblyKeyFileAttribute("")]
[assembly:AssemblyKeyNameAttribute("MyKeyContainer")];
```

Имеется и третий способ присваивания сборке строгого имени. Если воспользоваться утилитой `AL.exe` для связывания различных модулей и ресурсов и формирования декларации сборки, то нужно будет просто задать имя ключа с помощью опции `/key[file]:имя_файла`.

Применив любой из трех описанных методов добавления ключа, мы увидим, что открытый ключ, содержащийся в `SNK`-файле, появляется в декларации:



Утилита цифровой подписи *Signcode*

Строгое имя сборки является гарантированно уникальным и предотвращает несанкционированное использование имен (спуфинг). Это позволяет администратору определить, является ли сборка подлинной. С другой стороны, наличие строгого имени еще не гарантирует необходимый уровень доверия.

Утилита **Signcode.exe** позволяет разработчику подписать сборку, используя подлинный цифровой сертификат, полученный от стороннего органа сертификации. Цифровая подпись указывает автора сборки, что дает администратору возможность принимать решение о доверии к этой сборке.

В сборку можно добавить одновременно как строгое имя, так и цифровую подпись, либо один из этих компонентов, или же вовсе не использовать их. Однако если сборка должна содержать строгое имя, то оно добавляется в первую очередь. Утилита `Signcode` может одновременно подписывать лишь один файл. При использовании многофайловых сборок должен подписываться файл, содержащий декларацию.

Установка номера версии

Внимательный читатель, возможно, заметил, что в декларацию сборки добавляется номер версии. Файл `AssemblyInfo.cpp` задает и этот параметр.

Номер версии сборки состоит из четырех частей: старшей версии, младшей версии, номера компоновки и ревизии. Для созданной нами сборки `MyClock` номер версии имеет вид: 1.0:764:24274. Обычно изменения старшей и младшей версии означают, что нарушается совместимость. Изменение номера компоновки соответствует *возможному* нарушению совместимости. Если же меняется номер ревизии, то гарантируется совместимость с более ранними версиями.

Разработчик может установить любой номер версии, изменив параметр атрибута `AssemblyVersionAttribute` в файле `AssemblyInfo.cpp`, как показано ниже:

```
[assembly: AssemblyVersionAttribute("1.2.3.4");]
```

Используя установки по умолчанию, мы позволяем компилятору автоматически генерировать ревизию и/или номер компоновки:

```
[assembly: AssemblyVersionAttribute("1.2.*.*");]
```

или:

```
[assembly: AssemblyVersionAttribute("1.2.3.*")];
```

При автоматической генерации последних двух частей номера версии компилятор в качестве номера компоновки использует количество дней, прошедших с 1 января 2000 г. Номером же ревизии в этом случае становится деленное пополам число секунд (определенное по компьютерным часам), которые прошли начиная с полуночи.

Получение номера версии программным образом

Для того чтобы получить доступ к номеру версии сборки `MyClock` из сборки `Test-Stopwatch`, нужно добавить новое свойство `AssemblyVersion` в класс `Stopwatch`. Это свойство возвращает класс `Version`, имеющий некоторые полезные свойства, которые позволяют получить все четыре части версии.

Во-первых, нужно добавить ссылку на пространство имен `System.Reflection`, а также определение нового свойства в файл `MyClock.h`:

```
using namespace System::Reflection;

namespace MyClock
{
    public __gc class Stopwatch
    {
    public:
        „property Version *get_AssemblyVersion();
        // Остальной код
    };
}
```

Пространство имен `System.Reflection` содержит классы, которые облегчают программный доступ к метаданным, таким как информация о версиях (подробнее об этом см. в следующей главе). Теперь добавим реализацию свойства в класс `Stopwatch`:

```
Version *MyClock::Stopwatch::get_AssemblyVersion()
{
    Char del[] = new Char[1];
```

```

del[0] = "
Assembly <assembly = Assembly::GetExecutingAssembly();
String *name[] = assembly->FullName->Split(del);

return new Version(name[1]->Substring(name[1]->IndexOf('=')+1));
}

```

На самом деле свойство `AssemblyVersion` менее сложно, чем это может показаться. Вначале мы получаем исполняемую сборку, а затем — ее полное имя. Последнее представляет собой строку, включающую все составные части имени сборки, разделенные запятыми. Вторая часть этой строки представляет собой номер версии в виде: `Version= <старшая_версия>, <младшая_версия>, <номер_компоновки>, <ревизия>`. После получения полного имени мы отщепляем данные о версии с помощью метода `Split()`, а затем, используя метод `Substring()`, получаем фактический номер. Далее создаем новый экземпляр типа `Version` с помощью конструктора `new`.

Теперь для получения номера версии из сборки `TestStopwatch` необходимо обратиться к свойству `AssemblyVersion`:

```

int _tmain(void)
{
    Stopwatch Swatch = *new Stopwatch; // Ссылка на объект Stopwatch
    // Остальной код
    Console.WriteLine(watch.AssemblyVersion);
}

```

Если скомпоновать и запустить этот код, то в дополнение к той информации, которая выводилась на экран ранее, теперь будет отображаться и номер версии сборки `MyClock`.

Глобальный кэш сборок

Как свидетельствует само название, **глобальный кэш сборки (Global Assembly Cache, GAC)** представляет собой кэш, где хранятся сборки. Это звучит достаточно претенциозно, поскольку фактически GAC представляет собой всего лишь каталог (с открытым доступом и фиксированным местоположением), в котором сохраняются совместно используемые сборки. Сборки, помещаемые в GAC, могут иметь три различных формата:

- ☐ Машинонезависимый код, т.е. MSIL
- ☐ Собственный машинный код
- ☐ Кроме того, ASP.NET позволяет загружать сборки так, что они могут использоваться в страницах HTML. Поскольку ASP.NET автоматически не поддерживает C++, мы не рассматриваем этот вопрос в настоящей книге.

Глобальный кэш сборки представляет собой фиксированный каталог `<WINDIR>\assembly`. В большинстве систем этот каталог имеет вид: `C:\WinNT\assembly`. Поскольку это местоположение гарантировано, CLR не испытывает проблем при поиске совместно используемых сборок, необходимых для исполнения того или иного приложения.

Средство просмотра глобального кэша сборки

Для того чтобы увидеть все совместно используемые сборки, хранящиеся в системе, можно просто перейти в каталог `assembly` с помощью Windows Explorer:

Средство просмотра сборок достаточно изящно, однако оно скрывает от пользователя многие детали сборки (а также истинную структуру каталога). Единственным средством, позволяющим отобразить структуру каталога GAC, является командная строка. К счастью, фактическую структуру каталога и точное местоположение совместно используемой сборки можно получить на основе информации, предоставляемой средством просмотра:

```
<WINDIR>\assembly\GAC\<имя_сборки>\<версия>__<маркер_ключа>
```

В системе автора каталог совместно используемой сборки System.Xml имеет вид:

```
C:\WINNT\assembly\GAC\System.Xml\1.0.3300.0_b77a5c561934e089
```

В папке <WINDIR>\assembly имеется два других главных каталога. Первый называется Download и содержит все загружаемые сборки для использования в страницах HTML. Во втором каталоге Native Image сохраняются сборки, откомпилированные в собственный машинный код (структуру этого каталога нельзя определить с помощью Windows Explorer). Разработчик не должен иметь дело ни с одной из указанных структур каталогов непосредственно. С первым каталогом позволяет работать ASP.NET, а со вторым — генератор собственного кода (см. ниже).

Инсталляция совместно используемой сборки в GAC

После того как сборка подписана и задано ее строгое имя, она может быть установлена в GAC двумя способами, в результате чего будет создана совместно используемая сборка. Автор применяет только один из этих способов, состоящий в простом перетаскивании сборки мышью в средство просмотра GAC. Нет ничего проще (удалить совместно используемую сборку так же легко — для этого нужно выделить ее в окне средства просмотра и нажать клавишу *Delete*),

Второй способ заключается в использовании утилиты глобального кэша сборок gacutil.exe. Это средство не предоставляет никакой дополнительной функциональности по сравнению со средством просмотра GAC, однако, поскольку оно активируется из командной строки, его можно использовать в пакетных программах. Утилита gacutil.exe поддерживает несколько опций, однако чаще всего будут востребованы две из них, применяемые для инсталляции и деинсталляции. Сборка устанавливается с помощью команды:

```
> gacutil /i <имя_сборки>.dll
```

Команда, удаляющая сборку из кэша, имеет вид:

```
> gacutil /u <имя_сборки>, Version=<номер_версии>
```

Читатель может подумать, что описанные процедуры усложняются в случае перемещения в GAC многофайловых сборок. Однако на самом деле это не так. Чтобы переместить многофайловую сборку, необходимо перетащить мышью DLL-файл сборки (содержащий декларацию) в средство просмотра GAC или вызвать утилиту gacutil.exe для того же самого DLL-файла. После выполнения команды все остальные файлы, составляющие сборку, также автоматически перемещаются в GAC.

Генерация собственного машинного кода

Предварительно скомпилированными (precompiled) являются сборки, которые откомпилированы в собственный машинный код той платформы, на которой они должны исполняться. Поскольку такие сборки не нужно компилировать перед исполнением, они запускаются намного быстрее при вызове средой CLR. Если

нужно ускорить время запуска, применение предварительно скомпилированных сборок является удачным решением. Однако подобные сборки занимают больше дискового пространства, поскольку GAC требует, чтобы любая сборка в кэше собственного машинного кода (область GAC, куда генератор собственного кода помещает предварительно скомпилированные сборки) имела копию в стандартном GAC. Дополнительная копия используется для извлечения из нее метаданных, а также компилируется в тех случаях, когда проверка версий или изменений в защите дает некорректный результат.

В настоящий момент отсутствует автоматизированный способ предварительной компиляции сборок. Такую компиляцию необходимо делать вручную при установке сборок в GAC. Для создания машинного кода необходимо использовать утилиту генератора собственного кода ngen.exe. Как и в случае других утилит, данная программа поддерживает несколько опций, однако, если задать только одно имя сборки, в GAC будет сгенерирован машинный код:

```
> ngen MyClock
```

Теперь мы должны увидеть две версии сборки MyClock в GAC, причем для одной из них указывается тип Native Image (собственный код). Сборка данного типа сохраняется в GAC до ее удаления. Для удаления сборки следует снова запустить программу ngen.exe и задать опцию /delete вместе с именем сборки:

```
> ngen /delete MyClock
```

Работа с совместно используемой сборкой

Теперь воспользуемся сборкой приложения TestStopwatch, чтобы сослаться на совместно используемую версию библиотечной сборки MyClock. Для этого, во-первых, необходимо удалить файл MyClock.dll из каталогов проекта Debug и Release. Если этого не сделать, программа TestStopwatch.exe попытается искать сборку в указанных каталогах, а не в GAC. Кроме того, следует удалить (или закомментировать) все созданные нами ранее события компоновки в проекте TestStopwatch, которые вновь приводили бы к копированию файла MyClock.dll.

Если после этого откомпилировать и запустить проект TestStopwatch, на экран будет выведена такая же информация, какую мы получили при запуске закрытой версии сборки;

```
Interval time in milliseconds: 1863 (Интервал времени в миллисекундах - 1863)
End time in milliseconds: 3916 (Время завершения в миллисекундах - 3916)
Reset stopwatch elapsed time in milliseconds: 12363
(Время в миллисекундах, прошедшее с момента перезапуска секундомера, -12363)
1.0.764.24896
```

Замена ключа строгого имени на версию для распространения

Когда, наконец, совместно используемая сборка полностью протестирована и готова к развертыванию, необходимо подписать ее, используя пару ключей компании (предполагается, что до этого момента вся разработка проводилась с парой ключей программиста). Создание ключей компании и разработчика выполняется одинаково. Различие состоит лишь в том, кто имеет доступ к этим ключам:

```
> sn -k CompanyStrongNameKey.snk
```

Теперь, когда привилегированный пользователь имеет совместно используемую сборку и ключ организации, он должен лишь еще раз запустить утилиту строгого имени sn.exe, на этот раз в режиме замены:

```
> sn -R <имя_сборки> <имя_файла_ключа>
```

Опция `-R` указывает, что будет заменено строгое имя в той сборке, имя которой следует далее. После этого нужно только установить заново подписанную совместно используемую сборку в GAC. В результате среда CLR получит гарантию, что сборка с новой подписью является подлинной и что она не подвергалась никаким изменениям.

Поддержка управления версиями

Теперь вернемся назад и продемонстрируем в действии возможности .NET, предоставляемые для поддержки управления версиями. Следует иметь в виду, что сборки ссылаются друг на друга, используя конкретную версию. Поскольку в GAC уже есть одна копия сборки `MyClock.dll`, на которую ссылается сборка `TestStopwatch.exe`, необходимо лишь добавить в глобальный кэш еще одну версию `MyClock.dll` и сослаться на нее из другой сборки. Простейший способ сделать это заключается в перекомпиляции обоих проектов в режиме, создающем версию приложения для распространения, а не для отладки. Начнем с проекта `MyClock` и скопируем вновь созданную сборку `MyClock.dll` в GAC, используя средство просмотра GAC. В результате мы должны увидеть две сборки `MyClock.dll` с двумя различными версиями;

Global Ass...	Type	Version	Culture	Public Key Token
MyMSDDSLMP		7.0.3300.0		b03f5f7f11d50a3a
MyMSDDSP		7.0.3300.0		b03f5f7f11d50a3a
MyClock		1.0.1052.26049		f64921db005adfcb
MyClock		1.0.1052.24691		f64921db005adfcb
Regcode		1.0.3300.0		b03f5f7f11d50a3a
SoapSudsCode		1.0.3300.0		b03f5f7f11d50a3a
Stdole		7.0.3300.0		b03e5f7f11d50a3a

Далее перекомпилируем клиентское приложение `TestStopwatch`, которое теперь должно ссылаться на версию для распространения `MyClock`. Чтобы посмотреть, как выполняется управление версиями в .NET, запустим версии приложения `TestStopwatch.exe`, предназначенные для отладки и для распространения. Мы увидим, что они обращаются к сборкам с различными версиями — каждое приложение ссылается на ту версию, с которой оно компилировалось:

```

Command Prompt
C:\>cd C:\code\assemblies\TestStopwatch\Debug\
C:\code\assemblies\TestStopwatch\Debug>TestStopwatch
Interval time in milliseconds: 951
End time in milliseconds: 2083
Reset stopwatch elapsed time in milliseconds: 1182
1.0.1052.24691
C:\code\assemblies\TestStopwatch\Debug>

```

```

Command Prompt
C:\>cd C:\code\assemblies\TestStopwatch\Release\
C:\code\assemblies\TestStopwatch\Release>TestStopwatch
Interval time in milliseconds: 1011
End time in milliseconds: 2333
Reset stopwatch elapsed time in milliseconds: 931
1.0.1052.26049
C:\code\assemblies\TestStopwatch\Release>

```

Работа с ресурсами

Базовые классы .NET Framework предоставляют большие возможности для работы с ресурсами, к которым относятся изображения, пиктограммы, курсоры, строки и т.д. К сожалению, для реализации этих возможностей программистам C++ приходится приложить больше усилий, чем разработчикам, использующим другие .NET-совместимые языки. Платформа .NET поддерживает три способа работы с ресурсами:

- Встраивание файла (файлов) ресурсов непосредственно в сборку.
- Работа с файлом ресурсов, внешним по отношению к сборке. Это позволяет организовывать *динамические* файлы ресурсов (можно создавать и затем использовать внешний файл ресурсов во время выполнения приложения).
- Размещение файлов ресурсов в сопутствующих сборках, т.е. в таких, которые содержат только ресурсы. Подобные сборки могут использоваться для локализации приложений или для создания специфических ресурсов, относящихся к языку и основанных на культуре, реализующей сборку.

Создание файла ресурсов

Существует два редактируемых формата текстовых файлов, которые могут быть использованы для создания файла ресурсов: XML и пара "имя-значение". XML-файлы ресурсов имеют расширения .resX. Расширение файлов типа "имя-значение" может быть любым, однако обычно, в соответствии с имеющимся соглашением, используется расширение .txt.

Для программистов C++ нет необходимости работать с .resX-файлами, которые используются для сохранения ресурсов форм Windows и Web, создаваемых в среде разработки Visual Studio.NET. Язык Visual C++ .NET не поддерживает формы Web, а формы Windows позволяет создавать лишь вручную с минимальным участием IDE (см. главу 6). Поскольку формат .resX-файлов достаточно сложен (намного сложнее, чем для файлов типа "имя-значение") и не поддерживается IDE C++, мы не будем его обсуждать.

В файлы ресурсов помещаются рисунки, таблицы строк и многие другие объекты. Можно создать файл с расширением .resources, включающий один тип ресурсов либо несколько различных типов. Простейшим видом ресурса является таблица строк, преимущественно используемая для многоязычных приложений (на этот раз имеются в виду языки человеческого общения). Для создания такого ресурса можно использовать простой текстовый файл, содержащий пары "имя-значение":

```
имя = значение
```

Как имя, так и значение могут включать пробелы. Однако, если нужно выровнять знаки равенства (не вставлять лишних пробелов), следует проявить осторожность, поскольку дополнительные пробелы станут частью имени. Знак равенства может быть и частью значения — для форматирования важен лишь первый такой знак.

Утилита ResGen

Сам по себе текстовый файл не является файлом ресурсов. Его необходимо преобразовать в двоичный файл с расширением .resources, который будет использоваться средой CLR. Для выполнения такого преобразования платформа .NET Framework предоставляет утилиту resgen.exe. Например, чтобы преобразовать файл test.txt, в командной строке следует просто напечатать:

```
> resgen test.txt
```

После выполнения этой программы мы получим в текущем каталоге файл ресурсов с именем `test.resources`. Он может быть встроен в сборку приложения, в библиотечную либо в сопутствующую сборку, и, кроме того, с ним можно работать как с отдельным файлом. Позже мы покажем, каким образом файл ресурсов встраивается в сборку.

Класс `ResourceWriter`

К сожалению, утилита `resgen.exe` не поддерживает рисунки и другие объекты, не относящиеся к текстовому формату. Однако добавить такие объекты достаточно легко. Для этого служит класс `ResourceWriter`, относящийся к базовым классам .NET Framework. Для создания файла ресурсов, содержащего рисунки, нужно написать очень простой код. Ниже приведен весь код, необходимый для того, чтобы сгенерировать файл ресурсов с изображением логотипа издательства Wrox:

```
#using <mscorlib.dll>
#using <System.Drawing.dll>

using namespace System;
using namespace System::Resources;
using namespace System::Drawing;

int main(void)
{
    ResourceWriter *rw = new ResourceWriter("WroxLogo.resources");
    rw->AddResource("WroxLogo", Image::FromFile("wroxlogo.gif"));
    rw->Close();

    return 0;
}
```

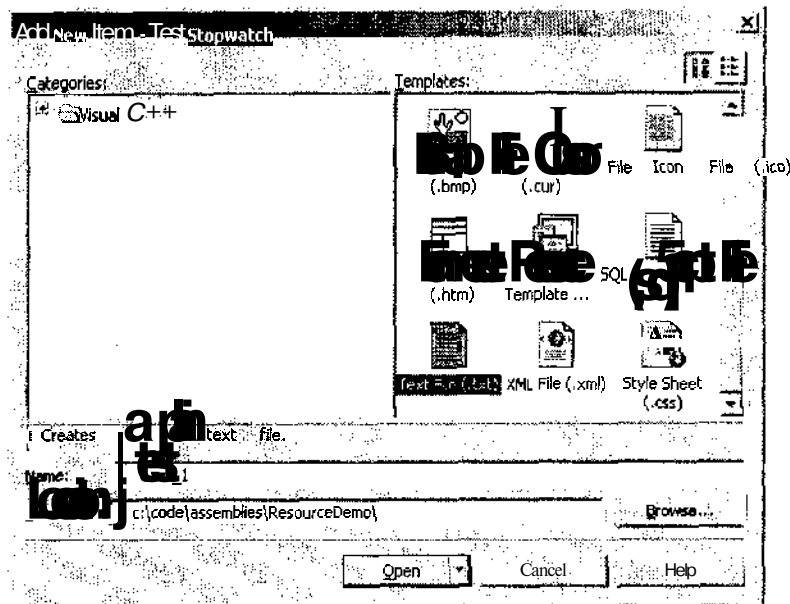
Здесь мы создаем экземпляр класса `ResourceWriter`, добавляем в него ресурсы с помощью метода `AddResource()` и закрываем его. Первый параметр перегруженного метода `AddResource()` всегда указывает имя ресурса, а второй параметр есть не что иное, как сам ресурс. Он может представлять собой строку, байтовый массив или родовой объект. Такие встроенные многоцелевые возможности означают, что указанный метод позволяет преобразовать в файл ресурсов практически все, что угодно, если полный размер файла не превышает 2 Гб. Если скомпоновать и запустить приведенный выше код, будет сгенерирован двоичный файл ресурсов `WroxLogo.resources`, предназначенный для работы в среде CLR.

Создание файлов ресурсов с помощью Visual Studio .NET

Обращение к командной строке всякий раз, когда требуется создавать файлы ресурсов, не очень-то удобно. Однако, хотя среда Visual Studio .NET автоматически не работает с ресурсами, можно использовать ее для этих целей, несколько изменив свойства.

Создадим приложение, которое демонстрирует методы работы с ресурсами всех типов. Начнем с простых встроенных ресурсов, затем сделаем небольшое отступление, чтобы проиллюстрировать, как выполняется локализация, и, наконец, обсудим использование динамических внешних ресурсов. В качестве внешнего ресурса будет служить файл изображения, загружаемого в форму Windows.

Вначале запустим Visual Studio .NET и создадим управляемое приложение Visual C++ (Managed C++ Application) с именем ResourceDemo. Затем добавим в проект три новых текстовых файла: test_1.txt, test_1.fr.txt и test_2.txt. Это можно сделать, щелкнув правой кнопкой мыши на имени проекта и выбрав в открывающемся контекстном меню пункт Add > Add New Item... (Добавить > Добавить новый элемент):



Указанные файлы будут содержать пары "имя-значение" для файлов ресурсов. Необходимо преобразовать их в файлы с расширением .resources, прежде чем использовать механизм управления ресурсами, предоставляемый платформой .NET. Очевидно, что ничто не мешает работать с ними, как с обычными текстовыми файлами. Добавим следующий текст в test_1.txt:

```
First = The first text.
Second = The second text = Equals included (works!)
Third = The third text.
Fourth = The fourth text.
```

Файл test_1.txt будет встроен в сборку ResourceDemo. В файл test_1.fr.txt добавим такой текст:

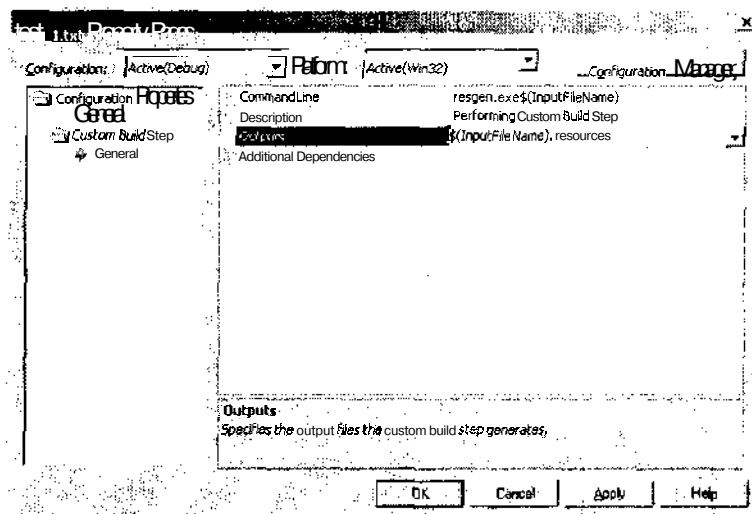
```
First = Le texte premiere.
Second = Le texte deuxieme.
Third = Le texte troisieme.
Fourth = Le texte quatrieme.
```

Мы преобразуем этот файл в сопутствующую сборку. Наконец, изменим файл test_2.txt так, чтобы его содержимое имело вид:

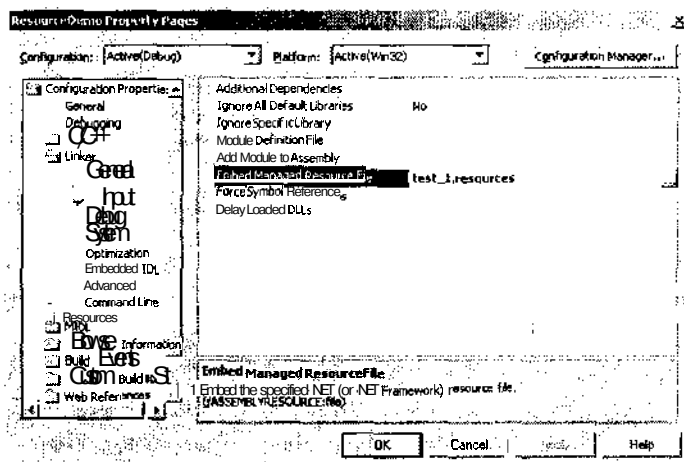
```
Fifth = The fifth text.
Sixth = The sixth text.
Seventh = The seventh text.
Eighth = The eighth text.
```

Этот файл будет преобразован во внешний файл ресурсов test_2.resources.

Для того чтобы среда разработки Visual Studio .NET могла преобразовывать эти файлы автоматически всякий раз, когда они изменяются, программисту следует приложить некоторые усилия. Однако идея здесь достаточно проста. Нужно активизировать утилиту `resgen.exe` на пользовательском этапе компоновки (custom build step) для каждого из текстовых файлов. Выберем файл `test_1.txt` и откроем диалоговое окно его свойств. Затем в папке **Custom Build Step** выберем элемент **General** и введем в поле **Command Line** (Командная строка) текст `ResGen.exe $(InputFileName)`, а в поле **Outputs** (Выходные файлы) — текст `$(InputName).resources`:



Повторим ту же самую процедуру для файлов `test_1_fr.txt` и `test_2.txt`. Теперь у нас есть три текстовых файла, которые будут конвертироваться в файлы ресурсов при каждой компоновке. Далее необходимо встроить файл `test_1.resources` в сборку `ResourceDemo.exe` с помощью диалогового окна свойств приложения. Для этого в папке **Linker** (Компоновщик) нужно выбрать элемент **Input** (Входные данные), после чего добавить имя файла `test_1.resources` в поле **Embed Managed Resource File** (Встроить управляемый файл ресурсов):



Можно убедиться, что эта операция выполнена успешно, так как после новой компоновки проекта в декларации сборки ResourceDemo появится следующая запись:

```
resource public test_1.resources
```

Локализация сборок

Мы уже создали файл ресурсов, содержащий французский эквивалент файла test_1.resources. Было бы очень удобно, если бы приложение автоматически переходило на французский язык при настройке компьютера на французскую культуру. Реализация такой возможности называется локализацией. К сожалению, сделать это средствами C++ сложнее, чем с помощью языков Visual Basic .NET или C#. Для этих двух языков решение подобной задачи сводится к установке свойства, задающего ту культуру, в рамках которой требуется разрабатывать приложение, и к последующему изменению пользовательского интерфейса, как это делалось при выборе языка, задаваемого по умолчанию. В результате среда разработки управляет в фоновом режиме всеми механизмами, относящимися к ресурсам, и программисту не приходится делать это вручную. Однако в случае C++ такая функциональность не поддерживается, поэтому управлять ресурсами приходится самому разработчику. Тем не менее с помощью C++ можно реализовать те же функции, относящиеся к управлению ресурсами, какие осуществляются другими языками неявным образом.

Позднее мы покажем, как осуществляется доступ к ресурсам, а вначале обратимся к вопросу о том, как создавать файлы ресурсов для выполнения локализации. Начнем с создания текстового файла, который содержит те же самые пары "имя-значение", что и в культуре по умолчанию, причем все значения должны быть переведены. Если их не перевести, в локализованной версии будут автоматически употребляться значения, принятые в культуре по умолчанию. Если, например, убрать вторую пару "имя-значение" из файла test1_fr.txt, то по умолчанию будет выводиться английский текст. Позже, чтобы убедиться в этом, мы удалим указанную строку из файла.

Для локализации приложения необходимо создать новый подкаталог в том месте, где находится сборка приложения. Имя подкаталога должно совпадать с маркером французской культуры ("fr"), заданным в спецификации RFC 1766. Добавим папку fr в отладочный каталог проекта ResourceDemo. В эту папку необходимо поместить нашу сопутствующую сборку, соответствующую французской культуре. Зададим имя сборки ResourceDemo.resources.dll.

Для создания сопутствующей сборки необходимо запустить компоновщик сборки (Assembly Linker, AL.exe). При этом следует задать несколько параметров: имя создаваемой сборки, ее версию, культуру, а также параметры встраивания ресурса в сборку:

```
> al /out:fr/ResourceDemo.resources.dll /version:1.0.0.0 /culture:fr  
/embedresource:test_1.fr.resources,test_1.fr.resources,Private
```

Можно добавить эту команду в событие проекта, наступающее после компоновки (post-build event):

```
> al/out:$(OutDir)/fr/ResourceDemo.resources.dll /v:1.0.0.0/c:fr  
/embed:test_1.fr.resources,test_1.fr.resources,Private
```

Эта команда автоматически обращается к файлу ресурсов, предварительно созданному Visual Studio .NET, и встраивает сопутствующую сборку в нужный каталог.

Доступ к файлам ресурсов

После встраивания ресурса в сборку или сохранения его как самостоятельного файла необходимо получить к нему доступ. Для этого платформа .NET Framework предоставляет два класса:

- `ResourceManager` — считывает ресурсы непосредственно из сборки.
- `ResourceReader` — реализует последовательный доступ к файлу ресурсов, извлекая ресурсы оттуда.

Класс `ResourceManager`

Для того чтобы создать экземпляр класса `ResourceManager`, нужно определить имя исполняемой сборки и использовать его вместе с именем файла ресурсов, к которому мы хотим получить доступ. Затем с помощью указанного экземпляра класса можно запросить значение ресурса, задав имя, соответствующее этому значению. Основная проблема здесь состоит в получении имени исполняемой сборки. К счастью, для решения этой проблемы платформа .NET Framework предоставляет статический метод `GetExecutingAssembly()`, принадлежащий к классу `System.Reflection.Assembly`. Приведенный ниже код демонстрирует, насколько легко можно получить доступ к файлу ресурсов.

// Файл ResourceDemo.cpp

```
((include "stdafx.h"
#using <mscorlib.dll>
```

```
using namespace System;
using namespace System::Reflection;
using namespace System::Resources;
```

```
int main(void)
```

```
{
```

```
// Считывание ресурсов непосредственно из сборки
Assembly* assembly = Assembly::GetExecutingAssembly();
ResourceManager* rm = new ResourceManager(S"test_1", assembly);
```

```
Console::WriteLine(rm->GetObject(S"First"));
```

```
Console::WriteLine(rm->GetObject(S"Second"));
```

```
Console::WriteLine(rm->GetObject(S"Third"));
```

```
Console::WriteLine(rm->GetObject(S"Fourth"));
```

```
// Остальной код (объясняется в последующих разделах)
```

```
}
```

Локализация приложений с помощью класса `ResourceManager`

Класс `ResourceManager` реализует дополнительные возможности, позволяющие автоматически управлять процессом локализации с помощью свойства `CurrentUICulture` текущего потока, к которому можно обратиться благодаря статическому методу `Thread::Thread::CurrentThread`. Чтобы изменить значение свойства `CurrentUICulture`, присвоим ему экземпляр объекта `CultureInfo`, который принадлежит к пространству имен `System.Globalization`. При инициализации `CultureInfo` задается строковое значение, представляющее собой маркер культуры. Описанная процедура позволяет получить объекты из файла ресурсов, как было сделано выше:

```
- // Доступ к ресурсам существующей сборки
```

```
Globalization::CultureInfo* cultureInfo =
```

```

        new Globalization.CultureInfo("fr");
        Threading.Thread.CurrentThread.CurrentCulture = cultureInfo;

        Console.WriteLine(rm->GetObject($"First"));
        Console.WriteLine(rm->GetObject($"Second"));
        Console.WriteLine(rm->GetObject($"Third"));
        Console.WriteLine(rm->GetObject($"Fourth"));

```

Приложение прекратит свою работу, если не сможет найти сопутствующую сборку для задаваемой культуры. Однако при этом нет необходимости дублировать все объекты. Если класс `ResourceManager` не находит объект ресурса в сопутствующей сборке, он извлечет этот объект из файла ресурса по умолчанию.

Класс `ResourceReader`

Работать с внешним файлом ресурсов можно двумя способами, в частности:

- Использовать статический метод `CreateFileBasedResourceManager` класса `ResourceManager` для создания объекта `ResourceManager`
- Применить класс `ResourceReader`

Рассмотрим класс `ResourceReader`. Этот класс позволяет работать с файлом ресурсов, как с любым другим последовательным файлом, и считывать одновременно лишь одну пару "имя-значение". Потенциальная проблема здесь заключается в том, что порядок считывания может отличаться от порядка записи (см. ниже).

Базовая последовательность операций при использовании класса `ResourceReader` та же самая, что и для любого другого потока (при желании можно использовать поток вместо файла ресурсов). Мы открываем файл ресурсов, создаем перечислитель, выполняем считывание или применяем метод `MoveNext()` для перемещения по парам "имя-значение" и закрываем файл ресурсов. Эту процедуру демонстрирует код, приведенный ниже:

```

// Доступ к внешнему файлу ресурсов
ResourceReader* rr = new ResourceReader(S"test_2.resources");
Collections.IDictionaryEnumerator* de = rr->GetEnumerator();
while(de->MoveNext())
{
    Console.WriteLine(de->Key);
    Console.WriteLine(S" = ");
    Console.WriteLine(de->Value);
}
rr->Close();

```

В главе 5 мы рассмотрим применение перечислителей для списков и коллекций, использующих классы коллекций .NET Framework.

Динамические ресурсы

Завершая обсуждение проекта `ResourceDemo`, продемонстрируем, каким образом можно динамически создать файл ресурсов, содержащий рисунок, и затем отобразить этот рисунок в простой форме Windows. Выше уже был приведен код, динамически создающий файл ресурсов. Новым элементом здесь является вызов формы Windows:

```

ResourceWriter* rw = new ResourceWriter(S"test_3.resources");
rw->AddResource(S"WroxLogo", Image.FromFile(S"wroxlogo.gif"));
rw->Close();

```

```
// Доступ к динамически созданному файлу ресурсов, содержащему
// изображение
Application.Run(new WinForm());
```

Ниже представлен код реализации класса `WinForm`, производного от класса `Form`. В конструкторе класса `WinForm` мы получаем доступ к файлу ресурсов, создаем графическое окно и задаем его свойства, необходимые для того, чтобы поместить туда рисунок. Затем код изменяет размер формы и добавляет на нее рисунок:

```
using System;
using System.Windows.Forms;

public class WinForm : public Form
{
    public WinForm()
    {
        ResourceManager *rm =
            ResourceManager::CreateFileBasedResourceManager("test_3", "", 0);
        PictureBox *pictureBox = new PictureBox();
        pictureBox->Image = dynamic_cast<Image*>(rm->GetObject("WroxLogo"));
        pictureBox->Location = Point(16, 8);
        pictureBox->Size = Drawing::Size(64, 72);

        this->ClientSize = Drawing::Size(100, 80);
        this->Controls->Add(pictureBox);
    }
};
```

Для работы с классами, позволяющими создавать формы `Windows`, необходимо импортировать две новые сборки и добавить в начало файла некоторые директивы `using namespace`:

```
#include "stdafx.h"

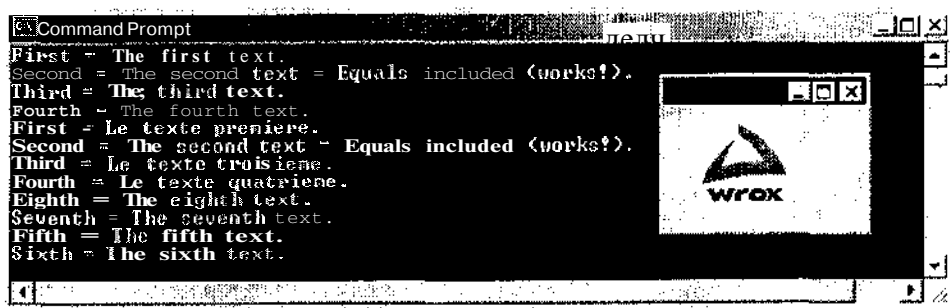
(fusing <mscorlib.dll>
#include <tchar.h>
(fusing <System.dll>
#using <System.Windows.Forms.dll>
(fusing <System.Drawing.dll>

using namespace System;
using namespace System::Resources;
using namespace System::Windows::Forms;
using namespace System::Drawing;
```

Из приведенного кода видно, что процесс создания формы или окна при работе с платформой `.NET` значительно проще, чем при использовании традиционных средств `C++`. Примеры работы с формами мы рассмотрим в главе 6.

Нам потребовалось квалифицировать имя класса `Size`, добавив к нему название пространства имен `Drawing` несмотря на то, что было объявлено пространство имен `System::Drawing`. Это сделано для того, чтобы отличать указанный класс от свойства `Size` класса `Form`.

Теперь, после полного завершения разработки проекта `ResourceDemo`, скомпилируем и запустим его. На следующем рисунке представлен окончательный вариант выводимой на экран информации:



При использовании класса `ResourceReader` необходимо проявлять осторожность. Дело в том, что последние четыре строки выводятся не в том порядке, в каком они располагаются в файле ресурсов. Вследствие этого нужно проверять имя объекта, прежде чем делать вывод о том, что это требуемый объект.

Развертывание сборок

При развертывании приложений и элементов управления, написанных для работы в среде .NET, следует в первую очередь убедиться, что на том компьютере, где должны исполняться созданные программы, установлена платформа .NET Framework. Имеются две свободно распространяемые версии платформы .NET Framework:

- Полная версия (fullversion) включает в себя всю платформу .NET Framework, кроме элементов, задействованных во время разработки. Эта версия используется для запуска полномасштабных приложений и элементов управления.
- Версия, предназначенная для работы элементов управления (control version), представляет собой вариант .NET Framework с минимальной функциональностью. Она разработана только для поддержки элементов управления .NET, содержащихся в браузере.

Для наших целей нужна полная версия, поскольку C++ не поддерживает форм Web и элементов управления. Консольные программы, приложения, разработанные на основе форм Windows, а также Web-службы, которые могут быть созданы с помощью кода C++, требуют полной версии .NET Framework.

После инсталляции платформы .NET Framework на целевую машину развертывание приложения сводится к копированию на эту машину тех сборок, которые необходимы для его работы. Такое копирование можно выполнить тремя способами — в зависимости от метода распространения и сложности приложения. Самый простой способ состоит в применении команд `xcopy.exe` или `unzip.exe` для пересылки каталога приложения на целевую машину. Местоположение этого каталога на машине может быть *любым*., поскольку закрытые сборки должны быть расположены в каталоге приложения или в специальном подкаталоге, который определяется культурой. Ясно, что этот вариант можно использовать только при необходимости поместить в CAS совместно используемые сборки, однако в этом случае ничто не мешает написать простую пакетную команду для выполнения копирования.

Второй способ развертывания приложения заключается в том, чтобы сделать его доступным для загрузки через Web. Проще всего это можно реализовать с помощью САВ-файлов, на которые накладываются следующие ограничения:

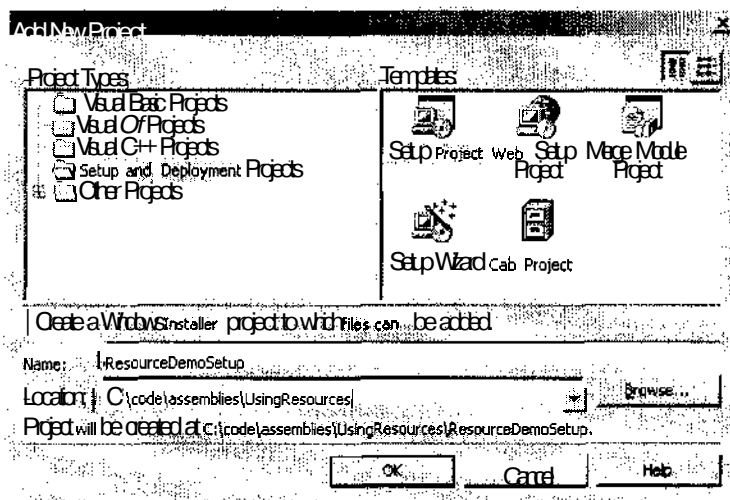
- В CAB-файл может быть помещена только одна сборка.
- Имя CAB-файла должно совпадать с именем сборки (естественно, это не относится к расширениям). Например, если сборка называется `lotafunc.dll`, CAB-файл должен иметь имя `lotafunc.cab`.

Среда CLR не выполняет поиск CAB-файлов для тех сборок, на которые делаются ссылки, поэтому в процессе инсталляции CAB-файлы должны распаковываться.

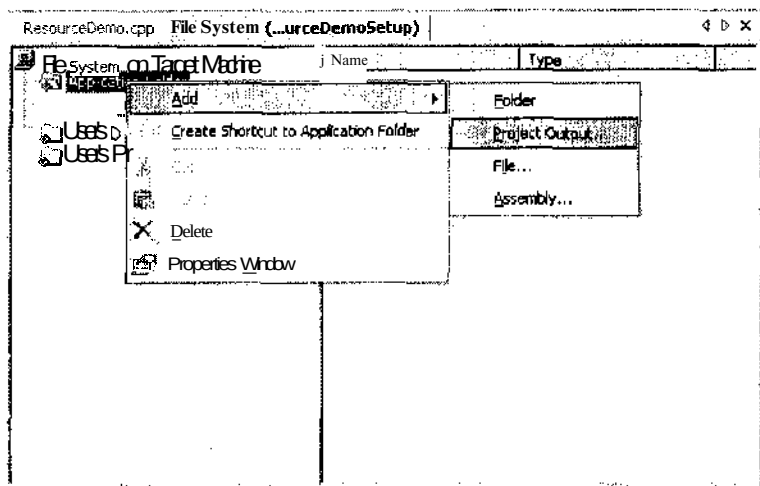
Важно подчеркнуть, что копирование сборок на целевую машину реально работает лишь в случае чистого управляемого кода. Вполне вероятно, что большинство управляемых приложений C++ будет также включать *неуправляемый* код. Один из основных мотивов использования управляемого кода C++ заключается в том, что этот код реализует взаимодействие между существующим неуправляемым кодом и управляемой средой .NET. Если включить в приложение неуправляемый код, оно, скорее всего, будет содержать зависимости от внешних библиотек, а также требовать ряда дополнительных манипуляций, таких как регистрация COM-компонентов. В подобных случаях потребуются более изощренные методы инсталляции, включающие использование Windows Installer и применение CD или DVD.

Последний метод предоставляет максимальную функциональность. Можно автоматизировать весь процесс, включая исправление и удаление сборок в зависимости от даты. Этот метод позволяет также задавать другие требования, относящиеся к процессу инсталляции. Пользователь, устанавливающий ПО, должен будет лишь вставить CD-ROM в устройство чтения и ответить на несколько простых вопросов.

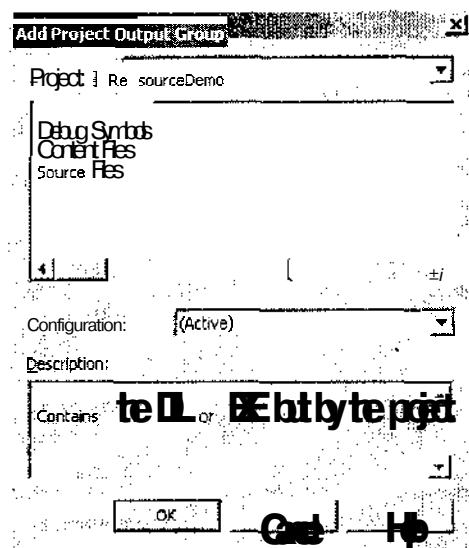
С помощью Windows Installer можно автоматизировать копирование всех совместно используемых сборок. Неважно, является ли сборка локальной для приложения или располагаются в GAC. Если сборка предназначена для копирования в GAC, Windows Installer автоматически запустит утилиту `gacutil.exe`. Для создания инсталляционного файла необходимо добавить проект установки (Setup Project) к текущему проекту, как показано ниже:



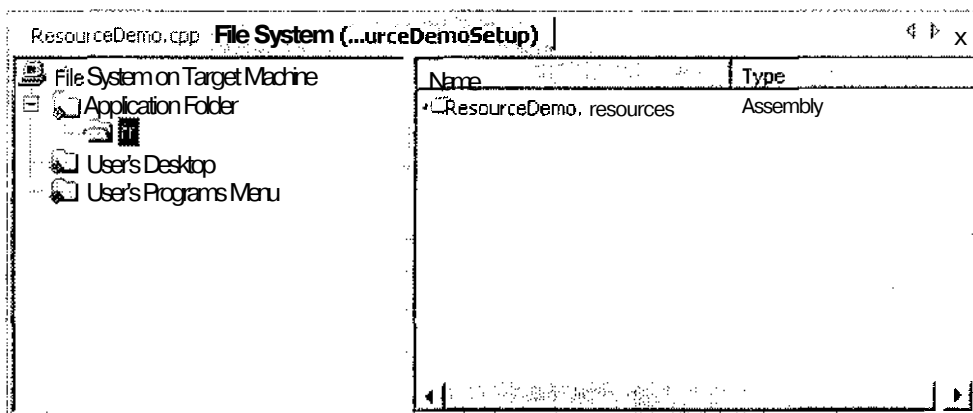
В зависимости от того, что мы хотим добавить в проект, выберем пункт **Project Output** (Выходные файлы проекта) в меню **Add** папки приложения (Application Folder) или пункт **Global Assembly Cache Folder** (Папка глобального кэша сборки):



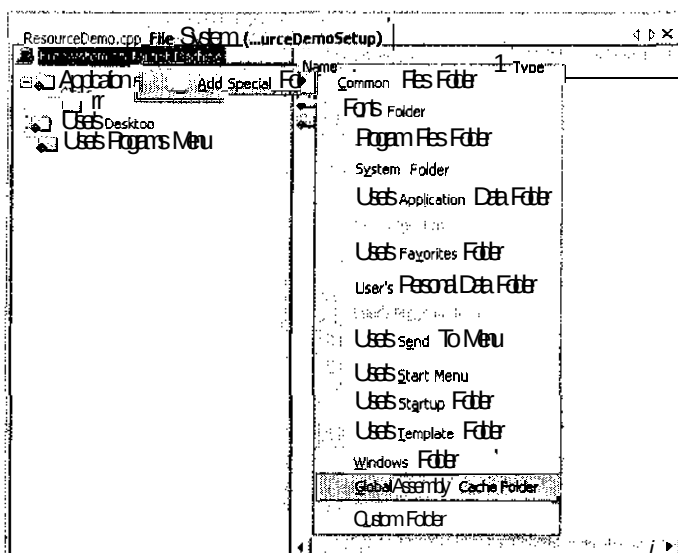
В результате откроется диалоговое окно Add Project Output Group (Группа выходных файлов проекта). В ниспадающем окне Project выберем первый загружаемый проект, т.е. проект, содержащий точку входа в приложение. Затем выберем элемент Primary Output (Первичные выходные файлы). Это приведет к добавлению всех сборок проекта в установочный файл:



Для добавления в проект сопутствующей сборки создадим в папке Application Folder подкаталог, имя которого совпадает с маркером культуры, определенном в спецификации RFC 1766 (в нашем случае имя маркера "fr"). После этого добавим в новый подкаталог сопутствующий ресурс:



Для того чтобы поместить сборку в GAC, необходимо добавить в проект специальную папку. Щелкнем правой кнопкой мыши на папке File System on Target Machine (Файловая система на целевой машине) и выберем команду меню Add Special Folder (Добавить специальную папку), а затем выберем пункт Global Assembly Cache Folder (Папка глобального кэша сборок). Далее необходимо лишь поместить в эту папку все совместно используемые сборки, которые должны находиться в GAC:



Имеется множество других свойств, которые также можно изменять, но, чтобы обсудить все эти свойства, потребуется написать еще одну книгу.

ГЛАВА 4



Атрибуты и отражение

Мы много говорили о том, что сборки .NET реализуют способ распространения не только управляемого кода, но и информации об этом коде, т.е. метаданных. Однако обсуждавшиеся нами метаданные почти исключительно относились к типам данных и номерам версий. Конечно, это очень полезная область применения, но при внимательном рассмотрении становится очевидно, что все указанные возможности могут быть реализованы и с использованием более традиционных методов. В настоящей главе мы покажем, каким образом в код добавляется информация, которая фактически изменяет его поведение, и как программы могут считывать эту информацию. Конкретно мы рассмотрим такие технологии .NET, как атрибуты и отражение.

Атрибуты являются дополнительными элементами информации о сущностях кода (таких, как классы, методы, свойства и др.). Атрибуты включаются в исходный код с помощью специального синтаксиса. Они могут модифицировать поведение упомянутых сущностей или же просто сохраняются как дополнительная информация в метаданных создаваемых сборок. Если сейчас это звучит несколько туманно, то ясность наступит, когда мы начнем рассматривать примеры.

Отражение представляет собой технологию, позволяющую программным образом получать доступ к информации, хранящейся в метаданных. Это значит, что программа может исследовать классы, методы, другие элементы сборки и даже саму себя. Поэтому многие задачи, решить которые средствами традиционных языков программирования было весьма трудно (или даже невозможно), теперь становятся совсем несложными.

Хотя атрибуты и отражение — это две весьма различные технологии, читатель, по-видимому, уже начинает представлять себе потенциал их совместного использования. Данные технологии связаны на уровне метаданных: атрибуты могут порождать метаданные, а отражение — исследовать их. Можно использовать атрибуты для добавления в программу дополнительной информации, описывающей классы и методы в коде, а позже программным образом извлекать эту информацию и на ее основе выполнять определенные действия. Чтобы дать читателю представление о пользе этих технологий, перечислим их некоторые возможные приложения:

- Средства разработки могут использовать отражение для автоматического предоставления информации о классах, методах и параметрах, содержащихся в библиотеках.
- Средство документирования WinCV позволяет использовать отражение для получения сведений о классах, принадлежащих к библиотеке классов .NET Framework.
- Если организация поставляет библиотеки классов другим компаниям, разрабатывающим ПО, то можно использовать атрибуты для указания даты последнего изменения класса или метода, а также группы, ответственной за создание данного раздела кода. Это позволяет клиентам использовать отражение для получения указанной информации, что делает более эффективным сопровождение ПО и ускоряет решение вопросов, связанных с его поддержкой.
- Если приложение ссылается на библиотеку, которая содержит неизвестное число похожих классов, то отражение позволяет точно определить, какие конкретно классы содержатся в указанной библиотеке (см. в настоящей главе пример разработки библиотеки музыкальных инструментов).
- Д Приложение, предоставляющее пользовательский интерфейс для работы с базой данных, может использовать атрибуты, указывающие, какие члены класса соответствуют тем или иным таблицам и полям в базе данных. Это упрощает код, необходимый для реализации операций чтения/записи в базах данных.
- а Если вы поставляете несколько версий приложения ("professional", "enterprise" и т.д.), атрибуты позволяют указать, какие функции приложения должны быть использованы в той или иной версии. А если приложение требует администрирования, можно с помощью атрибутов предоставлять каждой категории пользователей доступ лишь к его определенным функциям. Однако в последнем случае более удобными могут оказаться средства обеспечения защиты, встроенные в .NET.

Наличие таких широких возможностей приводит к тому, что объем материала, который мы должны обсудить, оказывается достаточно большим. Поэтому необходимо разумно структурировать дальнейшее изложение. Оставшаяся часть настоящей главы разбита на три основных раздела:

- **Атрибуты.** Мы рассмотрим некоторые наиболее полезные атрибуты, определяемые Microsoft, а также создадим свои собственные атрибуты.
- **Отражение.** Ниже будет представлен несложный пример, иллюстрирующий использование отражения для получения информации о свойствах класса.
- а **Совместное использование обеих технологий.** Мы рассмотрим пример, основанный на использовании пакета музыкальных программ, и покажем, как можно использовать атрибуты и отражение для получения перечня музыкальных инструментов, определенных в приложении.

Атрибуты

Мы уже отмечали, что атрибуты "могут модифицировать поведение сущностей [в коде] или же просто сохраняются как дополнительная информация в метаданных создаваемых сборок". Наличие таких внешне не связанных друг с другом

аспектов поведения атрибутов проистекает из того факта, что атрибуты могут принадлежать к двум разновидностям:

- О Атрибуты, которые, аналогично директивам препроцессора, иницируют добавление в программу дополнительного кода C++ непосредственно перед компиляцией. Эти атрибуты являются новыми в Visual C++ .NET, однако фактически они не имеют ничего общего с .NET Framework. В основном атрибуты данного типа разработаны для того, чтобы упростить COM-программирование. Поскольку такие атрибуты не имеют отношения к .NET, они не генерируют метаданных и их нельзя использовать совместно с отражением.
- Атрибуты, распознаваемые платформой .NET Framework. Они не добавляют в программу код C++, но обычно генерируют дополнительные метаданные, помещаемые в сборках. Их можно использовать вместе с отражением.

Атрибуты, относящиеся к COM-программированию, являются исключительно особенностью нового компилятора C++ (и, следовательно, специфичны для C++), а .NET-ориентированные атрибуты поддерживаются внутри самой платформы .NET Framework. Это означает, что они также доступны другим .NET-совместимым языкам, включая C# и Visual Basic .NET. Поскольку платформа .NET Framework позволяет не только использовать имеющиеся атрибуты, но и определять новые, то можно дополнительно разделить .NET-ориентированные атрибуты на две категории:

- Атрибуты, определенные и реализованные Microsoft как часть платформы .NET Framework. Они распознаются различными .NET-совместимыми компиляторами. Их применение может приводить к выполнению некоторых дополнительных действий при компиляции, а также к генерации добавочных метаданных.
- Пользовательские атрибуты, которые обычно определяются в коде сторонних организаций

Синтаксис использования атрибутов всех трех категорий совершенно одинаков. Можно просто добавить в код необходимый атрибут, не интересуясь его происхождением. Однако имена COM-атрибутов обычно пишутся строчными буквами (`__class`, `__uuid`), а атрибуты, распознаваемые .NET, используют соглашения, принятые в языке Pascal, согласно которым первая буква каждого слова должна быть заглавной (например, `StructLayout` или `AttributeUsage`). Поскольку разные типы атрибутов выполняют различные функции, мы рассмотрим их по отдельности.

Атрибуты C++

Те читатели, которым приходилось заниматься программированием компонентов COM и просматривать файлы, представленные в формате IDL (Interface Definition Language — Язык определения интерфейса), уже знакомы с атрибутами. Атрибуты представляют собой важную часть IDL. Поскольку функции атрибутов в C++ и в IDL аналогичны, можно упростить знакомство с атрибутами C++, если вспомнить, как работают атрибуты в IDL. Обратимся к определению интерфейса COM:

```
// Внимание. Это код IDL, а не C++
[
    __object,
    __uuid(82CB2B10-DD33-4B05-A0D1-E199EF32AEB8),
    __dual,
]
```

```
interface ISomeInterface : IDispatch
{
    [id] HRESULT DoSomething ([in] BSTR bstrDATA);
};
```

Атрибуты помещаются в квадратные скобки и являются *вездесущими*. Атрибут объекта сообщает компилятору IDL, что данный интерфейс есть часть определения класса COM. Атрибут `uuid` предоставляет идентификатор интерфейса IID, а атрибут `dual` указывает, что данный интерфейс происходит от интерфейса IDispatch (и, следовательно, поддерживает автоматизацию). Эти три атрибута относятся ко всему интерфейсу и потому, согласно синтаксису IDL, должны помещаться непосредственно перед определением интерфейса. Соответственно, атрибут `id`, предоставляющий идентификатор диспетчеризации, относится к методу интерфейса, а параметр метода модифицируется атрибутом `in`, указывающим, что этот параметр используется для получения значения от клиента, а не для передачи значения. Компилятор IDL учитывает информацию, содержащуюся в атрибутах, и без них было бы невозможно создать IDL-файл.

В Visual C++ .NET атрибуты представляют собой расширения, созданные Microsoft для ANSI C++, и компилятор C++ теперь распознает несколько атрибутов. При этом синтаксис работы с атрибутами остается точно таким же, как в IDL. Например, один из атрибутов (`coclass`) выполняет практически ту же самую функцию в C++, что и в IDL. Он информирует компилятор, что на основе определяемого класса будут создаваться объекты COM, и поэтому компилятор сгенерирует код для COM-объекта, а не для чистого класса C++. Приведенный ниже код определяет класс C++, который будет также компилироваться в COM-объект. Здесь используется атрибут `coclass` вместе с несколькими другими атрибутами, назначение которых нетрудно себе представить:

```
// Это настоящий код C++, хотя, возможно, он выглядит странно!
{
    coclass
    threading("apartment"),
    v_i_progid("ATLAttribs.MyCOMAttribObject"),
    progid("ATLAttribs.MyCOMAttribObject.1")
    version(1.0),
    uuid("BC7DCB98-67E6-4BA3-AC58-BEE653E56C29"),
    helpstring("MyCOMAttribObject Class")
}

class ATL_NO_VTABLE CMyCOMAttribObject : public IMyCOMAttribObject
{
    // Код класса
};
```

Тот, кто занимается COM-программированием, согласится, что `coclass` и другие атрибуты значительно облегчают работу, поскольку они заменяют многие старые макросы ATL и вовсе устраняют необходимость поддержки отдельных IDL-файлов программистами. Однако читателям, не знакомым с технологией COM, вряд ли приходилось иметь дело сразу с таким большим количеством атрибутов. Но сейчас им и не нужно изучать все приведенные атрибуты (подробнее о применении атрибутов в COM-программировании см. в главе 9).

Атрибуты.NET

Мы уже познакомились в общих чертах с синтаксисом, применяемым при работе с атрибутами. Чтобы показать, как работают атрибуты .NET, рассмотрим

пример использования атрибута `Obsolete`, который указывает, что применение данного метода более не рекомендуется:

```
void BoardPlaneO
{
    // Код
}
```

Определение метода `BoardPlaneO` вместе с этим атрибутом означает, что при попытке вызвать метод компилятор C++ во время компоновки выдаст предупреждение, текст которого задан в качестве параметра атрибута. Мы рассмотрим атрибут `Obsolete` более подробно, когда будем обсуждать избранные атрибуты .NET.

Разработчикам следует иметь в виду, что из-за ошибок, имеющихся в распространяемой версии компилятора Visual C++ .NET, некоторые описываемые здесь атрибуты не будут корректно работать в C++, хотя они нормально функционируют в других языках, таких как C# и Visual Basic .NET. Microsoft объявила, что для решения этой проблемы будут созданы пакеты обновления. В данном разделе мы объясним, каким образом должны действовать различные атрибуты, и отдельно укажем, в каких случаях код может не работать должным образом, если используется ранняя версия Visual Studio .NET. Описанная проблема возникает при использовании атрибута `Obsolete`.

Необходимо также иметь в виду, что, поскольку атрибуты .NET являются частью платформы .NET Framework, их можно использовать только в тех проектах, которые компилируются в управляемый код. На самом деле такие атрибуты имеют смысл лишь тогда, когда они относятся к управляемым объектам. Так, следующий код вызовет ошибку компиляции:

```
#pragma class Airport
{
    [Obsolete("Warning text here")]
    void BoardPlane
    {
        // Код
    }
    // Код
}
```

Из-за того, что объекты COM отличаются от объектов .NET, не имеет смысла применять к управляемым объектам COM-ориентированные атрибуты C++, такие как `coclass`.

Предопределенные атрибуты .NET

Microsoft уже определила большое количество атрибутов .NET, и весьма вероятно, что к ним добавятся новые. По этой причине невозможно привести здесь исчерпывающий список .NET-ориентированных атрибутов. Вместо этого мы рассмотрим некоторые наиболее важные из них, чтобы дать представление о том, как они работают.

Атрибут `Obsolete`

Атрибут `System::Obsolete` сообщает компилятору, что использование класса, структуры или одного из их членов более не рекомендуется. Этот атрибут можно применить к компоненту нашего кода, если нежелательно, чтобы другие разработчики использовали данный компонент. Атрибут `obsolete` принимает один или два

параметра либо употребляется вовсе без таковых. Когда задаются два параметра, первый из них представляет собой строку с текстом сообщения, которое будет выдавать компилятор, если он зафиксирует попытку обращения к соответствующему члену. Второй параметр имеет тип `Boolean`. Если он устанавливается равным `true`, то любая попытка обращения к тому члену, к которому относится атрибут, вызовет ошибку компиляции, а не предупреждение. Представленный ниже фрагмент кода отличается от рассмотренного ранее примера лишь тем, что в случае вызова метода `BoardPlane()` из любого места нашего кода мы получим ошибку компиляции:

```
[Obsolete($"BoardPlane() deprecated. Use CheckSecurityAndBoardPlane().", true)]
void BoardPlane(void)
{
    // Код
}
```

Если в атрибуте не используются параметры или задан только один из них, оставшиеся параметры принимают значения по умолчанию: предупреждение компилятора, а не генерация ошибки, и сообщение по умолчанию.

К моменту написания данной книги предупреждение или ошибка генерировались лишь в том случае, если метод с атрибутом `Obsolete` вызывался из кода другого проекта, а не из того же самого. Такая ситуация не возникает в языках C# или Visual Basic .NET.

Атрибут `Conditional`

Атрибут `System.Diagnostics.Conditional` представляет собой альтернативу директивам препроцессора `#ifdef`, используемым для компиляции того кода, который нужно иметь лишь в определенных версиях приложения. В частности, это может быть диагностический код, необходимый только в отладочных версиях. Например:

```
[Conditional("$!DEBUG")]
void rWriteUserDiagnosticsO
{
    Console::Write($"g_nUsers = ");
    Console::WriteLine(g_nUsers);
}
```

В данном случае атрибут `Conditional` приписывается к диагностическому методу, который выводит значение переменной `g_nUsers` (мы предполагаем, что эта переменная является глобальной).

Хотя здесь описывается предполагаемое использование атрибута `Conditional`, он (не считая генерации метаданных) игнорируется версией компилятора C++, доступной во время написания настоящей книги. Эта ошибка должна быть устранена в пакете обновления. Следует иметь в виду, что атрибут `Conditional` корректно работает в C# и Visual Basic .NET.

Данный атрибут применим только к методам и содержит один строковый параметр. Когда компилятор встречает атрибут, он считывает параметр и сравнивает его с символами препроцессора, определенными в данный момент. Проверяемый метод будет компилироваться лишь в том случае, если параметр совпадет с текущим символом препроцессора. Если же такой символ не будет обнаружен, компилятор проигнорирует не только определение метода, но и все вызовы этого метода, которые встречаются в любом месте исходного кода. Продолжая предыдущий пример, напомним такой код:


```
int _tmain(void)
{
    g_nUsers = 2;
    WriteUserDiagnostics();
    Console::WriteLine(S"Hello World");
    return 0;
}
```

Этот код будет откомпилирован, даже если выбрать версию компиляции для распространения, в которой символ препроцессора `__DEBUG` не определен (и, следовательно, метод `WriteUserDiagnostics()` не существует). Старые методы, в которых для выполнения условной компиляции использовались директивы препроцессора `iflifdef`, требовали присоединять эти директивы к каждому оператору вызова диагностического метода. Это делало код весьма неприглядным и неудобным для восприятия. В некоторой степени, такую трудность можно было обойти, определив макрос, аналогичный макросу `ASSERT` библиотеки MFC, однако полученный в результате код все же оставался неаккуратным. Если не использовать атрибут `Conditional`, то самый лучший способ, с помощью которого можно добиться того же эффекта, что и в приведенном выше примере, выглядит следующим образом:

```
// Эквивалентный код, не использующий атрибуты
#ifdef _DEBUG
void WriteUserDiagnostics()
{
    Console::Write(S"g_nUsers = "):
    Console::WriteLine(g_nUsers);
}

#define WRITEUSERDIAGNOSTICS WriteUserDiagnostics()
#else
#define WRITEUSERDIAGNOSTICS
#endif

int _tmain(void)
{
    g_nUsers = 2;
    WRITEUSERDIAGNOSTICS();
    Console::WriteLine(S"Hello World");
    return 0;
}
```

Атрибут `Conditional` можно применить к одной и той же функции несколько раз. В этом случае функция будет компилироваться при выполнении хотя бы одного условия:

```
[Conditional(S"Professional"), Conditional(S"Enterprise")]
void DoSomething()
{
    // Этот код будет компилироваться, если определен любой из символов:
    // Professional или Enterprise,
}
}
```

Очевидно, что идея условной компиляции, реализованная подобным образом, может быть воплощена в синтаксические конструкции, только если проверяемый метод возвращает тип `void`. Когда атрибут `Conditional` применяется к методу, возвращающему любой другой тип, генерируется синтаксическая ошибка. Кроме того, по вполне очевидным причинам метод не должен представлять собой подмену другого метода.

Атрибут *StructLayout*

Атрибут `System::Runtime::InteropServices::StructLayout` обеспечивает способ реализации объединений в стиле C/C++, которые платформа .NET непосредственно не поддерживает. Если попытаться определить объединение в управляемом коде, мы получим ошибку компиляции. Однако атрибут `StructLayout` — это значительно более мощное средство, чем объединения C/C++. Объединение позволяет указать, что два или более членов структуры занимают область памяти, начиная с одного и того же адреса, тогда как атрибут `StructLayout` предоставляет возможность задать относительное смещение каждого члена от начального адреса экземпляра структуры. Атрибут `StructLayout` может потребоваться для решения следующих задач:

- Создание в управляемом коде объединения такого же типа, как в C/C++.
- Определение структуры для передачи ее в неуправляемую функцию API с использованием механизма `P/Invoke` (см. главу 7), когда необходимо, чтобы размещение в памяти структуры .NET было в точности таким, какого ожидает неуправляемая функция API. В этом случае можно было бы определить обычную структуру C++, однако тогда ее нельзя было бы использовать при любых вызовах методов, требующих управляемых структур. Атрибут `StructLayout` обеспечивает гибкость, позволяя работать с одной и той же структурой как в управляемом, так и в неуправляемом коде.

В последнем случае атрибут `StructLayout`, как правило, применяется вместе с двумя другими атрибутами: `DllImport` и `MarshalAs` (подробнее о них см. ниже в настоящей главе и в главе 7).

Атрибут `StructLayout` может относиться к управляемым структурам, т.е. к классам и структурам, объявленным с ключевым словом `__value`. Однако данный атрибут нельзя применять к типам, в объявлении которых стоит ключевое слово `__gc`, поскольку платформа .NET Framework сама распределяет эти типы в памяти.

Мы проиллюстрируем применение атрибута `StructLayout` на простом примере, который называется `ComplexTypes`, где будут определены две структуры, представляющие собой комплексные числа. В нашем случае комплексное число рассматривается как пара действительных чисел типа `double`. Мы рассмотрим два комплексных типа. Первый из них (`Complex`) представляет собой структуру, определенную обычным образом, а второй (`ReImComplex`) использует атрибут `StructLayout`. Тип `ReImComplex` содержит два объединения, поскольку имеются две популярные пары имен для двух чисел типа `Double`, содержащихся в комплексном числе. Это имена *x* и *y*, *Re* и *Im*. Напомним, что в математике числа двойной точности, составляющие комплексное число, называются действительной (*Real*) и мнимой (*Imaginary*) частями. Атрибут `StructLayout` позволит свободно получать доступ к одним и тем же данным, используя обе пары имен.

Ниже приведен пример соответствующего кода. Для простоты мы поместили определения классов и тестовый код в один и тот же файл консольного приложения. Вначале для сравнения определим обычное комплексное число без атрибута `StructLayout`:

```
#include "stdafx.h"

(fusing <mcorlib.dll>
#include <tchar, h>

using namespace System;
using namespace System::Runtime::InteropServices;
```

```
value struct Complex
{
    double x;
    double y;
};
```

Этот код отражает сформулированную нами идею создания комплексного числа. Добавление директивы `using` для пространства имен `System::Runtime; InteropServices` необходимо, поскольку именно здесь находится атрибут `StructLayout`. Однако этот атрибут определен в сборке `<mscorlib.dll>`, поэтому нет необходимости ссылаться на другие сборки.

Так же, как классы .NET Framework, атрибуты .NET определяются в пространствах имен. В силу этого для использования атрибутов необходимо обязательно ссылаться на соответствующие пространства имен.

Теперь определим комплексное число, используя атрибут `StructLayout`:

```
[StructLayout(LayoutKind.Explicit)]
value struct RealComplex
{
    [FieldOffset(0)] double x;
    [FieldOffset(8)] double y;
    [FieldOffset(0)] double Re;
    [FieldOffset(8)] double Im;
};
```

Именно здесь начинается самое интересное. Из приведенного кода видно, что атрибут `StructLayout` принимает один обязательный параметр, который представляет собой перечисление, объявляемое с ключевым словом `„value` и принадлежащее к типу `LayoutKind`. В следующей таблице приведены возможные значения указанного параметра:

Значение параметра	Описание
<code>LayoutKind::Explicit</code>	Указывает, что мы явным образом будем сообщать компилятору, как должна размещаться структура в памяти.
<code>LayoutKind::Sequential</code>	Задает последовательное расположение полей в памяти. Выбор этого параметра означает также, что мы получаем возможность указать, насколько плотно должны быть размещены в памяти поля.
<code>LayoutKind::Auto</code>	Размещение структуры в памяти определяется средой CLR.

Мы вскоре рассмотрим параметр `LayoutKind::Sequential`. Выбор параметра `LayoutKind::Auto` дает такой же эффект, как отсутствие атрибута `StructLayout`, поэтому данный параметр более обсуждаться не будет.

Параметр **LayoutKind::Explicit**

Как показывает предыдущий пример, при выборе параметра `LayoutKind::Explicit` необходимо использовать еще атрибут `FieldOffset` для того, чтобы задать фактическое местоположение всех полей. Атрибут `FieldOffset` приписывается к каждому члену структуры и содержит один целочисленный параметр, указывающий, на сколько байт должен быть смещен данный член относительно начала структуры. Поскольку тип `Double` занимает 8 байт, мы задаем для `x` (первый элемент структуры) нулевое

смещение, а для y — смещение, равное 8 байт. Таким образом, элемент y будет следовать непосредственно за x . Затем определяются члены Re и Im , и для них задаются такие смещения, что они будут занимать ту же самую область памяти, какую занимают соответственно элементы x и y . Не обязательно перечислять члены в том же порядке, в каком они будут располагаться в памяти.

Протестируем наши новые типы. Присвоим значения членам x и y , принадлежащим к экземпляру структуры `ReImComplex`, после чего убедимся, что члены Re и Im также получили значения. Затем выведем на экран размер наших двух структур, который в обоих случаях должен составлять 16 байт. Для этого напишем следующий код:

```
int _tmain(void)
{
    ReImComplex c1;
    c1.x = 23.5;
    c1.y = 10.0;
    Console.WriteLine(S"c1.x = {0}", c1.x.ToString());
    Console.WriteLine(S"c1.y = {0}", c1.y.ToString());
    Console.WriteLine(S"c1.Re = {0}", c1.Re.ToString());
    Console.WriteLine(S"c1.Im = {0}", c1.Im.ToString());
    Console.WriteLine(S"sizeof(ReImComplex) = ");
    Console.WriteLine(sizeof(ReImComplex));
    Console.WriteLine(S"sizeof(Complex) = ");
    Console.WriteLine(sizeof(Complex));
    return 0;
}
```

При запуске этого кода получим следующий результат:

```
c1.x = 23.5
c1.y = 10
c1.Im = 23.5
c1.Re = 10
sizeof(ReImComplex) = 16
sizeof(Complex) = 16
```

При использовании атрибутов `StructLayout` и `FieldOffset` мы получаем полный контроль над размещением в памяти каждого члена. В нашем примере память распределена таким образом, что $Im(y)$ следует непосредственно за $Re(x)$. Однако, если потребуются добавить 8 байт свободного пространства между указанными элементами, этого легко можно добиться:

```
[FieldOffset(0)] double x;
[FieldOffset(16)] double y;
[FieldOffset(0)] double Re;
[FieldOffset(16)] double Im;
```

Выбрав подходящие смещения, можно даже сделать так, что члены будут частично перекрываться. Такой прием применяется в программировании для Windows, чтобы представить тип `LARGE_INTEGER` с помощью двух более коротких целых типов.

Параметр `LayoutKind::Sequential` и параметр уплотнения

Использование атрибута `StructLayout` с параметром `LayoutKind::Sequential` позволяет указать последовательное, а не явно определенное, размещение элементов структуры в памяти и задать параметр уплотнения. Это часто происходит автоматически, если атрибут `StructLayout` не применяется, однако не всегда можно полагаться на данный процесс. Даже если члены структуры последовательно располагаются в памяти, это вовсе не означает, что они будут занимать участки

памяти, непосредственно примыкающие друг к другу. В зависимости от типов полей между ними могут появляться пробелы (незанятые области памяти). Причина заключается в том, что с целью повышения производительности система по умолчанию выравнивает элементарные типы таким образом, что они занимают участки памяти, адреса которых начинаются с целых значений, кратных размерам упомянутых типов. Если член содержит поля различных размеров, параметр уплотнения определяется размером наибольшего поля, и в этом случае появляются пробелы. Рассмотрим следующую структуру:

```
__value struct TestPackStruct
{
    Byte c;
    double x;
    Byte y;
};
```

Длина типа `System::Byte` равна 1, тогда как тип `double` имеет размер 8 байт, поэтому объявленная структура должна всего занимать 10 байт памяти. Но, поскольку здесь присутствует тип `double`, система выравнивает все поля, привязав их границы к 8-байтовым значениям. Таким образом, в данном случае **параметр уплотнения (packing size)** будет равен 8, а полный размер структуры составит 24 байта. При желании можно изменить такое поведение следующим образом:

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
__value struct TestPackStruct
{
    Byte c;
    double x;
    Byte y;
};
```

В данном случае мы задаем значение параметра уплотнения равным 1, в результате чего полный размер структуры составит 10 байт. Поэтому данная структура будет занимать меньше памяти, хотя это может привести к потере производительности в связи с особенностями работы системного оборудования.

Параметр `Pack` может принимать значения 1, 2, 4, 8, 16, 32... или 0. В последнем случае система выбирает значение параметра уплотнения по умолчанию (обычно равное 8).

Из-за ошибки компилятора C++, которая не исправлена к моменту написания настоящей книги, параметр `Pack` игнорируется в C++, однако он корректно обрабатывается компиляторами C# и Visual Basic .NET. Указанная ошибка, по-видимому, будет устранена в одном из пакетов обновления.

Вместо простой передачи значения параметра в атрибут мы указываем имя параметра в объявлении атрибута: `Pack = 1`. Различные атрибуты используют тот или иной способ работы с параметрами. Здесь нет общего правила — это лишь вопрос определения каждого атрибута. Если вы не знаете, какой вариант выбрать, лучше обратиться к документации.

Атрибут **DllImport**

Атрибут `DllImport` исключительно полезен; это один из нескольких атрибутов, которые можно использовать как с управляемыми, так и с неуправляемыми методами. Он дает возможность сослаться на внешний метод, принадлежащий к библиотеке DLL с заданным именем, когда доступ к методу необходимо получить с помощью механизма `PInvoke`, позволяющему вызывать неуправляемый код. Атрибут `DllImport` обычно используется следующим образом:

```
[DllImport("user32.dll", EntryPoint = "MessageBox", CharSet = Unicode)]
int MessageBox(void* hWnd, wchar_t* lpText, wchar_t* lpCaption, unsigned
int uType);
```

Функция `MessageBox()` выбрана для того, чтобы проиллюстрировать синтаксис, используемый в данном случае. На практике мы вряд ли будем вызывать эту функцию из управляемого приложения, так как метод `MessageBox::Show()`, принадлежащий к пространству имен `System::Windows::Forms`, позволяет выполнить те же самые функции (см. главу 6). Однако, если потребуется получить доступ к возможностям интерфейса Windows API, которые не реализуются библиотекой классов .NET Framework, или работать с кодом сторонней организации, упакованным в классическую DLL, можно воспользоваться атрибутом `DllImport`.

Мы не будем обсуждать здесь атрибут `DllImport` или службы `PInvoke`, поскольку вопросы взаимодействия с наследованным неуправляемым кодом подробно освещаются в главе 7.

Атрибут `ThreadStatic`

Для того чтобы проиллюстрировать разнообразие атрибутов, коснемся вопроса работы с потоками (подробнее см. в следующей главе). Атрибут `System::ThreadStatic` помечает статический член класса как сущность, принадлежащую к одному потоку. Это значит, что каждый поток, имеющий доступ к классу, получает свою собственную независимую копию члена, которая не подвержена воздействию со стороны других потоков. Ниже приведен пример применения атрибута `ThreadStatic`, взятый из проекта главы 5:

```
_gc class TestThreadStatic
{
public:
    // Определение метода для краткости отпущено

private:
    [ThreadStatic] static int m_nThreadStatic;
    static int m_nNotThreadStatic;
};
```

Атрибуты сборки

Атрибуты, которые мы обсуждали до сих пор, помещаются внутри основного текста программы и относятся к методам, свойствам, классам, структурам и т.д. Однако можно приписать атрибут сборке в целом. В .NET Framework определено несколько атрибутов, предоставляющих такие сведения о сборке, как название, описание и информация об авторских правах. Кроме того, атрибуты могут сообщать компилятору важные данные о файлах сборки, которые должны помещаться в глобальный кэш сборки.

Имя атрибута сборки *всегда* начинается с префикса `assembly::`. Если создать управляемое приложение C++ с помощью Visual Studio .NET, в файле `AssemblyInfo.cpp` окажется набор атрибутов, объявленных по умолчанию, которые можно редактировать, как было показано в предыдущей главе:

```
[assembly: AssemblyTitle("")];
[assembly: AssemblyDescription("")];
[assembly: AssemblyConfigurationAttribute("")];
[assembly: AssemblyCompany("")];
[assembly: AssemblyProduct("")];
[assembly: AssemblyCopyrightAttribute("")];
[assembly: AssemblyTrademark("")];
```

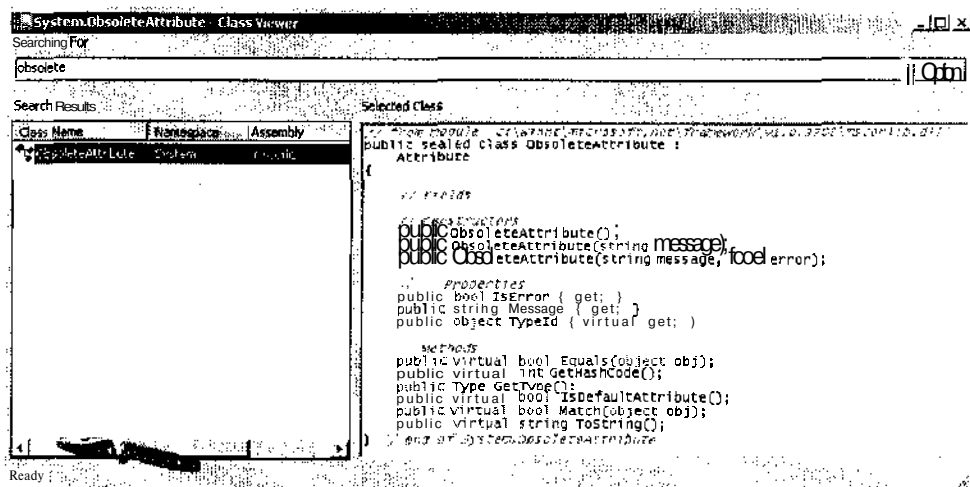
```
[assembly:AssemblyCultureAttribute("")];  
[assembly:AssemblyVersionAttribute("1.0.*")];
```

Атрибуты как классы

Теперь обсудим подробнее, каким образом атрибуты реализованы в .NET Framework. Основная концепция, которую здесь необходимо иметь в виду, состоит в том, что каждый атрибут .NET является классом.

Все атрибуты .NET реализованы как классы .NET.

Проще всего убедиться в этом, если вывести на экран определение класса для уже знакомого нам атрибута `Obsolete` с помощью средства `WinCV`, которое входит в Visual Studio .NET (файл `Microsoft Visual Studio .NET\FrameworkSDK\Bin\WinCV.exe`). При работе с `WinCV` можно просто впечатать имя класса, в результате чего на экране появится псевдокод определения этого класса. Ниже показано, какую информацию выдаст утилита `WinCV` для атрибута `Obsolete`, после того как будет впечатано имя "obsolete" и выбран класс `ObsoleteAttribute` в окне просмотра дерева классов. При этом для представления кода используется синтаксис C#, а не C++, однако в данном коде нетрудно разобраться.



Первая строка этого кода показывает, что класс называется `ObsoleteAttribute`, а не `Obsolete`. Согласно общей рекомендации, имена всех классов, представляющих атрибуты, должны заканчиваться словом `Attribute`, хотя для удобства восприятия кода компилятор может автоматически добавить указанную часть имени класса, когда он встречает атрибут в коде. При желании можно использовать и полное имя атрибута — это совершенно не влияет на выполнение кода.

При создании в файле `AssemblyInfo.cpp` атрибутов, которые можно редактировать, мастер использует полные имена этих атрибутов.

Класс `ObsoleteAttribute` происходит от базового класса .NET Framework `System.Attribute`. Все атрибуты должны быть производными от этого класса, поскольку он реализует механизм, с помощью которого компилятор распознает атрибуты.

Класс `System::Attribute` определяет несколько методов, которые, разумеется, перечислены в документации MSDN. Однако эти методы используются редко, поэтому мы не будем их обсуждать. Они в основном позволяют получать информацию о том, определен ли данный атрибут для конкретных классов, сборок и т.д.

Ранее мы видели, что атрибут `Obsolete` может принимать один или два параметра либо использоваться без параметров. Конструкторы класса `ObsoleteAttribute` предоставляют именно такой выбор. Общее правило здесь состоит в том, что имеющиеся в наличии конструкторы определяют параметры, которые необходимо задавать.

Может создаться впечатление, что при использовании атрибута в нашем коде компилятор инициализирует объект класса атрибута в зависимости от задаваемых аргументов. Однако на самом деле атрибут не инициализируется и конструктор не выполняется. Вместо этого компилятор проверяет передаваемые аргументы на совместимость с имеющимися конструкторами для соответствующего класса атрибута и помещает значения аргументов в метаданные сборки. Можно инициализировать объект атрибута с помощью отражения (см. ниже). При этом в конструктор будут передаваться значения, хранящиеся в метаданных.

Члены класса атрибута

Вернемся к полученному нами псевдокоду. Из него видно, что атрибут `ObsoleteAttribute` имеет два свойства, которые возвращают значения `Message` и `IsError` его аргументов. Как мы убедимся позже, очень важно определить подобные свойства. Если они отсутствуют, нельзя использовать отражение для получения значений параметров.

Остальные методы и третье свойство атрибута `ObsoleteAttribute` малоинтересны для нас в настоящий момент. В основном методы, перечисленные в окне на рисунке, определены классом `System::Object` и потому поддерживаются всеми объектами .NET. Атрибут `ObsoleteAttribute` включает также методы и свойства, унаследованные от класса `System::Attribute` (см. выше). К ним относятся: `Match()`, `IsDefaultAttribute()` и `TypeID`.

Рассмотренные нами свойства, определенные для атрибута `ObsoleteAttribute`, доступны только для чтения. Если свойство будет допускать запись, то при использовании атрибута можно обращаться к свойству по имени. Атрибут `ObsoleteAttribute` не имеет подобных свойств, но в атрибуте `StructLayout` для записи доступны три свойства: `Pack`, `CharSet` и `Size`. Ниже приведен псевдокод, отображаемый утилитой WinCV для атрибута `StructLayout`:

```
// из модуля c:\winnt\microsoft.net\framework\v1.0.3705\mscorlib.dll
public sealed class System.Runtime.InteropServices.StructLayoutAttribute :
    Attribute
{
    // Поля
    public System.Runtime.InteropServices.CharSet CharSet;
    public int Pack;
    public int Size;

    // Конструкторы
    public StructLayoutAttribute(short layoutKind);
    public StructLayoutAttribute(
        System.Runtime.InteropServices.LayoutKind layoutKind);
    // и т.д.
```


В данном случае соответствующие параметры атрибута фактически определены как открытые члены, а не свойства. Однако на процедуре использования атрибута это никак не сказывается. Например:

```
[StructLayout(LayoutKind.Sequential, Pack = 1)]
public __value struct TestPackStruct
```

Здесь используется синтаксис <свойство> = <значение>. В результате компилятор ищет открытое свойство или член с заданным именем для соответствующего класса атрибута. Если искомый член найден, компилятор присваивает ему задаваемое значение. В противном случае будет получена ошибка компиляции.

Атрибут *AttributeUsage*

Прежде чем приступить к созданию атрибута, следует прояснить еще один вопрос. Определяя наши собственные атрибуты, необходимо предоставить компилятору информацию о том, каким образом эти атрибуты будут использоваться, т.е. компилятор должен знать:

- К каким элементам (классам, структурам, методам, аргументам методов или сборкам в целом) может относиться атрибут.
- Может ли конкретный атрибут использоваться более одного раза с одним и тем же элементом.
- Должен ли атрибут автоматически передаваться в производный класс.

Эту информацию можно сообщить компилятору, добавив атрибут `System: AttributeUsageAttribute` (да-да, еще один атрибут!) в определение нашего атрибута. Можно использовать и более краткую запись имени атрибута: `AttributeUsage`. В следующем проекте `CustomAttributes` мы продемонстрируем определение и применение пользовательского атрибута `CompletionAttribute`. Потребуем, чтобы этот атрибут можно было использовать с классами и структурами. Он должен пометить написанный код элементами перечисления `CompletionStatus`, указывающими, какую работу следует выполнить программистам. Кроме того, `CompletionAttribute` должен иметь необязательный параметр, содержащий комментарий. Исходя из указанных требований, запишем начальную часть определения атрибута `CompletionAttribute`:

```
[AttributeUsage(AttributeTargets::Class | AttributeTargets::Struct,
                AllowMultiple = 'false', Inherited = false)]
__gc class CompletionAttribute : public Attribute
{
```

Атрибут `AttributeUsage` может присоединяться только к классам, причем к тем из них, которые будут использоваться в качестве атрибутов. Он принимает один обязательный и два необязательных параметра. Обязательный аргумент принадлежит к типу `AttributeTargets` и указывает, где можно использовать новый атрибут. Тип `AttributeTargets` представляет собой перечисление .NET и определяется следующим образом:

```
[Flags]
__value enum AttributeTargets {
    Assembly = 0x00000001,
    Module   = 0x00000002,
    Class    = 0x00000004,
    Struct    = 0x00000008,
    Enum     = 0x00000010,
    Constructor = 0x00000020,
```

```

Method      = 0x00000040
Property    = 0x00000080
Field       = 0x00000100
Event       = 0x00000200
Interface   = 0x00000400
Parameter   = 0x00000800
Delegate    = 0x00001000
ReturnValue = 0x00002000
All         = 0x00003fff
};

```

СМЫСЛ ВОЗМОЖНЫХ значений этого перечисления должен быть понятен читателю. В коде использован логический оператор OR, с помощью которого мы указали, что атрибут CompletionAttribute может присоединяться к классам или структурам. Это значит, что можно написать:

```

[Completion(CompletionStatus::ReadyToShip)]
// Атрибут присоединяется к управляемому классу
__gc class MyClass
{
    // код
};

```

Однако следующий код вызовет ошибку компиляции:

```

[Completion(CompletionStatus::ReadyToShip)]
// Данный атрибут нельзя приписывать методу
void DoSomething()
{
    // Код
}

```

Возможно, читатель обратил внимание на дополнительное обстоятельство, связанное с перечислением AttributeTargets, которое помечено еще одним атрибутом! Это System::FlagsAttribute, указывающий, что перечисление представляет собой последовательность флагов, которые можно комбинировать, используя побитовые операторы. Данный атрибут не оказывает влияния на код перечисления, однако он может быть обнаружен средствами разработки и использован для соответствующей модификации интерфейса. Например, Visual Studio .NET проверяет наличие атрибута Flags, прежде чем выбрать режим отображения подсказок IntelliSense для перечислений.

Второй параметр AllowMultiple атрибута AttributeUsage является необязательным. Он представляет собой булеву величину, указывающую, можно ли этот атрибут использовать с одним и тем же элементом более одного раза. В нашем примере выбрано значение параметра false, поэтому код, представленный ниже, будет ошибочным:

```

// Этот код не будет откомпилирован, поскольку выбрано значение параметра
// AllowMultiple равно false
[Completion(CompletionStatus::InProgress),
Completion(CompletionStatus::ReadyToShip)]
__gc class MyClass
{
    //Код
};

```

Если бы для параметра AllowMultiple было задано значение true, данный код оказался бы корректным. Когда значение параметра AllowMultiple не указывается, по умолчанию оно выбирается равным false. В рассматриваемом примере трудно

представить себе, что бы это значило, если бы класс имел более одного атрибута `CompletionAttribute`, определяющего статус данного класса. Поэтому имеет смысл выбрать для параметра `AllowMultiple` значение `false`.

Последний необязательный параметр `Inherited` атрибута `AttributeUsage` также представляет собой булеву величину. Когда он равен `true`, атрибут будет передаваться производным классам или методам, подменяющим данный метод. Поскольку производные классы могут быть написаны разными разработчиками, в нашем случае лучше задать для указанного параметра значение `false`. Если параметр `Inherited` не задан, по умолчанию он устанавливается равным `true`.

Написание пользовательских атрибутов

Все атрибуты, которые мы рассматривали до сих пор, влияли на процесс компоновки приложения или на создаваемый в итоге код. Однако, определяя пользовательские атрибуты, мы не можем сообщить компилятору специфическую информацию о них. Поэтому единственный эффект от применения таких атрибутов заключается в том, что в метаданные сборки будет помещаться дополнительная информация. Тем не менее, несмотря на столь очевидные ограничения, пользовательские атрибуты представляют собой очень мощное средство.

Пользовательские атрибуты могут применяться только для генерации метаданных. Они не оказывают влияния на процесс компиляции.

В настоящем разделе мы опишем создание атрибута `CompletionAttribute`, который будет использоваться для указания состояния завершенности для разрабатываемого класса или структуры. Предположим, что организация имеет систему контроля качества, определяющую четыре фазы написания кода ("разработка", "завершено", "протестировано" и "готово для распространения"). Пусть эти фазы определяются с помощью перечисления:

```
enum CompletionStatus {InProgress, Coded, Checked, ReadyToShip};
```

В реальной жизни подобный атрибут мог бы применяться с высокой степенью избирательности — к отдельным методам, конструкторам и свойствам. Но мы упростили задачу. Разрабатываемый нами атрибут примет два параметра:

- Значение, принадлежащее к перечислению `CompletionStatus`, указывающее степень готовности кода (обязательный параметр)
- Необязательный параметр с сопутствующим пояснением

Мы также будем придерживаться соглашения, используемого при написании атрибутов, согласно которому синтаксис употребления обязательного параметра должен иметь вид:

```
[Completion(CompletionStatus.InProgress)]
gc class SomeClass
{
    // Код
};
```

При наличии необязательного параметра для его записи должен применяться такой синтаксис:

```
[Completion(CompletionStatus::InProgress,
    Comment = "Methods not yet implemented")]
__gc class SomeClass
```

```
    // Код
};
```

В соответствии с этим обязательный параметр необходимо реализовать так, чтобы он передавался через конструктор, а необязательный параметр представлял собой редактируемое свойство или член. Полный код, определяющий атрибут `CompletionAttribute`, имеет вид:

```
ftprag ma, once
using namespace System;

__value enum CompletionStatus {InProgress, Coded, Checked, ReadyToShip};
[AttributeUsage(AttributeTargets::Class, AttributeTargets::Struct,
    AllowMultiple = false, Inherited = false)]
__gc class CompletionAttribute : public Attribute
{
public:
    CompletionAttribute(CompletionStatus status) : m_Status(status),
                                                m_pComment("")
    {
    }

    __property CompletionStatus get_Status()
    {
        return m_Status;
    }

    __property String* get_Comment()
    {
        return m_pComment;
    }

    // Этот метод доступа свойства позволяет передавать
    // в конструктор необязательный параметр
    __property void set_Comment(String* pComment)
    {
        m_pComment = pComment;
    }

private:
    CompletionStatus m_Status;
    String* m_pComment;
};
```

После определения атрибута его можно использовать. Код в файле `CustomAttributes.cpp`, где употребляется атрибут `CompletionAttribute`, выглядит следующим образом:

```
#include "stdafx.h"

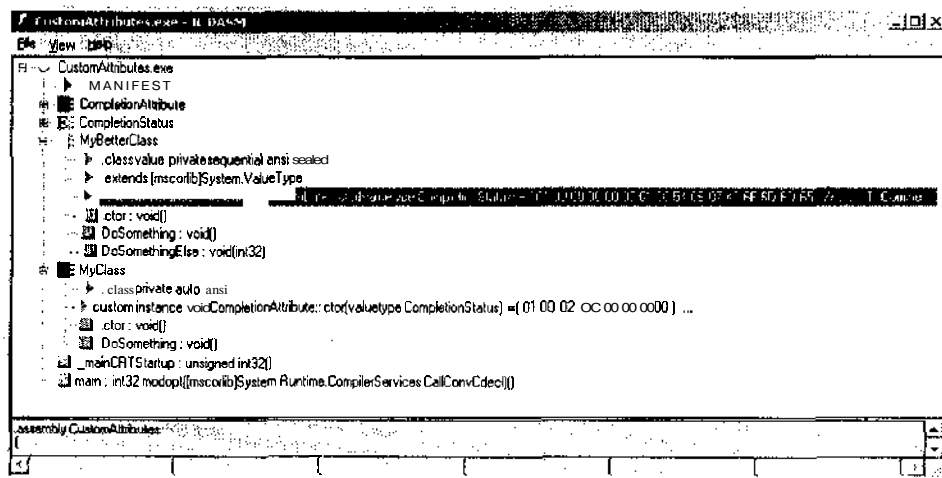
#using <mcorlib.dll>
#include <tchar.h>
#include "CustomAttributes.h"
using namespace System;
```

```
[Completion(CompletionStatus::Checked)]
__gc class MyClass
{
    void DoSomething() {}
};

[Completion(CompletionStatus::InProgress,
    Comment = S"Methods not yet implemented")]
__value class MyBetterClass
{
    void DoSomething() {}
    void DoSomethingElse(int x) {}
};

// ЭТО точка входа в приложение
int _tmain(void)
{
    Console::WriteLine(S"HelloWorld");
    return 0;
}
```

Если откомпилировать этот код, то с помощью утилиты ILDasm (см. предыдущую главу) можно убедиться, что значения атрибута переданы в сборку вместе с определениями классов MyClass и MyBetterClass:



Информация о новом атрибуте содержится в строках, начинающихся с текста `“.custom instance...“`. К сожалению, утилита ILDasm не переносит текст строки, из-за чего неудобно читать выходной файл. Однако если вывести в файл содержимое окна просмотра дерева (путем нажатия клавиш `<Ctrl> + <T>`), то можно получить больше информации. Для класса MyBetterClass соответствующая строка примет вид:

```
.custom instance void CompletionAttribute::ctor(valuetype
CompletionStatus) = ( 01 00 00 00 00 00 01 00 54 0E 07 43 6F 6D 6D 65
// . . . . . T..Comme...
```

Строка, относящаяся к классу MyClass, будет выглядеть так:

```
.custom instance void CompletionAttribute::ctor(valuetype
CompletionStatus) = ( 01 00 02 00 00 00 00 00 ) ...
```

К сожалению, в обоих случаях утилита ILDasm отказывается выводить в одну строку количество символов, превышающее определенное значение. Однако мы, по крайней мере, можем убедиться, что информация об атрибуте присутствует в сборке.

На этом мы завершим обсуждение нашего примера. Читатель может почувствовать разочарование, поскольку созданный нами код фактически ничего не делает. Функция `main()` просто возвращает значения и не использует ни один из определенных выше классов. В настоящий момент нам пока не удастся сделать больше, поскольку мы еще не рассмотрели вопрос об использовании дополнительной информации, помещаемой в метаданные. Ниже, при обсуждении кода, использующего отражение, мы приведем более убедительный пример, иллюстрирующий некоторые другие способы применения пользовательских атрибутов.

Отражение

Оставим на время атрибуты в стороне. Хотя, разумеется, отражение и атрибуты могут использоваться совместно, первая из двух технологий имеет более широкую область применения. Отражение позволяет создаваемым программам исследовать все метаданные, содержащиеся в сборках.

Проект ListColors

Ниже приведен код, использованный в простом проекте ListColors, который служит примером того, как можно использовать в программе отражение и почему эта технология является столь мощной. Эта несложная программа перечисляет имена некоторых цветов:



Здесь у читателя должен возникнуть вполне законный вопрос: какое отношение ко всему этому имеет отражение? Дело в том, что цвета, названия которых должно выдавать приложение ListColors, — это не те цвета, которые мы использовали ранее. В рассматриваемом примере мы будем работать с полным набором именованных цветов, определенных в .NET Framework, и определить их программным образом можно только с помощью отражения.

В NET Framework цвета представлены структурой `System.Drawing.Color`, которая применяется для раскраски различных элементов пользовательского

интерфейса в приложении. Данная структура реализует несколько статических свойств, возвращающих различные именованные цвета. Для того чтобы, например, структура `Color` возвращала пурпурный цвет, требуется написать следующий код:

```
Color::Purple = Color::Purple;
```

Итак, если нужен **именно** пурпурный цвет, проблем не возникает. Но как быть, если необходимо выяснить, какие вообще цвета имеются в нашем распоряжении? Конечно, можно прочесть документацию и затем задать все нужные цвета в коде приложения, однако это не самое удачное решение. При этом придется изрядно потрудиться, поскольку в настоящий момент реализовано 149 именованных цветов. Возможно даже, потом придется переписывать код, если окажется, что Microsoft добавила новые цвета. Следовательно, необходим способ, позволяющий выяснить программным образом, какие свойства реализует структура `Color`. Эту проблему можно решить с помощью отражения.

Имена цветов также заданы в перечислении `System::Drawing::KnownColor`. При обращении к нему не надо использовать отражение для выяснения имен цветов. Перечисление `KnownColor` содержит не только фактические именованные цвета, но и различные системные цвета, которые соответствуют разным именованным цветам, в зависимости от настройки конкретного компьютера. Например, цвет `ActiveBorder` соответствует установленному цвету границы активного окна, каким бы тот ни был. Если нужны только имена фактических цветов, то следует использовать отражение.

Хотя следующий пример содержит некоторые нюансы, которые трудно уловить на первый взгляд, все прояснится, когда мы более подробно будем рассматривать классы отражения. Ниже приведен код приложения `ListColors`:

```
<code>include "stdafx.h"

using namespace System;
using namespace System::Reflection;
using namespace System::Drawing;
int _tmain(void)
{
    Type* colorType = _typeof(Color);
    PropertyInfo* colorProps[] = colorType->GetProperties();

    // Перебор свойств
    for(int i=0; i < colorProps.Length; i++)
    {
        PropertyInfo* colorProp = colorProps[i];
        if (colorProp->GetMethod()->IsStatic &&
            colorProp->PropertyType == _typeof(Color))
        {
            Console::Write(colorProps[i]->Name);
            Console::Write(S"\t");
        }
    }

    return 0;
}</code>
```

Кратко обсудим этот код. В нем используются два пространства имен: `System;` `Drawing` (к которому относится структура `Color`) и `System::Reflection` (куда включены классы отражения), а также дополнительная библиотека `System.Drawing.dll`, содержащую структуру `Color`.

Основной текст программы начинается с определения переменной типа `System::Type`. Это базовый класс `.NET`, описывающий именованный тип, я он почти всегда служит точкой входа для отражения. Далее для получения объекта типа `Type` используется оператор `__typeof()`, после чего применяется метод `GetProperties()`, который возвращает массив экземпляров типа `PropertyInfo`. Каждый такой экземпляр содержит информацию об одном из свойств, реализованных типом (в нашем случае структурой `Color`). Тип `PropertyInfo` относится к пространству имен `System::Reflection`.

Коль скоро мы получили массив типа `PropertyInfo`, наша задача почти выполнена (и для этого потребовалось всего две строки кода!). Теперь нужно вывести на экран имена полученных свойств с помощью свойства `Name`:

```
Console::Write(colorProps[i]->Name);
```

Оператор `Console::Write()` находится внутри предложения `if`, поскольку структура `Color` определяет также некоторые другие свойства, помимо тех, которые нас интересуют. Например, свойство `Color::R` возвращает относительную интенсивность красного компонента цвета. Данное свойство не представляет для нас интереса, и мы отбрасываем его, основываясь на том факте, что оно возвращает тип `Byte`, а не `Color`, и является свойством экземпляра, а не статическим свойством.

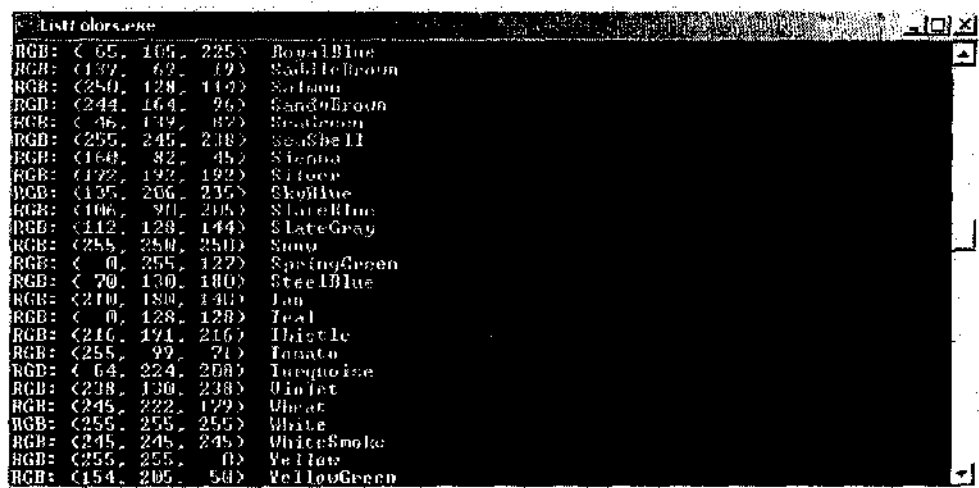
Для гарантии того, что наш код выбирает свойства, соответствующие именованным цветам, мы ограничиваемся статическими свойствами, которые возвращают тип `Color`:

```
if (colorProp->GetGetMethod()->IsStatic &&
    colorProp->PropertyType == __typeof(Color))
{
    ...
}
```

С помощью метода `PropertyInfo::PropertyType` мы получаем объект типа `Type`, который описывает возвращаемый тип свойства. Выяснить, является ли свойство статическим, несколько сложнее. Класс `PropertyInfo` не содержит члена, который мог бы непосредственно предоставить такую информацию. Вспомним, однако, что свойство представляет собой всего лишь пару методов доступа (`get` и `set`), и воспользуемся методом `PropertyInfo::GetGetMethod()`, который возвращает экземпляр типа `MethodInfo`, описывающий `get`-метод свойства. Класс `MethodInfo`, относящийся к пространству имен `System::Reflection`, как раз предназначен для описания методов, как класс `PropertyInfo` предоставляет информацию о свойствах. Тип `MethodInfo` имеет свойство `IsStatic`, которое возвращает булево значение, указывающее, является ли метод статическим.

Вызов члена класса

Проект `LiatColors` дает достаточно хорошее представление о возможностях отражения, однако кое-что мы упустили. Получение списка членов класса — это неплохо, но как вызвать то или иное свойство или метод? Прежде чем начать более детальное теоретическое обсуждение отражения, расширим проект `ListColors`, чтобы вывести на экран относительную интенсивность красного (`Red`), зеленого (`Green`) и синего (`Blue`) компонентов каждого цвета:



Из этого списка видно, что желтый цвет (Yellow), как и следовало ожидать, содержит красный и зеленый компоненты максимальной интенсивности (255), а синий компонент здесь полностью отсутствует, т.е. относительная интенсивность компонентов имеет вид: (255, 255, 0). Далее очевидно, что цвет WhiteSmoke ("Белая дымка") близок к белому, поскольку значения всех интенсивностей (248, 248, 248) мало отличаются от максимальных.

Для того чтобы вывести на экран всю эту информацию, нужно инициализировать объект Color, который представляет нужный цвет, и получить доступ к свойствам R, G и B данного объекта. Ниже приведен необходимый для этого обновленный код, в котором все изменения выделены:

```
int _tmain(void)
{
    Type* colorType = __typeof(Color);
    PropertyInfo* colorProps[] = colorType->GetProperties();

    /* Перебор свойств
    for(int i = 0; i < colorProps.Length; i++)
    {
        PropertyInfo colorProp = colorProps[i];
        MethodInfo pMethod = colorProp->GetMethod();
        if(pMethod->IsStatic && colorProp->PropertyType == __typeof(Color))
        {
            Object* colorAsObject = pMethod->Invoke(0, 0);
            Color* theColor = dynamic_cast<Color*>(colorAsObject);
            Console.WriteLine($"RGB: ({0, 3}, {1, 3}, {2, 3})",
                theColor->R.ToString(), theColor->G.ToString(),
                theColor->B.ToString());
            Console.WriteLine($" {0}", colorProps[i]->Name);
        }
    }

    return 0;
}
```

На сей раз, кроме получения имени свойства, мы фактически вызываем его, точнее, мы вызываем get-метод свойства. Это делается с помощью метода Invoke

объекта `MethodInfo`, который мы вновь получаем, используя метод `PropertyInfo::GetMethod()`. Метод `MethodInfo::Invoke` принимает два параметра:

- Указатель на объект, чей метод мы вызываем
- Набор передаваемых в метод параметров, представляющий собой массив объектов

В рассматриваемом случае оба параметра равны нулю по следующим причинам: во-первых, все свойства, используемые нами для получения именованных цветов, являются статическими, и не существует объекта, связанного с любым из этих свойств; во-вторых, метод `get` по определению никогда не принимает параметров.

Метод `InvokeO` позволяет получить возвращаемое значение вызываемого нами метода, однако данное значение приведено к типу `Object*`. Необходимость этого связана с тем, что метод `InvokeO` применим для вызова абсолютно любого метода. При этом фактический тип возвращаемой величины может быть любым управляемым типом. Далее мы используем оператор `dynamic_cast<>()`, чтобы привести полученную величину к типу `Color*`, который должен фактически возвращаться в данном случае:

```
Object* colorAsObject = pMethod->Invoke(0, 0);
Color* theColor = dynamic_cast<Color*>(colorAsObject);
```

Хотя выше использовано отражение для возвращения объекта `Color`, представляющего именованный цвет, нам не потребовалось прибегать к технологии отражения, чтобы получить доступ к свойствам `R`, `G`, и `B` указанного объекта. Мы обращаемся к этим свойствам обычным путем, поскольку во время компиляции точно известно, какие именно свойства должны вызываться:

```
Console::Write($"RGB: ({0, 3}, {1, 3}, {2, 3})",
    theColor->R.ToString(), theColor->G.ToString(),
    theColor->B.ToString());
```

Когда мы получали сами цвета `Colors`, методы, которые должны были вызываться, выявлялись на стадии выполнения программы. Благодаря использованию отражения не было необходимости знать заранее, что именно мы можем найти. Так, наш код работает со свойствами `Color::Yellow` и `Color::WhiteSmoke`, однако нигде в исходном коде не указаны имена этих свойств. Отражение позволило во время выполнения выявить данные свойства и получить к ним доступ. В этом и заключается мощь технологии отражения.

Класс `System.Type`

Доступ к технологии отражения осуществляется с помощью нескольких базовых классов .NET. Наиболее важным из них (как легко понять из последнего примера) является класс `System.Type`, который во многих отношениях может рассматриваться как программная точка доступа к функциональности отражения. Скорее всего, именно из-за своей важности и широкого применения этот класс определен в пространстве имен `System`, хотя почти все остальные базовые классы, связанные с отражением, отнесены к пространству `System.Reflection`.

Хотя это и не сказывается на использовании класса `System.Type`, стоит отметить, что он является частью весьма сложной иерархии классов. Данный класс непосредственно происходит от класса `System.Reflection.MemberInfo`, который может описывать любой член класса или структуры. Класс `System.Type` также реализует интерфейс `System.Reflection.IReflect`, определяющий различные методы получения членов типа.

Получение ссылки типа `System::Type`

Ссылка типа `System::Type` может быть получена двумя способами:

- С помощью метода `System::Object::GetType()`
- С использованием оператора `TypeOf()`

Если есть доступ к существующему экземпляру объекта и нужна информация о его типе, то можно воспользоваться методом `GetType()`. Предположим, например, что определен следующий класс:

```
__gc class MyClass
{
    // и т.д.
};
```

Тогда можно написать:

```
MyClass* pMine = new MyClass;
Type* pType = pMine->GetType();
```

Поскольку метод `GetType()` определен в классе `System::Object`, все управляемые объекты автоматически поддерживают этот метод. Иными словами, класс или структура C++, определяемые с ключевым словом `__gc` или `__value`, будут заведомо поддерживать метод `GetType()`, и для этого ничего больше не нужно делать. Однако типы, в объявлении которых стоит ключевое слово `__value` (типы-значения), необходимо упаковывать перед вызовом метода `GetType()`. Предположим, что определен тип-значение `MyStruct`:

```
__value struct MyStruct
{
    // и т.д.
};
```

Эта структура должна быть упакована, прежде чем будет вызван метод `GetType()`:

```
MyStruct myStruct;
Value* pmyObj = __box(myStruct);
Type* pType = pmyObj->GetType();
```

Преимуществом метода `GetType()` является то, что при его использовании в период компиляции не нужна информация о типе экземпляра объекта. Так, в приведенном выше примере указатель `pMine` не обязательно должен указывать на экземпляр класса `MyClass`. Вместо этого он может указывать на экземпляр какого-либо класса, производного от `MyClass`. В этом случае метод `GetType` позволяет получить объект `Type`, правильно описывающий производный класс.

Если доступ к конкретному экземпляру объекта отсутствует, однако на стадии компиляции известно имя класса, можно воспользоваться оператором `__typeof()`, как это делалось в предыдущих примерах:

```
Type* colorType = __typeof(Color);
```

Оператор `__typeof()` обеспечивает меньшую гибкость, чем метод `GetType()`, поскольку в первом случае во время компиляции необходимо знать имя класса, однако при этом не нужно инициализировать объект. Если требуется получить информацию о типе для абстрактного класса, то оператор `__typeof()`, очевидно, является единственным вариантом.

Важно иметь в виду, что экземпляры класса `Type` никогда не создаются непосредственно. Это невозможно сделать, поскольку `System::Type` является абстрактным

классом. Таким образом, можно сделать вывод (хотя об этом и не сказано в документации), что при определении любого управляемого типа платформа .NET Framework неявно создает соответствующий класс (производный от `System.Type`), описывающий исследуемый тип. Однако, с точки зрения разработчика, этот процесс является нематериальным: мы просто получаем ссылку на тип, используя одну из двух описанных выше методик.

Важно то, что нельзя создать более одного экземпляра объекта `Type`, описывающего каждый управляемый тип. Если попытаться получить два объекта `Type`, описывающих один и тот же класс, в результате мы будем иметь два указателя на один и тот же объект. Поэтому можно непосредственно сравнивать ссылки типа `Type`, чтобы узнать, относятся ли они к одному и тому же классу:

```
Type* pType = __typeof(MyClass);
Type* pAnotherType = __typeof(MyClass);
if(pType == pAnotherType) // Это условие всегда будет справедливо.
{
    // и т.д.
}
```

Свойства и методы класса `System.Type`

Объект `Type` поддерживает большое количество методов и свойств, позволяющих получать информацию о типе, который представлен этим объектом. Приведенные ниже таблицы содержат неполный перечень членов класса `Type`, начиная с его свойств:

Свойство	Назначение
Assembly	Возвращает экземпляр класса <code>System.Assembly</code> , описывающий сборку, в которой определен данный тип.
BaseType	Позволяет получить тип, от которого наследует данный тип. Свойство возвращает другой объект типа <code>Type</code> . Если двигаться далее вверх по цепочке наследования, можно обратиться к базовому типу возвращаемого объекта и т.д. До тех пор, пока мы не дойдем до класса <code>System.Object</code> .
IsAbstract	Указывает, является ли данный тип абстрактным.
IsArray	Показывает, является ли тип массивом.
IsClass	Возвращает значение, показывающее, относится ли тип к управляемым классам.
IsEnum	Указывает, принадлежит ли тип к перечислениям, объявленным с ключевым словом <code>enum</code> .
IsInterface	Показывает, является ли тип интерфейсом .NET.
IsPrimitive	Указывает, относится ли данный тип к типам-примитивам .NET (таким, как <code>Double</code> , <code>Byte</code> , <code>Short</code> и т.д.).
IsSealed	Показывает, является ли тип изолированным.
Namespace	Возвращает в строковом виде имя того пространства имен, к которому принадлежит данный тип.

Все эти свойства предназначены только для чтения. Невозможно использовать экземпляр типа `Type` для того, чтобы каким-либо образом изменить определение типа. Можно лишь получить информацию о существующем определении данного типа.

Метод	Назначение
<code>GetConstructor()</code>	Возвращает ссылку типа <code>ConstructorInfo</code> , которая представляет подробную информацию о конструкторе, принимающем заданные параметры.
<code>GetConstructors()</code>	Возвращает массив типа <code>ConstructorInfo</code> , представляющий детали всех конструкторов данного типа.
<code>GetEvent()</code>	Возвращает объект <code>EventInfo</code> , содержащий информацию о событии с заданным именем.
<code>GetEvents()</code>	Возвращает массив типа <code>EventInfo</code> , содержащий подробную информацию обо всех событиях, определенных в данном типе.
<code>GetField()</code>	Возвращает ссылку типа <code>FieldInfo</code> , описывающую поле с заданным именем.
<code>GetFields()</code>	Возвращает массив типа <code>FieldInfo</code> , который подробно описывает все поля, определенные в данном типе.
<code>GetInterface()</code>	Возвращает ссылку типа <code>Object</code> , описывающую интерфейс с заданным именем.
<code>GetInterfaces()</code>	Возвращает массив типа <code>Object</code> , содержащий данные обо всех интерфейсах, определенных в данном типе.
<code>GetMembers()</code>	Возвращает ссылку типа <code>MemberInfo</code> , которая описывает член, имеющий заданное имя.
<code>GetMembers()</code>	Возвращает массив типа <code>MemberInfo</code> , содержащий информацию обо всех членах, определенных в данном типе.
<code>GetMethod()</code>	Возвращает ссылку типа <code>MethodInfo</code> , которая описывает метод, имеющий заданное имя.
<code>GetMethods()</code>	Возвращает массив типа <code>MethodInfo</code> , содержащий информацию обо всех методах, определенных в данном типе.
<code>GetProperty()</code>	Возвращает ссылку типа <code>PropertyInfo</code> , которая описывает свойство с заданным именем.
<code>GetProperties()</code>	Возвращает массив типа <code>PropertyInfo</code> , содержащий информацию обо всех свойствах, определенных в данном типе.
<code>IsSubclassOf()</code>	Указывает, происходит ли данный тип прямо или косвенно от конкретного класса.

Видно, что многие из этих методов возвращают экземпляры других классов, таких как `MethodInfo`, `MemberInfo` и `PropertyInfo`. Мы уже сталкивались с классом `PropertyInfo`, который используется для получения информации о свойстве, а также с классом `MethodInfo`, выполняющим аналогичные функции по отношению к методу. Имена классов отражают их назначение. Так, класс `EventInfo` предоставляет информацию о событии и т.д. Поскольку таких классов довольно много, а их назначение интуитивно понятно, нет необходимости подробно обсуждать все эти классы.

Перечисление типов: пример с музыкальными инструментами

До сих пор мы использовали отражение для вызова свойств и методов заданного типа к даже для инициализации объекта этого типа. Однако нам заранее нужна

была информация о типе, полученная с помощью оператора `_.typeof()` (с указанием полностью квалифицированного имени типа) или с использованием метода `System::Object::GetType()`, применяемого к экземпляру объекта. В настоящем разделе мы обсудим ситуацию, когда нет даже такой информации, когда имеется только имя сборки, которую мы хотим исследовать. Задача состоит в том, чтобы определить, какие типы содержит эта сборка.

Предположим, что создается некоторое музыкальное ПО. Имеется набор музыкальных инструментов, представленный библиотекой классов, содержащихся в сборке, которая может быть изменена независимым образом с помощью любого клиентского приложения. Каждый класс представляет один инструмент; соответственно, указанная сборка будет содержать такие типы, как `Guitar` (Гитара) и `Clarinet` (Кларнет). К моменту завершения разработки приложения эти классы будут иметь достаточно сложную функциональность, однако пока каждый из них реализует только один метод `PlayNote()`, который принимает строку, описывающую исполняемую ноту. Мы не хотим углубляться в написание кода для интерфейса `DirectX audio`, и наша идея проигрывания ноты будет сводиться к выводу в окно консоли строки, которая должна содержать информацию типа: "Guitar is playing note C".

Музыкальное ПО должно иметь некоторый пользовательский интерфейс, который даст пользователю возможность выбирать нужный музыкальный инструмент из всех, имеющихся в наличии. При этом необходимо, чтобы в случае обновления библиотеки интерфейс автоматически адаптировался к любым новым инструментам. Это означает, что интерфейс должен иметь возможность проверять сборку, содержащую библиотеку, и программным образом определять, какие инструменты фактически там содержатся. Однако мы не можем просто предположить, что любой класс в сборке представляет музыкальный инструмент. Там, например, могут быть служебные классы и структуры. Поэтому реализуем отбор интересующих нас классов исходя из их базового класса. В нашем случае все музыкальные инструменты будут происходить от абстрактного класса `InstrumentBase`.

Теперь, когда задача сформулирована, обратимся к библиотеке, содержащейся в проекте `InstrumentLibrary`, который включен в загружаемый код для данной главы. Это библиотека создана как проект `Managed C++ Class Library`.

```
// Файл InstrumentLibrary.h.
```

```
frpragma once
```

```
using namespace System;
```

```
using namespace System::Collections;
```

```
namespace InstrumentLibrary
```

```
{
```

```
    public __gc __abstract class InstrumentBase
```

```
    public:
```

```
        virtual void PlayNote(String* note) = 0;
```

```
};
```

```
    public __gc class Guitar : public InstrumentBase
```

```
    {
```

```
    public:
```

```
        virtual void PlayNote(String* note)
```

```
        {
            Console.WriteLine(S"Guitar playing note '{0}'", note);
        }
    }
```



```

using namespace System;
using namespace InstrumentLibrary;

// Это точка входа в приложение
int _tmain(void)
{
    Assembly* library = Assembly::Load(S"InstrumentLibrary");
    Type* instBaseType =
        library->GetType(S"InstrumentLibrary.InstrumentBase");
    Type* types[] = library->GetTypes();
    for(int i = 0; i < types->Count; i++)
    {
        Type* inst = types[i];
        if(inst->IsAbstract)
            continue;
        if(inst->IsSubclassOf(instBaseType))
        {
            Console.WriteLine(types[i]->Name);
            Object* instObj = Activator.CreateInstance(inst);
            InstrumentBase* pinst = dynamic_cast<InstrumentBase*>(instObj);
            pinst->PlayNote(S"C");
        }
    }

    return 0;
}

```

Здесь после обычного начального кода мы загружаем сборку, используя метод `Assembly::Load()`:

```
Assembly* library = Assembly::Load(S"InstrumentLibrary");
```

Класс `Assembly` определен в пространстве имен `System::Reflection` и имеет статический метод `Load()`, загружающий в приложение сборку с заданным именем (если она не была загружена ранее). Обычно сборка загружается неявно, если был вызван какой-либо тип, определенный в данной сборке, но это не относится к нашему примеру. Метод `Load()` возвращает указатель на экземпляр класса `Assembly`, который описывает исследуемую сборку. Этот указатель затем используется для вызова нестатического метода `Assembly::GetType`:

```
Type* instBaseType = library->GetType(S"InstrumentLibrary.InstrumentBase");
```

Метод `GetType` имеет несколько перегруженных версий. Здесь используется самая простая из них. В качестве параметра она принимает полностью квалифицированное имя типа, определяемого в данной сборке, и возвращает указатель на объект `Type`, который описывает исследуемый тип. В данном случае мы получаем объект `Type`, описывающий класс `InstrumentBase`. Это нужно для того, чтобы проверить, являются ли другие типы в библиотеке производными от `InstrumentBase` (т.е. представляют ли они музыкальные инструменты).

Полностью квалифицированное имя класса `InstrumentBase` записывается с использованием синтаксиса `.NET(C#)`, а не `C++`. Поэтому пространства имен отделяются точками, а не парой двоеточий.

Далее мы вызываем метод `Assembly::GetTypes()`, похожий на `Assembly::GetType()`. Однако вместо возвращения объекта `Type` для типа с заданным именем метод `Assembly::GetTypes()` получает все типы, определенные в сборке, и возвращает массив объектов типа `Type`:


```
Type* types[] = library->GetTypes();
```

Теперь необходимо перебрать все элементы массива и проверить, какие типы происходят от класса `InstrumentBase`, инициализировать объекты для каждого из таких типов и вызвать метод `PlayNote()` для всех этих объектов. Выяснить, происходит ли данный конкретный тип от другого заданного типа, позволяет метод `System::Type::IsSubclassOf()`, возвращающий булево значение. Мы также проверяем, является ли класс абстрактным, и если это так, не выполняем дальнейшую обработку. Очевидно, что, если попытаться инициализировать абстрактный класс, возникнут проблемы!

```
for(int i=0; i < types->Count; i++)
{
    Type* inst = types[i];
    if(inst->IsAbstract)
        continue;
    if(inst->IsSubclassOf(instBaseType))
```

Инициализация объекта выполняется с помощью статического метода `Activator::CreateInstance()`. `Activator` — это еще один класс, определенный в пространстве имен `System::Reflection`. Он позволяет инициализировать объекты других классов. Простейшая перегруженная версия метода `CreateInstance` принимает один параметр типа `System::Type`, определяющий тот класс, экземпляр которого нужно создать. Эта версия применима, если исследуемый тип имеет конструктор без параметров. В нашем примере реализуется как раз такой случай. Если же в конструктор необходимо передавать параметры, можно использовать другие перегруженные версии метода `CreateInstance()`, принимающие дополнительные параметры и передающие их далее.

После создания объекта необходимо вызвать его метод `PlayNote()`. Но, поскольку метод `Activator::CreateInstance()` возвращает указатель типа `Object`, нужно привести этот указатель к типу `InstrumentBase*`, чтобы получить доступ к методу `PlayNote()`:

```
{
    Object* instObj = Activator::CreateInstance(inst);
    InstrumentBase* pinst = dynamic_cast<InstrumentBase*>(instObj);
    pinst->PlayNote("C");
}
```

На этом завершается написание кода для проекта `PlayInstruments`. В результате выполнения данного кода на экран будет выведена следующая информация:

```
Guitar (Гитара)
Guitar playing note C (Гитара исполняет ноту "до")
Saxophone (Саксофон)
Saxophone playing note C (Саксофон исполняет ноту "до")
Clarinet (Кларнет)
Clarinet playing note C (Кларнет исполняет ноту "до")
```

Как видим, клиентское приложение правильно выбрало из библиотеки только те классы, которые представляют музыкальные инструменты.

Комбинирование атрибутов и отражения

Теперь объединим две обсуждаемые нами технологии, расширив библиотеку музыкальных инструментов так, чтобы она использовала атрибуты. Наше приложение показывает, что можно идентифицировать музыкальные инструменты,

имеющиеся в библиотеке, инициализировать их и вызывать их методы. Однако *нельзя* получить дополнительную информацию об этих инструментах (не считан информации, доступной через статические свойства класса инструмента, которые в нашем случае отсутствуют). Например, музыкальные инструменты делятся на группы: деревянные духовые, медные духовые, струнные и т.д. Поэтому вполне естественно затребовать, чтобы приложение распознавало эти категории. Для реализации такой функциональности воспользуемся атрибутами.

Создадим два новых проекта: `InstrumentLibraryWithAttributes` и `PlayInstrumentWithAttributes`. Неудивительно, что эти проекты содержат почти такой же код, как `InstrumentLibrary` и `PlayInstruments`, но с добавлением некоторых новых элементов.

Вначале обратимся к библиотеке. Определим атрибут `InstrumentCategory`, который будет принимать один параметр, указывающий, к какой категории относится инструмент. Нужно также должны определить перечисление .NET, содержащее возможные категории. Код нашего нового атрибута имеет вид:

```
public value enum InstrumentCategory
{
    Strings, Woodwind, Percussion, Piano, Brass;
}

public __gc class InstrumentCategoryAttribute : public Attribute
{
public:
    InstrumentCategoryAttribute(InstrumentCategory category)
    {
        Category = category;
    }

    InstrumentCategory.....
```

Этот атрибут имеет один конструктор, который принимает элемент перечисления в качестве параметра. Для упрощения задачи мы не объявляем свойство для категории, а просто реализуем ее как открытый элемент данных.

После указания нового атрибута необходимо изменить код, определяющий классы в библиотеке инструментов, добавляя атрибут к нужным элементам. Ниже представлен обновленный код. Все изменения в нем выделены, а определения классов для краткости опущены:

```
public __gc__abstract class InstrumentBase
{
    // Определение класса
};

[InstrumentCategory(InstrumentCategory::Strings)]
public __gc class Guitar : public InstrumentBase
{
    // Определение класса
};

[InstrumentCategory(InstrumentCategory::Woodwind)]
public __gc class Saxophone : public InstrumentBase
{
    // Определение класса
};

[InstrumentCategory(InstrumentCategory::Woodwind)]
public __gc class Clarinet : public InstrumentBase
{
    // Определение класса
};
```

```

        // Определение класса
    };

    public __gc class InstrumentCollection
    {
        // Определение класса
    };

```

Как и в предыдущем примере, нужно откомпилировать данную библиотеку и скопировать полученную сборку в ту папку, где находится вызывающая сборка (или в подкаталог этой папки).

Далее скопируем код из проекта `PlayInstruments` в проект `PlayInstrumentsWithAttributes`, а затем изменим этот код таким образом, чтобы он мог считывать информацию об атрибуте `InstrumentCategory` и выводить на экран категорию каждого инструмента:

```

int _tmain(void)
{
    Assembly* library = Assembly::Load("InstrumentLibrary");
    Type* instBaseType =
        library->GetType("InstrumentLibrary.InstrumentBase");
    Type* types[] = library->GetTypes();

    for(int i = 0; i < types->Count; i++)
    {
        Type* inst = types[i];
        if(inst->IsAbstract)
            continue;
        if(inst->IsSubclassOf(instBaseType))
        {
            Console::WriteLine(types[i]->Name);
            Object* pAttribs[] = inst->GetCustomAttributes(
                typeof(InstrumentCategoryAttribute), true);
            for(int j = 0; j < pAttribs->Length; j++)
            {
                InstrumentCategoryAttribute* pCat =
                    dynamic_cast<InstrumentCategoryAttribute*>(pAttribs[j]);
                Console::WriteLine("box(pCat->Category)->ToString());
            }
            Object* instObj = Activator::CreateInstance(inst);
            InstrumentBase* pinst = __try_cast<InstrumentBase*>(instObj);
            pinst->PlayNote("C");
        }
    }
    return 0;
}

```

Для того чтобы получить атрибут класса, мы используем метод `Type::GetCustomAttributes()`, возвращающий массив указателей типа `Object`. Каждый из них указывает на экземпляр атрибута, который был добавлен к опрашиваемому типу. Имеется две перегруженных версии данного метода. Та версия, что мы используем, принимает два параметра: объект типа `Type`, описывающий исследуемый атрибут, и булев параметр, указывающий, хотим ли мы получать атрибуты, унаследованные от базовых классов. Эта версия позволяет извлекать только атрибуты `InstrumentCategory`, которые присоединены к типу. Вторая перегруженная версия метода — `GetCustomAttributes()` принимает только булев параметр и возвращает все

атрибуты, использованные для описания типа. Однако в обоих случаях возвращается массив, даже если нас интересует лишь один атрибут, поскольку данный атрибут может использоваться более одного раза (если при определении атрибута значение флага `AllowMultiple` было установлено равным `true`).

Именно в этот момент инициализируется объект типа `InstrumentCategoryAttribute`, и можно убедиться в полезности определения атрибутов как классов. После получения массива атрибутов необходимо перебрать элементы этого массива. При этом каждый элемент массива преобразуется к типу `InstrumentCategoryAttribute*`, после чего член `Category` нашего атрибута используется для вывода на экран категории инструмента. Ниже представлена информация, которую мы увидим на экране:

Guitar (Гитара)

Strings (Струнные инструменты)

Guitar playing note C (Гитара исполняет ноту "до")

Saxophone (Саксофон)

Woodwind (Деревянные духовые инструменты)

Saxophone playing note C (Саксофон исполняет ноту "до")

Clarinet (Кларнет)

Woodwind (Деревянные духовые инструменты)

Clarinet playing note C (Кларнет исполняет ноту "до")

Этот пример не только показывает, как можно получать данные, содержащиеся в пользовательских атрибутах, но и демонстрирует важность определения атрибутов как классов и реализации параметров как открытых методов или членов этих классов. Процесс извлечения информации, содержащейся в атрибуте, включает инициализацию объекта типа `Attribute` с помощью отражения и последующее использование свойств данного объекта для определения значений параметров атрибута.

Здесь у читателя, скорее всего, возникнет вопрос: а действительно ли в данном приложении были нужны пользовательские атрибуты? Разве нельзя добиться того же результата, создав во всех классах инструментов редактируемое свойство, указывающее категорию инструмента, или же реализовав наследование классами от интерфейса, определяющего такое свойство? В принципе так можно поступить. Назначение пользовательского атрибута (по крайней мере, в приложении к классам) заключается в предоставлении дополнительной информации о классах, что, собственно, делает и интерфейс. Поэтому практически любое решение, связанное с присоединением пользовательских атрибутов к классу, может быть реализовано путем комбинирования интерфейсов и свойств. В определенной степени выбор той или иной методики зависит от субъективного представления разработчика о том, какое решение данной проблемы будет более изящным. Однако во многих ситуациях читатель, несомненно, придет к выводу, что лучшим вариантом является применение атрибутов.

ГЛАВА 5



Классы-утилиты .NET Framework

Платформа .NET Framework — это не первая попытка предоставить разработчикам C++ набор классов для решения стандартных задач. MFC, ATL, STL и стандартная библиотека C++ обеспечивают широкие возможности, позволяющие выполнять различные математические операции, обрабатывать строки и работать с регулярными выражениями. Однако, если код должен работать в среде CLR, то наилучшим вариантом является использование библиотеки классов .NET Framework.

В настоящей главе мы рассмотрим некоторые классы-утилиты, предоставляемые .NET Framework, и продемонстрируем их использование в приложениях C++. В частности, мы обсудим применение этих классов для работы с файлами, текстом и структурами данных, а также для организации потоков. Данная глава не является исчерпывающим справочным руководством: мы сосредоточимся на нескольких классах, а также на некоторых наиболее важных членах этих классов. Полная информация о классах, их членах и функциональности содержится в документации MSDN.

Обработка текстовых данных

Пространство имен `System::Text` содержит классы, предназначенные для форматирования объектов строкового типа, кодирования и декодирования данных, а также для определения правил проверки данных. Мы рассмотрим следующие классы:

- `System::Text::StringBuilder` — используется для динамического изменения содержимого строки.
- `System::Text::RegularExpressions::Regex` — применяется для проверки комплексных данных с использованием регулярных выражений.

Класс `StringBuilder`

Мы обсудим здесь еще один строковый класс, поскольку класс `String`, рассмотренный в предыдущей главе, в определенных ситуациях может оказаться весьма

неэффективным, особенно если приходится вставлять в строку многочисленные фрагменты текста, чтобы так или иначе ее редактировать. Класс `String` представляет собой постоянный тип данных, которые *нельзя изменить*. Это значит, что применение таких методов, как `Concat()`, `Replace()` и `Remove()` приводит не к изменению первоначальной строки, а к созданию новой. Когда в строку необходимо вносить многочисленные и частые изменения, использование класса `String` может отрицательно сказаться на производительности, и поэтому стоит подумать о применении класса `StringBuilder`.

Класс `StringBuilder` аналогичен классу `String`, однако память для его объектов выделяется иначе. Когда а управляемой куче создается новый экземпляр класса `StringBuilder`, при этом выделяется больше памяти, чем фактически требуется строке. Поэтому в строку можно добавлять символы и изменять ее без копирования результата в новую строку, которая должна поместиться в другую область памяти.

По умолчанию вся эта процедура выполняется неявно, однако можно задать емкость строки вручную с помощью свойства `Capacity` класса `StringBuilder`, которое отличается от свойства `Length`. Последнее определяет фактическую длину строки, представляемой классом. Однако с точки зрения возможностей применения класс `StringBuilder` уступает классу `String`, так как предоставляет гораздо менее богатую функциональность. Поэтому решение вопроса об использовании класса `StringBuilder` зависит от того, какие операции со строками придется выполнять.

Ниже приведен пример приложения, в котором применяется класс `StringBuilder`:

```
using namespace System;
using namespace System::Text;

int _tmain(void)
{
    // Создание нового экземпляра класса StringBuilder
    StringBuilder* l_ptrSBValue = new StringBuilder;
    Console::WriteLine(S"Initial Capacity is {0}",
        l_ptrSBValue->Capacity.ToString());
    Console::WriteLine(S"Initial Length is {0}",
        l_ptrSBValue->Length.ToString());

    // Добавление данных
    l_ptrSBValue->Append(S"Hello!\n");
    l_ptrSBValue->AppendFormat(S"Name = {0}\nAddress = {1}",
        S"Niranjan", S"No 226, Cathedral Road");

    DateTime l_dtCurrTime = DateTime::Now;
    l_ptrSBValue->Append(S"\nCurrent Date = ");
    l_ptrSBValue->Append(l_dtCurrTime.ToString(S"f"));

    // Вывод результата
    Console::WriteLine(
        S"\nStringBuilder object now contains:\n{0}", l_ptrSBValue);

    Console::WriteLine("After appending data the Capacity is {0}",
        l_ptrSBValue->Capacity.ToString());

    Console::WriteLine("After appending data the Length is {0}",
        l_ptrSBValue->Length.ToString());

    // Изменение содержимого
    l_ptrSBValue->Remove(0, 7);
    l_ptrSBValue->Replace(S"Name = ", S"welcome ");
}
```

```

    l_ptrSBValue->Insert(0, S"My Personal Details are shown below:\n");
    Console.WriteLine(S"\nContents of StringBuilder after
        modification:\n{0}", l_ptrSBValue);

    // Уменьшение емкости до длины объекта
    l_ptrSBValue->Capacity = l_ptrSBValue->Length;
    Console.WriteLine(S"Finally the Capacity is {0}",
        l_ptrSBValue->Capacity.ToString());

    return 0;
}

```

В процессе выполнения данного кода начальная емкость и длина строки, представляемой классом `StringBuilder`, выводятся в окно консоли с помощью свойств `Capacity` и `Length`. Затем в строку добавляются данные, и результат вновь выводится на экран. Мы используем такие методы класса `StringBuilder`, как:

- ❑ `AppendFormat()`: метод принимает форматированную строку, содержащую спецификации формата. В рассматриваемом примере мы заменяем спецификации формата строками, в которых должны храниться имя и адрес.
- ❑ `Append()`: метод принимает параметр `String*`, указывающий на добавляемый объект типа `String`.

Для того чтобы добавить в экземпляр класса `StringBuilder` объект типа `DateTime`, вызывается метод `ToStringO` этого объекта. Причем в качестве аргумента используется константа "f", задающая формат даты с указанием заголовка, месяца и текущего времени.

Затем мы изменяем содержимое строки с помощью таких методов класса `StringBuilder`, как `Replace()`, `Remove()` и `Insert()`. Хотя в примере заменяется лишь один экземпляр подстроки, метод `Replace()` выполняет замену всех вхождений искомой подстроки на новую строку. Метод `Remove()` имеет два параметра (исходная позиция и длина) и удаляет символы, задаваемые этими параметрами. В нашем случае длина равна 7, включая символ новой строки.

Метод `InsertO` служит для вставки строки в объект, начиная с заданной позиции, однако имеются перегруженные версии данного метода, реализованные в большинстве базовых типов. Хотя этот метод и версия метода `String::Insert` используются похожим образом, их поведение различается. Версия класса `String` не оказывает влияния на вызывающий объект, а возвращает результат в виде указателя на новую строку. Версия же типа `StringBuilder` вставляет передаваемую строку в вызывающий объект.

После выполнения этих операций объект типа `StringBuilder` автоматически получает необходимую память, и для иллюстрации произошедших изменений мы вновь выводим на экран длину и емкость строки. Скомпоновав и запустив па исполнение код проекта, получим следующий результат:

```

Initial Capacity is 16 (Начальная емкость равна 16)
Initial Length is 0 (Начальная длина равна 0)

```

```

The StringBuilder object now contains (Содержимое объекта StringBuilder):
Hello!

```

```

Name = Niranjan (Имя = Ниранджан)

```

```

Address = No 226, Cathedral Road (Адрес = No 226, Cathedral Road)

```

```

Current Date = 04 February 2002 12.56 (Текущая дата = 4 февраля 2002 г. 12.56)

```

```

After appending data the Capacity is 128

```

(После добавления данных емкость равна 128)
 After appending data the Length is 93 (После добавления данных длина равна 93)

Contents of StringBuilder after modification
 (Содержимое объекта типа StringBuilder после внесенных изменений):
 My personal details are shown below
 (Моя персональная информация представлена ниже):
 Welcome Niranjana
 Address = No 226, Cathedral Road (Адрес = No 226, Cathedral Road)
 Current Date = 04 February 2002 12.56 (Текущая дата = 4 февраля 2002 г. 12.56)
 Finally, the Capacity is 128 (Емкость после внесенных изменений равна 128)
 Finally, the Length is 124 (Длина после внесенных изменений равна 124)

Отсюда видно, что начальная емкость объекта класса StringBuilder равна 16, хотя сама строка является пустой. Однако после добавления данных емкость автоматически увеличивается до 128 без всякого участия с нашей стороны. В процессе дальнейших манипуляций со строкой емкость остается прежней, хотя длина возрастает.

Можно попытаться вручную сократить емкость до величины, меньшей, чем длина текущей строки:

```
l_ptrSBValue->Capacity = 100;
```

Однако это приведет к генерации исключительной ситуации типа `ArgumentOutOfRangeException`. Поэтому, задавая свойство `Capacity` вручную, необходимо быть весьма осторожным.

Класс `Regex`

Платформа .NET Framework облегчает проверку сложных данных благодаря набору классов-утилит, входящих в пространство имен `System.Text.RegularExpressions`. Мы ограничимся рассмотрением важнейшего из этих классов, `Regex`, который выполняет сопоставление регулярных выражений со строковыми объектами. Класс `Regex` имеет несколько перегруженных методов, делающих более эффективной процедуру поиска сложных выражений.

Объект типа `Regex` можно инициализировать, передавая строку в конструктор. Приведенный ниже пример иллюстрирует процесс поиска строки "is" в другой строке:

```
String* myText = "This is a big silver whistle.";
Regex* firstRegex = new Regex("is");
MatchCollection* myMatches = firstRegex->Matches(myText);
Console::WriteLine("Looking for \"is\": number of matches = {0}",
    myMatches->Count.ToString());
```

Метод `Matches()` возвращает объект класса `MatchCollection`, представляющий собой коллекцию объектов типа `Match`, каждый из которых соответствует совпадению, обнаруженному в строке. Класс `Regex` имеет похожий метод `Match()`, который возвращает один объект типа `Match`, относящийся к первому найденному совпадению. Число совпадений можно узнать, вызвав свойство `Count` класса `MatchCollection`. В нашем примере таких совпадений 3.

Можно также вызвать метод `Matches` как статический метод класса `Regex`. При этом в качестве параметров должны передаваться как проверяемая строка, так и образец:

```
myMatches = Regex::Matches(myText, "this");
Console::WriteLine("Looking for \"this\": number of matches = {0}",
    myMatches->Count.ToString());
```


Поскольку по умолчанию регулярное выражение чувствительно к регистру, здесь совпадений не обнаружено. Однако можно указать, чтобы регистр игнорировался, используя одну из перечислимых величин типа `RegexOptions`. В нашем примере это будет величина `IgnoreCase`:

```
myMatches = Regex.Matches(myText, S"this", RegexOptions.IgnoreCase);
Console.WriteLine(
    S"Looking for \\this\\ (IgnoreCase): number of matches = {0}",
    myMatches.Count.ToString());
```

На этот раз будет обнаружено совпадение со строкой "this". Величина, принадлежащая к перечислению `RegexOptions`, может быть также передана как дополнительный параметр в конструктор класса `Regex`.

Часто бывает необходимо искать целое слово, а не подстроку. Вернемся к первому примеру и зададим регулярное выражение так, чтобы оно обозначало отдельное слово "is":

```
Regex secondRegex = new Regex(S"\\bis\\b");
myMatches = secondRegex.Matches(myText);
Console.WriteLine(S"Looking for \\is\\ (whole word): number of
    matches = {0}");
myMatches.Count.ToString());
```

Символ перехода `\\b` в образце регулярного выражения обозначает границу слова. В C++ необходимо использовать две обратные косые черты для того, чтобы этот символ передавался в регулярное выражение из строки. Имеются и другие символы перехода, которые при поиске можно использовать в качестве разделителей:

- `\\t` - табуляция
- `\\n` - новая строка
- `\\r` - возврат каретки

Класс `Regex` также поддерживает методы, которые позволяют изменять проверяемую строку в зависимости от результатов поиска. Например:

```
String* myNewText = Regex.Replace(myText, S"whistle", S"bell");
Console.WriteLine(myNewText);
```

Здесь используется метод `Replace()`, в котором первый параметр задает изменяемую строку, второй представляет собой искомый образец, а третий определяет фрагмент текста, который должен быть вставлен в строку вместо старого фрагмента. Данный метод возвращает измененную строку. Метод также имеет много перегруженных версий. Для получения их полного перечня необходимо обратиться к документации MSDN.

Образцы поиска

До сих пор мы имели дело лишь с простыми регулярными выражениями. Рассмотрим теперь более сложные образцы для поиска. Создадим приложение, которое проверяет входные данные, такие как номер телефона, адрес электронной почты и номер кредитной карты. Если входные данные некорректны, пользователю будет предложено повторить попытку. Начнем с ввода имени:

```
Console.WriteLine(S"Enter your name: ");
strName = Console.ReadLine();
Regex ptrREName = new Regex(S"^[A-Z][a-z]+[ ]?$");
while (!ptrREName.IsMatch(strName))
```

```
{
    Console.WriteLine("Name is not valid! Try again: ");
    strName = Console.ReadLine();
}
```

Этот код инициализирует объект, используя образец "[A-Z][a-z]+[]?+\$". Здесь символы [A-Z] указывают, что буква должна быть заглавной. Далее часть выражения [a-z]+ задает строчные буквы, где знак + имеет смысл "один или несколько". Затем следуют символы необязательного пробела, где знак ? указывает, что пробелов может быть 0 или 1. Если мы хотим задать 0 или более пробелов, необходимо использовать знак *. Потом все выражение помещается в скобки и к нему добавляется еще один знак +. Это позволяет проверять одно или более слов, начинающихся с заглавной буквы и разделенных пробелами, что представляет собой обычную форму записи имени. Наконец, обратите внимание на символы " и \$ в начале и в конце образца. Эти специальные символы указывают, в какой части текста располагается образец. Знак " говорит о том, что образец должен находиться в начале текста, а символ \$ — что образец должен быть в конце. Одновременное использование обоих символов исключает возможность наличия любого другого текста в проверяемой строке. Такой метод проверки очень полезен.

Вставшейся части раздела основное внимание мы обратим на сами регулярные выражения, а не на то, как выглядит весь код. Полностью познакомиться с текстом программы читатель может, загрузив код с сайта издательства Wrox.

Ниже представлено регулярное выражение, используемое для проверки телефонного номера:

```
Regex ptrREPhone = new Regex(@"^\d{3}[-\s]?\d{3}[-\s]?\d{4}$");
```

Здесь мы ищем последовательность цифр, указываемую с помощью символа перехода \d. Предполагается, что телефонный номер должен иметь такой формат: 3 цифры, за которыми следуют также 3 цифры, а затем идут 4 цифры. Между этими группами цифр могут помещаться необязательные разделители, представляющие собой пробелы или дефисы.

Далее записывается регулярное выражение для проверки адреса электронной почты:

```
Regex ptrREEmail = new Regex(@"^\w+([\.]\w+)*@\w+([\.]\w+)$");
```

Здесь знак перехода \w указывает на любой из символов, составляющих слово, такой как буква, цифра или символ подчеркивания. Мы предполагаем, что адрес электронной почты состоит из одного или нескольких слов, разделенных точками, после чего следует символ @, за которым вновь идет одно или несколько слов, отделенных друг от друга точками. В соответствии с этим определением корректными будут такие адреса:

□ bridget@wrox.com

□ bridget.jones@mail.wrox.co.uk

Более сложное выражение могло бы использоваться для проверки последней части адреса, чтобы убедиться, соответствует ли окончание адреса тому или иному домену верхнего уровня.

Наконец, с помощью регулярного выражения мы проверяем номер кредитной карты:

```
Regex ptrRECardNumber = new Regex(@"^\d{4}-?\d{4}-?\d{4}-?\d{4}$");
```

Данное выражение указывает, что номер может состоять из 16 цифр, разделенных (не обязательно) дефисами на группы по 4 цифры.

Мы создали лишь небольшое демонстрационное приложение, иллюстрирующее работу с регулярными выражениями. Однако даже простые примеры, приведенные здесь, показывают возможности, предоставляемые классами регулярных выражений .NET. Другие классы в пространстве имен `System::Text::RegularExpressions` включают функциональность, предназначенную для работы с "трофеями" (captures), т.е. с объектами, которые сохраняют предыдущие результаты поиска совпадений. Это позволяет устанавливать более сложные правила проверки и расширять возможности манипулирования строками.

Работа с файлами

Обсудим теперь классы .NET, с помощью которых можно работать с файлами. Это классы `System::IO::File`, `System::IO::FileStream`, `System::IO::StreamReader` и `System::IO::StreamWriter`.

Классы File и FileStream

Пространство имен `System::IO`, принадлежащее к платформе .NET Framework, содержит классы, необходимые для управления вводом и выводом данных. Сюда входят классы, позволяющие работать с файлами, исследовать каталоги, управлять массивами байтов, манипулировать текстовыми и двоичными данными, обрабатывать соответствующие исключительные ситуации и т.д.

Ниже рассмотрено приложение, принимающее имена двух файлов в качестве входных данных, а затем добавляющее содержимое второго файла в конец первого. Основное внимание мы уделим методам работы с файлами. Полный текст программы можно найти в загружаемом проекте `FileCat`.

Открытие файлов

Файлы можно открывать с помощью статических методов класса `File`. Нам необходимо открыть исходный файл в режиме чтения и целевой файл в режиме дозаписи:

```
FileStream* l_fsSource = File::OpenRead(l_strSourceFileName);
FileStream* l_fsTarget = File::Open(l_strTargetFileName, FileMode::Append);
```

Метод `File::OpenRead()` принимает в качестве параметра путь к файлу, открывает файл в режиме чтения и возвращает указатель типа `FileStream*` (это похоже на открытие файла в режиме "r" при вызове функции `fopen()` языка C). Второй аргумент в методе `File::Open()` указывает режим открытия файла (в нашем случае это режим дозаписи — `Append`) и также возвращает указатель типа `FileStream*`, что эквивалентно открытию файла в режиме "a" с помощью функции `fopen()`. Если файл, указанный в параметре какого-либо из двух методов, отсутствует, будет выдана исключительная ситуация `FileNotFoundException`. Поэтому необходимо поместить приведенный код в блок `try` и добавить блок `catch` для обработки указанной исключительной ситуации.

Чтение из файла и запись в файл

Для чтения и записи данных с помощью объекта `FileStream` необходим управляемый массив типа `unsigned char`. Следующий код объявляет такой массив, а также целую переменную `l_iBytesRead`, которая будет использована при чтении данных из исходного файла. Для константы препроцессора `MAX_VALUE` определено значение 255:

```
unsigned char l_a_rgContents_gc[] = new unsigned char_gc[MAX_VALUE];
int l_iBytesRead = 0;
```

Для чтения данных из объекта `FileStream` мы вызываем метод `FileStream::Read()`, передавая туда в качестве аргументов управляемый массив, позицию в массиве, с которой начинается запись, и количество считываемых байт. После успешного завершения чтения данных из файла метод возвращает количество прочитанных и сохраненных в массиве байт. Когда достигается конец файла, метод `Read()` возвращает нуль. Следующий фрагмент кода прочитывает весь исходный файл в цикле `while` и записывает полученные данные в целевой файл, используя метод `FileStream::Write()`:

```
Console.WriteLine("Contents of the Source file\n*****");
while (l_iBytesRead = l_fsSource->Read(l_arrContents, 0, MAX_VALUE))
{
    // Преобразуем массив unsigned char в строку и выводим ее в окно консоли.
    String *l_strContents = Encoding.ASCII->GetString(l_arrContents);
    Console.WriteLine(l_strContents);

    // Запись в файл
    l_fsTarget->Write(l_arrContents, 0, l_iBytesRead);
    l_fsTarget->Flush();

    // Инициализация массива
    int iCtr = 0;
    for (; iCtr < MAX_VALUE ; iCtr++)
    {
        l_arrContents->Item[iCtr] = '\0';
    }
}
```

Метод `FileStream::Write()` похож на свой аналог. Его аргументами являются массив, начальная позиция в массиве и количество записываемых байт. В цикле `while` мы сохраняем число полученных из файла байт, используя целую переменную `l_iBytesRead`. Кроме того, управляемый массив `unsigned char` преобразуется в строку с помощью класса `System::Text::Encoding`, что позволяет вывести содержимое массива на консоль. Метод `Flush()` записывает все буферизованные данные и очищает содержимое потока. Этот процесс аналогичен тому, который реализуется с помощью функции `fflush(FILE*)` языка C. Наконец, мы инициализируем массив в цикле `for`, чтобы очистить его для выполнения следующей операции чтения.

Закрываем файлы

Следующий фрагмент кода показывает, как закрыть файлы с помощью метода `FileStream::Close()`:

```
l_fsSource->Close();
l_fsTarget->Close();
```

Метод `FileStream::Close()` записывает данные, хранящиеся в буфере (если таковые имеются), и закрывает файл. Перед прекращением работы приложения среда исполнения сама закрывает все файлы, открытые программой, даже если мы не вызываем метод `FileStream::Close()`. Однако лучше делать это программным образом и не зависеть от системы.

Альтернативный способ реализовать описанную выше операцию чтения-записи состоит в том, чтобы выполнить все три этапа в одном цикле `while`, используя свойство `FileStream::Length`, которое возвращает длину потока. Можно объявить массив соответствующей длины, а затем осуществить чтение и запись. Представленный ниже фрагмент кода иллюстрирует такой подход:

```
// Определение длины
__int64 l_len = l_fsSource->Length;
```

```
// Объявление массива
unsigned char l_arrContents[_gc[]] = new unsigned char _gc[l_len]

// Чтение входного файла
l_fsSource->Read(l_arrContents, 0, l_len)

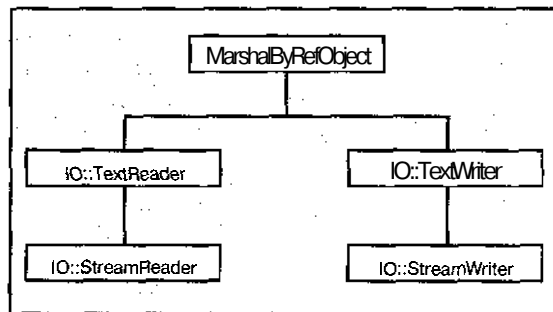
// Запись в выходной файл
l_fsTarget->Write(l_arrContents, 0, l_len);
```

Однако при использовании данного подхода могут возникнуть проблемы, если окажется, что входной файл имеет слишком большой размер (исчисляемый мегабайтами) и программа не сможет выделить необходимый объем памяти. Поэтому желательно выполнять в цикле операции чтения и записи, как это было сделано ранее, а не решать всю задачу в один прием.

Классы **StreamReader** и **StreamWriter**

ЕСЛИ платформа .NET включает класс `FileStream` для выполнения операций файлового ввода-вывода, то зачем нужны еще классы `StreamReader` и `StreamWriter`, реализующие те же функции? Дело в том, что эти классы работают с данными, представленными в строковой, а не в байтовой форме. Неразумно тратить время, преобразуя данные из строкового формата в управляемый массив `unsigned char` и обратно. Поэтому в тех случаях, когда нужно работать только со строками, указанные классы дают большое преимущество.

Оба рассматриваемых класса происходят от класса `MarshalByRefObject`. Соответствующая иерархия показана на рисунке:



Класс `StreamReader` применяется для открытия файла в режиме чтения, а с помощью класса `StreamWriter` файл открывается в режиме записи. Оба класса поддерживают кодирование при чтении и сохранении данных. Для иллюстрации классов `StreamReader` и `StreamWriter` напомним программу, которая копирует текстовые файлы, если в командной строке задаются два аргумента: имена исходного и целевого файлов. Полный текст этой программы можно найти в проекте `FileCopy`.

В приведенной ниже программе имена файлов передаются в аргументы конструкторов и создаются соответствующие потоковые объекты:

```
StreamReader *l_swSource = new StreamReader(l_strSourceFileName);
StreamWriter *l_swTarget = new StreamWriter(l_strTargetFileName);
```

Следующий шаг состоит в копировании файла с помощью методов `StreamReader::ReadLine()` и `StreamWriter::WriteLine()`. Операция копирования выполняется до тех пор, пока не будет достигнут конец файла. Для того чтобы определить этот момент, используется метод `StreamReader::Peek()`. Метод возвращает следующий

символ в потоке (без изменения текущей позиции) или -1, если процесс чтения дошел до конца файла. Представленный ниже фрагмент кода реализует основную логику приложения:

```

Console.WriteLine($"Contents of the file\n*****");
String *l_strContents = S"";

for (; l_swSource->Peek() > -1; )
{
    // Чтение строки
    l_strContents = l_swSource->ReadLine();
    // Запись в файл
    l_swTarget->WriteLine(l_strContents);
    l_swTarget->Flush();

    // Вывод на консоль
    Console.WriteLine(l_strContents);
}

Console.WriteLine($"*****");
Console.WriteLine($"The file '{0}' successfully copied to '{1}' ",
    l_strSourceFileName, l_strTargetFileName);

```

После копирования данных в целевой файл они выводятся на экран. Теперь остается лишь закрыть потоки с помощью метода `Close()`:

```

// Закрываем файлы
l_swSource->Close();
l_swTarget->Close();

```

Как и ранее, в загружаемом проекте весь этот код помещен в блок `try`, после которого в программе идут соответствующие блоки `catch`.

Коллекции

Теперь обратимся к некоторым из важнейших классов коллекций, предоставляемых платформой .NET Framework и относящихся к пространству имен `System::Collections`. Мы рассмотрим также интерфейсы, реализуемые этими классами. К наиболее важным интерфейсам, принадлежащим к указанному пространству имен, относятся:

- ❑ `ICollection` - определяет коллекцию объектов.
- ❑ `IList` — представляет собой коллекцию объектов, доступ к которым осуществляется с помощью индекса, так же, как к элементам массива.
 - `IEnumerator` — применяется для сравнения двух объектов.
- ❑ `IEnumerable` — позволяет клиенту получать доступ к перечислителю с помощью его метода `GetEnumerator()`.
 - `IEnumerator` — поддерживает перебор элементов в коллекции.
- ❑ `IDictionary` — представляет коллекцию пар "ключ-значение".

Мы подробно рассмотрим два из перечисленных интерфейсов: `IEnumerator` и `IEnumerator`. Однако вначале обратимся к весьма полезному классу `System::Collections::ArrayList`, принадлежащему к пространству имен `System::Collections` и реализующему интерфейс `IList`.

Класс ArrayList

Класс ArrayList ведет себя как динамический массив. Можно добавлять объекты любого типа в экземпляр ArrayList и затем выполнять перебор этих объектов. Метод `ArrayList::Add()` принимает параметр типа `Object*` и добавляет объект в массив. Отдельные элементы этого массива могут считываться с помощью свойства `ArrayList::Item`. Между классом ArrayList и обычным управляемым массивом имеются два различия:

- ❑ После создания объекта ArrayList туда могут динамически добавляться элементы. В случае управляемого массива это невозможно.
- ❑ Класс ArrayList может содержать любые типы данных, в то время как управляемый массив позволяет сохранять только данные своего типа. Например, нельзя сохранить строку в целочисленном управляемом массиве.

Представленный ниже код добавляет в экземпляр ArrayList объекты различных типов (`String`, `double`, `DateTime` и `int`) с помощью метода `Add()`. При этом необходимо конвертировать управляемые типы-значения в тип `Object*`, используя ключевое слово `__box`:

```

Collections::ArrayList* l_ptrALDetails = new Collections::ArrayList;

String* l_strName = S"Nirajan Kumar";
String* l_strAddress = S"KNiranja@chn.cognizant.com";
double l_dMonthlyInc = 7500.60;
DateTime l_dtDOB(1976, 05, 15, 16, 20, 00);
__box int* l_iNumber = __box(DateTime::Today.Day);

l_ptrALDetails->Add(l_strName);
l_ptrALDetails->Add(l_strAddress);
l_ptrALDetails->Add(__box(l_dMonthlyInc));
l_ptrALDetails->Add(__box(l_dtDOB));
l_ptrALDetails->Add(l_iNumber);

```

На следующем шаге мы считываем элементы массива с помощью свойства `ArrayList::Item`:

```

Console::WriteLine(S"The values in the ArrayList are: ");
for(int l_iCtr = 0; l_iCtr < l_ptrALDetails->Count; l_iCtr++)
{
    Console::WriteLine(l_ptrALDetails->Item[l_iCtr]->ToString());
}

```

Для удаления элемента из массива можно применить метод `Remove()`, который принимает параметр типа `Object*` и удаляет первый элемент заданного типа, входящий в массив. Метод `RemoveAt()`, использующий в качестве аргумента индекс, удаляет тот элемент, к которому относится данный индекс. После удаления элемента из списка позиции оставшихся элементов изменяются и соответственно меняются их индексы:

```

// Удаление последнего элемента
l_ptrALDetails->Remove(l_iNumber);

```

Сортировка

Класс ArrayList реализует еще один метод, `Sort()`, позволяющий отсортировать все элементы массива. Однако для того, чтобы данный метод мог быть использован, объекты в массиве должны иметь доступ к реализации интерфейса `IComparer`,

имеющего единственный метод `Compare()`. Этот метод сравнивает значения двух объектов и возвращает целое число, зависящее от результата сравнения, т.е. от того, какой объект "больше".

Следующие три аргумента в методе `Sort()` позволяют управлять сортировкой массива:

- Начальный индекс
- D Количество сортируемых элементов
- Реализация интерфейса `IComparer`

Если значение третьего параметра не задано, используется реализация метода `Sort()` по умолчанию. Это подходит для объектов, которые уже поддерживают интерфейс `IComparer` (например, для объектов типа `String`). В нашем примере мы отсортируем первые два элемента типа `String*`, входящих в массив, и затем вновь выведем список элементов:

```
- l_ptrALDetails->Sort(0, 2, 0);
Console.WriteLine(S"\nThe ArrayList now contains: ");
for(int l_iCtr = 0; l_iCtr < l_ptrALDetails->Count; l_iCtr++)
{
    Console.WriteLine(l_ptrALDetails->Item[l_iCtr]->ToString());
}
```

Ниже представлены данные, которые выводятся на экран после запуска приложения. Два объекта `String*` (имя и адрес электронной почты) правильно отсортированы во втором блоке выдачи:

The values in the ArrayList are (Объект `ArrayList` содержит следующие значения):

```
Niranjan Kumar
KNiranja@chn.cognizant.com
7500.6
5/15/76 4:20:00 PM
5
```

The ArrayList now contains (Обновленное содержимое объекта `ArrayList`):

```
KNiranja@chn.cognizant.com
Niranjan Kumar
7500.6
5/15/76 4:20:00 PM
```

Класс `SortedList`

Класс `System::Collections::SortedList` существенно отличается от класса `ArrayList`. Каждый элемент сохраняется в виде пары "ключ-значение". Класс `SortedList` ведет себя как комбинация массива и хэш-таблицы, позволяя обращаться к своим элементам двумя различными способами. Когда доступ к элементам осуществляется с помощью свойства `Item()`, поведение класса такое же, как у хэш-таблицы. В случае же использования метода `GetByIndex()` класс `SortedList` ведет себя как массив.

Мы рассмотрим класс `SortedList` и создадим приложение, где он будет использоваться. Вначале добавим в список элемент, используя метод `SortedList::Add()`, который имеет два параметра типа `Object*`. Первый из них представляет собой ключ, а второй — фактическое значение. Ключ должен быть уникальным. Попытка добавить повторяющийся ключ приведет к генерации исключительной ситуации типа `ArgumentException`. В нашем приложении мы вводим, а затем выдаем на экран 6 имен:


```

Collections::SortedList* l_ptrSLNames = new Collections::SortedList;
String* l_strName
String* l_strKey;

Console::WriteLine(S"Capacity of SortedList = {0}",
    l_ptrSLNames->Capacity.ToString());

// Добавление элементов в объект SortedList
for(int l_iCtr = 0; l_iCtr < 3; l_iCtr++)
{
    Console::Write(S"Enter Name: ");
    l_strName = Console::ReadLine();
    l_strKey = String::Concat(S"Name", l_iCtr.ToString());
    l_ptrSLNames->Add(l_strKey, l_strName);
}

```

Прежде чем этот фрагмент кода предлагает вводить данные, на экране отображается емкость списка, для чего используется свойство `SortedList::Capacity`. Вначале емкость равна 16. Если известно, что во время выполнения потребуются не все выделенные элементы, можно удалить те, которые не будут использоваться, вызвав метод `SortedList::TrimToSize()`:

```

l_ptrSLNames->TrimToSize();
Console::WriteLine(S"Now the capacity is {0}",
    l_ptrSLNames->Capacity.ToString());

```

Даже после удаления неиспользуемых элементов можно добавлять новые в объект `SortedList` с помощью метода `SortedList::Add()`. Приведенный ниже код добавляет два таких элемента, что иллюстрирует возможность объекта `SortedList` динамически увеличивать или уменьшать свой размер при изменении числа элементов:

```

// Добавление двух новых элементов
l_strKey = String::Concat(S"Name", l_ptrSLNames->Count.ToString());
l_ptrSLNames->Add(l_strKey, S"Murali");
l_strKey = String::Concat(S"Name", l_ptrSLNames->Count.ToString());
l_ptrSLNames->Add(l_strKey, S"Murugan");

```

Метод `GetKey()` возвращает ключ, связанный с передаваемым индексом, тогда как метод `GetByIndex()` принимает в качестве параметра индекс и возвращает соответствующий элемент:

```

Console::WriteLine(S"The keys and values of the SortedList using Index: ");
for(int l_iCtr = 0; l_iCtr < l_ptrSLNames->Count; l_iCtr++)
{
    Console::WriteLine(S"Key = {0}, Value = {1}",
        l_ptrSLNames->GetKey(l_iCtr), l_ptrSLNames->GetByIndex(l_iCtr));
}

```

При использовании свойства `Item` объект ведет себя как хэш-таблица:

```

Console::WriteLine(S"The values of the SortedList using Key: ");
for(int l_iCtr = 0; l_iCtr < l_ptrSLNames->Count; l_iCtr++)
{
    l_strKey = String::Concat(S"Name", l_iCtr.ToString());
    Console::WriteLine(l_ptrSLNames->Item[l_strKey]);
}

Console::WriteLine(S"Finally, the capacity = {0}",
    l_ptrSLNames->Capacity.ToString());

```

После компиляции и запуска проекта мы получим следующий результат:

```
Capacity of SortedList = 16 (Емкость объекта SortedList равна 16)
Enter Name: Aruna
Enter Name: Roberto
Enter Name: Giacomo
Now the capacity is 3 (Теперь емкость равна 3)
The keys and values of the SortedList using Index
(КЛЮЧИ И значения объекта SortedList, полученные с помощью индекса):
Key = Name0, Value = Aruna
Key = Name1, Value = Roberto
Key = Name2, Value = Giacomo
Key = Name6, Value = Murali
Key = Name7, Value = Murugan
The values ()f the SortedList using Key
(Значения объекта SortedList, полученные с помощью ключа):
Aruna
Roberto
Giacomo
Murali
Murugan
Finally the capacity = 6 (Окончательная емкость равна 6)
```

Вначале мы добавили в список три элемента, затем изменили емкость объекта, сделав ее равной числу элементов (т.е. трем) с помощью метода `TrimToSize()`. После этого в объект было добавлено еще 2 элемента. Однако, несмотря на то, что всего в списке стало $3 + 2 = 5$ элементов, значение емкости увеличилось до 6. Причина в том, что при необходимости добавления элементов объект `SortedList` выделяет память для такого количества элементов, которое равно текущей емкости. Когда мы стали добавлять новые элементы, емкость удвоилась и стала равной 6.

Интерфейс `IComparer`

Для сравнения двух объектов при вызове метода `ArrayList.Sort()` должен быть реализован интерфейс `IComparer`. В рассмотренном выше примере, посвященном классу `ArrayList`, мы отсортировали первые два элемента типа `String*` с помощью метода `ArrayList.Sort()`:

```
l_ptrALDetails->Sort(0, 2, 0);
```

Здесь третий параметр выбран равным 0. Это значит, что в данном случае реализация интерфейса `IComparer` не задана. При этом можно сортировать строковые элементы, поскольку класс `String` уже реализует `IComparer`.

Реализация интерфейса `IComparer`

Рассмотрим класс C++ под названием `Friend`, который предоставляет информацию о наших друзьях, такую как имя (`name`) и возраст (`age`). Можно добавить экземпляр этого класса в массив `ArrayList`, однако для сортировки элементов данного массива нельзя непосредственно использовать метод `ArrayList.Sort()`. Чтобы это стало возможным, необходимо передать методу `Sort()` в качестве третьего параметра пользовательскую реализацию интерфейса `IComparer`.

Реализация интерфейса `IComparer` весьма проста. Необходимо лишь создать класс, производный от `IComparer`, и реализовать его члены. Интерфейс `IComparer` имеет единственный открытый метод — `Compare()`, параметрами которого являются два указателя типа `Object`. Метод `Compare()` выполняет сравнение величин для двух объектов. Он может возвращать следующие значения:

- `C` — если значения обоих параметров одинаковы.

❑ < 1 — если первое значение меньше второго.

❑ > 1 — если первое значение больше второго.

Приступим к написанию класса Friend (представленный ниже код можно найти в загружаемом проекте CustomCompare). Этот класс будет иметь два нередактируемых свойства: Name (типа String*) и Age (принадлежащее к типу int). Данные свойства можно реализовать с помощью закрытых членов m_strName и m_iAge.

```
__gc class Friend
{
private:
    String *m_strName;
    int m_iAge;

public:
    Friend(String *l_strName, int l_iAge) : m_strName(l_strName),
                                           m_iAge(l_iAge)
    {}

    __property String* get_Name()
    {
        return m_strName;
    }
    __property int get_Age()
    {
        return m_iAge;
    }
};
```

Далее необходимо реализовать интерфейс IComparer в другом классе:

```
__gc class MyComparer : public IComparer
{
private:
    int m_iSortBy;

public:
    MyComparer() : m_iSortBy(0)
    {}

    // Свойство SortBy
    __property int get_SortBy()
    {
        return m_iSortBy;
    }
    __property void set_SortBy(int l_iValue)
    {
        m_iSortBy = l_iValue;
    }

    // Сравнение объектов
    int Compare(Object *l_ptr1, Object *l_ptr2)
    {
        int l_iRetVal = 0;
        try
        {
            Friend* l_ptrFriend1 = dynamic_cast<Friend*>(l_ptr1);
            Friend* l_ptrFriend2 = dynamic_cast<Friend*>(l_ptr2);
```

```

        // Сортировка по возрасту
        if(m_iSortBy == 0)
        {
            if(1_ptrFriend1->Age < 1_ptrFriend2->Age)
                l_iRetVal = -1;
            else if(1_ptrFriend1->Age > 1_ptrFriend2->Age)
                l_iRetVal = 1;
        }
        else
        {
            // Сортировка по имени
            l_iRetVal = String::Compare(1_ptrFriend1->Name,
                                      1_ptrFriend2->Name);
        }
    }
    catch(Exception* e)
    {
        Console::Write(e);
        return l_iRetVal;
    }
    return l_iRetVal;
}
}

```

Класс `MyComparer` может сортировать объекты по имени и по возрасту¹, в зависимости от установленного нами значения свойства `SortBy`. Метод `Compare`, в соответствии с определением интерфейса `IComparer`, принимает два параметра типа `Object*`, которые мы приводим к типу `Friend*`. Указанный метод возвращает целочисленную величину, начальное значение которой установлено равным 0. Если значение свойства `SortBy` равно 0, мы сравниваем свойства `Age` объектов `Friend` и соответствующим образом задаем возвращаемую величину. В случае же, когда свойство `SortBy` имеет значение 1, для сортировки выбирается свойство `Name` объекта `Friend`. Поскольку данное свойство имеет тип `String`, для сравнения во втором случае применяется статический метод `String::Compare()`, который устанавливает возвращаемое значение автоматически. Если код генерирует исключительную ситуацию, выполнение метода прекращается.

Посмотрим, что происходит внутри списка `ArrayList`. Для этого добавим в код вспомогательную функцию, которая выдает на экран значения свойств `Name` и `Age` экземпляров класса `Friend`, сохраняемых в объекте `ArrayList`:

```

void PrintElements(ArrayList* l_ptrList)
{
    for(int l_iCtr = 0 ; l_iCtr < l_ptrList->Count; l_iCtr++)
    {
        Friend* l_ptrSorted = dynamic_cast<Friend*>(l_ptrList->Item[l_iCtr]);
        Console::WriteLine("Name = {0}, Age = {1}",
                          l_ptrSorted->Name, l_ptrSorted->Age.ToString());
    }
}

```

Теперь напомним код для тестирования класса `MyComparer`:

```

int _tmain(void)
{
    Friend* l_ptrFriend;
    ArrayList* l_ptrFriendList = new ArrayList();
    int l_iCtr = 0;
}

```

```
// Добавляем информацию о друзьях в объект ArrayList
l_ptrFriend = new Friend("Roopa", 26);
l_ptrFriendList->Add(l_ptrFriend);
l_ptrFriend = new Friend("Zaratin", 24);
l_ptrFriendList->Add(l_ptrFriend);
l_ptrFriend = new Friend("Johnny", 32);
l_ptrFriendList->Add(l_ptrFriend);

Console.WriteLine("Before sorting the array");

// Печать перед сортировкой
PrintElements(l_ptrFriendList);

// Сортировка по возрасту с использованием класса MyComparer
MyComparer<I_obj> l_obj = new MyComparer();

// Сортировка по возрасту
l_ptrFriendList->Sort(0, 3, l_obj);

// Печать после сортировки
Console.WriteLine("After sorting the array on Age");
PrintElements(l_ptrFriendList);

// Сортировка по имени с использованием класса MyComparer
l_obj->SortBy = 1;
l_ptrFriendList->Sort(0, 3, l_obj);

// Печать после сортировки
Console.WriteLine("After sorting the array on Name");
PrintElements(l_ptrFriendList);

return 0;
}
```

Здесь мы создаем несколько экземпляров класса `Friend`, добавляем их в объект `ArrayList` и выводим на экран начальные элементы несортированного списка. Далее вызывается метод `Sort()` массива, причем в качестве третьего параметра методу передается указатель на новый экземпляр класса `MyComparer`. После этого результаты вновь выводятся в окно консоли. Наконец, мы изменяем свойство `SortBy` объекта `MyComparer`, присвоив данному свойству значение 1. Это позволяет отсортировать массив по имени, прежде чем выдать элементы на экран после очередной сортировки. Результат выполнения всей программы имеет вид:

```
Before sorting the array
(Перед сортировкой массива):
Name = Roopa, Age = 26
Name = Johnny, Age = 32
Name = Robert, Age = 41
After sorting the array on Age
(После сортировки массива по возрасту):
Name = Roopa, Age = 26
Name = Johnny, Age = 32
Name = Robert, Age = 41
After sorting the array on Name
(После сортировки массива по имени):
Name = Johnny, Age = 32
Name = Robert, Age = 41
Name = Roopa, Age = 26
```

Интерфейс (Enumerator)

Интерфейс `IEnumerator` определяет набор методов, облегчающих перемещение между элементами коллекции. Это базовый интерфейс для всех перечислителей. Важно подчеркнуть, что он не позволяет изменять данные. Вместо этого он предоставляет моментальный снимок. Классы, реализующие этот интерфейс, могут использоваться языками C# или Visual Basic .NET в цикле 'for each'

Цикл 'for each' в C# или VB аналогичен циклу for языка C, однако первый цикл автоматически обеспечивает инициализацию, проверку условия, перемещение % следующему элементу и т.д.

Методы интерфейса `IEnumerator` перечислены в следующей таблице:

Имя метода	Тип члена	Описание
<code>MoveNext()</code>	Открытый метод	Переводит перечислитель на следующий элемент
<code>Reset()</code>	Открытый метод	Устанавливает перечислитель на начальный элемент
<code>Current</code>	Открытое свойство	Возвращает текущий элемент из списка

Реализация интерфейса `IEnumerator`

Мы разработаем класс C++, реализующий интерфейс `IEnumerator`. Этот класс будет сохранять список имен — по одному в каждом элементе. Вначале мы создадим проект Managed C++ Class Library с именем `FriendArray`, включаемый в решение `MyFriends`. Это позволит позднее добавить в решение клиентский проект. В библиотеке `FriendArray` мы создадим класс C++ под названием `MyFriends`, который реализует интерфейс `IEnumerator`:

```
// Файл FriendArray.h
```

```
#pragma once
```

```
using namespace System;
```

```
using namespace System::Collections;
```

```
namespace FriendArray
```

```
{
```

```
public __gc class MyFriends : public IEnumerator
```

```
{
```

```
private:
```

```
    ArrayList* m_ptrALNames;
```

```
    int m_iPos;
```

```
public:
```

```
    MyFriends()
```

```
{
```

```
    m_ptrALNames = new ArrayList;
```

```
    m_iPos = -1;
```

```
}
```

```
    // Свойство Value
```

```
    __property set Value(String* l_strValue)
```

```
{
```

```
    m_ptrALNames->Add(l_strValue);
```

```
}
```

```
    // Метод, предназначенный для получения элемента
```

```

String* GetValue(int l_iCount);
{
    return dynamic_cast<String*>(m_ptrALNames->Item(l_iIndex));
}

// Свойство count
property int get_Count()
{
    return m_ptrALNames->Count;
}

// Возвращает средство реализации интерфейса IEnumerator
IEnumerator* GetEnumerator();

// Интерфейс IEnumerator
bool MoveNext();
property Object* get_Current();
void Reset();
};
)

```

Класс `MyFriends` имеет два закрытых члена: `m_ptrALNames`, содержащий список имен, и `m_iPos`, используемый при переборе элементов. Можно добавить элемент в массив, используя метод `set` свойства `Value` и получить значение элемента с помощью соответствующего метода `get`. В методе `get` необходимо выполнить преобразование типа, поскольку метод `ArrayList::get_Item` возвращает тип `Object*`. Свойство `Count` является *нечитательным* и возвращает количество элементов в списке `ArrayList`.

Следующие три метода позволяют клиентским приложениям, написанным на C# или Visual Basic .NET, перебирать элементы коллекции с помощью цикла `'for each'`. Для этого цикл использует реализацию интерфейса `IEnumerator`. Если нужно, чтобы данная функциональность была доступна другим языкам, следует реализовать метод `GetEnumerator()`. При этом мы фактически реализуем интерфейс `IEnumerable`. Ниже представлена реализация метода `GetEnumerator()` в файле `FriendArray.cpp`:

```

IEnumerator* MyFriends::GetEnumerator()
{
    Console::WriteLine(S"Inside C++ : GetEnumerator()");
    m_iPos = -1;
    return this;
}

```

Этот метод возвращает указатель `this`, поскольку интерфейс `IEnumerator` реализуется в том же самом классе. Мы также устанавливаем значение переменной `m_iPos` равным `-1`, поскольку цикл `'for each'` вначале будет вызывать метод `MoveNext()`, а затем обращаться к свойству `Current`. В результате при первом вызове текущим станет начальный элемент списка.

Приведем код метода `MoveNext()`:

```

bool MyFriends::MoveNext()
{
    Console::WriteLine(S"Inside C++ : MoveNext()");
    m_iPos++;
    return m_iPos < Count;
}

```

Здесь переменная `m_iPos` получает приращение и ее значение сравнивается с числом элементов объекта `m_ptrALNames` типа `ArrayList`. Если окажется, что `m_iPos` не

меньше полного числа элементов, выполнение цикла завершится. Пока метод возвращает значение `true`, цикл `for each` продолжается. В противном случае он прекратится.

Следующий фрагмент кода демонстрирует реализацию метода `get_Current`:

```
Object* MyFriends::get_Current()
{
    Console::WriteLine(S"Inside C++ : get_Current()");
    if(m_iPos == -1)
    {
        Console::WriteLine(S"Invalid Operation\n");
        return S"Error : Invalid Operation";
    }
    return GetValue(m_iPos);
}
```

После начальной проверки переменной `m_iPos` этот код с помощью метода `GetValue()` получает элемент с заданным индексом и возвращает его вызывающему методу. Теперь осталось реализовать еще один метод — `Reset()`, который не содержит аргументов и просто устанавливает начальное значение индекса. Для этого переменной `m_iPos` присваивается значение `-1`:

```
void MyFriends::Reset()
{
    Console::WriteLine(S"Inside C++; Reset()");
    m_iPos = -1;
    return;
}
```

Итак, мы написали на языке C++ класс `MyFriends`, реализующий интерфейс `IEnumerator<to>`. Теперь создадим клиентское приложение C#, использующее этот класс.

Создание клиентского приложения

Добавим в существующее решение новое консольное приложение C# с именем `TestClient`. Прежде чем писать код, необходимо включить в проект ссылку на класс `MyFriends`. Для этого нужно щелкнуть правой кнопкой мыши на узле проекта в окне `Solution Explorer` и выбрать пункт меню `Add Reference` (Добавить ссылку). В открывшемся окне проекта `TestClient` под узлом `References` (Ссылки) должен находиться проект `FriendArray`. Теперь можно приступить к написанию кода, чтобы протестировать класс C++. Откроем файл C#, который называется `Class1.cs`, и добавим приведенный ниже код в функцию `Main()`. В начало файла необходимо поместить директиву `using` для пространства имен `FriendArray`.

```
static void Main(string[] args)
{
    Console.WriteLine("Output from C# program: Beginning of C# TestClient");

    // Создание экземпляра класса MyFriends
    MyFriends l_objFriends = new MyFriendsO;

    // Добавление элементов в объект
    l_objFriends.Value = "Jack";
    l_objFriends.Value = "Prabhu";
    l_objFriends.Value = "Aparna";

    // Вывод элементов на экран
    Console.WriteLine("C# program: Before foreach()");
}
```



```
foreach(string str in l_objFriends)
{
    // Вывод элемента в окно консоли
    Console.WriteLine("C# program: {0}", str);
}
Console.WriteLine("C# program: End of foreach()");
Console.WriteLine("C# program: End of C# TestClient");
```

Вначале этот код создает экземпляр класса `MyFriends` с именем `l_objFriends` и добавляет в список 3 имени. Затем используется цикл `foreach` для перебора элементов списка и вывода их на экран. Результат выполнения всей программы имеет следующий вид:

```
Output from C# program: Beginning of C# TestClient
[Выходные данные приложения C#: начало программы TestClient, написанной на C#]
C# program: Before foreach() [Программа C#: начало цикла foreach()]
Inside C++; GetEnumerator()
Inside C++; MoveNext()
Inside C++; get_Current()
C# program: Jack
Inside C++; MoveNext()
Inside C++; get_Current()
C# program: Prabhu
Inside C++; MoveNext()
Inside C++; get_Current()
C# program: Aparna
Inside C++; MoveNext()
C# program: End of foreach() [Программа C#: конец цикла foreach()]
C# program : End of C# TestClient
[Приложение C#: конец программы TestClient, написанной на C#]
```

Отсюда видно, что цикл `foreach` для перебора элементов массива использует методы `GetEnumerator()`, `MoveNext()` и `get_Current()`. Можно также наблюдать ход выполнения программы, если поместить точки останова в файл `FriendArray.cpp` и запустить программу в режиме отладки.

Организация потоков

Теперь мы обсудим модель потоков, применяемую в .NET, и покажем, как использовать преимущества этой модели в управляемых приложениях C++ с помощью набора классов, определенных в пространстве имен `System::Threading`. В частности, мы рассмотрим классы `Thread` и `ThreadPool`, объекты синхронизации потоков, а также классы `Monitor` и `ReaderWriterLock`. Попутно мы познакомимся с рядом других аспектов, относящихся к потоковой модели .NET, например с исключительными ситуациями.

В модели .NET программа выполняется в одном или нескольких **доменах приложения (application domain)**; часто используется сокращение **AppDomain**. Домен приложения может рассматриваться как логический процесс, тогда как традиционные процессы Win32 являются физическими процессами. Один процесс Win32 может фактически включать в себя несколько доменов приложения. Каждый домен приложения имеет "надежную" границу, и код, исполняемый в одном домене, не может непосредственно получить доступ к коду, исполняемому в другом домене. В этом отношении домены приложения очень похожи на процессы Win32, каждый из которых протекает в своем собственном виртуальном адресном пространстве, как правило, независимо от других процессов.

Домен приложения может содержать одну или несколько логических групп, включающих условия исполнения кода. Такие группы называются **контекстами (contexts)**. Контексты формируют другой тип границ исполняемого кода. Объекты, существующие в одном контексте, не могут непосредственно получать доступ к коду другого контекста, поскольку условия исполнения кода могут быть различными и даже несовместимыми.

Внутри доменов приложения и контекстов реализуются потоки исполнения кода. Когда в приложении функционирует несколько потоков, оно называется многопоточным. Обратимся вначале к созданию потока в управляемом приложении, для чего используется модель потоков .NET Framework.

Классы управления потоками

Класс `System.Threading.Thread` содержит методы и свойства, предназначенные для управления отдельным потоком (включая его создание и уничтожение). Класс `System.Threading.ThreadPool` позволяет приложению использовать поток, создаваемый системой.

Класс `Thread`

Для создания потока в управляемой среде можно инициализировать экземпляр класса `Thread`. С помощью делегата `ThreadStart` модель потоков .NET Framework удачно отделяет создание потока и управление им от точки входа для этого потока.

Для создания экземпляра класса `Thread` необходимо передать делегат `ThreadStart` конструктору класса `Thread`. Целевой функцией делегата может быть любая функция, которая возвращает тип `void` и не содержит параметров. Это означает, что нельзя непосредственно передать информацию о состоянии в целевую функцию делегата `ThreadStart`. Вместо этого указанную целевую функцию мы делаем методом класса, выдающим информацию о состоянии, к которой будет получать доступ функция потока при его работе. Проиллюстрируем это на примере:

```
using namespace System;
using namespace System::Threading;

__gc class gcSimple
{
private:
    String* strToday;
public:
    // Открытый конструктор
    gcSimple()
    {
        strToday = DateTime::Now.ToShortDateString();
    }

    void ThreadFunc1()
    {
        Console::WriteLine(S"Today is {0}", strToday);
        for(int i = 0; i < 10; i++)
        {
            Console::WriteLine(S"Thread ID = {0}, i = {1}",
                AppDomain::GetCurrentThreadId().ToString(), i.ToString());
            for(int j = 0; j < 1000000; j++)
            {
                int i1 = i * j;
            }
        }
    }
}
```

Здесь мы вначале определяем управляемый класс `gcSimple`, который имеет метод `ThreadFunc1()`. Позже этот метод будет играть роль целевой функции для экземпляра делегата `ThreadStart`, который должен передаваться в конструктор класса `Thread`. Метод `ThreadFunc1()` включает цикл `for` для вывода на экран идентификатора потока и индекса цикла. В свою очередь, этот цикл запускает еще один вложенный цикл `for`, заставляя поток функционировать, а не находиться в режиме ожидания. Это позволяет наблюдать за потоком в состоянии исполнения. Наш класс потока содержит член `strToday` типа `String*`, которому в открытом конструкторе присваивается текущая дата. Переменная `strToday` используется функцией потока. Такой подход отличается от методов организации потоков в Win32 или MFC, где значения в функцию потока передаются как величины типа `void*`. При использовании же модели .NET в делегат `ThreadStart` передается весь объект целиком:

```
int _tmain(void)
{
    gcSimple* st = new gcSimple;
    Thread t1 = new Thread(new ThreadStart(st, gcSimple::ThreadFunc1));
    Console.WriteLine(S"t1 is in the {0} state", __box(t1->ThreadState));
}
```

Этот код инициализирует экземпляр класса `gcSimple`. Затем создается экземпляр класса `Thread`, куда передается новый экземпляр делегата `ThreadStart`, целевой функцией которого является метод `ThreadFunc1()`. В данный момент поток *не* исполняется. Класс `Thread` содержит несколько свойств. Одно из них, `ThreadState`, возвращает текущее состояние потока. После создания экземпляра потока на экран будет выведена информация о его состоянии:

t1 is in the Unstarted state (Поток t1 находится в состоянии Unstarted)

Далее мы запускаем поток t1, вызывая его метод Start():

```
Console.WriteLine(S"starting t1");
t1->Start();
Thread.Sleep(0);
Console.WriteLine(S"t1 is in the {0} state", __box(t1->ThreadState));
```

После того как основная программа запускает поток `t1`, вызывается метод `Thread.Sleep(0)`, что вызывает переключение контекста потока. Когда сам автор запустил программу, идентификатор потока оказался равным 1728. Приведенная ниже распечатка выдачи показывает, что поток `t1` имел достаточно времени для вывода на экран одной строки, прежде чем произошло переключение контекста и передача управления основному потоку. Основной поток, в свою очередь сообщает, что поток `t1` находится в состоянии исполнения:

```
starting t1 (Запуск потока t1)
Today is 1/30/02 (Сегодня 30.01.02)
Thread ID 1728, i = 1 (Идентификатор потока равен 1728, i = 1)
t1 is in the Running State (Поток t1 находится в состоянии исполнения)
```

В приведенных ниже примерах выводимая на экран информация зависит от того, как используется процессор, от его скорости, количества других работающих процессов и т.д. Более того, поведение приложения можно изменять, по-разному вызывая метод `Sleep()`. Например, если передать в этот метод значение параметра, равное 1000, функция потока успеет выдать больше информации, прежде чем возобновится исполнение программы `main()`.

В последующих примерах данной главы используется метод `Thread.Sleep()`, чтобы вызывать переключение контекста в наиболее подходящее время. Это помогает выявлять ошибки синхронизации.

Далее основная программа приостанавливает поток `t1`, вызывая метод `Suspend()`:

```
Console.WriteLine("suspending t1");
t1->Suspend();

Thread.Sleep(0);
Console.WriteLine($"t1 is in the {{0}} state", __box(t1->ThreadState));
```

В результате выполнения этого кода на экран выдается следующая информация;

```
suspending t1 (Приостановка потока t1)
t1 is in the Suspended state (Поток t1 находится в приостановленном состоянии)
```

Приостановив поток `t1`, основная программа вызывает метод `Sleep(0)`. После этого контекст вновь переключается на основной поток, который сообщает, что поток `t1` теперь находится в приостановленном состоянии. Далее поток `t1` возобновляется, поскольку вызывается его метод `Resume()`:

```
Console.WriteLine("resuming t1");
t1->Resume();

Thread.Sleep(0);
Console.WriteLine($"t1 is in the {{0}} state", __box(t1->ThreadState));
```

Этот код выдаст на экран такие сообщения:

```
resuming t1 (Возобновление потока t1)
Thread ID 1728, i = 2 (Идентификатор потока равен 1728, i = 2)
t1 is in the Running State (Поток t1 находится в состоянии исполнения)
```

После того как поток `t1` возобновился, он может вновь приступить к исполнению кода. Этот поток отображает единственную строку, прежде чем переключение контекста вновь приводит к передаче управления основному потоку, который выводит сообщение о том, что поток `t1` опять находится в состоянии исполнения.

Наконец, можно подождать, пока поток `t1` не закончит свою работу, вызвав его метод `Join()`:

```
Console.WriteLine("joining t1");
t1->Join();
Console.WriteLine("t1 has finished");
return 0;
}
```

Этот код выведет на экран следующую информацию:

```
joining t1 (Присоединение потока t1)
Thread ID 1728, i = 5
Thread ID 1728, i = 6
Thread ID 1728, i = 7
Thread ID 1728, i = 8
Thread ID 1728, i = 9
t1 has finished (Поток t1 завершен)
```

Метод `Join()` блокирует вызывающий поток до тех пор, пока не вернет управление [целевой метод делегата `ThreadStart`, инициализирующего поток `t1`. Затем метод `Join()` возвращает управление основному потоку, который продолжает свою работу, выводя в окно консоли сообщение о том, что работа потока `t1` завершена. Приведенная выше распечатка выдачи иллюстрирует процесс исполнения программы после того, как поток `t1` присоединяется к основной функции.

Незначительные изменения этого кода могут заставить программу работать совсем иначе. Если в основной функции закомментировать все операторы, вызывающие метод `Thread::Sleep(0)`, выходные данные будут другими. В этом случае после запуска дочернего потока `t1` основной поток не предоставит ему время и будет выполнять все свои операторы до тех пор, пока не достигнет оператора, вызывающего метод `Join()`. До этого времени поток `t1` будет находиться в состоянии 'Unstarted'.

Класс `ThreadPool`

Во многих приложениях потоки большую часть времени находятся в состоянии ожидания. Например, можно создать приложение, в котором только один поток управляет вводом-выводом данных через сокет для каждого подключенного клиента. Такая модель называется "thread-per-connection" (один поток на соединение). При этом каждый поток будет в основном находиться в приостановленном состоянии, ожидая отправки или приема данных.

Альтернативой такому подходу является **потоковый пул (thread pool)**, представляющий собой группу потоков, которые находятся в состоянии ожидания и используются для выполнения работы, когда это необходимо. В приложении, где нужно реализовать ввод-вывод через сокеты, потоковый пул может способствовать уменьшению количества необходимых потоков, позволяя при этом обслуживать столько же (если не больше) соединений. Благодаря использованию данной модели каждый поток работает одновременно с несколькими соединениями и таким образом оказывается более загруженным. Рассмотрим, как проблема создания множественных потоков для одновременного выполнения какого-либо задания решается в .NET.

Платформа .NET Framework содержит класс `ThreadPool`, который можно использовать для постановки рабочих элементов в очередь на исполнение потоками пула. Все члены класса `ThreadPool` являются статическими. Среда CLR сама заботится об управлении потоками, принадлежащими к пулу, увеличивая или уменьшая их число по мере необходимости. Эта процедура зависит от внутреннего алгоритма, учитывающего количество процессоров в машине, число элементов работы, выполняемых пулом, и многие другие детали. Потоковый пул позволяет также реализовать быстрый и простой способ выполнения асинхронного вызова функций.

Обратимся к простому примеру. Вначале нужно определить класс, содержащий метод, который может быть исполнен в потоке класса `ThreadPool`:

```
using System;
using System.Threading;

public class MyThreadContext
{
    public:
        // Простой пример использования пула
        void SimpleThreadFunc(Object* obj)
        {
            // Вывод передаваемого параметра
            Console.WriteLine("Today is {0}", obj);
            Console.WriteLine($"Thread ID {0} in SimpleThreadFunc()",
                AppDomain.GetCurrentThreadId().ToString());
            int nWorkerThreads;
            int nIOThreads;

            // Подсчет доступного количества потоков
            ThreadPool.GetAvailableThreads(&nWorkerThreads, &nIOThreads);
            Console.WriteLine($"Available Worker threads: {0}",
                nWorkerThreads.ToString());
            Thread.Sleep(1000);
        }
}
```

Класс `MyThreadContext` определяет метод `SimpleThreadFunc()`, имеющий один аргумент типа `Object*`. Данный метод используется в качестве целевой функции делегата `WaitCallback` (см. ниже). Приведенный код выводит в окно консоли передаваемый объект и идентификатор потока. Последняя величина может быть получена с помощью статического метода `AppDomain::GetCurrentThreadId()`, возвращающего идентификатор текущего потока. Затем определяется количество доступных потоков, которое может содержать пул, для чего используется статический метод `ThreadPool::GetAvailableThreads()`. Этот метод принимает два аргумента `Int32*` и присваивает им значения, равные количеству рабочих потоков и числу потоков ввода-вывода. После отображения описанной информации функция в течение одной секунды находится в режиме ожидания, позволяя выполнять работу другим потокам.

Теперь можно использовать класс `ThreadPool` для вызова метода `SimpleThreadFunc()`. Ниже приведен пример вызывающего кода:

```
// Объявление строки
String *strToday = DateTime::Today.ToShortDateString();
MyThreadContext tc = new MyThreadContext();

// Использование потоков класса ThreadPool уровня процесса для
// вызова метода SimpleThreadFuncO
for(int i = 0; i < 5; i++)
{
    ThreadPool::QueueUserWorkItem(new WaitCallback(tc,
        MyThreadContext::SimpleThreadFunc), strToday);
}

// Предоставляется достаточное время для завершения всех элементов работы.
Thread::Sleep(10000);
```

Этот код создает экземпляр класса `MyThreadContext`, а затем запускает цикл `for`, в котором вызывается метод `ThreadPool::QueueUserWorkItem()`. Первый параметр данного метода имеет тип `WaitCallback`. Данный тип является делегатом, при инициализации которого указывается метод `SimpleThreadFunc()` экземпляра класса `MyThreadContext`. Обратите внимание на отличие между использованием делегатов `ThreadStart` и `WaitCallback`. Во втором случае дополнительные параметры передаются в функцию потока с помощью метода `QueueUserWorkItem` (в приведенном коде передается параметр `strToday`). Далее пять рабочих элементов помещаются в очередь на исполнение пулом `ThreadPool`, после чего на 10 с включается режим ожидания, чтобы пул успел выполнить эти элементы работы. Выполнение кода приводит к следующему результату:

```
Today is 2/1/02 Thread ID 285 in SimpleThreadFunc() [Сегодня 01.02.02.
Поток исполняемый в функции SimpleThreadFuncO. Идентификатор потока 285]
Available Worker threads 24 [Доступное число рабочих потоков - 24]
Today is 2/1/02 Thread ID 271 in SimpleThreadFuncO
Available Worker threads 23
Today is 2/1/02 Thread ID 285 in SimpleThreadFuncO
Available Worker threads 23
Today is 2/1/02 Thread ID 271 in SimpleThreadFuncO
Available Worker threads 23
Today is 2/1/02 Thread ID 285 in SimpleThreadFuncO
Available Worker threads 23
```

Как видим, функция `SimpleThreadFuncO` вызывается 5 раз (для выполнения задания на машине автора потребовалось два потока из пула `ThreadPool`). Об этом свидетельствует число различных идентификаторов потоков (285 и 271). На другой

машине результаты могут быть иными. Возможна даже ситуация, когда для выполнения всех элементов работы потребуется лишь один поток.

Обратите внимание на соотношение между количеством потоков и доступным числом рабочих потоков. По умолчанию последнее значение равно 25. Когда используется первый поток с идентификатором 285, доступное число рабочих потоков становится равным 24. Затем оно уменьшается до 23, и дальнейшего снижения этой величины не происходит. Отсюда можно заключить, что класс `ThreadPool` использует новые потоки только тогда, когда это необходимо, и, если есть возможность, этот класс пытается выполнить задание лишь с помощью существующих потоков. В результате программист освобождается от написания кода для создания и уничтожения потоков.

В рассмотренном примере есть одна проблема, состоящая в том, что основной поток должен реализовать распределение времени, позволяющее завершить выполнение элементов работы. Основной поток находится в режиме ожидания в течение 10 с, прежде чем закончить свою работу. Если за указанное время все элементы работы не будут завершены, некоторые из них так и останутся невыполненными. Поэтому разработчик должен программным образом обеспечить выполнение каждого элемента работы до завершения основного потока*. События синхронизации потоков позволяют наилучшим образом решить данную проблему (см. ниже).

Классы синхронизации потоков

При написании многопоточного приложения необходимо обеспечить синхронизацию доступа к совместно используемым ресурсам. Платформа .NET содержит несколько классов, которые можно применять для синхронизации потоков. Каждый такой класс, принадлежащий к библиотеке классов .NET Framework, реализует свой стиль или метод синхронизации, и поэтому разные классы применимы для решения различных проблем. Рассмотрим поочередно все эти классы.

Класс *Interlocked*

Наиболее фундаментальная форма синхронизации достигается при использовании класса `Interlocked`, принадлежащего к пространству имен `System::Threading`. Этот класс содержит четыре метода, выполняющих атомарные операции инкремента, декремента, присваивания, а также сравнения и присваивания величины. Все члены данного класса являются статическими и перегруженными. Версии методов `Increment()` и `Decrement()` принимают указатели на значения `int` или `int64`, тогда как аргументы функций `Exchange()` и `CompareExchange()` могут иметь тип `int*`, `Object*` или `float*`.

Методы *Interlocked::Increment* и *Interlocked::Decrement*

Метод `Increment()` реализует атомарную операцию инкремента и возвращает увеличенное значение переменной. Этому методу необходимо передавать величины типа `int*` или `long*`. Метод защищает от потенциальной опасности переключения контекста потока, которое может иметь место при выполнении таких операций, как `p++`, где `p` — переменная, совместно используемая несколькими потоками. Описываемый метод очень просто использовать. Необходимо лишь помнить, что при создании многопоточных приложений следует избегать написания такого, например, кода:

```
int n; ... n++; ...
```

Вместо этого нужно использовать следующую конструкцию:

`Interlocked::Increment (&n);`

Метод `Interlocked::Decrement ()` работает аналогичным образом, за исключением того, что он (как говорит само название) выполняет атомарную операцию декремента и возвращает значение с отрицательным приращением.

Метод `Interlocked::Exchange`

Метод `Exchange()` реализует атомарную операцию, которая считывает и присваивает значение переменной. Он имеет перегруженные версии для величин типа `int`, `float` и `Object`. Он записывает заданное значение в определенную область памяти и возвращает чу величину, которая хранилась в этой области перед выполнением операции.

В следующем примере делается попытка реализовать простой механизм спин-блокировки. Чтобы проиллюстрировать возникающие проблемы, вначале приведем код, который не использует взаимную блокировку:

```
// Ожидание в режиме спин-блокировки
while(1 == n)
{
    Console.WriteLine($"Thread ID {0} waiting for spin lock to open",
        AppDomain.GetCurrentThreadId().ToString());
    Thread.Sleep(100);
}

Thread.Sleep(100);

// Установление контроля над спин-блокировкой
n = 1;

// Выполнение кода
Console.WriteLine($"Thread ID {0} has the spin lock",
    AppDomain.GetCurrentThreadId().ToString());

Thread.Sleep(100);

Console.WriteLine($"Thread ID {0} releasing the spin lock",
    AppDomain.GetCurrentThreadId().ToString());

// Отмена режима спин-блокировки
n = 0;
```

Вначале этот код сохраняет поток в спин-блоке, который реализуется с помощью переменной `n`. Пока `n == 1`, выполняется цикл `while` и при каждой прогонке задается режим ожидания продолжительностью в 100 мс. Когда код обнаруживает, что значение `n` перестало быть равным 1, цикл `while` завершается и еще на 100 мс устанавливается режим ожидания. Затем код получает контроль над спин-блокировкой, задавая значение `n` равное 1. В это время код является "владельцем" блока, а другие потоки ожидают своей очереди в цикле `while`. После выполнения некоторого кода (вывода на консоль идентификационной информации, относящейся к потоку) режим спин-блокировки отменяется.

В основном данный код работает корректно, однако из-за переключения контекста здесь не поддерживается безопасность потоков. Посмотрим на следующую распечатку выдачи, которая является результатом выполнения кода тремя параллельными потоками:

```
Thread ID = 1796 has the spin lock
(Поток с идентификационным номером 1796 имеет контроль над спин-блоком)
```



```
Thread ID = 1820 has the spin lock
Thread ID = 1808 has the spin lock
Thread ID = 1796 releasing the spin lock
(Поток с идентификационным номером 1796 отменяет режим спин-блокировки)
Thread ID = 1820 releasing the spin lock
Thread ID = 1808 releasing the spin lock
```

Очевидно, что код работает некорректно. Каждый из трех потоков считает, что он контролирует спин-блокировку. Проблема возникает из-за того, что переключение контекста потока имеет место между тем моментом, когда считывается величина p для проверки равенства $p = 1$, и моментом получения потоком контроля над спин-блокировкой, что достигается за счет установки значения p равным 1.

Для исправления кода необходимо использовать метод `Exchange()`, позволяющий проверить величину и присвоить ей значение 1 в рамках одной неделимой операции:

```
while(1 == Interlocked::Exchange( &n, 1))
{
    Console::WriteLine($"Thread ID {0} waiting for spin lock to open",
        AppDomain::GetCurrentThreadId().ToString());
    Thread::Sleep(100);
}

Thread::Sleep(100);

// Выполнение кода
Console::WriteLine($"Thread ID {0} has the spin lock",
    AppDomain::GetCurrentThreadId().ToString());
Thread::Sleep(100);

Console::WriteLine($"Thread ID {0} releasing the spin lock",
    AppDomain::GetCurrentThreadId().ToString());

// Отмена режима спин-блокировки
Interlocked::Exchange(&n, 0);
```

В начальный момент $p = 0$, поскольку никакой поток пока не контролирует блокировку. Далее в условии цикла `while` проверяется возвращаемое значение метода `Exchange()`, и, поскольку оно равно 1, будет исполняться указанный цикл. Первый поток, выполняющий приведенный код, устанавливает значение p равным 1 и поэтому получает контроль над блокировкой. Все остальные потоки будут исполняться внутри цикла `while` до тех пор, пока поток-владелец не отменит блокировку. Выполнение на машине автора настоящего примера с тремя параллельными потоками дало такой результат;

```
Thread ID 1812 has the spin lock
(Поток с идентификационным номером 1812 имеет контроль над спин-блоком)
Thread ID 384 waiting for spin lock to open
(Поток с идентификационным номером 384 ожидает снятия блокировки)
Thread ID 1480 waiting for spin lock to open
Thread ID 1812 releasing the spin lock
(Поток с идентификационным номером 1812 отменяет режим спин-блокировки)
Thread ID 384 has the spin lock
Thread ID 1480 waiting for spin lock to open
Thread ID 384 releasing the spin lock
Thread ID 1480 has the spin lock
Thread ID 1480 releasing the spin lock
```

Теперь в каждый момент времени блоком владеет только один поток. Другие потоки ожидают отмены режима блокировки.

Метод `Interlocked::CompareExchange`

Метод `CompareExchange()` выполняет неделимую операцию, в рамках которой переменная сравнивается с некоторой величиной, и, если равенство имеет место, переменной присваивается определенное значение. Этот метод также имеет три перегруженные версии, принимающие параметры типа `int`, `float` и `Object*`.

Представленный ниже код проверяет, равна ли пулю целочисленная переменная `p`. При наличии равенства, `p` присваивается значение 1. Здесь переменная `p` играет роль ключа, который разрешает выполнение блока кода, следующего за оператором `if`, только если `l = 0`:

```
if(p != 0)
{
    Thread::Sleep(100);
    p = 1;
    Console.WriteLine($"Thread ID {0} set p = 1",
        AppDomain.GetCurrentThreadId().ToString());
}
else
{
    Thread::Sleep(100);
    Console.WriteLine($"Thread ID {0}, p != 0",
        AppDomain.GetCurrentThreadId().ToString());
}
```

Если `p = 0`, то код присваивает `p` значение 1. Поэтому любые последующие проверки равенства `p = 0` дадут результат `false`, и блок кода, следующий за оператором `if`, более не будет исполняться. Однако при таком подходе вновь возникает проблема, если с кодом работает несколько потоков. В этом случае оказывается возможной ситуация, когда упомянутый блок кода, который идет вслед за оператором `if`, будет выполняться более чем одним потоком. Это подтверждает следующий пример выдачи данных, когда код исполняется тремя параллельными потоками.

```
Tread ID = 1716, set p = 1
(Идентификатор потока 1716. Переменной p присвоено значение 1)
Tread ID = 1420, set p = 1
Tread ID = 1740, set p = 1
```

Здесь результатом проверки условия (`l == 0`) для каждого потока является значение `true`, поскольку переключение контекста происходит до того, как любой поток успеет присвоить переменной `p` значение 1. Для устранения этой проблемы можно воспользоваться методом `Interlocked::CompareExchange()`:

```
if (Interlocked.CompareExchange(&p, 1, 0) == 0)
{
    Thread::Sleep(100);
    Console.WriteLine($"Thread ID {0} set p = 1",
        AppDomain.GetCurrentThreadId().ToString());
}
else
{
    Thread::Sleep(100);
    Console.WriteLine($"Thread ID {0}, p != 0",
        AppDomain.GetCurrentThreadId().ToString());
}
```

Выполнение данного кода тремя параллельными потоками даст следующий результат:

```
Tread ID = 1816, set n = 1
Tread ID = 1844, n != 0
Tread ID = 1644, n != 0
```

В этом случае все три потока проверяют результат, возвращаемый методом `CompareExchange()`, однако значение переменной `n` для двух из них не равно 0. Поэтому функция не выполняет присваивания и возвращает текущее значение `n = 1`.

Класс *WaitHandle*

Класс `System.Threading.WaitHandle` содержит методы, предоставляющие потоку возможность в течение регулируемого интервала времени владеть объектами синхронизации. Данный класс позволяет создавать объекты, ожидающие исключительного доступа к совместно используемым ресурсам. Хотя класс `WaitHandle` является абстрактным, на его основе созданы три производных класса, относящихся к библиотеке классов `.NET Framework`. Это классы `Mutex`, `AutoResetEvent` и `ManualResetEvent`.

Класс *Mutex*

Класс `Mutex` позволяет создать объект синхронизации (мьютекс), которым может владеть один из потоков в течение любого заданного времени. Когда поток пытается завладеть мьютексом, а тот уже находится в распоряжении другого потока (т.е. заблокирован им), первый поток будет блокироваться до тех пор, пока не истечет заданный лимит времени или не будет освобожден мьютекс. Следующий фрагмент кода иллюстрирует использование именованного экземпляра `WroxMutex` для синхронизации трех потоков:

```
Mutex* m = new Mutex(false, "WroxMutex");

for(int i = 0; i < 2; i++)
{
    Console.WriteLine("Thread ID {0} waiting to claim the mutex",
        AppDomain.CurrentThread.Id.ToString());
    if(m.WaitOne(1000, false))
    {
        Console.WriteLine("Thread ID {0} owns the mutex",
            AppDomain.CurrentThread.Id.ToString());
        Thread.Sleep(100);
        Console.WriteLine("Thread ID {0} releasing the mutex",
            AppDomain.CurrentThread.Id.ToString());
        m.ReleaseMutex();
    }
    else
    {
        Console.WriteLine("Thread ID {0} failed to claim the mutex",
            AppDomain.CurrentThread.Id.ToString());
    }
}
```

Здесь мы инициализируем экземпляр класса `Mutex`, передавая конструктору значение `false`, указывающее, что в начальный момент мьютекс не имеет владельца. Если задать параметр, равный `true`, владельцем мьютекса станет текущий поток. Кроме того, в конструктор передается строка, используемая в качестве имени экземпляра класса `Mutex`.

Метод `Mutex.WaitOne()` ожидает в течение заданного времени (в миллисекундах), когда текущий поток получит сигнал об истечении интервала времени. Когда

метод `ReleaseMutex()` вызывается первым потоком, следующий запускает метод `WaitOne()`, а другие ожидают, когда второй поток вызовет метод `ReleaseMutex()`. В нашем примере этот процесс реализуется в цикле `for` дважды для каждого потока.

Именованные мьютексы доступны из других процессов и, следовательно, поддерживают синхронизацию потоков как в различных процессах, так и в рамках одного процесса. В приведенном примере все потоки выполняются в одном процессе. Когда код выполняется тремя параллельными потоками, мы получаем следующий результат:

```
Thread ID = 217 waiting to claim the mutex
(Поток с идентификационным номером 217 ожидает доступа к мьютексу)
Thread ID = 217 owns the mutex
(Поток с идентификационным номером 217 владеет мьютексом)
Thread ID = 60 waiting to claim the mutex
Thread ID = 177 waiting to claim the mutex
Thread ID = 217 releasing the mutex
(Поток с идентификационным номером 217 освобождает мьютекс)
Thread ID = 60 owns the mutex
Thread ID = 217 waiting to claim the mutex
Thread ID = 60 releasing the mutex
Thread ID = 60 waiting to claim the mutex
Thread ID = 177 owns the mutex
Thread ID = 177 releasing the mutex
Thread ID = 177 waiting to claim the mutex
Thread ID = 217 owns the mutex
Thread ID = 217 releasing the mutex
Thread ID = 60 owns the mutex
Thread ID = 60 releasing the mutex
Thread ID = 177 owns the mutex
Thread ID = 177 releasing the mutex
```

Как видим, в каждый момент времени только один поток владеет мьютексом. Этот процесс похож на тот, что мы наблюдали в примере со спин-блокировкой (см. предыдущий раздел). Однако между двумя процессами имеется различие. В случае реализации механизма спин-блокировки поток остается в цикле. При этом он периодически переходит в режим ожидания, но в основном занят выполнением оператора цикла. Когда же поток ожидает освобождения мьютекса другим потоком, то он находится в приостановленном состоянии. При такой схеме системные ресурсы расходуются гораздо более эффективно.

События синхронизации

События представляют собой форму объектов синхронизации и позволяют посылать сигналы другим потокам, которые могут ожидать выполнения некоторых условий. В пространстве имен `System.Threading` определены два типа событий синхронизации:

- `AutoResetEvent`
- `ManualResetEvent`

Оба этих типа поддерживают метод `Set()`, который применяется для перевода объекта синхронизации в "сигнальное" состояние, что позволяет разблокировать один или несколько потоков. Кроме того, каждый тип имеет метод `Reset()`, приводящий объекты в "несигнальное" состояние, в результате чего блокируются все потоки, ожидающие наступления события. Потоки, которые должны ожидать сигнала о событии, могут использовать метод `WaitOne()` класса `WaitHandle`. Вызов этого метода блокирует поток, если событие находится в несигнальном состоянии.

Различие между двумя типами событий синхронизации определяется тем, каким образом разблокируются ожидающие потоки при передаче сигнала о событии. Тип `AutoResetEvent` позволяет разблокировать *один* поток, если событие переведено в сигнальное состояние путем вызова метода `Set()`. После этого система автоматически возвращает событие в несигналное состояние. Тип `ManualResetEvent` дает возможность отменить блокировку *всех* потоков, если с помощью метода `Set()` установлено сигнальное состояние события. Событие должно переводиться в несигналное состояние вручную потоком, вызывающим метод `Reset()`.

Следующий пример моделирует забег на 50 ярдов. Здесь используется тип `ManualResetEvent` для запуска группы потоков. Потоки "пробегают" дистанцию, используя цикл `for`, в котором счетчик принимает значения от 1 до 10:

```
__gc class Runner
{
public:
    Runner()
    {
        m_start = new ManualResetEvent(false);
        m_finished = 0;
    }

    void Race()
    {
        // Ожидание стартового сигнала
        m_start->WaitOne();

        // Начало забега
        for(int i = 1; i <= 10; i++)
        {
            Console::WriteLine(S"Runner {0} at track position {1}",
                               Thread::CurrentThread->Name, i.ToString());
            Thread::Sleep(AppDomain::GetCurrentThreadId());
        }

        m_position = Interlocked::Increment(&m_finished);

        static ManualResetEvent* m_start;
        int m_position;
    private:
        static int m_finished;
    };
};
```

Класс `Runner` содержит статический член `ManualResetEvent*`, который используется для того, чтобы все потоки начинали забег в одно и то же время. Данный класс также определяет статический член `m_finished`, которому дает приращение каждый поток, когда завершает работу. Оригинальное значение величины `m_finished` присваивается члену экземпляра `m_position`, чтобы сохранить информацию о том, каким по счету финиширует поток.

Конструктор создает новый экземпляр типа `ManualResetEvent` (первоначально в несигналном состоянии) и устанавливает значение `m_finished` равным 0.

Метод `Race()` моделирует забег, вызывая вначале метод `WaitOne()` события `m_start`. Это блокирует поток до тех пор, пока основной поток не передаст сигнал о событии. После того как метод `WaitOne()` возвращает управление, поток начинает выполнять цикл `for`, который отображает информацию о выполнении кода текущим потоком. Затем поток переходит в состояние ожидания на время, зависящее

от идентификатора потока. После выхода из цикла поток определяет, каким по счету он закончил гонку.

Следующий код инициализирует несколько бегунов (runners), присваивает новые экземпляры потоков методу `Race()` для каждого экземпляра класса `Runner` и объявляет старт забега:

```
Runner* r1 = new Runner();
Thread* t1 = new Thread(new ThreadStart(r1, Runner::Race));
t1->Name = S"W";

Runner* r2 = new Runner();
Thread* t2 = new Thread(new ThreadStart(r2, Runner::Race));
t2->Name = S"R";

Runner* r3 = new Runner();
Thread* t3 = new Thread(new ThreadStart(r3, Runner::Race));
t3->Name = S"O";

Runner* r4 = new Runner();
Thread* t4 = new Thread(new ThreadStart(r4, Runner::Race));
t4->Name = S"X";

t1->Start();
t2->Start();
t3->Start();
t4->Start();
```

В данный момент, несмотря на то что мы выполнили запуск потоков, все они ожидают передачи сигнала о стартовом событии.

```
Console: WriteLine(S"Runners ready!");
Thread: Sleep(1000);
```

```
Console: WriteLine(S"Bang!");
```

```
// Начало забега
```

```
Runner::m_start->Set();
```

Когда вызывается метод `set()` события `Runner::m_start`, сигнал передается всем ожидающим потокам, после чего они разблокируются и начинают выполнение кода. В результате на экран будет выведена такая информация:

```
Runners Ready! (Бегуны готовы!)
Bang! (Старт!)
Runner W at track position 1 (Бегун W находится на этапе 1)
Runner R at track position 1
Runner O at track position 1
Runner X at track position 1
Runner X at track position 2
Runner O at track position 2
Runner R at track position 2
Runner W at track position 2
```

Теперь основной поток ожидает завершения работы всех потоков, после чего выводит результаты забега:

```
// Ожидание завершения забега
```

```
t1->Join();
```

```
t2->Join();
```

```
t3->Join();
```

```

t4->Join();

// Вывод результатов бега
Console.WriteLine(S"Runner {0} finished {1}",
    t1->Name, t1->m_position.ToString());
Console.WriteLine(S"Runner {0} finished {1}",
    t2->Name, t2->m_position.ToString());
Console.WriteLine(S"Runner {0} finished {1}",
    t3->Name, t3->m_position.ToString());
Console.WriteLine(S"Runner {0} finished {1}",
    t4->Name, t4->m_position.ToString());

```

В целях ЭКОНОМИИ бумаги приведем результаты лишь для последних яровов забега:

```

Runner W at track position 10 (Бегун W находится на этапе 10)
Runner O at track position 10
Runner X at track position 8
Runner R at track position 9
Runner X at track position 9
Runner R at track position 10
Runner X at track position 10
Runner W finished 1 (Бегун W финишировал первым)
Runner R finished 3
Runner O finished 2
Runner X finished 4

```

Класс Monitor

До сих пор мы синхронизировали доступ к совместно используемой переменной или ресурсу. Однако часто бывает необходимо, чтобы целая область кода выполнялась неделимым образом. Такие области часто называются **критическими секциями (critical sections)**. В каждый момент времени только один поток может выполнять код в критической секции. Другие же потоки блокируются до тех пор, пока текущий поток не завершит свою работу в этой области кода.

Платформа .NET обеспечивает синхронизацию критических секций с помощью класса `System.Threading.Monitor`, который реализует статические методы `Enter()`, `TryEnter()` и `Exit()` для поддержки функциональности критических секций. При использовании класса `Monitor` экземпляр этого класса не создается обычным образом. Вместо этого код передает указатель на объект, к которому мы хотим применить один из методов класса `Monitor`.

Мы создадим приложение, которое обращается к объекту `Array1_list()`, используя несколько потоков. В этом приложении будет два потока. Один из них мы используем для добавления элементов в список `ArrayList`, а другой — для получения элементов из этого списка. Следующий фрагмент программы показывает, как можно использовать класс `Monitor` для того, чтобы в любой момент времени только один поток мог выполнять код для конкретного экземпляра объекта в заданной области кода:

```

using System;
using System.Collections;
using System.Threading;

public class gmMonitor
{
    private:
        ArrayList m_objList;

        // Число элементов
        int m_iCount;

    public:
        // Открытый конструктор

```

```

gcMonitor(int iCount)
{
    m_ptrList = new ArrayList();
    m_iCount = iCount;
}

// Функция, добавляющая элементы в массив
void Put()
{
    Monitor.Enter(this);
    // Добавление элементов в массив
    Monitor.Exit(this);

    // Остальной код
}

```

Метод Put() использует класс Monitor для гарантии того, что только один поток будет исполнять код между операторами вызова методов Enter() и Exit() для данного экземпляра класса Monitor. Если несколько потоков вызывают метод Enter() одного и того же объекта, они будут блокированы до тех пор, пока объект не освободится с помощью метода Exit(), вызванного тем же самым потоком, который уже запустил метод Enter().

Методы Monitor.TryEnter() и Enter() сходны друг с другом. Отличие между ними состоит лишь в том, что первый метод позволяет задать время ожидания (или свести интервал ожидания к нулю), если монитор заблокирован. Метод TryEnter() возвращает значение true, когда заблокированный объект освобождается до истечения времени ожидания. В противном случае возвращается значение false.

Приведенный ниже фрагмент кода демонстрирует реализацию функции Put(), которая добавляет элементы в список m_ptrList:

```
void Put()
```

```
{
```

```

    for(int i = 0; i < m_iCount; i++)
    {
        // Блокировка списка m_ptrList
        Monitor.Enter(m_ptrList);
        String l_strText = String.Format("Index = {0}", i.ToString());
        // Выполнение операции над списком m_ptrList
        m_ptrList.Add(l_strText);
        Console.WriteLine("Inside Put() = {0}", l_strText);
        // Моделирование длительной операции
        Thread.Sleep(500);
        // Снятие блокировки
        Monitor.Exit(m_ptrList);
    }
}

```

Смысл этой функции достаточно прозрачен. Внутри цикла мы блокируем список m_ptrList, используя метод Enter(), а затем добавляем элемент в список. Период ожидания в 500 миллисекунд задается для того, чтобы имитировать длительную операцию. Наконец, мы снимаем блокировку, вызвав метод Exit().

Теперь обратимся к реализации соответствующего метода `Get()`, который позволяет получать элементы внутри цикла из списка `m_ptrList`:

```
void Get()
{
    for(int i = 0; i < m_iCount; i++)
    {
        // Блокировка списка m_ptrList
        Monitor.Enter(m_ptrList);

        // Выполнение операций над списком m_ptrList
        String l_strText = dynamic_cast<String*>(m_ptrList->Item[i]);
        Console.WriteLine($"Inside Get = {0}", l_strText);

        // Моделирование длительной операции
        Thread.Sleep(100);

        // Снятие блокировки
        Monitor.Exit(m_ptrList);
    }
}
```

Эта функция похожа на метод `Put()` за исключением того, что здесь для получения величины используется свойство `ArrayList:Item`. Мы также вызвали метод `Sleep(100)`, чтобы имитировать длительный процесс.

Создадим в основной функции один экземпляр класса `gcMonitor` и передадим ссылки на него двум потокам А и В. В представленном ниже фрагменте программы поток А вызывает метод `Put()` для добавления элементов в список `gcMonitor:m_ptrList`, а поток В с помощью метода `Get()` получает значения из списка и выводит их на экран:

```
int tmain(void)
{
    // Создание одного объекта типа gcMonitor
    // Передача в конструктор количества элементов, равного 5
    gcMonitor* l_ptrMonitor = new gcMonitor(5);

    // Создание потока А
    Thread* ptrThreadA =
        new Thread(new ThreadStart(l_ptrMonitor, gcMonitor.Put));

    // Создание потока В
    Thread* ptrThreadB =
        new Thread(new ThreadStart(l_ptrMonitor, gcMonitor.Get));

    // Запуск потоков
    ptrThreadA->Start();
    Thread.Sleep(100); // Поток В запускается несколько позже
    ptrThreadB->Start();

    // Присоединение потоков А и В к основному потоку
    ptrThreadA->Join();
    ptrThreadB->Join();

    return 0;
}
```

Здесь при создании потока А в конструктор передается метод `gcMonitor:Put()`. Когда же создается поток В, ему передается метод `gcMonitor:Get()`. Между двумя

операторами запуска потоков вызывается метод `Sleep(100)`, с тем чтобы поток В начинал свою работу после А.

Пока поток А выполняет этот код, поток В успевает вызвать метод `Enter()` и ждет снятия блокировки списка `m_ptrList`. Когда это происходит, поток В немедленно блокирует данный список и получает значение первого элемента, после чего выводит его на экран. Затем поток В в течение некоторого времени находится в режиме ожидания. За это время поток А вызывает (на втором шаге цикла) метод `Enter()` для списка `m_ptrList` и ждет, пока поток В освободит данный список. Эта процедура выполняется 5 раз, поскольку именно это значение передано в качестве аргумента конструктору класса `gcMonitor`. Ниже представлена информация, выдаваемая программой на экран:

```
Inside Put() = 'Index = 0'
Inside Get = Index = 0
Inside Put() = 'Index = 1'
Inside Get = Index = 1
Inside PutC() = 'Index = 2'
Inside Get = Index = 2
Inside PutC() = 'Index = 3'
Inside Get = Index = 3
Inside Put() = 'Index = 4'
Inside Get = Index = 4
```

Хотя для каждой итерации потоку А требуется больше времени, чем потоку В, последний никогда не перебивает поток А, поскольку операторы вызова методов `Enter()` и `Exit()` управляют работой потоков. Если закомментировать указанные операторы и запустить программу, поток В попытается получить из списка те элементы, которые еще не добавлены, и это приведет к генерации исключения.

Кроме рассмотренных методов, класс `Monitor` имеет несколько других: `Pulse()`, `PulseAll()` и `Wait()`. Метод `Pulse()` используется для уведомления ожидающего потока об изменении состояния блокируемого объекта. Метод `PulseAll()` выдаст такое уведомление всем ожидающим потокам. Метод `Wait()` снимает блокировку текущего объекта, блокирует текущий поток и устанавливает режим ожидания того момента, когда текущий поток вновь завладеет объектом. Все эти функции должны вызываться из синхронизированного блока кода. В противном случае мы получим исключительную ситуацию `SynchronizedLockException`.

Класс `ReaderWriterLock`

Класс `ReaderWriterLock` поддерживает две блокировки: одну для синхронизации чтения, а другую — для синхронизации записи. Множественные потоки могут параллельно владеть блоком *чтения*, в то время как только один поток имеет возможность в любой заданный момент времени распоряжаться блоком *записи* и лишь тогда, когда ни один из потоков не контролирует блок чтения. Класс `ReaderWriterLock` реализует семантику, которая обычно называется "single writer, multiple reader" (один записывает, многие читают).

Данный класс очень полезен в ситуациях, когда несколько потоков имеют доступ к общему ресурсу, причем некоторые из них выполняют только чтение (не изменяющую операцию), а другие — только запись {изменяющую операцию}. В качестве примера рассмотрим совместно используемую структуру данных со многими членами. Если к ней имеет доступ несколько потоков, нам бы не хотелось, чтобы какой-нибудь поток изменил состояние структуры, пока выполняется процесс чтения. Это может привести к несовместимости данных. Класс `ReaderWriterLock` позволяет гарантировать, что процесс чтения не начнется, если какой-либо

другой поток изменяет информацию, относящуюся к состоянию структуры. Ниже описывается поведение класса `ReaderWriterLock`:

- Если один из потоков осуществляет запись, никакой другой поток не может выполнять чтение. Операция чтения будет блокирована до тех пор, пока не закончится запись.
 - Когда поток выполняет запись, ни один другой поток не может записывать данные. Операция записи другими потоками блокируется до того момента, пока первый поток не закончит запись.
 - Если поток считывает данные, никакой другой поток не может выполнять запись. Операция записи блокируется, пока не закончится чтение.
- Если поток осуществляет чтение, другие потоки могут также выполнять эту операцию.

В следующем примере экземпляр класса `ReaderWriterLock` используется для защиты списка `ArrayList` во время перебора элементов. Один поток считывает данные. Он перебирает элементы массива, складывая величины, содержащиеся в массиве, и выводя сумму на экран. Второй поток записывает данные, заменяя каждую величину в массиве новой случайной величиной. Вначале посмотрим, как выглядит класс, считывающий данные:

```

__gc class MyReader
{
public:
    static ArrayList* m_pList;
    ReaderWriterLock* m_pLock;

    MyReader(ArrayList* pList, ReaderWriterLock* pLock)
    {
        m_pList = pList;
        m_pLock = pLock;
    }

    void ThreadFunc()
    {
        for (int i = 0; i < m_pList->Count; i++)
        {
            Work();
        }
    }
};

Console.WriteLine(S"ThreadID {0}\n",
    AppDomain.GetCurrentThreadId().ToString());

// Получение доступа к блоку чтения и считывание строки
m_pLock->AcquireReaderLock(Timeout: Infinite);

Console.WriteLine(S"ThreadID {0}\n",
    AppDomain.GetCurrentThreadId().ToString());
Thread.Sleep(1000);

int sum = 0;
for(int i = 0; i < m_pList->Count; i++)

```

```

    {
        int value = *(dynamic_cast<__box int*>(m_pList->Item[i]));
        sum += value;
    }

    Console.WriteLine(S.ThreadID {0}, sum of data = {1}",
        AppDomain.GetCurrentThreadId().ToString(), sum.ToString());
    Console.WriteLine(S.ThreadID {0}, releasing Reader lock.",
        AppDomain.GetCurrentThreadId().ToString());
    m_pLock->ReleaseReaderLock();

    Console.WriteLine(S.ThreadID {0}, Reader lock released",
        AppDomain.GetCurrentThreadId().ToString());
}
}

```

Класс MyReader имеет метод ThreadFunc(), который используется в качестве целевой функции делегата ThreadStart. Этот метод трижды вызывает метод Work() в цикле for. Последний ждет получения доступа к блоку чтения, для чего вызывается метод AcquireReaderLock(), в аргументе которого задается неограниченное время ожидания. Когда метод AcquireReaderLock() возвращает управление, поток суммирует значения из списка и выводит результат. Затем поток освобождает блок чтения с помощью метода ReleaseReaderLock().

Обратимся теперь к классу, который используется для записи:

```

__gc class MyWriter
{
public:
    static ArrayList* m_pList;
    ReaderWriterLock* m_pLock;
    MyWriter(ArrayList* pList, ReaderWriterLock* pLock)
    {
        m_pList = pList;
        m_pLock = pLock;
    }

    void ThreadFunc()
    {
        for(int i=0; i < 3; i++)
        {
            void Work()
            {
                Console.WriteLine(S.ThreadID {0}, acquiring Writer lock",
                    AppDomain.GetCurrentThreadId().ToString());

                // Получение доступа к блоку записи и запись данных
                m_pLock->AcquireWriterLock(Timeout::Infinite);

                Console.WriteLine(S.ThreadID {0}, Writer lock acquired",
                    AppDomain.GetCurrentThreadId().ToString());

                // модифицируем данные
                AppDomain.GetCurrentThreadId().ToString();

                Thread.Sleep(1000);
            }
        }
    }
}

```

```

        for(int i = 0; i < m_pList->Count; i++)
        {
            int value = (new Random(i))>Next(1000);
            int oldval = *(dynamic_cast<__box int*>(m_pList->Item[i]));
            int newval = oldval + value;
            m_pList->Item[i] = __box(newval);
        }

        Console.WriteLine($"ThreadID {0} releasing Writer lock...",
            AppDomain.GetCurrentThreadId().ToString());

        m_pLock->ReleaseWriterLock();

        Console.WriteLine($"ThreadID {0} Writer lock released",
            AppDomain.GetCurrentThreadId().ToString());
    }
}
f;

```

Как и класс `MyReader`, `MyWriter` определяет метод `ThreadFunc()`, который используется как целевая функция делегата `ThreadStart`. Метод `ThreadFunc()` три раза вызывает функцию `Work()` в цикле `for`. Метод `Work()` ждет получения доступа к блоку записи с помощью функции `AcquireWriterLock()`, и аргументе которой задается неограниченный интервал ожидания. После того как метод `AcquireWriterLock()` возвращает управление, код перебирает элементы вектора, обновляя каждое значение. Затем блок записи освобождается с помощью метода `ReleaseWriterLock()`.

Наконец, соберем все фрагменты вместе и запишем код, который запускает потоки чтения и записи:

```

int _tmain(void)
{
    ArrayList* myList = new ArrayList();
    for(int i = 0; i < 10; i++)
    {
        int value = (new Random(i))>Next(1000);
        myList->Add(__box(value));
    }

    ReaderWriterLock* pLock = new ReaderWriterLock();

    MyReader* r = new MyReader(myList, pLock);
    MyWriter* w = new MyWriter(myList, pLock);

    Thread* t1 = new Thread(new ThreadStart(r, MyReader::ThreadFunc));
    Thread* t2 = new Thread(new ThreadStart(r, MyReader::ThreadFunc));
    Thread* t3 = new Thread(new ThreadStart(w, MyWriter::ThreadFunc));
    Thread* t4 = new Thread(new ThreadStart(w, MyWriter::ThreadFunc));

    t1->Start();
    t2->Start();
    t3->Start();
    t4->Start();

    t1->Join();
    t2->Join();
    t3->Join();
    t4->Join();

    return 0;
}

```

Этот код создает по два потока чтения и записи. В результате его выполнения на экран выдается текст большого объема, часть которого представлена ниже. Напомним, что, в отличие от потоков записи, потоки чтения могут совместно владеть блокировкой.

```
ThreadID = 1756, Acquiring Reader lock...
(Поток с идентификационным номером 1756 получает доступ к блоку чтения)
ThreadID = 1756, Reader lock acquired...reading data
(Поток с идентификационным номером 1756 получил доступ к блоку чтения...
Считывание данных)
ThreadID = 600, Acquiring Reader lock...
ThreadID = 600, Reader lock acquired...reading data
ThreadID = 1572, Acquiring Writer lock...
(Поток с идентификационным номером 1572 получает доступ к блоку записи)
ThreadID = 1564, Acquiring Writer lock...
ThreadID = 1756, sum of data = 497
(Поток с идентификационным номером 1756. Сумма данных равна 497)
ThreadID = 1756, Releasing Reader lock...
(Поток с идентификационным номером 1756 освобождает блок чтения)
ThreadID = 1756, Reader lock released...
(Поток с идентификационным номером 1756 освободил блок чтения)
ThreadID = 1756, Acquiring Reader lock...
ThreadID = 600, sum of data ≈ 497
ThreadID = 600, Releasing Reader lock...
ThreadID = 1572, Writer lock acquired...modifying data
(Поток с идентификационным номером 1572 получил доступ к блоку записи...
Обновление данных)
Thread ID ≈ 600, Reader lock released
ThreadID = 600, Acquiring Reader lock...
ThreadID = 1572, Releasing Writer lock...
(Поток с идентификационным номером 1572 освобождает блок записи)
ThreadID = 1572, Writer lock released...
(Поток с идентификационным номером 1572 освободил блок записи)
ThreadID = 1572, Acquiring Writer lock...
Thread ID ≈ 1756, Reader lock acquired...reading data
ThreadID = 600, Reader lock acquired...reading data
ThreadID = 1756, sum of data = 4994
```

« Продолжение распечатки выводимых данных сюда не включено »

Сделаем еще одно замечание, относящееся к классу `ReaderWriterLock`. Вызовы блоков кода, поставленные в очередь, всегда обрабатываются последовательно. В качестве примера рассмотрим приложение, имеющее три потока. Первый поток получает доступ к блоку чтения, и для выполнения задания ему требуется длительное время. Второй поток посылает запрос на предоставление доступа к блоку записи и ждет освобождения блока чтения. Если при этом третий поток запрашивает доступ к блоку чтения, ему придется ждать не только завершения чтения потоком 1, но и окончания записи потоком 2.

Исключительные ситуации, генерируемые при работе потоков

Платформа .NET Framework определяет несколько классов для работы с исключительными ситуациями, которые могут иметь место при использовании классов, относящихся к пространству имен `System.Threading`. Исключительные ситуации не передаются через границы потоков. Мы подробно рассмотрели все типы исключительных ситуаций и привели примеры кода, вызывающего передачу конкретных исключительных ситуаций. Исходный код, обсуждаемый в данном разделе, использует следующий класс:

```

public class MyThreadContext
{
    public:
        static void ThreadFunc()
        {
            Console.WriteLine(S"In ThreadFunc() - ");
            try
            {
                DoWork();
            }
            catch(Threading.ThreadAbortException*)
            {
                Console.WriteLine(S"In ThreadFunc() Thread Aborted!");
            }
            catch(Threading.ThreadInterruptedException*)
            {
                Console.WriteLine(S"In ThreadFunc() Thread Interrupted!");
            }
            finally
            {
                Console.WriteLine(S"In ThreadFunc() finally clause");
            }
        }

        static void DoWork()
        {
            Console.WriteLine(S"In DoWork()");
            try
            {
                Thread.Sleep(100000);
            }
            catch(Threading.ThreadAbortException*)
            {
                Console.WriteLine(S"In DoWork() Thread Aborted!");
            }
            finally
            {
                Console.WriteLine(S"In DoWork() finally clause");
            }
        }
};

```

В этом коде нет ничего необычного. Управляемый класс `MyThreadContext` определяет два метода:

- Метод `ThreadFunc()` будет использоваться нашим кодом как целевая функция делегата `ThreadStart`. Реализация данного метода включает блок `try...catch...finally`. Этот блок перехватывает различные исключительные ситуации, возникающие при работе потоков, которые могут генерироваться в результате вызова метода `DoWork()`.
- Метод `DoWork()` вызывается функцией `ThreadFunc()` и делает примерно то же самое, если не считать перехода в режим ожидания на достаточно длительное время (100 с). Это дает возможность исходному коду выполнять действия, приводящие к исключительным ситуациям.

Исключительная ситуация `SynchronizationLockException`

Выше мы рассматривали класс `Monitor`. Если бы мы попытались вызвать один из методов `Pulse()`, `PulseAll()` или `Wait()` из блока кода без синхронизации его классом `Monitor`, то получили бы исключительную ситуацию `SynchronizationLockException`.

```

Thread* t = new Thread(new ThreadStart(0, MyThreadContext, ThreadFunc));

try
{
    Console.WriteLine("Calling Monitor::Pulse()...");
    Monitor.Pulse(t);

    // Вызов методов PulseAll и Wait приведет к тупости
    // исключения SynchronizationLockException
    //Monitor.PulseAll(t);
    //Monitor.Wait(t);
}
catch(Threading.SynchronizationLockException*)
{
    Console.WriteLine("SynchronizationLockException caught...");
}

try
{
    Console.WriteLine("Calling Monitor::Exit()...");
    Monitor.ExitCt();
}
catch(Threading.SynchronizationLockException*)
{
    // Интересно отметить, что исключение не передается...
    Console.WriteLine("SynchronizationLockException caught...");
}

```

При выполнении этого кода на экран будет выдана следующая информация:

```

Calling Monitor::Pulse()... [Вызов метода Monitor::Pulse()]
SynchronizationLockException caught..
[Перехвачена исключительная ситуация SynchronizationLockException]
Calling Monitor::Exit()... [Вызов метода Monitor::Exit()]

```

Исключительная ситуация *ThreadAbortException*

Исключительные ситуации типа *ThreadAbortException* возникают, когда в потоке вызывается метод *ThreadAbort()*. Это особая категория исключительной ситуации, поскольку при ее перехвате она автоматически передается системой, заставляя срабатывать все обработчики *catch* для данной исключительной ситуации в стеке вызовов. Следующий код иллюстрирует данную исключительную ситуацию:

```

Thread* t = new Thread(new ThreadStart(0, MyThreadContext, ThreadFunc));

Console.WriteLine("Starting thread...");
t->Start();

Thread.Sleep(1000);

Console.WriteLine("Aborting thread...");
t->Abort();

```

Запустив этот код, получим:

```

Starting thread... [Запуск потока]
In ThreadFunc()... [Функция ThreadFunc()]
In DoWorkO... [Метод DoWork()]
Aborting thread... [Преждевременное прекращение работы потока]
In DoWork(): Thread Aborted!
[Метод DoWorkO. Работа потока преждевременно прекращена]
In DoWorkO : __finally clause [Метод DoWorkO. Предложение „finally"]
In ThreadFunc() : Thread Aborted!

```


[Функция ThreadFunc(). Работа потока преждевременно прекращена]
 In ThreadFunc():__finally clause [Функция ThreadFunc(). Предложение__finally]

Исключительная ситуация перехватывается как в методе DoWork(), так и в функции ThreadFunc(). Для иллюстрации мы обработали исключительную ситуацию в обеих функциях. Более стандартной практикой является обработка исключительной ситуации в вызываемой функции и возврат информации об исключительной ситуации вызывающему методу.

Исключительная ситуация ThreadInterruptedException

Модель исключительных ситуаций .NET Framework позволяет прерывать поток, когда он блокирован (т.е. находится в приостановленном состоянии или в состоянии ожидания). Для этого можно использовать метод Thread::Interrupt():

```
Thread* t = new Thread(new ThreadStart(0, MyThreadContext::ThreadFunc));
Console::WriteLine($Starting thread..");
t->Start();

Thread::Sleep(1000);

Console::WriteLine($Interrupting thread. ...");
t->Interrupt();
```

При выполнении этого кода на экран будет выведена такая информация:

```
Starting thread.., [Запуск потока]
In ThreadFuncO... [Функция ThreadFunc()]
In DoWork()... [Метод DoWorkO]
Interrupting thread... [Прерывание потока]
In DoWork():__finally clause [Метод DoWork(). Предложение __finally]
In ThreadFuncO : Thread Interrupted!
[Функция ThreadFunc(). Работа потока прервана]
In ThreadFunc():__finally clause [Функция ThreadFuncO. Предложение __finally]
```

Исключительная ситуация ThreadStateException

Некоторые члены-функции класса Thread могут быть вызваны лишь тогда, когда поток находится в определенном состоянии. Исключительная ситуация типа ThreadStateException генерируется, если метод вызывается в тот момент, когда экземпляр класса Thread находится в состоянии, несовместимом с этим вызовом метода. Представленный ниже код демонстрирует подобную ситуацию, вызывая метод Resume () для того потока, который еще не запущен и находится в приостановленном состоянии.

```
Thread* t = new Thread(new ThreadStart(0, MyThreadContext::ThreadFunc));
try
{
    Console::WriteLine($Resuming thread..");
    t->Resume();
}
catch (Threading::ThreadStateException*)
{
    Console::WriteLine($ThreadStateException caught..");
}
```

Результат выполнения этого кода имеет вид:

```
Resuming thread.. (Возобновление работы потока)
ThreadStateException caught...
(Исключительная ситуация ThreadStateException перехвачена)
```

ГЛАВА 6



Формы Windows

В предыдущей главе мы познакомились с некоторыми классами-утилитами, которые входят в библиотеку классов .NET Framework. Другие части этой библиотеки служат для обеспечения безопасности и доступа к данным, используются для работы с XML-данными и т.д. В данной главе рассматриваются классы, с помощью которых можно создавать **формы Windows**. Эти классы составляют подмножество библиотеки .NET Framework и позволяют разрабатывать приложения Windows с графическим интерфейсом пользователя.

С самого начала подчеркнем, что управляемый код C++ не является самым естественным средством разработки приложений для Windows. В Visual Studio .NET имеются такие инструменты, как Visual Basic и C#, которые позволяют графически компоновать приложения аналогично тому, как создается графический интерфейс пользователя с помощью библиотеки MFC. Однако Visual C++ .NET не поддерживает подобные возможности. Тем не менее, поскольку управляемый код C++ обладает всеми правами гражданства в .NET, ничто не мешает нам самостоятельно поработать в указанном направлении. В результате мы получим возможность лучше познакомиться с основными принципами разработки управляемого кода C++ и программирования для платформы .NET Framework.

В данной главе мы дадим читателю общее представление о формах Windows и сравним их с тем, что можно сделать с помощью библиотеки классов MFC (Microsoft Foundation Classes — Библиотека базовых классов Microsoft). Мы также рассмотрим создание формы с помощью управляемого кода C++, определение для нее различных свойств и добавление событий. Мы обсудим использование элементов управления .NET в формах Windows, созданных с помощью управляемого кода C++. Затем мы познакомимся с отображением модальных и немодальных диалоговых окон, а также с разработкой МТЛ-приложений на основе управляемого кода C++, В конце главы будет рассмотрено программирование операций drag-and-drop.

Формы Windows и библиотека MFC

Выполнение приложения, разработанного на основе MFC, начинается с метода `InitInstance()`, относящегося к потомку класса `CWinApp`, в котором мы создаем окно

или поток, обрабатываем сообщения Windows, после чего выполнение продолжается. Однако в управляемом коде C++ (и фактически во всех .NET-совместимых языках) первой всегда вызывается функция `main()`, откуда запускается статический метод `Run()` класса `System::Windows::Forms::Application` для создания окна и обработки его сообщений. Цикл обработки сообщений Windows выполняется в том потоке, откуда был вызван метод `Run()`.

В библиотеке MFC класс `CWnd` является базовым для всех окон и подчиненных элементов управления. Что касается форм Windows, то здесь аналогичную роль играет класс `System::Windows::Forms::Form`. Другое важное различие между рассматриваемыми инструментами состоит в том, что при использовании MFC диалоги сохраняются в файлах ресурсов, причем вид и содержимое диалогов описываются в этих файлах с помощью другого языка. В случае форм Windows диалоги и их подчиненные элементы управления представлены в исходном файле как операторы C++, что облегчает модификацию указанных графических компонентов приложения с помощью программного кода.

Кроме этих архитектурных изменений, формы Windows характеризуются наличием многих новых элементов управления, которые в случае использования библиотеки MFC должны разрабатываться программистами самостоятельно. Например, элемент управления `LinkLabel` может отображать гиперссылку. К другим новым элементам управления форм Windows относятся: `CheckedListBox`, `Splitter`, `ImageList`, `HelpProvider`, `ToolTip`, `ContextMenu`, `ToolBar`, `StatusBar`, `NotifyIcon`, а также все общие диалоги об открытии ресурсов. Ни один из перечисленных объектов не поддерживается в MFC, так чтобы его можно было просто перетащить мышью в окно приложения, используя средства редактора ресурсов времени разработки. Необходимо динамически инициализировать указанные элементы управления или создавать подклассы соответствующих базовых классов для достижения похожей функциональности.

С помощью MFC можно разрабатывать приложения, использующие интерфейсы SDI и MDI, а также диалоговые приложения. Причем приложения этих трех типов создаются на основе различных классов. В технологии .NET для разработки приложений всех указанных типов используется единственный класс `Form`. Различные свойства класса `Form` позволяют ему играть роль родительского либо подчиненного окна MDI, формы SDI, диалогового окна и т.д.

Создание приложения Windows с помощью управляемого кода C++

В данном разделе мы рассмотрим создание формы с помощью управляемого кода C++. Как упоминалось выше, дизайнер форм Visual Studio .NET доступен только в проектах C# и Visual Basic. Поэтому в нашем примере разработка, создание и добавление событий к форме будут выполнены только с помощью кода.

Откроем Visual Studio .NET и создадим новое управляемое приложение C++ (**Managed C++ Application**) с именем `FirstApp`. Добавим в файл `FirstApp.cpp` следующий код, который создаст форму, называемую `My First Managed C++ Windows Form`:

```
#include "stdafx.h"
#include <mscorlib.dll>
using namespace System;
using namespace System::Windows::Forms;
#include <tchar.h>

using namespace System;
using namespace System::Windows::Forms;
```

```

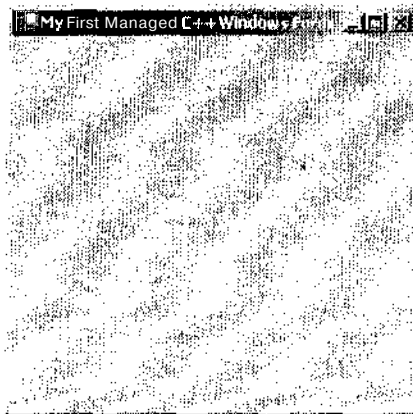
using System;
using System.Windows.Forms;

public class FirstApp : public Form
{
public :
    FirstApp()
    {
        Text = "My First Managed C++ Windows Form";
    }
};

int _tmain(void)
{
    Application::Run(new FirstApp);
    return 0;
}

```

Здесь импортируются две сборки: `System.dll` и `System::Windows::Forms.dll`, необходимые приложению для корректной работы. Затем, с целью упрощения кода добавляется оператор `using` для пространства имён `System::Windows::Forms` (в документации MSDN для каждого класса .NET указываются сборка и пространство имён, которые следует импортировать). Функция `main()` вызывает статический метод `Application::Run()`, передавая ему в качестве параметра указатель на новый экземпляр нашего класса формы `FirstApp`. В результате в текущем потоке создается новый цикл обработки сообщений Windows. Класс `FirstApp` очень простой. Он лишь присваивает в методе конструктора строковое значение свойству `Text` класса формы. Тем самым задается заголовок формы. Откомпилировав и запустив приложение, мы увидим, что оно работает:



При запуске этого приложения из интегрированной среды разработки на экран вначале выводится окно консоли, а затем отображается форма.

Эта форма содержит пиктограмму, задаваемую по умолчанию в левом верхнем углу, а также кнопки свертывания, развертывания и изменения размеров окна. Форма всегда имеет определенные начальные размеры. По умолчанию ее ширина и высота устанавливаются равными 300 пикселям, однако эти размеры можно изменять. Как легко заметить, при запуске приложения операционная система Windows располагает окно так, чтобы оно не совпадало с любыми предварительно открывавшимися окнами на рабочем столе. В следующем разделе мы покажем, как можно изменить поведение формы, заданное по умолчанию.

Настройка форм и добавление событий

Данный раздел мы посвятим изменению свойств и обработке событий формы. Вначале создадим новое управляемое приложение C++ иод названием CustomizedForm. Изменим в конструкторе некоторые свойства формы и добавим в код два обработчика событий мыши:

```

gc class CustomizedForm : public Form
{
public:
    CustomizedForm()
    {
        Text = S"Customized Form";
        // Установка свойства WindowState
        WindowState = FormWindowState::Maximized;
        // Добавляем обработчик событияMouseDown
        MouseDown += new MouseEventHandler(this, OnMouseDown);
        // Добавляем обработчик событияMouseMove
        MouseMove += new MouseEventHandler(this, OnMouseMove);
    }

private:
    void OnMouseDown(Object* ptrSender, MouseEventArgs* ptrMouse)
    {
        static_cast<Form*>(ptrSender)->Text = String::Concat(
            S"Customized Form, Mouse down at X = ", ptrMouse->X.ToString(),
            S", Y = ", ptrMouse->Y.ToString());
    }
    void OnMouseMove(Object* ptrSender, MouseEventArgs* ptrMouse)
    {
        static_cast<Form*>(ptrSender)->Text = String::Concat(
            S"Customized Form, Mouse move at X = ", ptrMouse->X.ToString(),
            S", Y = ", ptrMouse->Y.ToString());
    }
};

```

Можно максимизировать размер окна с помощью свойства WindowState. Другими возможными значениями этого свойства являются Minimized и Normal (задается по умолчанию). Представляет интерес та часть приложения, которая содержит код обработки событий. Данный код напоминает тот, что обсуждался в главе 2. В случае использования библиотеки MFC при работе с такими событиями, как перемещение мыши, мы вызываем мастер классов (Class Wizard), чтобы добавить макрос в карту сообщений, связывающую событие с методом класса окна. Однако при работе с платформой .NET используется делегат:

```

// Добавляем обработчик событияMouseDown
MouseDown += new MouseEventHandler(this, OnMouseDown);

// Добавляем обработчик событияMouseMove
MouseMove += new MouseEventHandler(this, OnMouseMove);

```

Для обработки событий MouseDown и MouseMove необходимо создать два экземпляра делегата MouseEventHandler и затем добавить их в список делегатов, который имеется в каждом событии. Конструктор делегата MouseEventHandler принимает два параметра:

указатель на объект, реализующий метод, который будет использоваться для обработки события, и имя самого метода. После запуска кода, когда наступает событие формы `MouseDown`, вызывается обработчик `OnMouseDown()`. Соответственно, при наступлении события `MouseMove` вызывается обработчик `OnMouseMove()`.

События `MouseDown` и `MouseMove` определены в классе `System::Windows::Forms::Control`, от которого происходит класс `Form`. Применительно к делегату `MouseEventHandler` соответствующие операторы объявления имеют вид:

```
__event MouseEventHandler* MouseDown;  
__event MouseEventHandler* MouseMove;
```

Класс `Form` определяет множество других полезных событий. Для получения полной информации о них необходимо обратиться к документации. Ниже перечислены лишь некоторые их категории:

- События, наступающие при изменении вида формы: `Activated`, `Closed`, `FontChanged`, `Resize` и `Invalidated`
- События клавиатуры: `KeyDown`, `KeyPress`, `KeyUp` и `TextChanged`
- События мыши: `MouseOver`, `MouseDown`, `MouseUp` и `MouseEnter`
- События операции `drag-and-drop`: `DragOver`, `DragEnter`, `DragLeave` и `DragDrop`

После того как задан механизм обработки событий, следует написать сами методы, обрабатывающие эти события. Реализация указанных методов весьма проста. Они обновляют надпись на форме, выводя строку, содержащую координаты курсора мыши `X` и `Y` относительно левого верхнего угла формы. Эти величины сохраняются свойствами `X` и `Y` объекта `EventArgs`. В данном случае не обязательно приводить тип `Object*` к типу `Form*`, поскольку все рассматриваемые нами методы и свойства являются членами одного и того же объекта.

Добавление на форму подчиненных элементов управления

Обратимся теперь к вопросу о добавлении элементов управления на формы Windows для того, чтобы можно было выполнять подтверждение ввода данных и отображение результатов. Элементы управления, используемые в формах Windows, можно отнести к следующим типам:

- o Отображающие текстовую информацию (например, `TextBox`, `Label` и `RichTextBox`)
- D Позволяющие отображать рисунки и манипулировать ими (например, `PictureBox`)
- a Помогающие выполнять различные действия (например, `Button`)
 - Предлагающие варианты выбора (например, `CheckBox`, `RadioButton`, `ListBox`, `ComboBox`, `MainMenu` и `ContextMenu`)
- D Имеющие расширенные возможности (например, `TreeView`, `ListView` и `TabControl`)
- `ActiveX`

Все эти элементы управления принадлежат к пространству имен `System::Windows::Forms` и происходят от класса `Control`. Иерархия наследования для них изображена на схеме:


```

void InitForm()
{
    Text = S'Managed C++ Button Sample';
    StartPosition = FormStartPosition::CenterScreen;
}

void AddFormEvents()
{
    // Добавляем обработчик события Resize
    Resize += new EventHandler(this, OnFormResize);
}

```

После создания формы необходимо создать кнопочный элемент управления:

```

void CreateChildControls()
{
    int l_iButtonWidth = 100;
    int l_iButtonHeight = 25;

    // Находим центр окна
    int l_iNewX = ClientSize.Width / 2 - l_iButtonWidth / 2;
    int l_iNewY = ClientSize.Height / 2 - l_iButtonHeight / 2;
    Point l_ptStart(l_iNewX, l_iNewY);

    // Создаем кнопку и устанавливаем значения свойств
    m_ptrbnButton = new Button;
    m_ptrbnButton->Name = S'm_btnButton1;
    m_ptrbnButton->Width = l_iButtonWidth;
    m_ptrbnButton->Height = l_iButtonHeight;
    m_ptrbnButton->Text = S'&Click here';
    m_ptrbnButton->Location = l_ptStart;

    // Добавляем обработчик события Click
    m_ptrbnButton->Click += new EventHandler(this, OnButtonClick);

    // Добавляем кнопку на форму
    Controls->Add(m_ptrbnButton);
}

```

Для того чтобы поместить кнопку в центр формы, нужно знать размер клиентской области для формы. Эта область не включает участки, занятые полем заголовка, меню, панелью инструментов, строкой состояния и т.д. Клиентская область представлена свойством формы `ClientSize` (объектом `Size`). Сам элемент `ClientSize` также имеет свойства `Width` и `Height`, которые используются для расчета координат центра формы. Затем эти координаты передаются объекту `Point`. По умолчанию размеры выражаются в пикселях.

Классы `Size` и `Point` включены в сборку `System::Drawing.dll`, поэтому в начало программы необходимо добавить директивы, импортирующие эту сборку, и пространство имен `System::Drawing`.

Далее мы создаем новый экземпляр кнопки и устанавливаем некоторые из ее свойств с помощью объекта `Point`, который специально создан для задания значения свойства `Location`. Затем код добавляет обработчик события кнопки `Click`. И, наконец, на форму помещается новый подчиненный элемент управления с помощью метода `Add()`, принадлежащего к коллекции `Controls`, встроенной в форму. Функция `Add()` имеет единственный аргумент, представляющий собой указатель на добавляемый элемент управления.

Обратимся теперь к обработчикам событий. Начнем с метода `OnFormResize()`, который должен заново найти центр формы и переместить туда элемент управления;


```

void OnFormResize(Object* l_ptrSender, EventArgs* l_ptrEventArgs)
{
    // Расчет координат центра
    int l_iNewX = ClientSize.Width / 2 - m_ptrbtnButton->Width / 2;
    int l_iNewY = ClientSize.Height / 2 - m_ptrbtnButton->Height / 2;
    Point l_ptNew(l_iNewX, l_iNewY);
    // Перемещение кнопки в центр окна
    m_ptrbtnButton->Location = l_ptNew;
    m_ptrbtnButton->Refresh();
}

```

После расчета изменившихся координат центра формы мы присваиваем новый объект Point свойству кнопки Location (сравните этот подход с вызовом функции `CWnd::MoveWindow()` в приложении, созданном на основе MFC!). Затем код вызывает метод `Refresh` объекта Button, в результате чего кнопка перерисовывается.

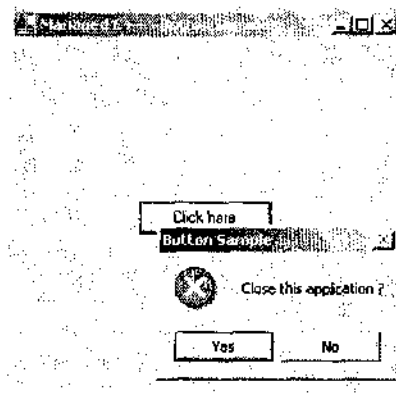
На последнем этапе разработки приложения необходимо написать обработчик события Click для кнопки:

```

void OnButtonClick(Object* l_ptrSender, EventArgs* l_ptrEventArgs)
{
    if (MessageBox::Show("Close this application ? ", "Button Sample",
        MessageBoxButtons::YesNo, MessageBoxIcon::Stop) == DialogResult::Yes)
    {
        Application::Exit();
    }
}

```

Для того чтобы выдать приглашение пользователю, вызывается статический метод `System::Windows::Forms::MessageBox::Show`. Хотя в начале программы мы создали кнопку, мы не предпринимаем никаких действий, чтобы уничтожить ее, поскольку сборщик мусора сделает эту работу. Окно нашего приложения будет иметь вид:

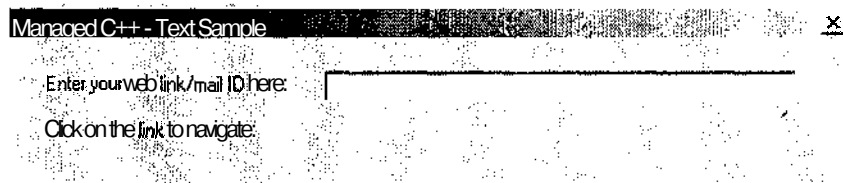


Работа с текстовыми элементами управления

В этом разделе мы познакомимся с элементами управления Label (Метка), LinkLabel (Гиперссылка) и TextBox (Текстовое поле), которые соответственно отображают статический текст, добавляют гиперссылку и позволяют пользователю вводить текстовую информацию. Элемент управления LinkLabel отсутствует в предыдущих версиях MFC. При работе с этими версиями для получения соответствующей

функциональности необходимо создать класс, производный от `CStatic`. Однако для Windows XP и более поздних версий корпорация Microsoft разработала новый класс MFC, который называется `CLinkCtrl`.

Для того чтобы больше узнать об упомянутых элементах управления, создадим новое приложение Windows Forms с именем `TextSample`. Единственная форма этого приложения будет содержать две метки, одно текстовое поле и одну гиперссылку. Текстовое поле предназначено для ввода указателя URL, и на основе этой информации должен динамически обновляться элемент управления `LinkLabel`:



Вначале создадим форму и добавим в ее класс члены, относящиеся к различным элементам управления:

```
_gc class TextSample : public Form
{
private:
    Label* m_ptrLabel1;
    Label* m_ptrLabel2;
    TextBox* m_ptrTBWebLink;
    LinkLabel* m_ptrLLWebLink;

public:
    // Добавление открытых методов
};
```

Мы инициализируем форму как диалоговое окно без кнопок сворачивания и разворачивания. Для этого свойству `FormBorderStyle` присваивается перечислимое значение `FixedDialog`, а свойства `MaximizeBox` и `MinimizeBox` устанавливаются равными `false`:

```
void InitForm()
{
    Text = S"Managed C++ - Text Sample";
    FormBorderStyle = FormBorderStyle::FixedDialog;
    WindowState = FormWindowState::Normal;
    StartPosition = FormStartPosition::CenterScreen;
    MaximizeBox = false;
    MinimizeBox = false;
}
```

Следующий фрагмент кода создает элементы управления и устанавливает значения их свойств:

```
void CreateChildControls()
{
    // Создание метки Label1
    m_ptrLabel1 = new Label;
    Point l_pt(20, 20);
    m_ptrLabel1->Location = l_pt;
    m_ptrLabel1->Name = S"label1";

    System::Drawing::Size l_sizeLabel1(175, 15);
```

```

m_ptrLabel1->Size = l_sizeLabel1;
m_ptrLabel1->Text = S"Enter your web link/mail ID here: ";
m_ptrLabel1->TextAlign = System::Drawing::ContentAlignment::MiddleLeft;

// Создание метки Label2
m_ptrLabel2 = new Label;
l_pt.X = 20;
l_pt.Y = 50;
m_ptrLabel2->Location = l_pt;
m_ptrLabel2->Name = S"Label2";

System::Drawing::Size l_sizeLabel2(175, 15);
m_ptrLabel2->Size = l_sizeLabel2;
m_ptrLabel2->Text = S"Click on the link to navigate. ";
m_ptrLabel1->TextAlign = System::Drawing::ContentAlignment::MiddleLeft;

// Создание текстового поля
m_ptrTBWebLink = new TextBox;
l_pt.X = 200;
l_pt.Y = 20;
m_ptrTBWebLink->Location = l_pt;
m_ptrTBWebLink->Name = S"m_TBWebLink";

System::Drawing::Size l_sizeTB(300, 20);
m_ptrTBWebLink->Size = l_sizeTB;

// Создание элемента управления LinkLabel
m_ptrLLWebLink = new LinkLabel;
l_pt.X = 200;
l_pt.Y = 50;
m_ptrLLWebLink->Location = l_pt;
m_ptrLLWebLink->Name = S"m_LLWebLink";

System::Drawing::Size l_sizeLL(300, 15);
m_ptrLLWebLink->Size = l_sizeLL;

// Остальной код
}

```

Здесь мы используем свойство `Size`, чтобы задать размер (высоту и ширину) всех элементов управления в пикселях, а инициализация данного свойства выполняется с помощью объекта `Size`. При создании объекта `Size` необходимо указать его пространство имен `System::Drawing`, чтобы устранить неоднозначность ссылок на этот объект и на свойство `Size` самого объекта `Form`.

Как и при создании кнопки в предыдущем примере, мы добавляем все эти элементы управления на форму, используя метод `Add()`, который относится к свойству формы `Controls`.

Добавим теперь события для элементов управления `TextBox` и `LinkLabel`. Когда пользователь вводит в текстовое поле указатель URL, можно одновременно обновить элемент управления `LinkLabel`, обработав событие текстового поля `TextChanged`:

```

m_ptrTBWebLink->TextChanged += new EventHandler(this, OnTextChanged);

```

Этот оператор добавляет обработчик `OnTextChanged()` к событию `TextChanged`. Реализация обработчика очень проста. При наступлении события значение свойства `Text` текстового поля присваивается одноименному свойству элемента управления `LinkLabel`:

```
void OnTextChanged(Object* l_ptrSender, EventArgs* l_ptrArgs)
{
    m_ptrLLWebLink->Text = m_ptrTBWebLink->Text;
}
```

Когда пользователь щелкает на ссылке LinkLabel, инициируется событие LinkClicked. Ниже представлен код обработки этого события:

```
m_ptrLLWebLink->LinkClicked += new LinkLabelLinkClickedEventHandler(this,
                                                                    OnLinkClicked);
```

Для данного события необходим делегат другого типа — LinkLabelLinkClicked-EventHandler. Реализация функции-обработчика вызывает метод System.Diagnostics::Process::Start() для открытия ресурса, на который указывает Web-ссылка:

```
void OnLinkClicked(Object* l_ptrSender,
                  LinkLabelLinkClickedEventArgs* l_ptrArgs)
{
    String* l_ptrURL = m_ptrLLWebLink->Text;
    l_ptrURL->Trim();

    if(l_ptrURL->Length == 0)
    {
        MessageBox::Show(S"URL cannot be empty.",
                        S"Managed C++ Text Sample",
                        MessageBoxButtons::OK, MessageBoxIcon::Information);
        return;
    }

    // Открытие ресурса
    System::Diagnostics::Process::Start(m_ptrLLWebLink->Text);
}
```

В нашем приложении при использовании протокола mailto может быть открыт почтовый клиент по умолчанию. Например, текст mailto:KNiranja@chn.cognizant.com помещенный в элемент управления TextBox, откроет почтовый клиент с адресом KNiranja@chn.cognizant.com в списке "To".

В приведенном примере мы продемонстрировали использование элемента управления LinkLabel для работы с единственной ссылкой, однако в один элемент управления может быть добавлено и несколько ссылок. Следующий код помещает в элемент управления LinkLabel одну строку и две ссылки:

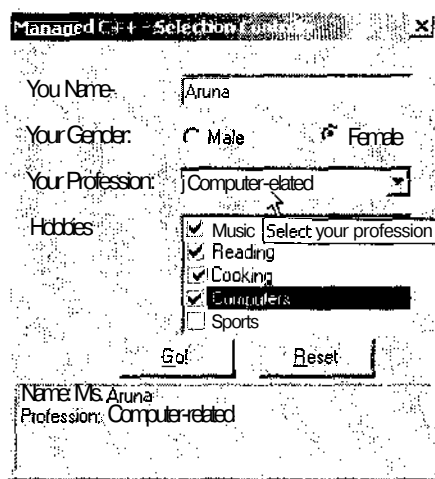
```
m_ptrLLWebLink->Text =
    S"For more information visit us at www.cognizant.com or contact N. Kumar";

// Добавление ссылок
m_ptrLLWebLink->Links->Add(34, 17, S"www.cognizant.com");
m_ptrLLWebLink->Links->Add(63, 14, S"mailto:KNiranja@chn.cognizant.com");
```

Здесь метод Add() добавляет ссылку HTTP www.cognizant.com и ссылку mailto: в соответствующие разделы текста. Для обработки события, наступающего при щелчке пользователя мышью на любой из этих ссылок, необходимо использовать аргумент LinkLabelLinkClickedEventArgs обработчика события LinkClicked. Данный аргумент содержит ссылку, для которой было инициировано событие Click. Ранее в нашем примере использовалось только свойство Text элемента управления LinkLabel, поскольку там мы имели лишь один указатель URL. Однако теперь один элемент управления содержит несколько ссылок, поэтому для добавления этих ссылок нужно использовать свойство LinkData аргумента LinkLabelLinkClickedEventArgs. Попробуйте самостоятельно написать код для получения ссылки, на которой пользователь щелкает мышью, и для открытия ресурса, определяемого этой ссылкой.

Элементы управления, предназначенные для выбора вариантов

В данном разделе мы опишем создание приложения `SelectionControls`, которое даст возможность пользователю вводить некоторую персональную информацию и будет выдавать на экран результат в отформатированном виде после занесения в форму каждого элемента данных. Мы используем текстовое поле для ввода имени пользователя и кнопку-переключатель (`RadioButton`) для указания пола. Комбинированное окно (`ComboBox`) отобразит список профессий, а элемент управления `CheckedListBox` (Список с флажками) позволит выбрать хобби. Каждый из перечисленных элементов будет снабжен всплывающими подсказками с помощью элемента управления `ToolTip`. Ниже показано, как должна выглядеть наша форма:



Метка в нижней части формы будет отображать введенные пользователем данные после нажатия кнопки `Go!`. Поскольку мы уже знакомы с созданием форм и некоторых элементов управления, рассмотрим лишь новые элементы кода.

Как и прежде, мы реализуем метод `CreateChildControls()`, который будет создавать и инициализировать все элементы управления. Начнем с создания текстового поля и добавим обработчик для события `LostFocus`, связанного с этим полем:

```
... m_ptrTBName->LostFocus += new EventHandler(this, OnLostFocus);
```

Для того чтобы пользователь мог указать свой пол, создадим две кнопки-переключателя. Когда изменяется состояние переключателей, инициируется событие `CheckedChanged`. Представленный ниже код задает также обработчик `OnCheckedChanged-RBMale()` для этого события:

```
... m_ptrRBMale = new RadioButton;
... l_pt.X = 110;
... l_pt.Y = 50;
... m_ptrRBMale->Location = l_pt;
... m_ptrRBMale->Name = S"m_rbMale";

System::Drawing::Size l_sizeRBMale(50, 20);
m_ptrRBMale->Size = l_sizeRBMale;
m_ptrRBMale->Text = S"Male";
```

```
m_ptrRBMale->CheckedChanged +=
    new EventHandler(this, OnCheckChangedRBMale);
m_ptrRBMale->TabIndex = 2;
```

Аналогичный код необходимо написать для второго переключателя, с помощью которого указывается женский пол, но в этом случае не нужно добавлять обработчик события. Поскольку имеется всего два переключателя, для них достаточно одного метода обработки события: если щелкнуть переключатель, указывающий женский пол, первый переключатель будет автоматически сброшен и инициирует событие `CheckedChanged`.

Поведение комбинированного окна по умолчанию таково, что в него можно вводить текст и добавлять новые элементы в список, содержащийся в этом элементе управления. Чтобы изменить такое поведение, необходимо сконфигурировать комбинированное окно как ниспадающий список, установив свойство `DropDownStyle` равным `DropDownList` (другими возможными вариантами являются `Simple` и `DropDown`). При изменении значения комбинированного окна будет инициировано событие `SelectedIndexChanged`, которое мы обработаем с помощью функции `OnSelectedIndexChanged()`:

```
m_ptrCBProfession = new ComboBox;
l_pt.X = 110;
l_pt.Y = 80;
m_ptrCBProfession->Location = l_pt;

System.Drawing.Size l_sizeCBProfession(150, 20);
m_ptrCBProfession->Size = l_sizeCBProfession;
m_ptrCBProfession->DropDownWidth = 100;
m_ptrCBProfession->DropDownStyle = ComboBoxStyle.DropDownList;
m_ptrCBProfession->TabIndex = 4;
m_ptrCBProfession->SelectedIndexChanged +=
    new EventHandler(this, OnSelectedIndexChanged);
```

Теперь обратимся к созданию списка с флажками (`CheckedListBox`), события которого мы не будем обрабатывать. Приведенный ниже фрагмент кода создает этот элемент управления и помещает его на форму:

```
m_ptrCLBHobbies = new CheckedListBox;
l_pt.X = 110;
l_pt.Y = 110;
m_ptrCLBHobbies->Location = l_pt;

System.Drawing.Size l_sizeCLBHobbies(150, 80);
m_ptrCLBHobbies->Size = l_sizeCLBHobbies;
m_ptrCLBHobbies->TabIndex = 5;
```

Следующий шаг заключается в создании двух кнопок с надписями `Go!` и `Reset`, а также метки, отображающей выходные данные. В предыдущих примерах мы уже добавляли на форму похожие элементы управления, поэтому не будем приводить здесь соответствующий код.

Наконец, создадим элемент управления `ToolTip` для вывода сообщений в виде всплывающей подсказки, после чего зададим сообщения для всех подчиненных элементов управления, используя метод `SetToolTip()` класса `ToolTip`. Следующий фрагмент кода иллюстрирует описанные действия:

```
m_ptrTtip = new ToolTip;
m_ptrTtip->SetToolTip(m_ptrTBName, S"Enter your name");
m_ptrTtip->SetToolTip(m_ptrCBProfession, S"Select your profession");
m_ptrTtip->SetToolTip(m_ptrCLBHobbies, S"Select your hobbies");
```

```
ffl_ptrTip->SetToolTip(rlptrbnGQ, S"Click here to display the output");
m_ptrTip->SetToolTip(m_ptrbnReset, S"Clear all inputs");
m_ptrTip->SetToolTip(m_ptrLabelOutput, S"Displays the output");
```

После создания всех подчиненных элементов управления и связывания с ними текста всплывающих подсказок поочередно вызовем для каждого из этих элементов метод `Controls->Add`, чтобы добавить их на форму.

Затем для комбинированного окна и списка с флажками необходимо выполнить некоторые установки по умолчанию, которые будут отображаться при первом появлении указанных элементов управления на экране. Для этого реализуем функцию `SetDefaultValuesO`:

```
void SetDefaultValuesO
{
    // Добавление элементов в комбинированное окно
    m_ptrCBProfession->Items->Add(S"None");
    m_ptrCBProfession->Items->Add(S"Student");
    m_ptrCBProfession->Items->Add(S"Computer related");
    m_ptrCBProfession->Items->Add(S"Government");
    m_ptrCBProfession->Items->Add(S"Private");
    m_ptrCBProfession->Items->Add(S"Self employed");
    m_ptrCBProfession->Items->Add(S"Retired");
    // По умолчанию задаем элемент "None" (Нет профессии)
    m_ptrCBProfession->SelectedIndex = 0;

    // Составляем список увлечений
    m_ptrCLBHobbies->Items->Add(S"Music", false);
    m_ptrCLBHobbies->Items->Add(S"Reading books", false);
    m_ptrCLBHobbies->Items->Add(S"Cooking", false);
    m_ptrCLBHobbies->Items->Add(S"Computers", false);
    m_ptrCLBHobbies->Items->Add(S"Sports", false);
}
```

Член `m_ptrCBProfession->Items` возвращает указатель на объект типа `ObjectCollection`. Метод `Add()` этого типа добавляет строку в объект `ComboBox`. Аналогично, первый параметр метода `m_ptrCLBHobbies->ItemsAdd` представляет собой строку, добавляемую в элемент управления `CheckedListBox`, а второй параметр булева типа указывает, будет ли флажок установлен или сброшен (как в нашем случае).

Последний этап создания приложения заключается в реализации обработчиков всех событий. Для удобства мы напишем общую функцию, которая просматривает текущее состояние всех объектов ввода данных, создает выходную строку и отображает ее в метке вывода:

```
void DisplayOutput()
{
    String* l_strName;
    String* l_strGender;

    // Получение имени пользователя
    l_strName = ffl_ptrTBName->Text;

    // Составляем строку, в которой указывается пол пользователя
    if(m_bMaleFlag)
        l_strGender = S"Name: Mr. ";
    else
        l_strGender = S"Name: Ms. ";

    // Определяем увлечения, отмеченные в списке
    String* l_strTemp = S"";
```

```

CheckedListBox: :CheckedItemCollection* l_ptrHobbiesCollection =
                                m_ptrClBHobbies->CheckedItems;
int nSelectedCount = l_ptrHobbiesCollection->Count;
for(int nCount = 0; nCount < nSelectedCount; nCount++)
{
    String* l_strItem = l_ptrHobbiesCollection->Item[nCount]->ToString();
    if(l_strTemp->Length == 0)
        l_strTemp = String::Concat(S"Hobbies: ", l_strItem);
    else
        l_strTemp = String::Concat(l_strTemp, S", ", l_strItem);
}
m_strHobbies = l_strTemp;

// Формируем выходную строку
String* l_strOutput;
l_strOutput = String::Concat(l_strGender, l_strName);
l_strOutput = String::Concat(l_strOutput, S"\n");
l_strOutput = String::Concat(l_strOutput, m_strProfession);
l_strOutput = String::Concat(l_strOutput, S"\n");
l_strOutput = String::Concat(l_strOutput, m_strHobbies);

m_ptrLabelOutput->Text = l_strOutput;
}

```

Наиболее интересна та часть этой функции, где определяется список увлечений, отмеченных пользователем. Свойство `CheckedItems` списка `CheckedListBox` возвращает указатель на коллекцию `CheckedItemCollection`, которая представляет собой список выбранных пользователем увлечений в элементе управления. Вначале мы получаем размер данной коллекции, для чего вызывается ее свойство `Count`. Затем используем цикл `for` для перебора всех отмеченных элементов и получения содержащегося в них текста с помощью метода `ToString()`. Список увлечений сохраняется в переменной `m_strHobbies`, которая представляет собой элемент формы. Данные из этой переменной последними добавляются в выходную строку.

Теперь кратко обсудим методы обработки событий. В основном они заключаются в определении величин, выбранных пользователем в различных элементах управления, и в вызове метода `DisplayOutput()` для обновления метки.

Мы еще нигде не присвоили значения переменной `m_bMaleFlag`. Сделаем это в обработчике событий `OnCheckChangedRBMale()`. Свойство `Checked` кнопки-переключателя возвращает значения `true` и `false`, если кнопка находится соответственно в положениях "включено" и "отключено":

```

void OnCheckChangedRBMale(Object* l_ptrSender, EventArgs* l_ptrArgs)
{
    m_bMaleFlag = m_ptrRBMale->Checked;
    DisplayOutput();
    m_ptrCBProfession->Focus();
}

```

Название профессии сохраняется в переменной `m_strProfession`, значение которой задается в обработчике событий `OnSelectedIndexChanged()`:

```

void OnSelectedIndexChanged(Object* l_ptrSender, EventArgs* l_args)
{
    m_strProfession =
        String::Concat(S"Profession: ", m_ptrCBProfession->SelectedItem);
    DisplayOutput();
}

```


После ввода имени пользователя в текстовое поле и потери фокуса этим элементом управления выходные данные обновляются. Теперь они будут содержать новое имя, а фокус передается кнопке-переключателю Male. Эти операции выполняются в методе OnLostFocus путем вызова функции DisplayOutput() и установки свойства Focus для кнопки Male:

```
void OnLostFocus(Object* l_ptrSender, EventArgs* l_ptrArgs)
{
    DisplayOutputO;
    m_ptrRBMale->Focus();
}
```

Обработчик события Click для кнопки GO! похож на приведенный выше:

```
void OnClickGOButton(Object* l_ptrSender, EventArgs* l_ptrArgs)
{
    DisplayOutputO;
}
```

Осталось написать лишь обработчик события Click кнопки Reset. Для реализации этого обработчика необходимо очистить текстовое поле и метку вывода данных, а также сбросить все флажки в списке увлечений. При этом следует иметь в виду, что в элементе управления CheckedListBox "выбор" элемента означает его выделение другим цветом, а при установке флажка в соответствующем прямоугольном поле появляется метка. Необходимо сбросить оба типа установок, прежде чем передать фокус текстовому полю Name с помощью функции Textbox::Focus():

```
void OnClickResetButton(Object* l_ptrSender, EventArgs* l_ptrArgs)
{
    // Очистка текстового поля
    m_ptrTBName->Text = S"";

    // По умолчанию задаем элемент "None" (Нет профессии)
    m_ptrCBProfession->SelectedIndex = 0;

    // Сбрасываем все флажки
    m_ptrCLBHobbies->ClearSelected();
    int nTotalCount = m_ptrCLBHobbies->Items->Count;
    for (int nCount = 0; nCount < nTotalCount; nCount++)
    {
        if(m_ptrCLBHobbies->GetItemChecked(nCount))
        {
            m_ptrCLBHobbies->SetItemChecked(nCount, false);
        }
    }

    // Очищаем метку вывода данных
    m_ptrLabelOutput->Text = S"";

    // Передаем фокус текстовому полю Name
    m_ptrTBName->Focus();
}
```

Другие подчиненные элементы управления

В приложениях для Windows в тех случаях, когда программа выполняется достаточно длительное время, часто виден индикатор выполнения, который отображает состояние программы. В другой распространенной ситуации пользователь должен выбрать некоторое значение: ему предлагается допустимый диапазон

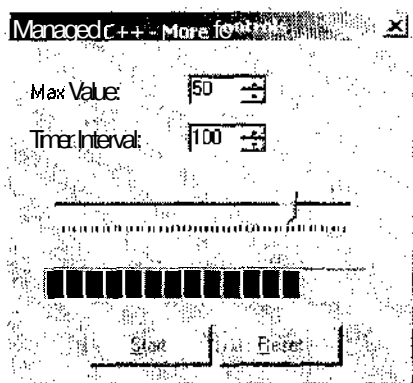
величин, которые можно задавать с помощью движка (пример — регулятор громкости). В данном разделе мы рассмотрим еще несколько элементов управления: `NumericUpDown`, `TrackBar`, `ProgressBar` и `Timer`. Первый из них применяется для того, чтобы пользователь мог ввести одну из величин, принадлежащих к фиксированному диапазону. Этот элемент управления фактически представляет собой комбинацию классов MFC `CSpinButtonCtrl` и `CEdit`. Элементы управления `TrackBar` (Движок) и `ProgressBar` (Индикатор выполнения) прямо соответствуют классам `CSliderCtrl` и `CProgressCtrl`.

Элемент управления `NumericUpDown` содержит свойства `Minimum` и `Maximum`, определяющие диапазон значений, а также свойство `Value`, в котором сохраняется текущая величина (она должна всегда находиться между значениями `Minimum` и `Maximum`). Когда пользователь щелкает на кнопке со стрелкой, значение свойства `Value` увеличивается или уменьшается, в зависимости от свойства `Increment`. Если свойство `Value` установлено так, что оно оказывается меньше, чем `Minimum` или больше, чем `Maximum`, генерируется исключительная ситуация типа `ArgumentException`.

Хотя движок и индикатор выполнения внешне существенно отличаются от элемента управления `NumericUpDown`, они имеют примерно такие же свойства и функциональность, которая связана с выбором величины из заданного диапазона. Таким образом, элементы управления `TrackBar` и `ProgressBar` обладают свойствами `Minimum`, `Maximum` и `Value`, причем последнее определяет положение движка в элементе `TrackBar` и состояние в индикаторе выполнения (`ProgressBar`).

Таймеры обычно применяются для управления регулярными фоновыми процессами. Этого может потребовать система, когда, например, необходимо периодически обновлять файлы и папки в `Windows Explorer`. Ниже мы рассмотрим создание объекта `Timer`, установку его свойства `Interval`, а также вызов методов `Start()` и `Stop()`. Такой подход отличается от используемого в MFC, где таймеры реализуются с помощью методов `SetTimer()` и `KillTimer()` класса `CWnd`.

Обратимся к форме, показанной на следующем рисунке. Она содержит все упомянутые элементы управления (код для нее представлен в проекте `MoreControls`). Максимальные значения для движка и индикатора выполнения заданы равными 50. Эта установка реализуется с помощью первого элемента управления `NumericUpDown`. Второй элемент управления этого типа задает значение свойства `Interval` для объекта `Timer`. В нашем случае это значение составляет 100 мс. Кнопка `Start` запускает таймер, который дает приращение свойствам `Value` движка и индикатора выполнения до тех пор, пока значение `Value` не достигает максимума. Кнопка `Reset` сбрасывает значения элементов управления `TrackBar` и `ProgressBar`, устанавливая начальные величины. Пока работает таймер, обе кнопки недоступны:



Приведенный ниже фрагмент кода показывает, каким образом создается первый элемент управления `NumericUpDown` и устанавливаются его свойства. Свойствам `Minimum`, `Maximum`, `Value` и `Increment` соответственно присваиваются значения 1, 100, 100 и 1. Когда этот элемент управления впервые появляется на экране, значение его свойства `Value` равно 100.

```
// Создание элемента управления NumericUpDown для задания диапазона
m_ptrNUDMaxValue = new NumericUpDown;
l_pt.X = 110;
l_pt.Y = 20;
m_ptrNUDMaxValue->Location = l_pt;
m_ptrNUDMaxValue->Name = S"m_NumericUpDown1";

System::Drawing::Size l_sizeNUD1(50, 20);
m_ptrNUDMaxValue->Size = l_sizeNUD1;
m_ptrNUDMaxValue->TabIndex = 1;
m_ptrNUDMaxValue->Minimum = 1;
m_ptrNUDMaxValue->Maximum = 100;
m_ptrNUDMaxValue->Value = 100;
m_ptrNUDMaxValue->Increment = 1;
```

Второй элемент управления `NumericUpDown` задает интервал таймера. На этот раз значения `Minimum`, `Value` и `Increment` устанавливаются равными 100, а свойству `Maximum` присваивается значение 1000:

```
m_ptrNUDTimer = new NumericUpDown;
l_pt.X = 110;
l_pt.Y = 50;
m_ptrNUDTimer->Location = l_pt;
m_ptrNUDTimer->Name = S"m_NumericUpDown2";

System::Drawing::Size l_sizeNUD2(50, 20);
m_ptrNUDTimer->Size = l_sizeNUD2;
m_ptrNUDTimer->TabIndex = 2;
m_ptrNUDTimer->Minimum = 100;
m_ptrNUDTimer->Maximum = 1000;
m_ptrNUDTimer->Value = 100;
m_ptrNUDTimer->Increment = 100;
```

Создание движка и индикатора выполнения очень похоже на процесс создания элемента управления `NumericUpDown`, хотя свойству `Value` мы теперь присваиваем другие значения:

```
// Создание элемента управления TrackBar
m_ptrTrackBar = new TrackBar;
l_pt.X = 20;
l_pt.Y = 90;
m_ptrTrackBar->Location = l_pt;

System::Drawing::Size l_sizeTrackBar(200, 20);
m_ptrTrackBar->Size = l_sizeTrackBar;
m_ptrTrackBar->TabIndex = 0;
m_ptrTrackBar->Minimum = 1;
m_ptrTrackBar->Maximum = 100;
m_ptrTrackBar->Value = 1;

// Создание элемента управления ProgressBar
m_ptrProgressBar = new ProgressBar;
l_pt.X = 20;
l_pt.Y = 140;
m_ptrProgressBar->Location = l_pt;
```

```
System: Drawing Size 1_sizeProgressBar(200, 20);
m_ptrProgressBar->Size = 1_sizeProgressBar;
m_ptrProgressBar->TabIndex = 0;
m_ptrProgressBar->Minimum = 1;
m_ptrProgressBar->Maximum = 100;
m_ptrProgressBar->Value = 1;
```

Далее создадим таймер и добавим обработчик для события Tick, которое будет наступать всякий раз, когда истечет время, равное интервалу таймера:

```
m_ptrTimer = new Timer;
m_ptrTimer->Tick += new EventHandler(this, OnTimerTick);
```

Интервал таймера устанавливается в другом месте, но обязательно до вызова метода Start(). Если этого не сделать, событие никогда не наступит.

Завершив написание кода для таймера, создадим две кнопки и добавим все элементы управления на форму. Исключение составляет лишь таймер, поскольку он не имеет визуального представления. Когда элемент управления невидим, на форму его добавлять не нужно. К невидимым элементам управления относятся Timer, DataSet и OleDbConnection.

Следующий шаг состоит в получении значений от элементов управления NumericUpDown и в запуске таймера. В обработчике события Click кнопки Start запишем код, считывающий значения свойств первого элемента управления NumericUpDown и сохраняющий их в членах класса MoreControls. Затем на основе полученных величин устанавливаются свойства движка и индикатора выполнения. Свойство таймера Interval задается, исходя из значения свойства Value второго элемента управления NumericUpDown. Затем вызывается метод таймера Start() и кнопки переводятся в недоступное состояние, чтобы пользователь не смог нажимать их, пока работает таймер:

```
void OnClickStartButton(Object* lptrSender, EventArgs* 1_ptrArgs)
```

```
// Получение значений от элементов управления NumericUpDown
m_nCurrMaxValue = static_cast<int>(m_ptrNUDMaxValue->Value);
m_nMinValue = static_cast<int>(m_ptrNUDMaxValue->Minimum);
m_nMaxValue = static_cast<int>(m_ptrNUDMaxValue->Maximum);
m_nIncValue = static_cast<int>(m_ptrNUDMaxValue->Increment);
m_nCurrentValue = m_nStartValue = m_nMinValue;
```

```
// Установка диапазона для движка
m_ptrTrackBar->Minimum = m_nMinValue;
m_ptrTrackBar->Maximum = m_nCurrMaxValue;
m_ptrTrackBar->Value = m_nMinValue;
```

```
// Установка диапазона для индикатора выполнения
m_ptrProgressBar->Minimum = m_nMinValue;
m_ptrProgressBar->Maximum = m_nCurrMaxValue;
m_ptrProgressBar->Value = m_nMinValue;
```

```
// Установка интервала таймера
```

```
m_ptrTimer->Interval = static_cast<int>(m_ptrNUDTimer->Value);
```

```
// Запуск таймера
```

```
m_ptrTimer->Start();
```

```
// Перевод обеих кнопок в недоступное состояние
```

```
m_ptrbtnStart->Enabled = false;
```

```
m_ptrbtnReset->Enabled = false;
```

```
}
```

В обработке события Tick объекта Timer мы задаем приращение величины `m_nCurrentValue`, равное `m_nIncValue`. Затем значение `m_nCurrentValue` сравнивается с максимальной величиной. Если максимум не достигнут, то операторы в коде устанавливают текущие значения свойств Value для движка и индикатора выполнения. Использование этих операторов приводит к изменению внешнего вида указанных элементов управления после вызова метода Refresh() для их перерисовки. Если переменная `m_nCurrentValue` больше максимального значения, таймер останавливается, а кнопка **Reset** становится доступной:

```
void OnTimerTick(Object* l_ptrSender, EventArgs* l_ptrArgs)
{
    try
    {
        m_nCurrentValue += m_nIncValue;
        if (m_nCurrentValue > m_nCurrMaxValue)
        {
            m_ptrTimer->Stop();
            // Делаем доступной кнопку Reset
            m_ptrbtnReset->Enabled = true;
            return;
        }

        // Задаем текущие значения для элементов управления
        m_ptrProgressBar->Value = m_nCurrentValue;
        m_ptrTrackBar->Value = m_nCurrentValue;

        // Вручную обновляем элементы управления
        m_ptrProgressBar->Refresh();
        m_ptrTrackBar->Refresh();
    }
    catch (ArgumentException *e)
    {
        // Остановка таймера
        m_ptrTimer->Stop();

        // Делаем обе кнопки недоступными
        m_ptrbtnStart->Enabled = false;
        m_ptrbtnReset->Enabled = false;
        MessageBox::Show(e->Message, S"Managed C++ - More controls",
            MessageBoxButtons::OK, MessageBoxIcon::Stop);
    }
}
```

Здесь блок Catch обрабатывает исключение ArgumentException, которое инициируется, если текущие значения движка или индикатора выполнения выходят за пределы допустимых диапазонов.

При нажатии кнопки Reset элементы управления TrackBar и ProgressBar заново инициализируются, после чего становится доступной кнопка Start:

```
void OnClickResetButton(Object* l_ptrSender, EventArgs* l_ptrArgs)
{
    m_nStartValue = rajiMinValue;
    m_ptrProgressBar->Value = m_nStartValue;
    m_ptrTrackBar->Value = m_nStartValue;

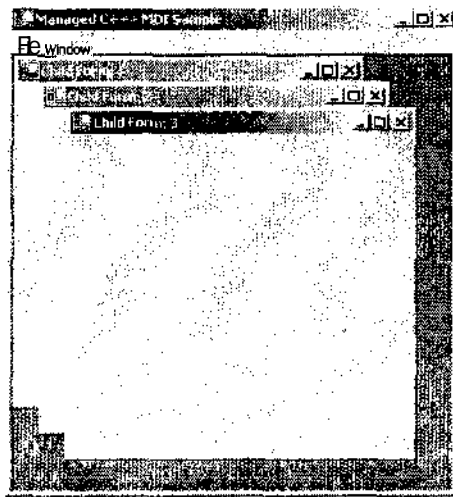
    // Делаем доступной кнопку Start
    m_ptrbtnStart->Enabled = true;
}
```

Меню и формы с многодокументным интерфейсом

Можно выделить три основных типа разработки приложений с графическим интерфейсом пользователя. Это создание диалогов, приложений с **однодокументным** интерфейсом (SDI) и программ с многодокументным интерфейсом (MDI). До сих пор мы рассматривали SDI- и диалоговые приложения. Такое поведение было задано путем присваивания свойству формы `FormBorderStyle` значения `Sizable` (устанавливается по умолчанию) для SDI-приложений и значения `FixedDialog` для диалогов.

MDI-приложения позволяют работать одновременно с несколькими формами. Создание MDI-приложений с помощью форм Windows оказывается очень простым. Класс `Form` имеет свойство `IsMdiContainer`. Когда это свойство устанавливается равным `true`, форма ведет себя как контейнер MDI и внутри нее можно открывать несколько других форм. При этом свойство `MdiParent` любой подчиненной формы должно указывать на контейнер.

В случае использования MFC для разработки MDI-приложений требуется пять классов. Несмотря на то что мастер классов генерирует их, понять и затем реализовать нужную архитектуру очень непросто. Создание же приложения на основе форм Windows по существу сводится к заданию нескольких свойств. Приведенный ниже снимок экрана демонстрирует четыре формы, причем внешняя форма содержит три остальных:



На этом рисунке можно видеть элементы главного меню `File` и `Window`. Меню `File` (Файл) содержит команды `New Form` (Новая форма) и `Close` (Заккрыть). При открытии меню `Window` (Окно) мы получим доступ к командам `Cascade` (Расположить каскадом), `Tile Horizontal` (Выстроить по горизонтали) и `Tile Vertical` (Выстроить по вертикали). В формах Windows меню создаются с помощью класса `MainMenu`, который отвечает за реализацию строки главного меню. Подчиненные элементы меню создаются на основе класса `MenuItem`.

До сих пор код наших приложений размещался в одном .cpp-файле, где одна за другой содержались функции-члены. Начиная с этого примера, мы будем включать определения в заголовочные файлы.

В следующем объявлении класса MDIForm мы декларируем меню, их элементы и добавляем обработчики событий для отдельных элементов:

```
__gc class MDIForm : public Form
{
private:
    // Главное меню
    MainMenu* m_ptrMainMenu;

    // Элементы меню File и Window
    MenuItem* m_ptrFileMenu;
    MenuItem* m_ptrWindowMenu;

    // Подчиненные элементы меню File
    MenuItem* m_ptrNewFormMenu;
    MenuItem* m_ptrCloseMenu;

    // Подчиненные элементы меню Window
    MenuItem* m_ptrCascadeMenu;
    MenuItem* m_ptrHorizontalMenu;
    MenuItem* m_ptrVerticalMenu;

    void InitForm();
    void CreateMenu();

    // Обработчик события Click элемента меню File->Close
    void OnFileClose(Object* l_ptrSender, EventArgs* l_ptrArgs);

    // Обработчик события Click элемента меню File->New Form
    void OnFileNewForm(Object* l_ptrSender, EventArgs* l_ptrArgs);

    // Обработчик события для всех элементов меню Window
    void OnArrangeWindows(Object* l_ptrSender, EventArgs* l_ptrArgs);

    // Количество подчиненных форм
    int m_nFormCount;

public:
    // Открытый конструктор
    MDIForm();
};
```

Метод InitForm() инициализирует форму, тогда как функция CreateMenu() создает элементы меню. Обе эти функции вызываются из конструктора:

```
// Конструктор класса MDIForm
MDIForm::MDIForm()
{
    InitForm();
    CreateMenu();
}
```

Метод InitForm() устанавливает значение свойства IsMdiContainer равным true, превращая форму в контейнер MDI:

```
void MDIForm::InitForm()
{
    Text = S"Managed C++ - MFC Sample";
    WindowState = FormWindowState::Maximized;
    StartPosition = FormStartPosition::CenterScreen;
```

```

        IsMdiContainer = true;
        m_nFormCount = 0;
    }

```

Следующий этап заключается в реализации меню и присоединении их к контейнеру MDI. Вначале создадим объект `MainMenu`, а затем — подчиненные элементы меню, которые будут представлены объектами `MenuItem`:

```

void MDIForm::CreateMenu()
{
    // Создание главного меню
    m_ptrMainMenu = new MainMenu;

    // Создание элементов главного меню
    m_ptrFileMenu = new MenuItem(S"&File");
    m_ptrWindowMenu = new MenuItem(S"&Window");
}

```

Аргумент, передаваемый в конструктор класса `MenuItem`, представляет собой название меню. Далее необходимо создать подчиненные элементы для меню `File`. Для этого в конструкторе класса `MenuItem` следует задать два параметра, последний из которых будет указывать на функцию обработки события, наступающего при выборе элемента данного меню:

```

// Создание подчиненных элементов для меню File
m_ptrNewFormMenu = new MenuItem(S"&New Form",
                                new EventHandler(this, OnFileNewForm));
m_ptrCloseMenu = new MenuItem(S"&Close",
                               new EventHandler(this, OnFileClose));

```

После этого добавим подчиненные элементы в меню `File`, а последнее — в главное меню, используя метод `Add()` коллекции `MenuItems`, принадлежащей к объекту `MenuItem`:

```

// Добавление подчиненных элементов в меню File
m_ptrFileMenu->MenuItems->Add(0, m_ptrNewFormMenu);
m_ptrFileMenu->MenuItems->Add(1, m_ptrCloseMenu);

// Добавление меню File в главное меню
m_ptrMainMenu->MenuItems->Add(0, m_ptrFileMenu);

```

Подчиненные элементы для меню `Window` добавляются и связываются с обработчиками событий аналогичным образом. В данном случае все три элемента меню связываются с единственным обработчиком `OnArrangeWindows()`:

```

// Создание подчиненных элементов для меню Window
m_ptrCascadeMenu = new MenuItem(S"&Cascade",
                                new EventHandler(this, OnArrangeWindows));
m_ptrHorizontalMenu = new MenuItem(S"&Tile &Horizontal",
                                   new EventHandler(this, OnArrangeWindows));
m_ptrVerticalMenu = new MenuItem(S"&Tile &Vertical",
                                 new EventHandler(this, OnArrangeWindows));

// Добавление подчиненных элементов а меню Window
m_ptrWindowMenu->MenuItems->Add(0, m_ptrCascadeMenu);
m_ptrWindowMenu->MenuItems->Add(1, m_ptrHorizontalMenu);
m_ptrWindowMenu->MenuItems->Add(2, m_ptrVerticalMenu);

// Добавление меню Window в главное меню
m_ptrMainMenu->MenuItems->Add(1, m_ptrWindowMenu);

```


Наконец, объект `MainMenu` присоединяется к контейнеру MDI:

```
// Присоединения меню к форме
Menu = m_ptrMainMenu;
}
```

Осталось реализовать обработчики событий для различных элементов меню, а также класс `ChildForm`. При выборе пользователем элемента меню `File | New Form` должна создаваться и отображаться на экране новая подчиненная форма:

```
// Обработчик событий для элемента меню File | New Form
void MDIForm::OnFileNewForm(Object* l_ptrSender, EventArgs* l_ptrArgs)
{
    ChildForm* l_ptrChildForm = new ChildForm(this, ++m_nFormCount);
    l_ptrChildForm->Show();
}
```

Ниже мы вкратце обсудим реализацию класса `ChildForm`. Конструктор этого класса принимает указатель на родительский контейнер, а также целое число, используемое контейнером для отслеживания подчиненных форм.

Когда пользователь выбирает пункт меню `File | Close`, на экран выдается сообщение. Если пользователь подтверждает закрытие приложения, вызывается метод `Close()` родительской формы:

```
// Обработчик события для элемента меню File | Close
void MDIForm::OnFileClose(Object* l_ptrSender, EventArgs* l_ptrArgs)
{
    if(MessageBox::Show(
        "Do you wish to close this application?", "Managed C++ - MDI Sample",
        MessageBoxButtons::YesNo, MessageBoxIcon::Stop) == DialogResult::Yes)
    {
        Close();
    }
}
```

Почти все MDI-приложения предоставляют пользователям возможность выстраивать формы в определенном порядке, и мы уже добавили соответствующие элементы в меню **Window**. Метод `Form::LayoutMdi()` позволяет легко реализовать указанную возможность путем задания одного из трех параметров: `Cascade`, `TileHorizontal` и `TileVertical`, в зависимости от которых формы выстраиваются определенным образом. Эти параметры соответствуют имеющимся элементам в меню **Window**. Событие, наступающее при выборе любого из этих элементов, обрабатывается методом `OnArrangeWindows()`. Поэтому необходимо, чтобы данный метод вызывался с параметром, указывающим, какой из объектов `MenuItem` инициировал событие причем, для определения объекта мы используем свойство `Index`:

```
// Обработчик событий для всех элементов меню Window
void MDIForm::OnArrangeWindows(Object* l_ptrSender, EventArgs* l_ptrArgs)
{
    MenuItem* l_ptrCurrItem = dynamic_cast<MenuItem*>(l_ptrSender);

    // Определение индекса и соответствующее выстраивание окон
    switch(l_ptrCurrItem->Index)
    {
        case 0:
            LayoutMdi(MdiLayout::Cascade);
            break;
        case 1:
            LayoutMdi(MdiLayout::TileHorizontal);
    }
}
```

```

        break;
    case 2:
        LayoutMdi(MdiLayout::TileVertical);
        break;
    default:
        break;
}

```

Осталось лишь написать реализацию класса `ChildForm`:

```

__gc class ChildForm : public Form
{
public:
    // Основной конструктор
    ChildForm(Form* ptrParent, int l_nCount);
};

// Конструктор класса ChildForm
ChildForm::ChildForm(Form* ptrParent, int l_nCount)
{
    Text = String::Concat(S"Child Form: ", l_nCount.ToString());
    WindowState = FormWindowState::Normal;
    MdiParent = ptrParent;
}

```

Экземпляру класса `ChildForm` присваивается номер, который передается в конструктор. Важно отметить, что мы присвоили свойству `MdiParent` указатель на родительский контейнер. Этот указатель представляет собой первый аргумент конструктора. Если не установить данное свойство, подчиненная форма не будет создана внутри границ окна MDI и ее нельзя будет разместить в пределах родительской формы. При закрытии родительской формы подчиненные формы закрываются автоматически.

Модальные и немодальные формы

Пока модальная подчиненная форма не закрыта, она не позволяет передавать фокус родительской форме. Для создания модальной формы в приложении Windows Forms необходимо использовать метод `ShowDialog()`:

```

l_ptrChildForm = new ChildForm(S"Child Form - Modal");
l_ptrChildForm->ShowDialog();

```

Для отображения немодальной формы следует вызывать метод `Show()`, как сделано в нашем примере:

```

l_ptrChildForm = new ChildForm(S"Child Form - Modeless");
l_ptrChildForm->Show();

```

При использовании библиотеки MFC модальный диалог открывается с помощью метода `CWnd::DoModal()`. Чтобы отобразить немодальный диалог, нужно динамически выделить память для объекта диалога, создать его с помощью метода `CWnd::Create()`, а затем вызвать его метод `CWnd::ShowWindow()` для вывода на экран окна диалога.

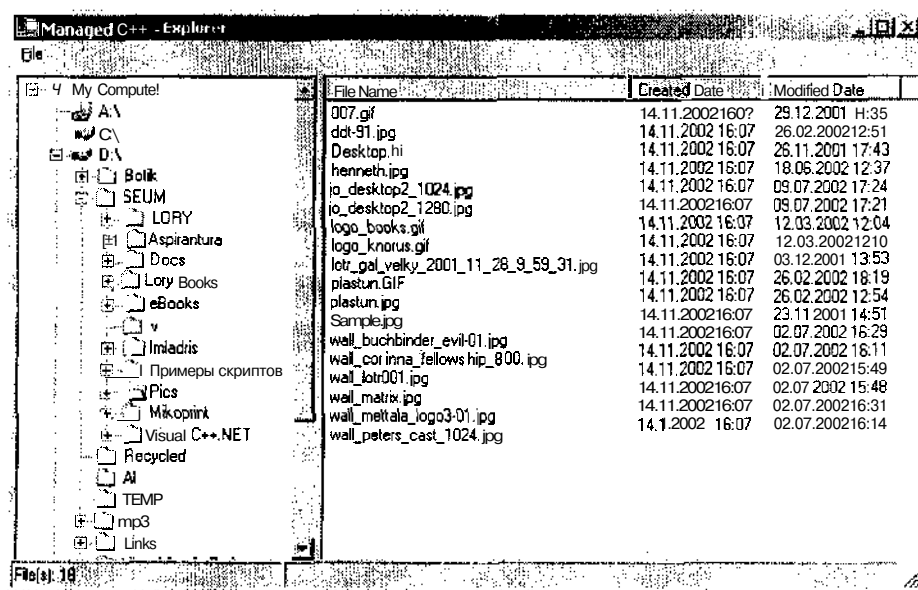
Элементы управления Windows Forms с расширенными возможностями

Мы уже обсудили некоторые элементы управления, доступные в проектах Windows Forms. В данном разделе мы рассмотрим еще несколько элементов

управления, обладающих расширенными возможностями: TreeView, ListView, StatusBar и ImageList. Затем мы покажем, как реализуется операция drag-and-drop в формах Windows.

Разработка приложения Windows Explorer

Разработаем приложение Explorer, которое моделирует поведение проводника Windows Explorer. Это приложение будет выводить системные диски, папки и файлы. Для его реализации добавим диски и папки в элемент управления TreeView и файлы — в список ListView. При выборе диска или папки в дереве каталогов TreeView будет обновляться список ListView, содержащий файлы. Мы также предоставим пользователю возможность копировать файлы, перетаскивая их мышью из элемента управления ListView и оставляя в папке дерева каталогов TreeView. Окно нашего приложения имеет следующий вид:



Для вывода на экран узлов в элементе управления TreeView используются изображения, тогда как в списке ListView показаны имена файлов, а также даты их создания и изменения. Элемент управления StatusBar (Строка состояния) применяется для отображения количества файлов и информации о состоянии. Приложение также содержит два элемента меню File | Refresh и File | Close. Полный код этого приложения содержится в загружаемых файлах, поэтому мы остановимся лишь на наиболее интересных фрагментах.

Вначале необходимо определить класс ExplorerForm и объявить переменные-указатели для всех элементов управления. В частности, нужно создать указатели типа TreeView (для дерева каталогов), ListView (для списка файлов), ImageList (чтобы связать изображения с узлами дерева каталогов), StatusBar, а также два объекта StatusBarPanel и элементы меню:

```

...gc class ExplorerForm : public Form
{
private

```

```
// Подчиненные элементы управления
TreeView* m_ptrTVFolders;
ListView* m_ptrLVFiles;
ToolTip* m_ptrToolTip;
ImageList* m_ptrImageList;
StatusBar* m_ptrStatusBar;
StatusBarPanel* m_ptrSBPanel1;
StatusBarPanel* m_ptrSBPanel2;

// Меню
MainMenu* m_ptrMainMenu;
MenuItem* m_ptrFileMenu;
MenuItem* m_ptrRefreshMenu;
MenuItem* m_ptrCloseMenu;
```

Процесс создания формы включает такие этапы, как инициализация, создание меню и подчиненных элементов управления, а также последующее заполнение дерева каталогов TreeView. Эта последовательность отражена в реализации конструктора класса ExplorerForm:

```
ExplorerForm::ExplorerForm(>
{
    InitForm();
    CreateMenu();
    CreateChildControls();
    PopulateDrives();
}
```

Поскольку мы уже имели дело с инициализацией форм и созданием меню, сконцентрируемся на реализации новых для нас элементов управления. Метод CreateChildControls() добавляет на форму все элементы управления и связанные с ними обработчики событий. Следующий фрагмент кода иллюстрирует создание дерева TreeView и списка ListView:

```
// Создание элемента управления TreeView
m_ptrTVFolders = new TreeView;
m_ptrTVFolders->AllowDrop = true;

// События элемента управления TreeView
m_ptrTVFolders->AfterSelect +=
    new TreeViewEventHandler(this, OnAfterSelectTVFolders);
m_ptrTVFolders->DragEnter +=
    new DragEventHandler(this, OnDragEnterTVFolders);
m_ptrTVFolders->DragOver +=
    new DragEventHandler(this, OnDragOverTVFolders);
m_ptrTVFolders->DragLeave +=
    new EventHandler(this, OnDragLeaveTVFolders);
m_ptrTVFolders->DragDrop +=
    new DragEventHandler(this, OnDragDropTVFolders);

// Создание элемента управления ListView
m_ptrLVFiles = new ListView;
m_ptrLVFiles->MultiSelect = false;
m_ptrLVFiles->View = View::Details;
m_ptrLVFiles->FullRowSelect = true;
m_ptrLVFiles->Columns->Add(
    S"File Name", 325, HorizontalAlignment::Left);
m_ptrLVFiles->Columns->Add(
    S"Created Date", 100, HorizontalAlignment::Left);
```

```

m_ptrLVFiles->Columns->Add(
    S"Modified Date", 100, HorizontalAlignment::Left);
// Событие ItemDrag
m_ptrLVFiles->ItemDrag +=
    new ItemDragEventHandler(this, OnItemDragLVFiles);

```

ДЛЯ ТОГО чтобы мы могли поместить перетаскиваемый мышью объект в дерево каталогов `TreeView`, свойство `AllowDrop` должно быть установлено равным `true`. Позже в приложение будет добавлен код, реализующий эту функциональность.

Событие `AfterSelect` наступает сразу же после того, как пользователь выбирает узел в дереве каталогов. Мы обрабатываем это событие, поскольку оно генерируется в самый подходящий момент для того, чтобы отобразить файлы, содержащиеся в папке. Как нетрудно догадаться, события `DragEnter`, `DragOver`, `DragLeave` и `DragDrop` добавлены для реализации функциональности `drag-and-drop`.

Свойство `MultiSelect` списка `ListView` позволяет этому списку одновременно содержать несколько выделенных элементов. Для упрощения примера мы присваиваем указанному свойству значение `false`, и пользователь будет иметь возможность выбрать лишь один элемент. Свойство `View` задает стиль списка `ListView`. Если установить значение данного свойства равным `View : Details`, на экран будут выводиться не только сами элементы, но и их подчиненные элементы.

Когда значение свойства `FullRowSelect` равно `true`, при выборе элемента в списке `ListView` будет выделяться вся строка (по умолчанию выделяется лишь первый элемент). В этом случае список `ListView` содержит три столбца: **File Name (Имя файла)**, **Created Date (Дата создания)** и **Modified Date (Дата изменения)**. Когда элемент перетаскивается мышью из списка `ListView`, наступает событие `ItemDrag`, которое мы также будем обрабатывать.

Для элементов управления `TreeView` и `ListView` мы *не задали* свойства `Location` и `Size`. Эти свойства будут устанавливаться в зависимости от размера и ширины формы, передаваемых в обработчик события `Resize`.

Следующий фрагмент кода создает элемент управления `StatusBar` с панелями:

```

// Создание элемента управления Status Bar
m_ptrStatusBar = new StatusBar;
m_ptrStatusBar->ShowPanels = true;

// Панель для отображения количества файлов
m_ptrSBPanel1 = new StatusBarPanel;
m_ptrSBPanel1->Text = S"";
m_ptrSBPanel1->Width = 200;
m_ptrSBPanel1->BorderStyle = StatusBarPanelBorderStyle::Sunken;
m_ptrStatusBar->Panels->Add(m_ptrSBPanel1);

// Панель для отображения сообщений
m_ptrSBPanel2 = new StatusBarPanel;
m_ptrSBPanel2->Text = S"";
m_ptrSBPanel2->BorderStyle = StatusBarPanelBorderStyle::Sunken;
m_ptrSBPanel2->AutoSize = StatusBarPanelAutoSize::Spring;
m_ptrStatusBar->Panels->Add(m_ptrSBPanel2);

```

Элемент управления `Status Bar` (Строка состояния) разделен на две панели. Вместо задания значения свойства `Width` для второй панели мы присваиваем свойству `AutoSize` перечислимую величину `Spring`. Это означает, что панель будет автоматически подстраивать свой размер, чтобы при изменении размеров формы она могла занять всю площадь, предназначенную для строки состояния.

Следующий раздел кода создает список `ImageList`, добавляет в список изображения, а сам список — в элемент управления `TreeView`:

```
// Создание элемента управления ImageList
m_ptrImageList = new ImageList;
m_ptrImageList->TransparentColor = Color::Transparent;
m_ptrImageList->ColorDepth = ColorDepth::Depth32Bit;

// Добавление изображений в ImageList
m_ptrImageList->Images->Add(Image::FromFile(S"MyComputer.bmp"));
m_ptrImageList->Images->Add(Image::FromFile(S"FloppyDrive.bmp"));
m_ptrImageList->Images->Add(Image::FromFile(S"Drive.bmp"));
m_ptrImageList->Images->Add(Image::FromFile(S"FolderClose.bmp"));
m_ptrImageList->Images->Add(Image::FromFile(S"FolderOpen.bmp"));
m_ptrTVFolders->ImageList = m_ptrImageList;
```

Мы почти закончили реализацию метода `CreateChildControls()`. Остается лишь создать всплывающие подсказки, задать местоположение различных элементов управления и, наконец, добавить их на родительскую форму. Второй из трех перечисленных этапов реализуется с помощью метода `ArrangeControlsO`, который также вызывается, когда пользователь изменяет размер формы.

В методе `ArrangeControlsO` мы задаем значение свойства `Width` элемента управления `TreeView` так, чтобы это значение составляло 1/3 от ширины клиентской области, а оставшиеся 2/3 будут заняты элементом управления `ListView`. Кроме того, высота дерева каталогов и списка `ListView` уменьшена на 26 пикселей, чтобы в нижней части окна можно было разместить строку состояния. Для улучшения внешнего вида графического интерфейса пользователя мы создаем некоторый промежуток между элементами управления `TreeView` и `ListView`, добавив 5 пикселей к координате `X` дерева каталогов `TreeView`:

```
// Функция, задающая расположение элементов управления при
// изменении размеров формы
void ExplorerForm::ArrangeControls()
{
    Point l_pt(2, 6);

    int l_nHeight = ClientSize.Height;
    int l_nWidth = ClientSize.Width;

    // Задаем ширину элемента управления, равную 1/3 от ширины формы
    int l_nTreeViewWidth = l_nWidth / 3;

    // Создание элемента управления TreeView
    m_ptrTVFolders->Location = l_pt;
    System::Drawing::Size l_sizeTreeView(l_nTreeViewWidth, l_nHeight - 26);
    m_ptrTVFolders->Size = l_sizeTreeView;
    l_pt.X = l_nTreeViewWidth + 5;
    l_pt.Y = 6;

    // Создание элемента управления ListView
    m_ptrLVFiles->Location = l_pt;
    System::Drawing::Size l_sizeListView(
        l_nWidth - (l_nTreeViewWidth + 5), l_nHeight - 26);
    m_ptrLVFiles->Size = l_sizeListView;
    return;
}
```

В последнюю очередь конструктор класса `ExplorerForm` вызывает метод `PopulateDrives()`, который добавляет в элемент управления `TreeView` имена всех дисков,

имеющихся на текущей машине. Если элемент управления уже содержит узлы, все они удаляются до того, как создается корневой узел с помощью класса `TreeNode`. В нашем приложении элемент управления `TreeView` имеет корневой узел `MyComputer`, куда добавляются имена всех дисков как подчиненные узлы.

Конструктор класса `TreeNode` принимает метку узла, индекс изображения и индекс выбранного изображения. Первый из двух индексов относится к изображению, которое выводится на экран, когда узел находится в нормальном состоянии. Второй индекс соответствует изображению выбранного узла. Начальным значением для этих индексов является нуль.

Для добавления в корневой узел подчиненных узлов нужно иметь указатель `TreeNodeCollection*` на коллекцию корневой узла и вызвать ее метод `Add()`. В нашем коде это коллекция `l_ptrCollection`, которая инициализируется сразу же после добавления корневой узла в дерево `TreeView`. Затем функция `PopulateDrives()` продолжает добавлять имена дисков в элемент управления, делая их подчиненными по отношению к корневому узлу. Для этого используется цикл `for` и функция `Environment::GetLogicalDrives()`, которая возвращает список дисков в виде массива типа `String*`. Для каждого узла в дереве каталогов `TreeView` необходимо создавать новый объект `TreeNode`:

```
void ExplorerForm::PopulateDrives()
{
    try
    {
        // Удаляем элементы, если они уже имеются в дереве
        if(m_ptrTVFolders->Nodes->Count > 0)
        {
            m_ptrTVFolders->Nodes->Clear();
        }
        m_ptrRootNode = new TreeNode(S"My Computer", 0, 0);
        m_ptrTVFolders->Nodes->Add(m_ptrRootNode);
        TreeNodeCollection* l_ptrCollection = m_ptrRootNode->Nodes;
        // Отображение дисков
        String* l_ptrDrives[] = Environment::GetLogicalDrives();
        for(int l_nCount = 0; l_nCount < l_ptrDrives->Length; l_nCount++)
        {
            String l_strDrive = l_ptrDrives[l_nCount];
            if(String::Equals(l_strDrive, S"A:\\"))
            {
                TreeNode* l_ptrCurrNode = new TreeNode(l_strDrive, 1, 1);
                l_ptrCollection->Add(l_ptrCurrNode);
                continue;
            }

            // Добавляем подчиненный узел
            TreeNode* l_ptrCurrNode = new TreeNode(l_strDrive, 2, 2);
            l_ptrCollection->Add(l_ptrCurrNode);
        }
    }
    catch(Exception* l_ptrException)
    {
        MessageBox::Show(l_ptrException->Message);
    }
}
```

Этот код передает узлу `MyComputer` нулевые значения для индексов регулярного и выделенного изображения. Затем диску `A:` предаются значения индексов,

равные 1, а остальным дискам — значения 2; последние соответствуют графическому файлу Drive.bmp. Если возникают какие-либо ошибки выполнения, выдается исключительная ситуация, при обработке которой ее свойство `Message` передается в метод `MessageBox::Show()`.

На этом завершается процесс отображения дисков в элементе управления `TreeView`. Следующими этапами являются добавление в дерево каталогов папок, относящихся к текущему диску, и отображение соответствующих файлов в списке `ListView`, когда выбирается узел в элементе управления `TreeView`. Для этого мы используем событие `AfterSelect`.

В обработчике события `AfterSelect` необходимо выполнить некоторую начальную проверку данных. Если пользователь щелкнул мышью на корневом узле, то при этом нет необходимости отображать какие-либо файлы в списке `ListView`, а нужно очистить список от всех файлов, которые там имеются. Если же пользователь щелкает на диске, то нужно заново добавить в дерево каталогов `TreeView` папки, относящиеся к этому диску, и отобразить все файлы выбранного диска в списке `ListView`. Когда пользователь щелкает на папке, следует вывести все файлы, содержащиеся в этой папке, в список `ListView`:

```
void ExplorerForm::OnAfterSelectTVFolders(Object* l_ptrSender,
                                         TreeViewEventArgs* l_ptrEventArgs)
{
    try
    {
        TreeNode* l_ptrCurrNode = l_ptrEventArgs->Node;
        if(l_ptrCurrNode != m_ptrRootNode)
        {
            // Очищаем все элементы списка, ListView, если выбирается
            // корневой узел
            m_ptrLVFiles->Items->Clear();
            return;
        }

        Cursor = Cursors::WaitCursor;

        // Если нет подчиненных узлов, добавляем папку
        // Обновляем папки только для узлов, представляющих диски
        if(l_ptrCurrNode->Nodes->Count == 0 &&
           (l_ptrCurrNode->ImageIndex == 1 || l_ptrCurrNode->ImageIndex == 2))
        {
            m_ptrSBPanel2->Text = String::Concat(
                S"Refreshing Drive '" + l_ptrCurrNode->Text +
                S"' Please wait...");
            PopulateFolders(l_ptrCurrNode->Text, l_ptrCurrNode->Nodes);
        }

        // Добавим файлы, относящиеся к выбранной папке
        int l_iFileCount = PopulateFiles(l_ptrCurrNode->FullPath);
        String l_strText = String::Concat(S"File(s) " +
                                           l_iFileCount.ToString());

        Cursor = Cursors::Arrow;
        m_ptrSBPanel1->Text = l_strText;
        m_ptrSBPanel2->Text = S"";

        // Сохраним текущий узел
        m_ptrSourceNode = l_ptrCurrNode;
    }
}
```



```

        catch(Exception* l_ptrException)
        {
            Cursor = Cursors::Arrow;
            String* l_strError = String::Concat(
                S"Error occurred while refreshing files. ",
                l_ptrException->Message);
            MessageBox::Show(l_strError, S"Managed C++ - Explorer",
                MessageBoxButtons::OK, MessageBoxIcon::Error);
            l_ptrException->Finalize();
        }
        return;
    }
}

```

ЕСЛИ пользователь щелкает мышью на корневом узле, элементы списка `ListView` удаляются и выполнение функции завершается. Второй условный оператор `if` определяет с помощью свойства `ImageIndex`, является ли выбранный узел диском. Если это так, вызывается метод `PopulateFolders()`:

```

void ExplorerForm::PopulateFolders(
    String* l_strCurrDir, TreeNodeCollection* l_ptrCollection)
{
    try
    {
        String* l_strFolders[] = Directory::GetDirectories(l_strCurrDir);
        for(int l_nCount = 0; l_nCount < l_strFolders->Length; l_nCount++)
        {
            String* l_strFilePath = l_strFolders->Item[l_nCount]->ToString();
            String* l_strFileName = GetFileNameFromPath(l_strFilePath);
            TreeNode* l_ptrTreeNode = new TreeNode(l_strFileName, 3, 4);
            l_ptrCollection->Add(l_ptrTreeNode);
            PopulateFolders(l_strFilePath, l_ptrTreeNode->Nodes);
        }
    }
    catch(Exception* l_ptrException)
    {
        String* l_strError = String::Concat(
            S"Error occurred while refreshing folder; ",
            l_ptrException->Message);
        MessageBox::Show(l_strError, S"Managed C++ - Explorer",
            MessageBoxButtons::OK, MessageBoxIcon::Error);
        l_ptrException->Finalize();
    }
    return;
}

```

Статический метод `Directory::GetDirectories()` принимает в качестве параметра путь к каталогу и возвращает массив строковых указателей, в котором каждый элемент содержит имя каталога. Цикл `for` добавляет эти каталоги в текущий узел, куда передается имя каталога, а также значения 3 и 4, представляющие собой соответственно индексы регулярного и выделенного изображений. Затем метод `PopulateFolders()` вызывается рекурсивно, причем в качестве аргументов ему передаются путь к подчиненному каталогу и указатель `TreeNodeCollection*`. Этот процесс продолжается до тех пор, пока все каталоги не будут добавлены в элемент управления `TreeView`.

В аргумент функции `GetFileNameFromPath()` передается полный путь, после чего функция возвращает заключительную часть строки, ограниченную обратной косой чертой. Например, если передать строку `C:\Niranjana\Sairam.exe`, указанная функция выдаст имя файла `Sairam.exe`. Символ ограничителя задается свойством `PathSeparator` класса `TreeView`, и по умолчанию этот символ представляет собой обратную косую черту.

Вернемся к обработчику события AfterSelect. Когда пользователь выбирает папку в дереве каталогов, условие во втором операторе if не выполняется и вызывается метод PopulateFiles(). Этот метод принимает полный путь текущего узла и отображает в элементе управления ListView файлы, содержащиеся в узле. Метод также возвращает количество найденных файлов, которое выводится на первой панели строки состояния (StatusBar). Затем в методе PopulateFiles() указателю m_ptrSourceNode присваивается значение, соответствующее текущему узлу. Позже мы воспользуемся этим обстоятельством при реализации обработчика события DragDrop, связанного с элементом управления TreeView.

Добавить файлы в список ListView очень просто. Статический метод Directory::GetFiles() принимает имя каталога и возвращает список файлов в виде массива указателей. Мы используем цикл for для перебора всех элементов и включаем в список ListView имена файлов, а также даты их создания и изменения. Для выполнения последней операции необходимо создать объект ListViewItem, конструктор которого принимает массив типа String', содержащий различные поля элементов. После создания указанного объекта он передается в метод Add() коллекции Items, принадлежащей к списку ListView. Этот метод добавляет элемент в список:

```
int ExplorerForm::PopulateFiles(String* l_strCurrDir)
{
    int l_nFileCount = 0;

    try
    {
        String* l_strSubItems[] = {S"", S"", S"", S""};
        String* l_strCreationTime = S"";
        DateTime l_dtCreatedTime;
        DateTime l_dtModifiedTime;
        String* l_strModifiedTime = S"";
        m_ptrLVFiles->Items->Clear();

        // Удаляем строку "My Computer" из текущего пути
        String* l_strCurPath = l_strCurrDir->Remove(0, 12);
        String* l_strFiles[] = Directory::GetFiles<l_strCurPath>;
        for( l_nFileCount < l_strFiles->Length; l_nFileCount++)
        {
            l_strSubItems[0] = GetFileNameFromPath(l_strFiles[l_nFileCount]);
            l_dtCreatedTime = File::GetCreationTime(l_strFiles[l_nFileCount]);
            l_dtModifiedTime = File::GetLastWriteTime(l_strFiles[l_nFileCount]);
            l_strSubItems[1] = String::Concat(l_dtCreatedTime.ToShortDateString(),
                                              S" ", l_dtCreatedTime.ToShortTimeString());
            l_strSubItems[2] = String::Concat(l_dtModifiedTime.ToShortDateString(),
                                              S" ", l_dtModifiedTime.ToShortTimeString());
            ListViewItem* l_ptrLVItem = new ListViewItem(l_strSubItems);
            m_ptrLVFiles->Items->Add(l_ptrLVItem);
        }
    }
    catch(Exception* l_ptrException)
    {
        Cursor = Cursors::Arrow;
        String l_strError = String::Concat(
            S"Error occurred while refreshing files: ", l_ptrException->Message);
        MessageBox::Show(l_strError, S"Managed C++ - Explorer",
            MessageBoxButtons::OK, MessageBoxIcon::Error);
        l_ptrException->Finalize();
    }

    return l_nFileCount;
}
```

Перед вызовом метода `Directory.GetFiles()` мы обращаемся к методу `String.Remove()` для удаления из текущего узла первых 12 символов (которые представляют собой не что иное, как строку "My Computer\").

Реализация операции drag-and-drop в формах Windows

Наконец обсудим основные принципы операции drag-and-drop и ее реализацию в формах Windows на примере элементов управления `TreeView` и `ListView`. Список `ListView` можно рассматривать как источник, откуда перетаскиваются файлы, а дерево каталогов `TreeView` — как целевой элемент управления, куда эти файлы помещаются. Мы уже сделали определенные заготовки, необходимые для реализации механизма drag-and-drop. Так, значение свойства `AllowDrop` элемента управления `TreeView` установлено равным `true`, а также добавлены обработчики для всех соответствующих событий. Теперь необходимо реализовать эти обработчики. Процесс реализации механизма drag-and-drop подразделяется на следующие этапы:

1. Установка значения `true` для свойства `AllowDrop`, относящегося к целевому элементу управления.
2. Обработка события `ItemDrag` для элемента управления, являющегося источником, чтобы разрешить перетаскивание объекта из него.
3. Вызов метода `DoDragDrop()` и установка эффектов перетаскивания, которые будут проявляться, когда перемещаемый объект находится над целевым элементом управления и генерируется событие `ItemDrag`.
4. Обработка событий `DragEnter`, `DragOver`, `DragLeave` и `DragDrop` для целевого элемента управления

Ниже приведена последовательность событий, происходящих при перетаскивании объекта из источника и помещения его в целевой элемент управления:

1. Событие `ItemDrag` в элементе управления, который является источником: наступает, когда пользователь перетаскивает объект из источника.
2. Событие `DragEnter` в целевом элементе управления: генерируется при вхождении курсора мыши в целевой элемент управления.
3. Событие `DragOver` в целевом элементе управления: происходит, когда курсор, перемещающий объект, движется над целевым элементом управления.
4. Событие `DragDrop` в целевом элементе управления: наступает, когда пользователь помещает объект в целевой элемент управления.

Если пользователь перемещает объект над целевым элементом управления, однако потом отводит курсор в сторону, так и не оставив объект в целевом элементе, наступает событие `DragLeave`.

Познакомившись с основами операции drag-and-drop, обратимся к соответствующему коду. В нашем приложении необходимо сначала реализовать обработку события `ItemDrag` элемента управления `ListView`. Возвращаясь к обсуждению метода `ExplorerForm.CreateChildControls()`, напомним, что данный метод содержит следующую строку кода:

```
m_ptrLVFiles->ItemDrag += new ItemDragEventHandler(this, OnItemDragLVFiles);
```

Первый аргумент, передаваемый в обработчик, представляет собой указатель на объект, который инициирует событие, а второй — указатель на объект типа

ItemDragEventArgs. Свойство Item объекта ItemDragEventArgs возвращает указатель на перетаскиваемый объект. Мы приводим этот указатель к типу ListViewItem и сохраняем результат в переменной m_ptrDragLViewItem. Далее вызывается метод DoDragDrop(), которому в качестве параметров передаются указатель ListViewItem и значение DragDropEffects::Copy:

```
// Обработчик события ListView -> ItemDrag
void ExplorerForm::OnItemDragLVFiles(
    Object* l_ptrSender, ItemDragEventArgs* l_ptrArgs)
{
    m_ptrDragLViewItem = static_cast<ListViewItem*>(l_ptrArgs->Item);
    DoDragDrop(m_ptrDragLViewItem, DragDropEffects::Copy);
    return;
}
```

Этот код позволяет перетаскивать объект из списка ListView. Теперь необходимо обработать события перетаскивания, которые мы добавили ранее в элемент управления Treeview. Перечислим соответствующие обработчики событий:

- OnDragEnterTVFolders(). Здесь необходимо задать эффекты перетаскивания в курсоре мыши.
- OnDragOverTVFolders(). Этот обработчик событий должен выделить текущий узел, изменив цвета фона и изображения.
- OnDragLeaveTVFolders(). Данный метод должен переводить элемент управления в нормальное состояние.
- OnDragDropTVFolders(). Здесь идентифицируется перетаскиваемый объект, определяется узел, куда этот объект должен помещаться, и выполняется копирование файла в целевую папку.

Реализация метода OnDragEnterTVFolders() весьма проста. Мы соответствующим образом устанавливаем свойство Effect объекта DragEventArgs, благодаря чему изменится форма указателя мыши. Необходимо использовать одну и ту же величину DragDropEffects во всех функциях, иначе операция drag-and-drop не будет работать:

```
// Обработчик события Treeview -> DragEnter
void ExplorerForm::OnDragEnterTVFolders(
    Object* l_ptrSender, DragEventArgs* l_ptrArgs)
{
    l_ptrArgs->Effect = DragDropEffects::Copy;
}
```

Когда объект перемещается над элементом управления TreeView, наступает событие DragOver. Второй параметр обработчика этого события также имеет тип DragEventArgs*. Данный обработчик должен выделить цветом текущий узел, когда пользователь двигает курсор над деревом каталогов TreeView, что является достаточно сложным процессом.

Объект DragEventArgs позволяет определить текущие координаты курсора мыши, которые затем передаются во вспомогательную функцию FindTreeNode(). Эта функция возвращает указатель на текущий узел. Если координаты не соответствуют ни одному из узлов, возвращается нулевой указатель. Когда функция FindTreeNode() выдает действительный узел, отличный от корневого узла, свойства BackColor и ForeColor изменяются и получают соответственно значения DarkBlue и White. В этот же момент указатель на предварительно выбранный узел сохраняется в переменной m_ptrOldNode, а сам узел приобретает первоначальный цвет:

```

void ExplorerForm::OnDragOverTVFolders(Object* l_ptrSender,
                                       DragEventArgs* l_ptrArgs)
{
    // Получение текущего узла
    TreeNode* l_ptrCurrNode = FindTreeNode(l_ptrArgs->X, l_ptrArgs->Y);
    if(l_ptrCurrNode)
    {
        // Если имеется действительный узел
        if(l_ptrCurrNode->ImageIndex > 0)
        {
            // Если это не корневой узел
            l_ptrCurrNode->BackColor = Color::DarkBlue;
            l_ptrCurrNode->ForeColor = Color::White;
            if(m_ptrOldNode && (m_ptrOldNode != l_ptrCurrNode))
            {
                // Возвращаем первоначальный цвет предыдущему узлу:
                m_ptrOldNode->BackColor = m_ptrRootNode->BackColor;
                m_ptrOldNode->ForeColor = m_ptrRootNode->ForeColor;
            }
            m_ptrOldNode = l_ptrCurrNode;
        }
    }
}

```

Реализация метода FindTreeNode() принимает координаты курсора мыши и возвращает указатель на узел. В этой функции метод TreeNode::Bounds::Contains() принимает объект типа Point и возвращает значение true, если точка попадает на узел. В цикле while функция FindTreeNode() выполняет поиск, начиная с корневого узла и заканчивая последним доступным узлом. Если узел удается найти, функция возвращает указатель типа TreeNode*. В противном случае возвращается нулевой указатель:

```

TreeNode* ExplorerForm::FindTreeNode(int l_iXPos, int l_iYPos)
{
    TreeNode* l_ptrCurrNode = m_ptrRootNode;
    Point l_pt(l_iXPos, l_iYPos);
    l_pt = PointToClient(l_pt);
    while(l_ptrCurrNode)
    {
        if(l_ptrCurrNode->Bounds.Contains(l_pt))
        {
            // Если узел обнаружен, возвращается это значение
            return l_ptrCurrNode;
        }
        l_ptrCurrNode = l_ptrCurrNode->NextVisibleNode;
    }
    // Если узел не найден, возвращается нулевой указатель
    return 0;
}

```

Когда пользователь убирает курсор мыши с элемента управления TreeView, не оставляя там объект, необходимо вернуть прежние значения свойствам BackColor и ForeColor для последнего измененного узла. Это делается в ответ на событие DragLeave. Кроме того, обработчик данного события присваивает нулевой указатель переменной m_ptrOldNode:

```

void ExplorerForm::OnDragLeaveTVFolders(Object* l_ptrSender,
                                       EventArgs* l_ptrArgs)
{
    if(m_ptrOldNode)
    {

```

```

if(m_ptrOldNode->ImageIndex > 0)
{
    m_ptrOldNode->BackColor = m_ptrRootNode->BackColor;
    m_ptrOldNode->ForeColor = m_ptrRootNode->ForeColor;
}
m_ptrOldNode = 0;
}

```

Осталось обработать лишь событие `DragDrop`, для чего вновь можно использовать вспомогательную функцию `FindTreeNode()`. Если узел, в который должен помещаться объект, является действительным и не корневым, определяется свойство объекта `FullPath`, которое возвращает папку назначения. Исходная папка содержится в переменной `m_ptrSourceNode`, которая инициализируется в обработчике события `AfterSelect` элемента управления `TreeView`, а имя исходного файла можно получить из свойства `m_ptrDragLVItem->Text`. На основе полученных данных формируются полные пути к папкам источника и назначения, а затем вызывается метод `File::Copy()` для копирования файла. Наконец, с помощью свойства `SelectedNode` выделяется текущий узел (куда был помещен перетаскиваемый объект) и элементу управления `TreeView` передается фокус:

```

void ExplorerForm::OnDragDropTVFolders(Object* l_ptrSender,
                                         DragEventArgs* l_ptrArgs)
{
    try
    {
        // Поиск текущего узла, куда будет помещаться объект
        TreeNode* l_ptrCurrNode = FindTreeNode(l_ptrArgs->X, l_ptrArgs->Y);
        if(l_ptrCurrNode)
        {
            // Изменение цвета текущего узла
            if(l_ptrCurrNode->ImageIndex > 0)
            {
                l_ptrCurrNode->BackColor = m_ptrRootNode->BackColor;
                l_ptrCurrNode->ForeColor = m_ptrRootNode->ForeColor;
            }
            m_ptrOldNode = 0;
        }
        else
        {
            // Если возвращается нулевое значение
            return;
        }
        // Если сделана попытка оставить объект в корневом узле
        if(l_ptrCurrNode == m_ptrRootNode)
        {
            m_ptrTVFolders->SelectedNode = m_ptrRootNode;
            m_ptrTVFolders->Focus();
            m_ptrDragLVItem = 0;
            MessageBox::Show(
                S"Cannot copy the dragged item to 'My Computer' node",
                S"Managed C++ - Explorer", MessageBoxButtons::OK,
                MessageBoxIcon::Information);
            return;
        }
        // Копирование перемещаемого файла в текущую папку
        Cursor = Cursors::WaitCursor;
        String* l_strTargetDir = l_ptrCurrNode->FullPath;
        String* l_strSourceFullPath = String::Concat(

```

```

        m_ptrSourceNode->FullPath, S"\\", m_ptrDragLVItem->Text);
    1_strTargetDir = 1_strTargetDir->Remove(0, 12);
    1_strSourceFullPath = 1_strSourceFullPath->Remove(0, 12);
    String* 1_strSourceFile = GetFileNameFromPath(1_strSourceFullPath);
    String* 1_strTargetFullPath = String::Concat(
        1_strTargetDir, S"\\", 1_strSourceFile);

    // Копирование файла
    File::Copy(1_strSourceFullPath, 1_strTargetFullPath);

    // Изменение папки назначения
    m_ptrTVFolders->SelectedNode = 1_ptrCurrNode;
    m_ptrTVFolders->Focus();

    // Присваиваем указателю m_ptrDragLVItem нулевое значение
    m_ptrDragLVItem = 0;
    Cursor := Cursors::Arrow;
}
catch(Exception* 1_ptrException)
{
    Cursor := Cursors::Arrow;
    String* 1_strError = String::Concat(
        S"Error occurred while copying file: ", 1_ptrException->Message);
    MessageBox::Show(1_strError, S"Managed C++ - Explorer",
        MessageBoxButtons::OK, MessageBoxIcon::Error);
    1_ptrException->Finalize();
}
}
}

```

Наконец, но не в последнюю очередь, нужно создать форму в функции приложения `main()`. Поскольку механизм `drag-and-drop` относится к функциональности OLE, для его активизации необходимо вызвать статический метод `Application::OleRequired()`. Этот метод инициализирует OLE в текущем потоке. Кроме того, для текущего потока должен быть задан тип STA с помощью метода `Thread::set_ApartmentState()`.

```

int _tmain(void)
{
    try
    {
        Application::OleRequired();
        Application::Run(new ExplorerForm);
    }
    catch(Exception* 1_ptrException)
    {
        String* 1_strText = String::Concat(
            S"Error in creating form: ", 1_ptrException->Message);
        MessageBox::Show(1_strText, S"Managed C++ - Explorer",
            MessageBoxButtons::OK, MessageBoxIcon::Stop);
    }
    return 0;
}

```

На этом завершается создание приложения, которое можно использовать для просмотра локального жесткого диска. Это приложение позволяет копировать файлы, перетаскивая их из списка `ListView` в элемент управления `TreeView`. Для простоты приложение поддерживает только функцию копирования, однако можно легко расширить его, реализовав операции перемещения, удаления, открытия файлов (когда пользователь дважды щелкает мышью на элементе в списке `ListView`) и т.д.

ГЛАВА 7



Управляемый и неуправляемый код

Возможно, было бы неплохо забыть о миллионах строк кода C++, которые были написаны еще до появления первых слухов о технологии .NET, однако этот неуправляемый код присутствует везде: от настольных приложений до операционных систем. Наши читатели почти наверняка сохраняют код в исходных файлах C++, статических библиотеках и DLL. Данный код отлажен и прекрасно работает. Спасибо за это его создателям! Но при переходе на технологию .NET перед многими программистами C++ встает ключевой вопрос, нужно ли трансформировать существующий код в управляемую форму, оставить ли его неуправляемым или совмещать код обоих типов.

Как и следовало ожидать, Microsoft реализовала средства взаимодействия, которые позволяют связать между собой два мира. Ведь в конце концов сама операционная система Windows, ее системные библиотеки DLL и функции Win32 API представляют собой машинный код, и такая ситуация вряд ли изменится в ближайшем будущем. Программисты C++ имеют уникальные возможности, поскольку C++ в настоящее время является единственным .NET-совместимым языком, позволяющим создавать код и данные как управляемого, так и неуправляемого типа. Настоящая глава посвящена следующим вопросам:

- Интеграция управляемого и неуправляемого кода. Мы рассмотрим компиляцию неуправляемого кода C++ с флагом `/clr`, применение механизма Microsoft "It Just Works" (IJW), а также использование новых директив `#pragma` для управления компиляцией различных секций исходного кода.
- Создание прокси-классов (или оболочек) вокруг существующего неуправляемого кода C++, обеспечивающих доступ к нему со стороны управляемого кода. Мы обсудим конкретные вопросы: работу с конструкторами, деструкторами, функциями-членами и перегруженными операциями.

- а Работа механизма, связывающего управляемый и неуправляемый мир. Будет рассмотрен маршalling между управляемым и неуправляемым кодом. Мы покажем, когда необходимо использовать маршalling, и обсудим применение класса `System::Runtime::InteropServices::Marshal`. Мы также подробно познакомимся со службами Platform Invocation Services (PInvoke), их назначением и использованием.

Прочитав эту главу, разработчики C++ получают полное представление о том, что такое управляемый и неуправляемый код, в чем заключается различие между двумя типами кода, как их можно совместно использовать и каким образом они связываются друг с другом при создании приложений.

Специальные средства, разработанные Microsoft для взаимодействия с COM, являются предметом следующей главы.

Соединение управляемого и неуправляемого кода

Решение перейти от неуправляемого кода к управляемому не дается легко. Из-за дополнительных возможностей, реализованных в среде CLR, использование управляемого кода может привести к снижению производительности и гибкости приложений, и не во всех ситуациях оно является оправданным. Прежде чем приступить к обсуждению процессов преобразования, вполне уместно перечислить причины, по которым разработчики могут отказаться от написания управляемого кода:

- о Неуправляемый код почти всегда быстрее, поскольку его применение не связано с издержками, возникающими при работе служб CLR, таких как сборка мусора и отслеживание ссылок. Для многих приложений (например, интерфейс пользователя) это несущественно. Однако в других случаях (создание системных утилит, инструментов, игр и т.д.) указанное обстоятельство может играть важную роль.
- Неуправляемый код позволяет непосредственно вызывать функции Win32 API, а также существующий код C++ без дополнительных затрат, связанных с переходом от управляемого кода к неуправляемому.
- Неуправляемый код более "детерминирован", чем управляемый. Например, нельзя предсказать, когда будет запущен сборщик мусора, и это может вызвать проблемы при работе с неуправляемыми ресурсами, такими как соединения с базами данных, файлы и коммуникационные порты. Если нужно точно знать, когда происходит тот или иной процесс, — это веская причина для написания неуправляемого кода.

Более того, необходимо учитывать и некоторые специфические ограничения, свойственные управляемым классам, а именно:

- Управляемые классы могут быть производными только от других управляемых классов. Равным образом класс, производный от управляемого типа, всегда является управляемым.
- Такие классы способны наследовать только от одного класса (хотя платформа .NET поддерживает наследование от многих интерфейсов). Если программист имеет дело с кодом, который в значительной степени использует множественное наследование, и хочет в дальнейшем придерживаться той же стратегии, то ему придется работать с неуправляемым кодом.

- Управляемые классы не могут иметь "дружественных" функций или классов.
- Управляемые классы не поддерживают перегрузку операторов `new` или `delete`.
- Эти классы нельзя использовать с операторами `sizeof()` и `offsetof()`.
- Управляемый класс не может быть шаблоном. Платформа .NET не поддерживает родовое программирование или шаблоны в своей первой версии (хотя ходят слухи, что такая поддержка будет реализована в версии 2). Если разработчик хочет писать программы на основе шаблонов, он должен делать это средствами неуправляемого кода C++.
- Управляемый класс может содержать ссылки на другие управляемые типы, однако, если необходимо, чтобы такой класс содержал полные ссылки на , неуправляемые типы, здесь возникают существенные ограничения, поскольку типы должны быть очень простыми. В этом случае можно использовать лишь POD-типы (т.е. типы Plain Old Data, указанные в стандарте C++).

Флаг /clr и механизм It Just Works (IJW)

Предположим, что после анализа всех перечисленных обстоятельств мы все же решили воспользоваться возможностями .NET и приступить к написанию управляемого кода C++. Такое решение не препятствует тому, чтобы часть созданного нами ранее кода осталась неуправляемой. Правильное применение этой методики избавит нас от необходимости переписывать многие строки кода в существующих приложениях.

Простейший способ начать переход к использованию управляемого кода заключается в перекомпиляции неуправляемого кода с флагом `/clr`. Microsoft называет эту технологию **It Just Works (IJW)** — "Этот код работает!". Согласно формулировке Microsoft, "флаг `/clr` позволяет получить доступ к управляемым расширениям и сделать код управляемым". Но столь неопределенное описание мало говорит о том, что же происходит в действительности и почему этот флаг так важен для интеграции с управляемым кодом. Фактически, средство IJW — большой успех команды Visual C++ .NET: можно взять приложение C++, созданное на основе MFC, добавить к нему шаблоны, контейнеры STL, использовать множественное наследование, затем выполнить компиляцию с флагом `/clr`, и окажется, что "этот код работает!".

Однако в первую очередь здесь следует иметь в виду, что ключ `/clr` не может чудесным образом преобразовать весь неуправляемый код в управляемый. Проблемы, перечисленные в предыдущем разделе, не решаются так просто. В действительности же флаг `/clr` изменяет цель компиляции. При использовании обычного компилятора в результате генерируется собственный машинный код на языке ассемблера. В случае же компиляции с флагом `/clr` генерируется код MSIL, представляющий собой сборку. Если выбран второй вариант компиляции, все службы CLR (например, средства безопасности .NET) становятся доступными для функций, а типы POD преобразуются в управляемые типы-значения — и это все, что происходит в результате.

Ограничения IJW

Если не внести изменения в существующий код, память для любых объектов будет выделяться в неуправляемой куче и среда CLR будет воспринимать эти объекты как прозрачные блоки двоичных данных. Только когда типы помечены спецификаторами `__ds` или `__value`, они становятся управляемыми данными, и именно это является причиной некоторых ограничений в том списке, который представлен ниже:

- Нельзя автоматически перекомпилировать все библиотеки классов, а затем реализовать наследование от них в управляемой среде.
- Указатели управляемого кода не могут быть просто переданы в ваши функции и классы.
- Использование встроенного ассемблера проблематично, поскольку при этом вам должны быть известны расположение стека на конкретной машине, соглашения о вызовах за пределами текущей функции или некоторая другая низкоуровневая информация о компьютере. Код может не работать, если эта информация будет применяться по отношению к стековому фрейму для управляемой функции. Функции, содержащие код на встроенном ассемблере, генерируются как неуправляемые.
- Функции `vprintf()` не могут вызываться из программы, откомпилированной с флагом `/clr`.
- Использование классов с атрибутами `dllexport` или `dllimport` не разрешается, если компиляция выполнена с флагом `/clr`.
- Если код компилируется с флагом `/clr`, не разрешается использование функций, не имеющих полного прототипа.

Некоторые из этих ограничений, например связанные с применением встроенного ассемблера, не обсуждаются настоящей книге, но мы коснемся более общих проблем. Механизм IJW позволяет совершенно свободно передавать ссылки в случае, если код написан в стиле языка C, до тех пор, пока этот код остается неуправляемым. Однако если попытаться передать управляемый тип в неуправляемую функцию, немедленно возникнут проблемы. Обратимся к такому фрагменту кода:

```
#include "stdio.h"

using <mscorlib.dll>
using namespace System;

// Здесь создается управляемый тип данных (структура)
__managed struct ManagedStruct
{
    int x;
    int y;
};

void swap1(int* px, int* py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}

int main()
{
    ManagedStruct* ps = new ManagedStruct;
    ps->x = 1;
    ps->y = 2;

    swap1(&ps->x, &ps->y);
    printf("x = %d y = %d\n", ps->x, ps->y);
    return 0;
}
```

Эта программа не будет откомпилирована, и мы получим следующее сообщение об ошибке:

```
error c2664: 'swap1'
cannot convert parameter 1 from 'int__gc *__w64' to 'int *'
(невозможно преобразовать параметр 1 типа 'int__gc *__w64' к типу 'int *').
```

Мы попытались передать управляемые указатели (адреса членов структуры) в функцию, параметры которой представляют собой неуправляемые указатели, и компилятор не позволил этого сделать. Даже если бы компилятор разрешил такую операцию, код не был бы безопасным. Дело в том, что в процессе уплотнения управляемой кучи сборщиком мусора объекты будут перемещаться и ссылки на соответствующие участки памяти изменятся.

Если **НУЖНО** решить эту проблему так, чтобы не переписывать неуправляемую функцию `swap1()`, следует использовать ключевое слово `__pin` (см. ниже).

Директивы `#pragma managed` и `#pragma unmanaged`

Как отмечалось выше, в результате компиляции программы без использования ключа `/clr` генерируется неуправляемый машинный код, а компиляция с этим ключом приводит к созданию кода MSIL. Таким образом, можно задать режим компиляции на уровне модулей. Однако иногда требуется более детальный контроль над механизмом JIT. Для этой цели используются две новые директивы `pragma`, которые позволяют явно смешивать в одном модуле управляемый и неуправляемый код:

```
• // файл mixed.cpp
// ...управляемый код по умолчанию...

#pragma unmanaged
// .. неуправляемый код...

#pragma managed
// ...управляемый код...
```

Эти указания компилятору особенно полезны в процессе миграции обычного кода C++ в управляемый код. Они позволяют переносить наш код последовательно без тех затрат, которые могут быть связаны с трансформацией всего приложения сразу. Автор оценил всю пользу рассматриваемых директив, которые помогли ему выполнить поэтапное преобразование большого объема кода C++, созданного на основе STL, для того, чтобы данный код мог работать в управляемой среде.

Отметим, что, если флаг `/clr` отсутствует, эти директивы полностью игнорируются и по умолчанию генерируется неуправляемый код.

Ключевое слово `__pin`

В управляемом коде ключевое слово `__pin` объявляет указатель на объект (или на встроенный объект управляемого класса) и не позволяет перемещать данный объект во время сборки мусора. Это особенно полезно, когда необходимо передать адрес управляемого типа в неуправляемую функцию, поскольку таким образом исключаются неожиданные изменения адреса в процессе вызова указанной функции.

Следующая строка кода иллюстрирует, как выполняется фиксация (pinning) указателя `pinnedObject` на управляемый объект:

```
ClivesOnManagedHeap __pin* pinnedObject = pManagedHeapObject;
```

Необходимо иметь в виду, что такая операция связана с определенными издержками. Зафиксированный объект может сильно снизить эффективность сжатия управляемой кучи, а также увеличить время, требуемое для выделения памяти и сборки мусора. Если в куче зафиксировать каждый пятый управляемый объект, выделение памяти объектам и процедура очистки потребуют в 25 раз больше времени, чем в случае, когда ключевое слово `__pin` не используется.

Сборщик мусора разработан таким образом, что он должен устранять фрагментацию кучи, и фиксация указателей существенно уменьшает его производительность. Поэтому важно свести до минимума то время, в течение которого указатели остаются зафиксированными.

Если более не нужно фиксировать объект, можно открепить его, присвоив прикрепляемому указателю нулевое значение или сделав так, чтобы этот указатель вышел за пределы области видимости:

```
pinnedObject = 0;
```

Вернемся к нашему простому примеру и зафиксируем управляемые указатели, прежде чем передавать их в неуправляемую функцию. Это послужит гарантией того, что объекты будут "закреплены" в памяти. Теперь функция `main()` примет вид:

```
int main()
{
    ManagedStruct* ps = new ManagedStruct;
    ps->x = 2;
    ps->y = 4;
    int __pin* pinnedX = &ps->x;
    int __pin* pinnedY = &ps->y;

    swap1(pinnedX, pinnedY);
    printf("x = %d y = %d\n", ps->x, ps->y);
    return 0;
}
```

В результате код будет вести себя так, как нам нужно, и при этом нет никакой необходимости изменять наши "старые" неуправляемые функции.

Использование управляемых типов в неуправляемом коде

Рассмотрим вопрос взаимодействия двух типов кода с другой стороны. При смешивании управляемого и неуправляемого кода разработчики вскоре столкнутся с проблемой, состоящей в том, что неуправляемый код C++ не может непосредственно получать доступ к управляемым типам. Ниже приведен пример кода, который не будет работать:

```
#using <mscorlib.dll>
#include <tchar.h>

using namespace System;

__gc class Foo
{
}
```

```

public:
    void ShowSomeFoo(int number)
    {
        Console.WriteLine(number.ToString());
    }
};

#pragma unmanaged
void UseFoo(Foo* pFoo)
{
    pFoo.ShowSomeFoo(2);
}

#pragma managed
int _tmain(void)
{
    Foo* pFoo = new Foo();
    UseFoo(pFoo);
    return 1;
}

```

Без директив `#pragma` этот код откомпилируется и будет выполнен (если установлен флаг `/clr`). Однако из-за того, что он содержит директиву `#pragma unmanaged`, он пытается использовать неуправляемую функцию `UseFoo()` для вызова управляемого метода. При этом также делается попытка манипулировать типом управляемых данных с помощью неуправляемой функции. Компилятор быстро выразит нам свое неудовольствие:

```

error C3821: 'pFoo':
managed type cannot be used in an unmanaged function
(управляемый тип не может использоваться в неуправляемой функции)

```

```

error C3175: 'Foo::ShowSomeFoo':
cannot call a method of a managed type from unmanaged function 'UseFoo'
(невозможно вызвать метод управляемого типа из неуправляемой функции 'UseFoo')

```

В данном случае необходим механизм, позволяющий поместить управляемый тип в какой-либо объект, чтобы к нему можно было получить доступ из неуправляемого кода. Кроме того, этот механизм должен проинформировать среду CLR о том, что имеется неуправляемая ссылка на управляемый объект. Решение заключается в использовании типа-значения `System::Runtime::InteropServices::GCHandle`, который позволяет создавать управляемый тип в управляемой куче, но так, чтобы указатель на данный тип принадлежал к типу-значению. Применить этот механизм очень просто. Для создания нужного указателя вызывается метод `GCHandle::Alloc()`, а для его освобождения применяется функция `GCHandle::Free()`. Таким образом, используя тип-значение `GCHandle`, можно написать следующий код:

```

ftusing<mscorlib.dll>
#include<tchar.h>

using namespace System;
using namespace System::Runtime::InteropServices;

__gc class Foo
{
public:
    void ShowSomeFoo(int number)

```

```

        Console.WriteLine(number.ToString());
    }
};

__nogc class FooWrapper
{
private:
    void* handle;

public:
    FooWrapper()
    {
        handle = GCHandle::op_Explicit(GCHandle::Alloc(new Foo)).ToPointer();
    }

    ~FooWrapper()
    {
        (GCHandle::op_Explicit(handle)).Free();
    }

    void ShowSomeFoo(int number)
    {
        Foo* pFoo = static_cast<Foo*>(GCHandle::op_Explicit(handle).Target);
        pFoo->ShowSomeFoo(number);
    }
};

#pragma unmanaged
void UseFoo(FooWrapper* pFooWrapper)
{
    pFooWrapper->ShowSomeFoo(2);
}

#pragma managed
int _tmain()
{
    FooWrapper* pFooWrapper = new FooWrapper;
    UseFoo(pFooWrapper);
    return 1;
}

```

В данном примере используется базовый класс-оболочка (подробнее см. в следующем разделе). В этом классе объявляется член типа `void*`. Затем в конструкторе вызываются два метода: `GCHandle::Alloc()` и `GCHandle::op_Explicit()`, чтобы можно было получить целочисленный тип `System::IntPtr` и использовать метод этого типа `ToPointer()` для преобразования целого числа в указатель. В деструкторе вызываются методы `GCHandle::op_Explicit()` и `GCHandle::Free()`.

Метод `FooWrapper::ShowSomeFoo()` использует свойство `GCHandle::Target`, чтобы задать объект, который представляет указатель `handle`. Поскольку свойство `Target` возвращает тип `System::Object*`, мы преобразуем его в указатель на объект типа `Foo`. После внесения в код всех коррективов функции `main()` и `UseFoo()` будут использовать класс-оболочку, однако вторая функция по-прежнему останется неуправляемой. При этом можно использовать управляемый класс без каких-либо изменений.

Здесь читатель может решить, что, хотя код работает, он слишком громоздкий. И с этим трудно не согласиться. В действительности для того, чтобы упростить описанный процесс, Microsoft предлагает шаблон `gcroot`, который представляет собой интеллектуальный указатель, аналогичный `GCHandle`. Используем этот новый указатель и посмотрим, каков будет результат:

```
#include <groot.h>

using <mscorlib.dll>
#include <tchar.h>

using namespace System;

using namespace System::Runtime::InteropServices;

// Для краткости код опущен

__nogc class FooWrapper
{
private:
    • groot<Foo*> m_pFoo;
public:
    FooWrapper()
    {
        m_pFoo = new Foo;
    }

    ~ FooWrapper()
    {
    }

    void ShowSomeFoo(int number)
    {
        m_pFoo->ShowSomeFoo(number);
    }
};
```

Трудно не согласиться, что это более простое и ясное решение. Данный пример служит хорошим введением в более общую тему написания классов-оболочек, которой посвящен следующий раздел.

Написание управляемых прокси-классов

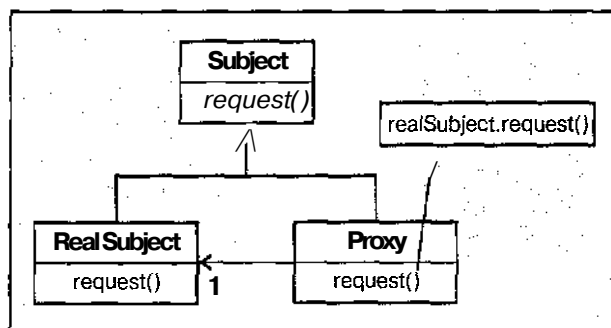
В последнем примере, который был успешно откомпилирован, неуправляемый код вызывает управляемый класс. Однако чаще всего, когда говорят о взаимодействии между управляемым и неуправляемым кодом, подразумевают противоположный процесс. Ранее мы уже реализовали такой процесс с помощью механизма *IJW*. Чем же может не устраивать подобный подход? Для поиска иного решения есть несколько причин. Например, весьма существенно то, что *IJW* представляет собой механизм C++, который в основном осуществляет взаимодействие между неуправляемым и управляемым кодом C++. Но что, если нужно предоставить функциональность неуправляемого кода C++ для использования в C#, Visual Basic .NET или любом другом .NET-совместимом языке?

Если говорить о политике программирования, то существенная доля разработок для платформы .NET будет реализована с помощью C# или Visual Basic .NET в силу тех достоинств, которыми обладают эти языки. Тот, кто уверен, что управляемый мир .NET дает разработчикам большие преимущества, несомненно, захочет предоставить свой код в распоряжение всех управляемых клиентов. Поэтому для реализации полного доступа со стороны управляемого кода к нашим неуправляемым классам C++ имеет смысл обратиться к написанию управляемых прокси-классов.

Образец “прокси”

Концепция оболочек широко известна и используется для решения многих проблем, связанных с разработкой ПО. В своем основополагающем труде *Design Patterns: Elements of Object-Oriented Software* четверо авторов, известных как “Банда четырех” (“Gang of Four”, GOF), ввели в обиход для таких решений понятие “образцов” проектирования ПО. Один из этих образцов они назвали “прокси” (proxy – заместитель, посредник). Пользуясь этой терминологией, можно сказать, что, когда мы создаем класс-оболочку, мы реализуем образец проектирования “прокси”.

Образец “прокси” “позволяет создать заместителя или представителя для другого объекта с целью управления доступом к нему” (GOF, стр. 297). Это именно то, что нам нужно от класса-оболочки, который должен обеспечить доступ к неуправляемому коду. “Прокси применяется, когда возникает потребность в создании более универсальной или сложной ссылки на объект, чем простой указатель” (GOF, стр. 208). Образец “прокси” можно представить с помощью следующей диаграммы классов UML:



Основные этапы создания класса-оболочки

Microsoft определяет следующие этапы создания управляемой оболочки для неуправляемого класса C++:

1. Создание управляемого класса и объявление единственного члена, который указывает на неуправляемый класс.
2. Определение для каждого конструктора неуправляемого класса соответствующего конструктора управляемого класса, который создает экземпляр неуправляемого класса с помощью неуправляемого оператора new, вызывающего первоначальный конструктор.
3. Определение деструктора, который вызывает оператор delete для члена-указателя на неуправляемый класс (если управляемый класс содержит единственную ссылку на неуправляемый класс).
4. Работа с перегруженными операторами.
5. Объявление для каждого оставшегося метода неуправляемого класса идентичного метода, который просто передает вызов в неуправляемую версию, выполняя любой необходимый маршalling параметров.

Кроме того, при создании класса-оболочки необходимо учитывать ряд факторов. В частности, следует ответить на вопрос, что именно нужно заключать в оболочку. На первый взгляд, кажется, что создавать оболочку необходимо для каждой

функции-члена или элемента данных, принадлежащего к классу C++, однако это не всегда имеет смысл. Одна из возможностей состоит в изменении открытого интерфейса. При построении управляемых оболочек часто будут обнаруживаться такие элементы, где первоначальный дизайн и реализация класса C++ могут быть улучшены и/или представлены иначе. Кроме того, некоторые открытые методы не имеют смысла для управляемых клиентов. Еще одно соображение состоит в том, что не следует открывать управляемому коду больше того, что было открыто для неуправляемого кода C++. Из последнего обстоятельства вытекают две рекомендации:

- ❑ Закрытый член класса был разработан не для того, чтобы к нему получали доступ неуправляемые классы. Следовательно, этот член не должен быть доступен и для управляемых классов.
- ❑ Вспомогательные функции в классе являются внутренними и не должны заключаться в оболочку.

Продемонстрируем реализацию этих рекомендаций. Рассмотрим связанный список, который очень хорошо подходит в качестве примера и не требует излишнего усложнения проекта. Вначале кратко обсудим определение неуправляемого класса, а затем покажем, как можно заключить его в управляемый класс-оболочку. Затем протестируем код, вызвав первоначальный неуправляемый класс C++ из проекта C# с помощью управляемого прокси-класса.

Неуправляемый связанный список

Полная реализация рассматриваемого примера IntList вместе с другим загружаемым кодом, использованным в данной книге, находится на сайте www.wrox.com. Здесь мы рассмотрим лишь определения класса, содержащиеся в файле IntList.h:

```
#pragma once

#pragma unmanaged

#include <iostream>

// Класс, представляющий реальные элементы в списке
class Item
{
public:
    Item(int value, Item *ItemToLinkTo = 0);

    int value() const
    { return m_value; }

    void next(Item *link)
    { m_next = link; }

    Item* next()
    { return m_next; }

private:
    int m_value;
    Item *m_next;
};

inline Item::Item(int value, Item *item) : m_value(value)
{
    if(!item)

```

```

        m_next = 0;
    else
    {
        *m_next = item->m_next;
        item->m_next = this;
    }
}

// Это реальный целочисленный связанный список
class IntList
{
public:
    // Конструкторы, оператор и деструктор
    IntList(): m_front(0), m_end(0), m_current(0), m_size(0)
    {}

    IntList(const IntList &rhs): m_front(0), m_end(0), m_current(0)
    { insert_all(rhs); }

    IntList& operator=(const IntList &rhs)
    {
        remove_all();
        insert_all(rhs);
        return *this;
    }

    ~IntList()
    { remove_all(); }

    // Методы доступа
    Item* front() const
    { return m_front; }

    int size()
    { return m_size; }

    // Методы вставки
    void insert(Item* ptr, int value);
    void insert_all(const IntList &rhs);
    void insert_end(int value);
    void insert_front(int value);

    // Методы удаления
    int remove(Item* ptr);
    int remove(int value);
    void remove_front();
    void remove_all();

    // Методы поиска и перебора членов
    Item* find(int value);
    Item* next_iter();

    Item* init_iter()
    { return m_current = m_front; }

    // Методы concat и reverse
    void concat(const IntList&);
    void reverse();
    IntList concat_copy(const IntList& const;
    IntList reverse_copy() const;

```

```

// Отображение списка
void display();

private:
// Вспомогательные функции
void bump_up()
{ ++m_size; }

void bump_down()
{ --m_size; }
Item *m_front;
Item *m_end;
Item *m_current;
int m_size;
};

```

Построение класса-оболочки

Начнем построение управляемого класса, который будет служить оболочкой для нашего связанного списка. Во-первых, создадим новый проект Managed C++ Class Library и назовем его ManagedList. Затем скопируем файлы IntList.h и IntList.cpp (содержащие реализацию неуправляемого связанного списка) в папку проекта, прежде чем добавить их в сам проект.

Этап 1. Создание члена — указателя на неуправляемый класс

Определим управляемый класс MList и объявим элемент данных IntList*, тип которого является указателем на класс, заключаемый в оболочку. Ниже приведен соответствующий код, содержащийся в файле ManagedList.h:

```
// Управляемый класс-оболочка для примера IntList
```

```

#pragma once
#include "IntList.h" #pragma managed
#pragma managed
#include <mscorlib.dll>

using namespace System;

namespace ManagedList
{
    public __gc class MList
    {
    public:
        private:
            IntList __nogc* m_pIntList;
    };
}

```

Этап 2. Заключение в оболочку конструкторов

При построении класса-оболочки не всегда имеет смысл создавать конструктор-оболочку для каждого конструктора неуправляемого класса. Прежде чем приступить ко второму этапу, необходимо ответить на два вопроса:

а Действительно ли нужен этот конструктор?

а Будет ли данный конструктор иметь смысл для управляемой среды?

Отвечая на первый вопрос, можно быстро прийти к выводу, что некоторые конструкторы в созданных нами ранее неуправляемых классах больше не нужны. Следует помнить, что есть возможность изменить наш класс.

Что касается второго вопроса, то конструктор может инициализировать типы, которые более не используются в управляемом окружении. Пусть, например, имеется класс, выполняющий прозрачное сжатие и восстановление аудиоданных с помощью мультимедийных служб Windows (версия 1 библиотеки классов .NET Framework не поддерживает мультимедийную функциональность). Подобный класс может выглядеть следующим образом:

```
public class MyAudio
{
    MyAudio(LPCSTR FileName);
    MyAudio(WAVEFORMATEX format, LPCSTR filename);
}
```

Управляемые клиенты никогда не будут работать со структурой WAVEFORMATEX. Поэтому нет никакой необходимости открывать второй конструктор этим клиентам.

Решения подобного рода следует принимать для каждого конкретного приложения. Если управляемый клиент не будет использовать конкретную структуру данных или другой элемент, представляемый конструктором, такой конструктор не нужно заключать в оболочку. Возвращаясь к рассматриваемому примеру, отметим, что наш неуправляемый класс имеет следующие конструкторы:

```
IntList(); m_front(0), m_end(0), m_current(0), m_size(0)
{}

IntList(const IntList &rhs) : m_front(0), m_end(0), m_current(0)
{ insert_all(rhs); }
```

Конструктор по умолчанию

Обратимся вначале к конструктору по умолчанию, так как он наиболее простой. Реализация этого конструктора в прокси-классе заключается в инициализации класса, который помещается в оболочку:

```
MList: MList()
{
    m_pIntList = new IntList;
}
```

Конструктор копирования

В связи с конструктором копирования возникает ряд интересных нюансов. Работая с традиционным кодом C++, мы часто приходим к выводу о необходимости определить пользовательский конструктор копирования и оператор присваивания копии, чтобы не допустить простого копирования членов. Для выполнения эквивалентного действия в управляемом коде C++ нужно реализовать интерфейс `ICloneable` в классе `MList`:

```
public __gc class MList : public ICloneable
{
public:
    MListO;

    virtual Object* Clone() ...
```

```

{
    // Создание экземпляра управляемого списка
    MList* ml = new MList;
    *(ml->m_pIntList) = *m_pIntList; // Вызов конструктора копирования
                                     // класса IntList

    // Полное копирование других членов из this->... в ml->...
    return ml;
}

```

Данный код демонстрирует реализацию метода Clone(), в котором мы создаем экземпляр класса MList и вызываем конструктор копирования неуправляемого класса IntList через управляемый указатель. В этом примере нет дополнительных элементов данных. Если бы они были, мы бы также выполнили полное копирование (deep copy) каждого из них в новый управляемый класс.

Этап 3. Создание оболочки для деструкторов

Процедура создания оболочки для деструкторов проще, чем для конструкторов. Нам лишь необходимо иметь в управляемой оболочке деструктор, который вызывает деструктор неуправляемого класса. Применительно к нашему примеру, следует написать:

```

~MList() { m_pIntList->~IntList(); }

```

Может возникнуть вопрос, зачем здесь нужен деструктор. Разве не твердят на каждом шагу, что платформа .NET не требует вызова оператора delete? Однако следует помнить, что автоматическая сборка мусора применима лишь для освобождения памяти от управляемых объектов *и не более того!* Если класс обращается к таким ресурсам, как соединения с базами данных, то их по-прежнему необходимо закрывать. В этом случае мы имеем указатель на неуправляемый объект и должны явно вызывать деструктор в неуправляемом классе, чтобы удалить соответствующий неуправляемый код.

Этап 4. Работа с перегруженными операторами

Аналогично конструктору копирования перегруженный оператор operator=() не поддерживается в управляемом коде C++. Однако можно определить метод Assign() и вызвать его явным образом:

```

// Оператор присваивания
virtual MList* Assign(MList* other)
{
    if (this != other) ...
    {
        *m_pIntList = *(other->m_pIntList);

        // Полное копирование других членов из this->... в ml->...

        return this;
    }
}

```

Что касается других операторов, то платформа .NET Framework предусматривает перегрузку с использованием синтаксиса op_XXXX(), и управляемые расширения C++ позволяют реализовывать перегруженные версии для всех этих операторов.

Этап 5. Создание оболочки для функций-членов

Часто функции-члены могут быть помещены в оболочку путем передачи реализации функции в управляемый класс. Ниже представлены функции, объявленные в заголовочном файле:

```
// Методы вставки
void insert_begin(ptr, int value);
void insert_end(int value);
void insert_front(int value);
void insert_all(const IntList &rhs);

// Методы удаления
int remove(int value);
void Remove_Front();
void Remove_All();

// Методы поиска и перебора членов
Item *find(int value); V
void display(); V
```

Следовательно, при реализации нужно просто вызвать соответствующие функции в управляемом классе:

```
void MList::insert_begin(ptr, int value)
{
    m_pIntList->insert_begin(value);
}

void MList::insert_end(int value)
{
    m_pIntList->insert_end(value);
}

void MList::insert_front(int value)
{
    m_pIntList->insert_front(value);
}

void MList::insert_all(const IntList& rhs)
{
    m_pIntList->insert_all(rhs);
}

// Методы удаления
int MList::remove(int value)
{
    return(m_pIntList->remove(value));
}

void MList::RemoveFront()
{
    m_pIntList->remove_front();
}

void MList::RemoveAll()
{
    m_pIntList->remove_all();
}
```

```
// Методы поиска и перебора членов
Item* Mlist::find(int value)
{
    return m_pIntList->find(value);
}

// Отображение списка
void Mlist::display()
{
    m_pIntList->display();
}
```

Однако не все функции-члены можно так просто заключить в оболочку. **Могут** встретиться три группы функций, для которых необходим другой подход:

а Функции доступа

- Функции, содержащие аргументы по умолчанию
- Функции с переменным числом аргументов

В нашем связанном списке присутствуют лишь функции первого типа, поэтому вначале мы обсудим именно их. Два других случая будут рассмотрены позже.

Функции доступа

Традиционный язык C++ использует стиль, определяющий функции для получения доступа к закрытым переменным-членам. Управляемый код C++ реализует полную поддержку свойств и благодаря им формализует понятие средств доступа (accessors).

Свойства фактически представляют собой методы, к которым клиенты получают доступ как к переменным-членам. Когда в клиентском коде указывается свойство, вызов метода реализуется с учетом того, как к этому свойству должен осуществляться доступ. Это позволяет клиентскому коду взаимодействовать с объектом, используя четко определенные, простые синтаксические конструкции.

Вернемся к нашему классу IntList. В этом классе определены два метода доступа, причем оба представляют собой методы get:

```
Item* front() const
{ return m_front; }

int size()
{ return m_size; }
```

Метод доступа front() позволяет получить значение самого первого ("головного") члена списка IntList. Это нередактируемая переменная-член, о чем говорит отсутствие метода set. Метод size() имеет аналогичную природу.

Оба этих метода являются хорошими кандидатами на то, чтобы стать свойствами в нашем управляемом классе-оболочке. В заголовочном файле мы объявили следующие свойства:

```
_property Item* get_Front();
_property int get_Size();
```

Реализация этих свойств содержится в .cpp-файле:


```

Item Mlist.get_Front()
{
    return(m_pInList->front());
}

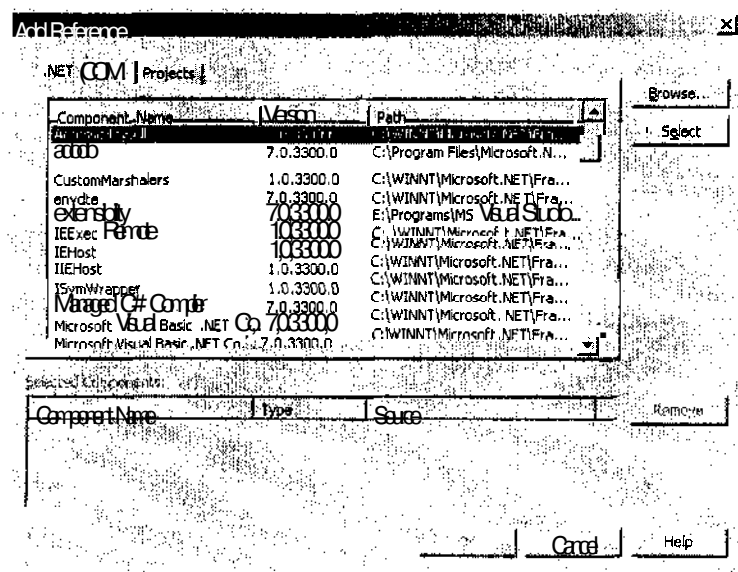
int Mlist.get_Size()
{
    return(m_pInList->size());
}

```

Теперь неуправляемый класс списка заключен в оболочку и можно приступить к его тестированию.

Тестирование управляемого класса-оболочки с помощью клиента C#

Воспользуемся средой разработки Visual Studio .NET для создания нового приложения Visual C# Console Application (Консольное приложение Visual C#), которому присвоим имя ManagedListTest. Можно оставить то имя класса, которое задается по умолчанию (Class1), поскольку в данном случае оно ни на что не влияет. Щелчком правой кнопкой мыши на пункте References в Solution Explorer и в открывшемся меню выберем команду Add Reference:



На вкладке Projects (Проекты) перейдем в папку Debug проекта ManagedList и выберем файл ManagedList.dll. После добавления ссылки на проект ManagedList необходимо сообщить компилятору C# о том, какое пространство имен будет использоваться. Добавим следующую строку кода в класс Class1.cs:

```
using ManagedList;
```

Теперь можно вызывать методы и тестировать наш класс-оболочку. В методе Main() вначале инициализируем новый экземпляр управляемого списка:

```
Mlist sharpList = new Mlist();
```

Затем вызовем методы вставки значений в начало и в конец списка, а также метод, отображающий список:

```
for(int ix = 0; ix < 10; ++ix)
{
    sharpList.InsertFront(ix);
    sharpList.InsertEnd(ix);
}
sharpList.display();
```

Далее используем два свойства. Мы определили свойство `Size` в управляемом коде C++, поэтому можем обратиться к этому свойству, указав его имя:

```
int size = sharpList.Size;
Console.WriteLine("Size of the managed list is {0} items", size.ToString());
```

Проверим теперь весьма важную функциональность метода `Clone()`, который играет роль конструктора копирования C++. Добавим следующий код в файл `Class1.cs`:

```
MList newList = (MList)sharpList.Clone();
size = newList.Size;
Console.WriteLine("Size of the clone is {0} items", size.ToString());
Console.WriteLine("And the members are...");
newList.display();
```

Наконец, протестируем оператор присваивания:

```
newList.InsertFront(100);
newList.InsertFront(1000);
size = newList.Size;
Console.WriteLine("Size of the clone is NOW {0} items", size.ToString());
Console.WriteLine("And the members are...");
newList.display();

// Присваиваем первоначальному списку новый объект для
// тестирования оператора присваивания
sharpList.Assign(newList);
size = sharpList.Size;
Console.WriteLine("Size of the original is NOW {0} items", size.ToString());
Console.WriteLine("And the members are...");
sharpList.display();
```

Ниже представлена информация, выводимая на экран после компоновки и запуска приложения:

```
20 ) ( 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 )
Size of the managed list is 20 items (Управляемый список содержит 20 элементов)
Size of the clone is 20 items (Клон содержит 20 элементов)
And the members are... (Это следующие элементы...)

20 ) ( 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 )
Size of the clone is NOW 22 items (Теперь клон содержит 22 элемента)
And the members are... (Это следующие элементы...)

22 ) (1000 100 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 )
Size of the original is NOW 22 items (Первоначальный список теперь содержит
22 элемента)
And the members are... (Это следующие элементы...)

22 ) (1000 100 9 8 7 6 5 4 3 2 1 0 0 1 2 3 4 5 6 7 8 9 )
```

Видно, что код работает так, как ожидалось. Теперь можно использовать функциональность неуправляемого класса C++ в *любом* .NET-совместимом языке.

Другие проблемы, возникающие при создании класса-оболочки

Мы уже упомянули два случая, когда при создании оболочки для функций-членов и конструкторов надо проявлять особую осторожность. Однако при разработке связанного списка мы не столкнулись с подобными ситуациями. Они имеют место, если приходится заключать в оболочку следующие типы функций:

- Функции с переменным числом параметров
- Функции, содержащие параметры по умолчанию

Функции с переменным числом параметров

При работе с функциями, содержащими переменное число параметров, управляемый код C++ не делает того, чего можно было бы ожидать. Компилятор будет автоматически компилировать любую такую функцию в неуправляемый код. Как правило, это не то, что требуется, однако не существует способа непосредственно заключить в оболочку функцию подобного типа. Поэтому необходимо искать альтернативные пути.

Одно из решений заключается в перегрузке функций, хотя количество перегруженных версий зависит от конкретного приложения и может быть очень большим. Microsoft рекомендует использовать перегруженные функции, которые принимают 1, 2 или 3 аргумента, а также массив типа `Object*`.

Функции, содержащие параметры по умолчанию

Управляемый код C++ не поддерживает функций с аргументами по умолчанию, поскольку это несовместимо со спецификацией Common Language Specification. Это означает, что нельзя непосредственно передавать реализацию членов такого типа. Решение проблемы заключается в том, чтобы написать функцию-член как две функции, одна из которых принимает два аргумента, а вторая — только один, но при этом вызывает первую функцию с нужным значением.

Если в конструкторах определены аргументы по умолчанию, ситуация несколько усложняется, поскольку C++ не позволяет вызывать конструктор из другого конструктора. Чтобы обойти эту проблему, следует использовать вспомогательную функцию с тем же количеством параметров, что и в оригинальном конструкторе. Такая функция может вызываться группой перегруженных конструкторов, передающих жестко заданные значения для аргументов, которые им самим не передаются.

Маршалинг между управляемым и неуправляемым кодом

До сих пор наши попытки ликвидировать брешь между управляемым и неуправляемым кодом в основном сводились к решению проблем, связанных с указателями. Однако может возникнуть реальная ситуация, когда неуправляемые данные, определяемые указателем, будут просто несовместимы с управляемым кодом. Чтобы продолжить наше продвижение по выбранному пути, необходимо детально обсудить различия между управляемым и неуправляемым кодом.

ДЛЯ ТОГО чтобы сосуществование управляемого и неуправляемого кода оказалось возможным, должен быть реализован механизм, который позволяет коду, выполняющемуся внутри (и, следовательно, под управлением) CLR взаимодействовать с кодом, находящимся за пределами среды исполнения. Когда существует взаимодействие между управляемым и неуправляемым кодом, работает один из трех механизмов реализации этого взаимодействия:

□ Внутренний вызов через указатель на функцию

D Механизм "It Just Works" (IJW)

- Службы Platform Invoke (PInvoke)

Мы уже обсуждали механизм IJW как средство переноса кода, а теперь следует рассмотреть этот механизм с точки зрения взаимодействия различных типов кода. Мы обратимся к внутренней реализации процесса перехода от одного типа кода к другому, обсудим три варианта перехода и покажем, когда имеет место **маршалинг**. В связи с этим мы подробно рассмотрим класс `System::Runtime::InteropServices::Marshal` и его использование для выделения управляемой памяти, копирования блоков неуправляемой памяти и преобразования управляемых типов в неуправляемые.

Процессы перехода между управляемым и неуправляемым кодом

На границе между управляемым и неуправляемым кодом проявляются отличия одного типа кода от другого. Эти различия можно разделить на следующие категории:

□ Различаются представления типов данных.

- Различно размещение структур в памяти.
- Отличаются соглашения о вызовах.
- Управляемый код не выполняет подсчет ссылок на объекты, как это делается в технологии COM.

О Управляемый код не освобождает выделенную память.

□ Управляемый код помещает динамически загружаемые библиотеки в закрытые или совместно используемые сборки.

- Управляемый код удаляется сборщиком мусора. При этом ссылки на объекты могут сдвигаться.

Тем, кто хорошо знаком с COM+, все это должно напоминать различия между контекстами времени выполнения, для преодоления которых используется перехватчик. Среда исполнения CLR предоставляет "переходную секцию" (раздел кода, который компилятор вставляет между двумя контекстами). Переходная секция не позволяет выполнять сборку мусора в течение вызова функции, обеспечивает распространение процесса обработки исключительных ситуаций и в случае необходимости реализует маршалинг между управляемым и неуправляемым кодом.

Процессы перехода, относящиеся к внутренним вызовам

Первая из перечисленных категорий перехода представляет собой механизм внутренних вызовов. Методы внутреннего вызова генерируются средой исполнения для использования в процессах, протекающих в рамках CLR. Такие методы применяются, например, управляемыми типами для передачи управления внутри

самой CLR. Описываемый переход осуществляется с помощью указателя на функцию, и, пока не применяетсяmarshaling, на вызываемый код накладываются строгие ограничения; он должен быть **IJW-совместимым** и содержать менее 100 инструкций. Этот тип перехода малоинтересен для программистов Visual C++ .NET, и мы упоминаем его для полноты изложения. С методами внутреннего вызова связано использование класса `System::GC` и метода `System::Object::GetHashCode()`.

IJW-переходы

IJW-переходы имеют место в результате функционирования компилятора Visual C++ .NET, а не работы программиста. Редактор связей реализует процесс разрешения ссылок, а загрузчик класса CLR создает переходные секции кода для преодоления различий между управляемым и неуправляемым кодом. Эти переходные секции определяют распространение процесса обработки исключений и сборку мусора, а также в случае необходимости обеспечивают некоторый базовый marshaling. Однако, если в неуправляемый код передаются данные, находящиеся в управляемой куче, marshaling необходимо реализовывать самостоятельно.

Переходы, связанные сработойслужбPlatformInvoke (PInvoke)

Технология PInvoke обеспечивает переход от управляемого кода к неуправляемому, если последний реализован в библиотеке для конкретной платформы, например в DLL Win32 API. Использование служб PInvoke не обязательно означает "вызов платформы Win32", а в общем случае описывается как "вызов платформозависимого кода". Текущая реализация Microsoft реализует вызов Win32, но в принципе это не обязательно.

При работе механизма PInvoke среда исполнения .NET сама заботится о marshalingе всех аргументов для передачи их в функцию неуправляемой DLL, а также о восстановлении возвращаемых значений и выходных параметров. Хотя большинство служб операционной системы доступно нам благодаря библиотеке классов .NET Framework, могут возникнуть ситуации, когда необходимо напрямую вызывать функции Win32 API. В этом случае на помощь приходит механизм PInvoke. Что касается внутренней реализации, то многие методы указанной библиотеки классов также используют службы PInvoke!

Имеется два различных типа служб PInvoke: базовые службы (обычно их и называют PInvoke) и COM-сервисы (называемые COM interop). Эти типы представляют собой два отдельных слоя, поскольку технология COM предъявляет более высокие требования, чем те, что связаны с процессом вызова функций. Так, к требованиям COM относятся обеспечение подсчета ссылок и корректный вызов метода `QueryInterface()`.

Позже мы более подробно обсудим технологию PInvoke, в том числе и в следующей главе, где рассматривается взаимодействие с COM.

Когда нужен marshaling

ЕСЛИ неуправляемый код **вызывается** из управляемого, в большинстве случаев необходимо выполнять marshaling передаваемых аргументов. С учетом этого следует ответить на два вопроса:

- Насколько простым или сложным должен быть marshaling?
- Кто реализует marshaling: разработчик или среда исполнения?

Понимание некоторых ключевых концепций поможет нам приподнять завесу таинственности над механизмами взаимодействия кода и маршалингом. Мы уже говорили о существовании границы между управляемым и неуправляемым кодом. Необходимо также иметь в виду, что при передаче вызова из управляемого кода (или в управляемый код) аргументы передаются в стек вызовов. Пока все просто. Но теперь мы подходим к основной концепции: стек вызовов содержит параметры, представляющие собой экземпляры типов, которые имеют определенный смысл для обеих сред — управляемой и неуправляемой. Значение (value) в стеке фактически представлено двумя типами: управляемым и неуправляемым. *На базовом уровне можно говорить о маршалинге, относящемся к взаимодействию кода, как о наборе правил преобразования управляемых типов в неуправляемые или наоборот.*

Следующая концепция связана с **изоморфными типами**. Хотя это понятие выглядит сложным, на деле оно лишь указывает на необходимость преобразования. Когда управляемый тип содержит только изоморфные типы данных и определено его размещение в памяти, представление этого типа оказывается идентичным как в управляемом, так и в неуправляемом коде. В этом случае преобразование не нужно. В любой другой ситуации, кроме той, в которой мы имеем дело только с изоморфными типами, необходимо преобразование (маршалинг), чтобы управляемый тип стал понятен неуправляемому коду. При этом создается копия управляемых данных и передается в неуправляемый код. Когда данные возвращаются, копия преобразуется обратно в управляемую форму и заменяет оригинал.

Иногда Microsoft использует термины "блитуемый" (blittable) и "неблитуемый" (non-blittable) типы вместо понятий "изоморфный" и "неизоморфный",

Как и в концепции перехвата COM⁺, необходимо, чтобы две стороны процесса в точности совпадали. Если хотя бы один параметр не относится к изоморфному типу, стековый фрейм должен преобразовываться в формат, понятный для неуправляемой среды. Преобразование также должно происходить и в другом направлении (для возвращаемых значений). Обсуждение этих требований естественным образом приводит нас к необходимости рассмотреть класс System::Runtime::InteropServices::Marshal.

Класс InteropServices::Marshal

Класс System::Runtime::InteropServices::Marshal содержит множество полезных служебных методов, помогающих решать задачи взаимодействия. Методы этого класса являются статическими, не считая тех, которые унаследованы от System::Object. Поэтому почти никогда не нужно инициализировать данный класс. Согласно документации Microsoft, класс Marshal "предоставляет набор методов, относящихся к выделению неуправляемой памяти, копированию блоков неуправляемой памяти и преобразованию управляемых типов в неуправляемые". Многие возможности этого класса предназначены для реализации взаимодействия с COM, и, хотя они не являются основной темой настоящей главы, их также можно здесь использовать.

Преобразование строк

Одна из наиболее полезных функций, реализуемых классом Marshal, заключается в преобразовании управляемой строки (неизоморфный тип) в неуправляемую строку, так что ее можно использовать в операторе вызова традиционной функции Win32 API. Следующая таблица Microsoft иллюстрирует преобразование строк, используемое при маршалинге:

wtypes.h	C++	Управляемые расширения	Единая среда исполнения
CHAR	char	Char	Char
LPSTR	char*	String* [in], StringBuilder* [in, out]	String [in], StringBuilder [in, out]
LPCSTR	Const char*	String*	String
LPWSTR	wchar_t*	String* [in], StringBuilder* [in, out]	String [in], StringBuilder [in, out]
LPCWSTR	Const wchar_t*	String*	String

Обратимся к примеру. Представленные в таблице типы CHAR, LPSTR, LPCSTR, LPWSTR и LPCWSTR часто встречаются в коде C++, написанном для Windows. При вызове функции, которая принимает один из этих типов в качестве параметра, необходимо выполнитьmarshaling. Мы создадим управляемую строку, содержащую имя машины, с помощью управляемого класса System::Environment, а затем воспользуемся классом Marshal для передачи этой строки в функцию Win32 MessageBox().

Можно также вызвать функцию API с помощью PInvoke, однако в настоящем примере будет использован механизм IJW.

```
#using <mscorlib.dll>
#include <tchar.h>
#include <windows.h>

using namespace System;
using namespace System::Runtime::InteropServices;

// Библиотеки, которые мы хотим использовать
#pragma comment(lib, "user32.lib")

int _tmain(void)
{
    Text::StringBuilder* buildMeAString = new Text::StringBuilder();
    buildMeAString->Append("The name of your machine is ");
    buildMeAString->Append(System::Environment::MachineName);

    // Преобразуем управляемую строку в строку LPCWSTR
    IntPtr aString;
    aString = Marshal::StringToHGlobalAuto(buildMeAString->ToString());
    MessageBox(0, static_cast<LPCWSTR>(aString.ToPointer()),
        "Look Ma! I, marshaled!", MB_OK);

    // Освободим память
    Marshal::FreeHGlobal(aString);

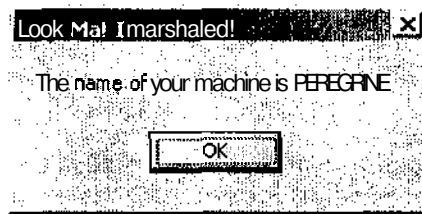
    return 0;
}
```

Откомпилируем этот код с флагом /clr, и механизм IJW сделает все, кроме marshalinga. С помощью ключевого слова pragma мы сообщаем компилятору, что программа будет использовать функцию из библиотеки user32.dll, а затем создаем строку, состоящую из некоторого текста и имени машины. Для понимания того, что происходит дальше, необходимо знать прототип функции Win32 MessageBox():

```
int MessageBox(HWND hwnd, LPCWSTR lpText, LPCWSTR lpCaption, UINT uType);
```

Проблема в том, что тип `LPCWSTR` не является управляемым и потому несовместим с `CLS`. Если нужно вызвать функцию `MessageBoxW()` из окружения `.NET`, необходимо выполнить маршalling между типом `LPCWSTR` и управляемым типом `String`.

Статические методы `StringToXXXX` класса `Marshal` возвращают величины управляемого типа `System::IntPtr`, который наследует от класса `System::ValueType`. Мы используем метод `StringToHGlobalAuto()` для копирования содержимого управляемой строки в глобальную кучу Windows. В случае необходимости строка может быть преобразована в формат ANSI, до того как будет использован метод `IntPtr::ToPointers` для преобразования значения экземпляра типа `IntPtr` в указатель на неспецифический тип (т.е. в указатель типа `void*`). Затем мы приводим указатель к типу, необходимому для функции `MessageBoxW()`. После этого вызывается метод `Marshal::FreeHGlobal()` для освобождения экземпляра типа `IntPtr` и приложение завершает работу. При запуске кода на машине автора результат выглядит следующим образом:



Альтернативой методам `Marshal::StringToXXXX()` является функция `PtrToStringCharsO`. Эта функция определена в файле `vcclr.h` и предоставляет прямой доступ к символам объекта `String`, обеспечивая высокопроизводительные вызовы неуправляемых функций, которые принимают строки типа `wchar_t*`. Однако этот метод не лишен недостатков, в частности он возвращает управляемый указатель, который необходимо фиксировать для вызова функции `Win32`:

```
const System::Char* PtrToStringChars(const System::String* s)
```

Другая проблема состоит в том, что этот метод возвращает внутренний управляемый указатель, который указывает на первый символ строки в объекте `System::String`. Внутренние указатели являются достаточно ресурсоемкими, поскольку они требуют специальной обработки сборщиком мусора и, следовательно, дополнительно загружают вычислительные ресурсы. Внутренний указатель можно напрямую увеличить или уменьшить. Наконец, метод `PtrToStringCharsO` возвращает строку в формате Unicode, которую в определенных случаях приходится приводить к формату ANSI.

Несмотря на все эти недостатки, функция `PtrToStringCharsO` обеспечивает в рассматриваемом нами случае некоторый выигрыш производительности по сравнению с методом `Marshal::StringToHGlobalAuto()`. Она позволяет заменить строки кода:

```
IntPtr aString;
aString = Marshal::StringToHGlobalAuto(buildMeAString->ToString());
MessageBoxW(0, static_cast<LPCWSTR>(aString.ToPointer()),
            L"Look Ma! I marshaled!", MB_OK);
Marshal::FreeHGlobal(aString);
```

следующими строками:

```
const wchar_t* pin* aString PtrToStringChars(buildMeAString->ToString());
MessageBoxW(0, aString, L"Look Ma! I marshaled!", MB_OK);
```


Механизм PInvoke: Вызов неуправляемых функций из управляемого кода

По существу, технология PInvoke представляет собой средство перехода от управляемого кода к неуправляемому в комбинации с механизмом, который преобразует точку входа для статического метода в точку входа для файла Windows DLL и загрузчик DLL. Мы уже немного говорили о том, как средство PInvoke обеспечивает переход между управляемым и неуправляемым кодом. Поэтому сейчас сконцентрируемся на остальных двух упомянутых элементах.

Для понимания механизма преобразования точки входа в технологии PInvoke полезно кратко обсудить структуру DLL и ее представление в файле формата Windows PE. Файлы DLL содержат экспортируемые функции, и Visual C++ поддерживает два способа экспорта таких функций: с использованием модификатора `__declspec` и с помощью файлов, имеющих расширение `.def`. Когда компилятор находит один из этих двух элементов, он заносит некоторую дополнительную информацию в результирующий `.obj`-файл. Затем компоновщик обнаруживает добавленную информацию об экспортируемой переменной, функции или классе и автоматически создает `.lib`-файл, содержащий экспортируемую таблицу символов, а также их относительные адреса.

Существует два способа вызвать метод, содержащийся в DLL. Если использовать динамическую связь, устанавливаемую при загрузке программы (явная связь), то код связывается с библиотекой импорта для DLL и выполняет явные вызовы функций. В случае использования динамической связи периода выполнения (неявная связь) применяется метод `LoadLibrary()` для загрузки DLL в адресное пространство во время выполнения. После этого вызывается метод `GetProcAddress()` для получения адресов экспортируемых функций DLL. Косвенные ссылки на эти функции обеспечиваются указателями, возвращаемыми методом `GetProcAddress()`.

Механизм PInvoke выполняет свои функции путем преобразования декларации статического метода, помещенной в управляемый код, в точку входа для DLL, которая определяется с помощью методов `LoadLibrary()` и `GetProcAddress()`. Стековый фрейм кодируется в декларации управляемого метода, однако сам метод находится в другом месте! Он содержится во внешней, неуправляемой DLL. Здесь важно отметить, что механизм PInvoke использует **отложенную загрузку (delay loading)**. Это означает, что до тех пор, пока мы в первый раз не вызовем метод, вызов функции `LoadLibrary()` не выполняется и соответственно не происходит утечка ресурсов.

Чтобы пометить метод как внешний, т.е. принадлежащий к неуправляемой DLL, нужно задать информацию о том, как вызывается DLL. Это делается с помощью атрибута `System::Runtime::InteropServices::DllImport`, который предоставляет сведения, относящиеся к конечной точке функции, соглашению о вызовах, предоставлению символов (Unicode или ANSI), а также (в случае COM interop) к манипуляциям сигнатурами. С помощью этого атрибута можно явно задавать кодировку Unicode/ANSI или позволить платформе использовать установки по умолчанию (что в большинстве случаев вполне оправдано).

Другая функция PInvoke — этоmarshaling параметров, для выполнения которого механизму PInvoke нужна определенная информация. Для предоставления этой информации используется атрибут `System::Runtime::InteropServices::MarshalAs`.

Применение PInvoke для вызова экспортируемых функций в неуправляемых DLL

Изучение механизма PInvoke начнем с выбора некоторой функции Win32 API и анализа того, как она вызывается из управляемого кода. В качестве опытного

образца используем функцию API `sndPlaySound()`, содержащуюся в файле `winmm.dll`. В первую очередь нужно написать управляемое определение для неуправляемой функции, а затем использовать атрибут `DllImport` для указания того, что эта функция будет вызываться с помощью `PInvoke`. Первый аргумент этого атрибута задает имя неуправляемой DLL, которая экспортирует функцию. Кроме этого, атрибут содержит и другие, именованные аргументы, позволяющие задать набор символов, псевдонимы, соглашение о вызовах и т.д.

Функция Win32 API `sndPlaySound()`, содержащаяся в файле `winmm.dll`, имеет следующую сигнатуру:

```
BOOL sndPlaySound(LPCSTR lpszSound, UINT fuSound);
```

Первый параметр этой функции задает путь к WAV-файлу или представляет зарегистрированный системный звук Windows. С помощью второго аргумента устанавливается флаг, определяющий, как будет проигрываться звук. Есть два способа импорта данной функции в управляемое приложение C++. Можно добавить в код заголовочный файл, связь с библиотекой импорта и использовать механизм `IJW` (как было в примере с функцией `MessageBox()`). Это простое решение, однако его недостаток состоит в том, что нужно явно задавать маршалинг в коде (вместо того, чтобы использовать атрибуты) со всеми проистекающими отсюда сложностями. Второй вариант заключается в использовании механизма `PInvoke`. В этом случае необходимо написать управляемое определение (с ключевыми словами `static` или `extern`) для функции `sndPlaySound()` и затем пометить это определение атрибутом `DllImport`.

```
using class MyWin32Bridge
{
public:
    [DllImport("winmm.dll", CharSet = CharSet.Auto)]
    static bool sndPlaySound(String* pstrWaveFile, int nSoundFlag);
};
```

Такие функции всегда лучше объявлять как статические методы управляемого класса. Однако ничто не мешает задать в управляемом определении глобальную область видимости. Для этого необходимо в начало определения поставить ключевое слово `extern`, а не `static`:

```
[DllImport("winmm.dll", CharSet = CharSet.Auto)]
extern bool sndPlaySound(String* pstrWaveFile, int nSoundFlag);
```

Мы заменили неуправляемые типы данных (возвращаемый тип `BOOL`, а также типы параметров `LPCSTR` и `UINT`) их управляемыми эквивалентами: `bool`, `String*` и `int`. Первый параметр атрибута `DllImport` указывает, что функция `sndPlaySound()` экспортируется из файла `winmm.dll`, а именованный параметр `CharSet` задает набор символов, используемый при маршалинге строковых параметров. Этот параметр может принимать одно из следующих значений перечисления `CharSet`:

Перечислимое значение CharSet	Описание
Ansi	Маршалинг с использованием только строк ANSI
Unicode	Маршалинг, использующий только строки Unicode
Auto	Автоматический маршалинг на основе набора символов операционной системы (Windows NT/2000/XP или Windows 9x/ME)

Для вызова функции `sndPlaySoundO` с помощью механизма `PInvoke` необходимо просто вызвать определение управляемой функции, передавая туда необходимые аргументы:

```
int _tmain(void)
{
    MyWin32Bridge::sndPlaySound(S"C:\\winnt\\Media\\ringin.wav", 0);
    return 0;
}
```

В управляемом определении можно даже присвоить неуправляемой функции псевдоним. Именованный параметр `EntryPoint` позволяет задать фактическое имя неуправляемой функции. Сделав это, мы получаем возможность выбрать для управляемой функции любое другое имя:

```
_gc class MyWin32Bridge
{
public:
    [DllImport("S\\winnt.dll", CharSet = CharSet.Auto,
        EntryPoint = "sndPlaySound")]
    static bool PlayMyWaveFile(String* pstrWaveFile, int nSoundFlag);
};
```

Теперь можно вызвать функцию `sndPlaySound()`, используя ее псевдоним `PlayMyWaveFile()`:

```
MyWin32Bridge::PlayMyWaveFile("C:\\winnt\\Media\\ringin.wav", 0);
```

Получение уведомлений обратного вызова в управляемом приложении C++ с помощью PInvoke

Методы обратного вызова реализуют механизм, позволяющий функции многократно вызывать зарегистрированную функцию-обработчик. Функция может вызывать обработчик в любой момент, когда ей это нужно, для передачи уведомления или некоторых данных. Методы обратного вызова часто применяются в Win32 API, особенно в функциях, использующих перечисления, причем эти функции вызывают обработчик для передачи информации об элементе перечисления.

С помощью `PInvoke` можно зарегистрировать управляемый обработчик обратного вызова и передать его в неуправляемую функцию, поддерживающую обратные вызовы. Мы покажем, как это делается, на примере вызова функции `API EnumWindows()` из управляемого приложения C++ и регистрации управляемого обработчика обратного вызова для взаимодействия с данной функцией. Функция-обработчик сможет получать уведомления, как только открытые в тот или иной момент окна будут перечислены с помощью функции `EnumWindows()`. Сигнатура функции `API EnumWindows()` имеет вид:

```
BOOL EnumWindows(WNDENUMPROC lpEnumFunc, LPARAM lParam);
```

Здесь первый параметр принадлежит к типу `WNDENUMPROC` и представляет собой указатель на функцию обратного вызова, которая имеет следующую сигнатуру:

```
BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lParam);
```

Функция `EnumWindowsO` будет многократно обращаться к обработчику обратного вызова, передавая ему дескрипторы окон переднего плана, открытых в данный момент. Чтобы с помощью `PInvoke` зарегистрировать управляемый обработчик обратного вызова для взаимодействия с функцией `EnumWindowsO`, вначале необходимо объявить делегата, исходя из сигнатуры обработчика `EnumWindowsProc()`:

```
__delegate bool EnumWindowsProcCallback(long hwnd, long lParam);
```

Следующий этап разработки нашего приложения состоит в определении управляемой функции, которая представляет собой обработчик обратного вызова. На это определение будет указывать делегат EnumWindowsProcCallback, и, следовательно, мы должны использовать ту же самую сигнатуру. С помощью обработчика обратного вызова выведем на экран заголовок окна, который будет получен из дескриптора окна, передаваемого обработчику функцией EnumWindows():

```
static bool DisplayWindowTitle(long hwnd, long lParam)
{
    // Использование обработчика для отображения заголовка окна.
}
```

Как обычно, нужно написать определение управляемой функции для функции API EnumWindows() и использовать атрибут DllImport, чтобы указать DLL, которая экспортирует EnumWindows():

```
[DllImport("user32.dll")]
static bool EnumWindows(EnumWindowsProcCallback* pfnCallback, long lParam);
```

Прежде чем вызывать функцию EnumWindows(), необходимо создать экземпляр делегата EnumWindowsProcCallback, передав ему адрес обработчика обратного вызова DisplayWindowTitle():

```
EnumWindowsProcCallback* pfnCallback =
    new EnumWindowsProcCallback(0, &MyWin32Bridge.DisplayWindowTitle);
```

В нашем примере предполагается, что обработчик обратного вызова DisplayWindowTitle() является статическим методом управляемого класса MyWin32Bridge. Теперь требуется передать параметр, представляющий собой экземпляр делегата, в функцию EnumWindows() с помощью механизма PInvoke:

```
MyWin32Bridge.EnumWindows(pfnCallback, 0);
```

Благодаря этому оператору функция EnumWindowsO будет вызывать обработчик обратного вызова DisplayWindowTitle() всякий раз, как только найдет окно переднего плана. Ниже приведен полный код, демонстрирующий все вышесказанное:

```
using namespace System;
using namespace System::Text;
using namespace System::Runtime::InteropServices;

// Объявление делегата для функции обратного вызова WNDENUMPROC
__delegate bool EnumWindowsProcCallback(long hwnd, long lParam);

// Константа, задающая максимальную длину заголовка окна
#define MAX_TITLE_CHARS 512

__gc class MyWin32Bridge
{
public:
    // Управляемое определение функции API EnumWindows()
    // (библиотека USER32.DLL)
    [DllImport("user32.dll")]
    static bool EnumWindows(EnumWindowsProcCallback* pfnCallback,
                           long lParam);

    // Управляемое определение функции API GetWindowText()
    // (библиотека USER32.DLL)
    [DllImport("user32.dll")]
```

```

static int GetWindowText(long hWnd, StringBuilder* lpstrTitle,
                        int nMaxCount);

// Обработчик обратного вызова представлен делегатом
// EnumWindowsProcCallback. Функция EnumWindows() посылает свой
// обратный вызов этому обработчику.
static bool DisplayWindowTitle(long hWnd, long lParam)

// Создание нового экземпляра класса StringBuilder
StringBuilder* lpstrTitle = new StringBuilder(MAX_TITLE_CHARS);

// Получение заголовка окна
GetWindowText(hWnd, lpstrTitle, MAX_TITLE_CHARS);

// Проверка того, что окно имеет заголовок
if(!lpstrTitle->ToString()->Equals(""))
{
    // Отображение заголовка
    Console.WriteLine(lpstrTitle);
}

return true;
}

int_tmain(void)
{
    // Создание экземпляра делегата EnumWindowsProcCallback и передача
    // ему адреса соответствующего обработчика обратного вызова
    EnumWindowsProcCallback* pfnCallback =
        new EnumWindowsProcCallback(0, &MyWin32Bridge::DisplayWindowTitle);

    // Вызов функции API EnumWindows, содержащейся в библиотеке user32.dll
    MyWin32Bridge::EnumWindows(pfnCallback, 0);

    return 0;
}

```

Использование **PInvoke** для передачи структур функциям, содержащимся в неуправляемых DLL

Встречаются ситуации, когда управляемый код должен вызывать неуправляемую функцию, которая в качестве аргумента принимает неуправляемую структуру или определенный пользователем тип данных. В таких случаях необходимо создавать управляемый класс или структуру, которая физически таким же образом располагается в памяти и имеет такой же порядок следования байтов, как и неуправляемая структура. Тогда можно передавать управляемую функцию вместо неуправляемой структуры. Имеются дополнительные возможности для управления физическим размещением в памяти управляемого класса или структуры благодаря атрибуту `StructLayout` (см. главу 4).

Теперь рассмотрим неуправляемую библиотеку Win32 DLL, которая экспортирует функцию `GetDepartmentDetails()`. Эта функция заносит данные в структуру, называемую `DEPARTMENT_DETAILS`, которая в конечном счете будет передаваться в оператор вызова из управляемого приложения C++. Ниже приведен код функции `GetDepartmentDetails()`, содержащейся в библиотеке `DepartmentDetails.dll`:

```

#include "stdafx.h"
#include <tchar.h>

BOOL WINAPI DllMain(HANDLE hModule, DWORD ul_reason, LPVOID lpReserved)

```

```
{
    return TRUE;
}
```

```
typedef struct tagDEPARTMENT_DETAILS
```

```
{
    DWORD m_dwEmployeeID;
    BOOL m_bIsEmployeeManager;
    CHAR m_chEmployeeSection;
    INT m_nDepartmentNumber;
    WORD m_wDivisionNumber;
    DWORD m_dwReportingManagerID;
} DEPARTMENT_DETAILS;
```

```
extern "C" __declspec(dllexport) void __stdcall
GetDepartmentDetails(int nEmployeeID, DEPARTMENT_DETAILS* pDetails)
```

```
{
    pDetails->m_dwEmployeeID = 10236;
    pDetails->m_bIsEmployeeManager = TRUE;
    pDetails->m_chEmployeeSection = 'A';
    pDetails->m_nDepartmentNumber = 3;
    pDetails->m_wDivisionNumber = 726;
    pDetails->m_dwReportingManagerID = 34752;
}
```

ДЛЯ ТОГО чтобы максимально упростить функцию `GetDepartmentDetails()`, мы заносим в структуру `DEPARTMENT_DETAILS` набор величин, которые всегда одинаковы. При этом не нужно обращать внимание на определяемый первым параметром идентификационный номер служащего (`EmployeeID`), для которого должны быть получены данные о подразделении (`DEPARTMENT_DETAILS`).

Создадим теперь управляемое приложение C++, вызывающее эту функцию с помощью механизма `PInvoke`. Во-первых, необходимо воссоздать структуру `DEPARTMENT_DETAILS` с точно таким же физическим размещением в памяти. Ниже представлено определение управляемого класса `DepartmentDetails`, причем его члены организованы так, что их физическое размещение в памяти совпадает с расположением неуправляемой структуры `DEPARTMENT_DETAILS`.

```
[StructLayout(LayoutKind::Sequential)]
public class DepartmentDetails
{
public:
    unsigned long m_lEmployeeID;
    bool m_bIsEmployeeManager;
    char m_chEmployeeSection;
    int m_nDepartmentNumber;
    unsigned short m_wDivisionNumber;
    unsigned long m_lReportingManagerID;
};
```

Если при объявлении членов управляемого класса используются управляемые типы данных, которые соответствуют неуправляемым типам членов неуправляемой структуры, это служит гарантией того, что порядок следования байтов и их расположение в памяти в обоих случаях будут одинаковыми. Сделав всю подготовительную работу, создадим управляемое определение для функции `GetDepartmentDetails()`, которая экспортируется из библиотеки `DepartmentDetails.dll`:

```

__gc class DepartmentDetailsBridge
{
public:
    [DllImport("DepartmentDetails.dll")]
    static void GetDepartmentDetails(int nEmployeeID,
                                     DepartmentDetails* portable);
};

```

Можно вызвать функцию GetDepartmentDetails(), передав ей указатель на управляемый класс DepartmentDetails вместо неуправляемой структуры DEPARTMENT_DETAILS:

```

DepartmentDetails* pObjDetails = new DepartmentDetails;
DepartmentDetailsBridge::GetDepartmentDetails(1011, pObjDetails);

```

Ниже приведен полный исходный код управляемого приложения C++, которое вызывает неуправляемую функцию, принимающую в качестве параметра неуправляемую структуру:

```

#using <mscorlib.dll>
#include <tchar.h>

using namespace System;
using namespace System::Text;
using namespace System::Runtime::InteropServices;

// Определение управляемого класса, размещаемого в памяти как структура
[StructLayout(LayoutKind::Sequential)]
class DepartmentDetail
{
public:
    unsigned long m_lEmployeeID;
    bool m_bIsEmployeeManager;
    char m_cEmployeeSection;
    int m_nDepartmentNumber;
    unsigned short m_wDivisionNumber;
    unsigned long m_lReportingManagerID;
};

__gc class DepartmentDetailsBridge
{
public:
    [DllImport("DepartmentDetails.dll")]
    static void GetDepartmentDetails(int nEmployeeID,
                                     DepartmentDetails* pDetails);
};

```

```

// Создание экземпляра объекта DepartmentDetails
DepartmentDetails* pObjDetails = new DepartmentDetails;

// Вызов неуправляемой функции для получения сведений о подразделении
DepartmentDetailsBridge::GetDepartmentDetails(1011, pObjDetails);

// Вывод результатов
Console.WriteLine($Employee ID: {0}",
                  pObjDetails->m_lEmployeeID.ToString());
Console.WriteLine($Is Manager: {0}",
                  pObjDetails->m_bIsEmployeeManager ? "YES" : "NO");

```

```

Console.WriteLine(S"Belongs to: {0}",
    pObjDetails->m_chEmployeeSection == 'A' ? S"Section A": S"Section 8" );
Console.WriteLine(S"Dept. Number: {0}",
    pObjDetails->m_nDepartmentNumber.ToString());
Console.WriteLine(S"Division Number: {0}",
    pObjDetails->m_wDivisionNumber.ToString());
Console.WriteLine(S"Reporting Manager ID: {0}",
    pObjDetails->m_lReportingManagerID.ToString());

return 0;
}

```

После запуска приложения на экран будет выведена следующая информация, подтверждающая, что все работает так, как описывалось выше:

```

Employee ID: 10236
Is Manager: YES
Belongs to: Section A
Dept. Number: 3
Division Number: 726
Reporting Manager ID: 34752

```

Явный маршалинг для "неблитуемых" типов данных

Как говорилось выше, "блитуемые" типы данных имеют одинаковое представление в управляемой и в неуправляемой памяти. Иными словами, они не требуют явного преобразования, если участвуют в процессе маршалинга между управляемым и неуправляемым кодом. Если передать в неуправляемый код структуру, содержащую только "блитуемые" типы, вся структура будет "блитуемой". То же самое верно для массива, чьи элементы относятся к "блитуемым" типам данных. Ниже перечислены некоторые общие типы данных, которые являются "блитуемыми":

- ELEMENT_TYPE_I1(char)
 - ELEMENT_TYPE_UI1(unsigned char)
- ELEMENT_TYPE_I2(short)
 - a ELEMENT_TYPE_UI2(unsigned short)
 - ELEMENT_TYPE_I4(int, long)
- ELEMENT_TYPE_UI4(unsigned int, unsigned long)
- D ELEMENT_TYPE_R4(float)
- ELEMENT_TYPE_I8(__int64)
- a ELEMENT_TYPE_U8(unsigned __int64)
 - ELEMENT_TYPE_R8(double)
- D ELEMENT_TYPE_PTR(void)

С другой стороны, "неблитуемые" типы данных имеют множественные представления в неуправляемой памяти для заданных управляемых типов. Например, такой управляемый тип, как String, может быть представлен в неуправляемой памяти строками ANSI (LPSTR), Unicode (LPWSTR) или BSTR. В следующей таблице приведены некоторые "неблитуемые" типы данных и их возможное представление в неуправляемой памяти:

"Неблитуемый" тип	Возможные сценарии преобразования
ELEMENT_TYPE_BOOLEAN	BOOL (4 байта), VARIANT_BOOL (2 байта)
ELEMENT_TYPE_CHAR	Символ ANSI, символ Unicode
ELEMENT_TYPE_STRING	Строка ANSI (LPSTR), строка Unicode (LPWSTR), BSTR
ELEMENT_TYPE_VALUTYPE	Struct (пользовательский тип с фиксированным размещением в памяти)
ELEMENT_TYPE_CLASS	Class, interface
ELEMENT_TYPE_OBJECT	Interface, VARIANT
ELEMENT_TYPE_ARRAY	Interface, ~~SAFEARRAY~~

Можно использовать атрибут `MarshalAs` для указания того, как "неблитуемый" тип должен преобразовываться при передаче в неуправляемую память. Предположим, например, что для обращения строки необходимо вызвать функцию `C_wcsrev()` из библиотеки периода выполнения. Приведем сигнатуру этой функции:

```
wchar_t* _wcsrev(wchar_t* string);
```

В качестве параметра функция принимает строку Unicode и выдает обращенную строку того же формата. Если необходимо вызвать эту функцию из управляемого приложения C++ с помощью механизма `PInvoke`, следует задать инструкцию механизму маршалинга, чтобы тот преобразовал строку в формат Unicode (`LPWSTR`) для передачи в неуправляемую функцию. Затем необходимо выполнить обратное преобразование строки, возвращаемой в формате Unicode. Следующий код демонстрирует управляемое определение для функции `_wcsrev()` в управляемом приложении C++:

```
public _gc_class MyPInvokeBridge
{
    public
        [DllImport("msvcrt.dll"), MarshalAs(UnmanagedType.LPWSTR)]
        static String _wcsrev([MarshalAs(UnmanagedType.LPWSTR)] String*);
};
```

Вначале следует использовать атрибут `DllImport`, чтобы задать библиотеку `msvcrt.dll`, экспортирующую функцию `_wcsrev()`. Затем с помощью атрибута `MarshalAs` можно указать, что параметр функции и возвращаемый тип должны быть преобразованы, причем в преобразовании будет участвовать неуправляемый тип `LPWSTR` (соответствующий формату Unicode). Приведенная ниже программа вызывает функцию `_wcsrev()`:

```
int main(void)
{
    String* pstrReversed = MyPInvokeBridge._wcsrev(S"Hello there!");
    Console.WriteLine(pstrReversed);
    return 0;
}
```

Отметим, однако, что (как всегда!) при передаче "неблитуемых" типов в неуправляемые функции имеет место определенная потеря производительности из-за преобразования, которое должно быть выполнено механизмом маршалинга.

Вопросы производительности

На протяжении всей данной главы у нас постоянно возникал вопрос: "А во что все это обойдется?" Microsoft в своей документации утверждает, что использование

`PInvoke` "сопровождается издержками, составляющими от 10 до 30 инструкций для процессора x86 на один вызов". Автор пришел к выводу, что снижение производительности в значительной степени зависит от типа и объема выполняемого маршалинга.

В спецификации Microsoft Partition II Metadata (раздел 14.5.6.3) сказано: "Переходы от управляемого кода к неуправляемому связаны с небольшими дополнительными издержками, необходимыми для того, чтобы позволить механизмам исключительных ситуаций и сборки мусора корректно определять контекст выполнения. На процессоре x86 в наилучших условиях для этих переходов между управляемым и неуправляемым кодом необходимо примерно 5 инструкций на один вызов/возврат. Кроме того, любой метод, включающий вызовы с переходами, требует 8 дополнительных инструкций, распределенных между прологом и эпилогом вызывающего метода". "Наилучшие условия", о которых говорит Microsoft, имеют место тогда, когда не используется маршалинг.

Общая рекомендация по повышению производительности состоит в том, что лучше реже вызывать механизм `PInvoke` с большим объемом данных, нежели чаще вызывать этот метод для работы с малыми порциями данных. Важно отметить, что, даже если применяется маршалинг, издержки, связанные с работой служб `PInvoke` и созданием оболочек C++, меньше, чем при взаимодействии с COM. Причина состоит в том, что в первом случае используется на один уровень меньше. Механизм COM interop должен заключать в оболочку класс C++, а также генерировать оболочку RCW (см. следующую главу).

ГЛАВА 8



Интероперабельность COM

Компонентное ПО обеспечивает повторное использование кода, модульность и интероперабельность (способность к взаимодействию), а также целую гамму других общепризнанных преимуществ. Это побудило многие компании использовать компонентные платформы, такие как COM и CORBA, для решения задач по разработке ПО. Стимулом к применению компонентного подхода служит также его поддержка средствами разработки, которые могут генерировать значительную часть стандартного кода для удовлетворения требований объектных моделей. Это позволяет разработчикам сконцентрироваться на бизнес-логике приложений, а не погружаться в рутинное написание кода. Существует огромный спрос на программные компоненты, и организации вкладывают в разработку компонентного ПО большие средства, рассматривая эту деятельность как главное требование бизнеса, выполнение которого обязательно для сохранения конкурентоспособности на рынке ПО.

Тот, кто когда-либо занимался разработками с использованием технологий Microsoft, так или иначе обязательно имел дело с COM. До появления .NET технология COM была краеугольным камнем деятельности Microsoft по развитию компонентного подхода к созданию ПО, на основе которого разработаны ключевые службы платформы Windows. Теперь же перед разработчиками встают вопросы: не приведет ли появление технологии .NET к снижению роли COM и какое место будет занимать COM в новом порядке вещей, установленном платформой .NET Framework.

В настоящей главе мы попытаемся ответить на эти вопросы. Вначале основное внимание мы уделим способам повторного использования компонентов в управляемых приложениях C++. После этого мы подойдем к проблеме с другой стороны и обсудим, как неуправляемые клиенты, распознающие COM, могут работать с управляемыми компонентами C++.

Переход от COM к .NET

Технология COM реализует исключительно элегантную объектную модель и широко применяется корпорациями для разработки повторно используемых

программных компонентов. Компании всего мира интенсивно инвестируют в создание приложений и компонентов на основе COM.

С появлением платформы .NET Framework новое воплощение технологии компонентно-ориентированного программирования оказалось более простым и позволяющим разработчикам создавать компоненты в более короткие сроки. Компании, которые долгое время добивались успеха на волне COM, должны отдавать себе отчет в том, что технология COM не движется вперед в направлении .NET. Платформа .NET поддерживает программирование, основанное на модели интерфейсов, с которой нас познакомила COM. Однако парадигма разработки компонентов, реализованная в .NET, имеет мало общего с такими чувствительными точками COM, как записи в реестре, проблемы, связанные с управлением версиями, необходимость работать с загадочными интерфейсами IUnknown, IDispatch и т.д.

Для того чтобы получить все преимущества, предлагаемые моделью программирования .NET и средой CLR, программисты для написания своих компонентов могут использовать любой из .NET-совместимых языков разработки. Выходящие в свет продукты Microsoft, а также грядущие решения этой корпорации будут тесно связаны с платформой .NET, давая компаниям все больше оснований для того, чтобы пересмотреть свою стратегию разработки компонентного ПО и обеспечить эффективный перенос существующих компонентов на платформу .NET.

Необходимость взаимодействия

Некоторые основные службы, необходимые для бизнес-компонентов .NET и приложений масштаба предприятия, такие как автоматическая обработка транзакций, работа с пулами объектов и активизация по требованию, все еще обеспечиваются только через сервисы COM+. Более того, хотя все признаки COM-программирования исчезли из платформы .NET, технология COM присутствует на внутренних уровнях CLR. Компании, которые сделали крупные вложения в COM-разработки, вряд ли в одночасье отложат в сторону все свои компоненты и прыгнут на подножку экспресса .NET только потому, что здесь предлагают более совершенную и простую модель программирования. Перенос же всех существующих компонентов на платформу .NET — это поистине геркулесова задача.

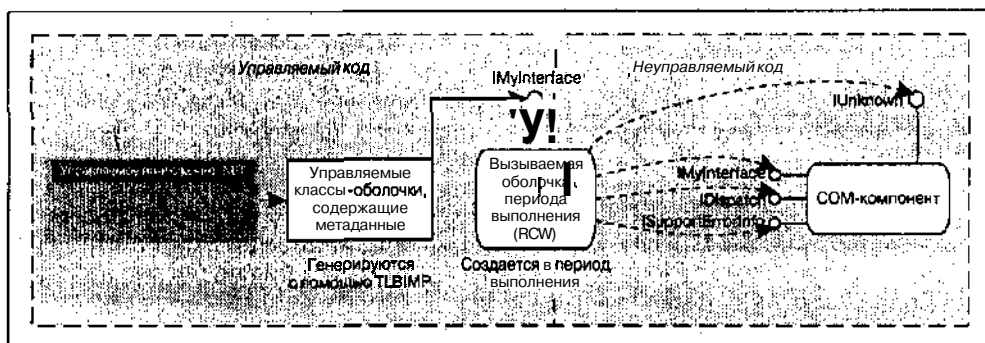
Microsoft осознала эту проблему и поняла необходимость реализации механизма, который обеспечивал бы взаимодействие COM и .NET. В первой половине настоящей главы мы уделим основное внимание использованию данного механизма для того, чтобы унаследованные компоненты COM могли работать в приложениях .NET.

С другой стороны, компании не захотят лишаться тех преимуществ, которые предлагает .NET. Предположим, что наша организация решила последовать за восходящей звездой и приступила к созданию компонентов на платформе .NET, скорее всего, с помощью .NET-совместимого языка, такого как C# или Visual Basic .NET. Мы наверняка захотим, чтобы эти управляемые компоненты были доступны унаследованным приложениям, т.е. неуправляемому коду. И вновь инфраструктура взаимодействия позволяет сделать это бесшовным образом. Во второй половине главы мы подробно разъясним, как можно добиться того, чтобы компоненты .NET предстали в качестве COM-компонентов перед неуправляемыми клиентами, способными работать с COM.

Использование COM-компонентов в управляемом коде C++

Управляемое приложение не может непосредственно работать с COM-компонентом, поскольку оно совершенно ничего не знает об интерфейсах IUnknown и

IDispatch. К счастью, .NET Framework делает процедуру использования в приложениях .NET компонентов COM настолько простой, будто эти компоненты являются управляемыми. В этом случае "обманутое" приложение .NET считает, что оно имеет дело с родственным объектом. Прежде чем подробно рассмотреть, за счет чего создается такая иллюзия, обратимся к рисунку, поясняющему процесс взаимодействия между управляемым клиентом и компонентом COM:



Приложение .NET общается с компонентом COM только через управляемые классы-оболочки, которые содержат метаданные, описывающие классы, интерфейсы, методы, свойства и события, связанные с COM-компонентом. Эти оболочки генерируются с помощью утилиты **type library importer** (средство импорта библиотеки типов, файл `tlbimp.exe`), которая представляет собой часть платформы .NET Framework. После этого мы можем создавать и вызывать методы COM-компонентов, создавая и вызывая соответствующие методы в классах-оболочках так, будто мы обращаемся к методу управляемого компонента.

Каким же образом эти управляемые оболочки взаимодействуют с COM-компонентами? Когда мы создаем COM-компонент через посредство класса-оболочки, то во время выполнения среда CLR генерирует некоторый промежуточный слой, известный как вызываемая оболочка периода выполнения (**runtime callable wrapper — RCW**). RCW действует как клиент COM-компонента. При этом целью вызова метода, выполняемого управляемым клиентом, является не сам компонент COM, а оболочка RCW. RCW знает, как преобразовать управляемый вызов в специфичную форму запроса COM путем обращения к необходимым COM-интерфейсам, так что приложение .NET уверено, будто оно взаимодействует с управляемым компонентом .NET. При этом COM-компонент полагает, что он вызывается неуправляемым клиентом, ориентированным на работу с COM. Среда CLR создает отдельную оболочку RCW для каждого экземпляра COM-компонента.

Оболочка RCW осуществляет маршalling управляемых типов в типы данных, с которыми работают COM-компоненты, а также выполняет обратное преобразование типов, возвращаемых компонентами COM. Кроме того, RCW трансформирует в исключительные ситуации .NET коды ошибок **HRESULT**, возвращаемые компонентами COM. RCW управляет временем жизни COM-компонента, используя подсчет ссылок. Когда управляемому клиенту более не нужны управляемые объекты-оболочки, содержащие метаданные, оболочка RCW помечается соответствующим образом для удаления сборщиком мусора. В конечном счете среда CLR удаляет оболочку RCW, в результате чего освобождаются ссылки со стороны RCW на COM-компонент и последний удаляется из неуправляемой памяти.

Продemonстрируем теперь, как все эти механизмы работают на практике. Мы обсудим различные сценарии взаимодействия между COM и .NET. В настоящем разделе рассматриваются следующие вопросы:

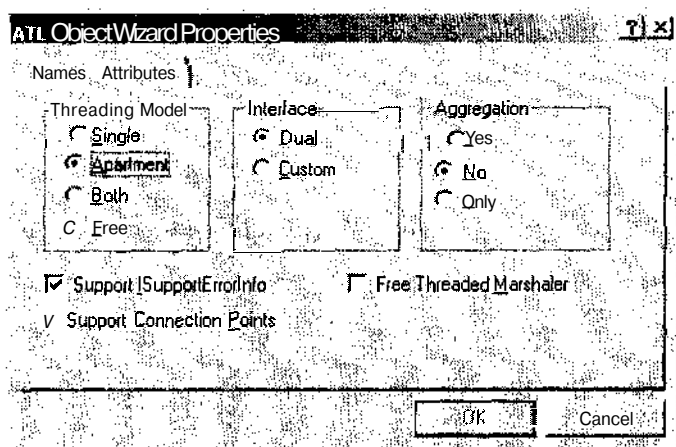
- D Использование раннего и позднего связывания при работе управляемого кода C++ с компонентами COM
- O Применение управляемого кода C++ для приема уведомлений о событиях, инициируемых компонентами COM
- ▣ Работа управляемых приложений C++ с классами коллекций COM
 - Повторное использование элементов управления ActiveX в контейнерах .NET, таких как формы Windows
- ▣ Применение в управляемом коде моделей повторного использования для COM-компонентов
- O Объявление управляемыми потоками предпочтений, указывающих типы "апартаментов" COM

Вызов компонента COM из среды .NET

С помощью библиотеки ATL 3.0 создадим простой, классический компонент COM, который назовем MyTimeKeeper. Этот компонент представляет собой ежедневный табель учета количества часов, которые отработал служащий. Затем мы покажем, как на основе библиотеки типов компонента создается сборка для оболочки, содержащей метаданные. После этого обсудим использование компонента COM MyTimeKeeper в управляемом приложении C++.

Создание компонента COM

Приступим к созданию компонента MyTimeKeeper с помощью библиотеки ATL. Откроем окно мастера ATL COM AppWizard в интегрированной среде разработки Visual C++ 6.0 IDE. Выберем опцию Dynamic Link Library (DLL), которая устанавливается по умолчанию, и щелкнем на кнопке Finish. Добавим в этот проект новый объект COM с помощью известной нам команды меню insert | New ATL Object, выберем тип Simple Object и назовем объект MyTimeKeeper. Чтобы сделать компонент максимально простым, выберем следующие атрибуты:



Компонент `MyTimeKeeper` поддерживает два интерфейса: `IMyTimeKeeper` и `ITimeTeller`. Первый из них генерируется мастером автоматически, а второй нужно добавить в файл IDL вручную. Кроме того, следует соответствующим образом изменить определение класса `СMyTimeKeeper`, что вскоре и будет сделано. Ниже представлены изменения в файле IDL и определение `coclass`:

```
[
    object,
    uuid(E5A89168-4E1F-41BE-8017-92D5D8C69845),
    dual,
    nonextensible,
    helpstring("IMyTimeKeeper Interface"),
    pointer_default(unique)
]
interface IMyTimeKeeper : IDispatch
{
    [id(1), helpstring("method GetEmployeeWorkHours")]
    HRESULT GetEmployeeWorkHours([in] long lEmpCode,
                                  [out,retval] float* pVal);
};

[
    object,
    uuid(B3866080-D898-4d59-90E8-0E6E952D8551),
    dual,
    helpstring("ITimeTeller Interface"),
    pointer_default(unique)
]
interface ITimeTeller : IDispatch
{
    [propget, id(1), helpstring("property CurrentTime")]
    HRESULT CurrentTime([out, retval] BSIR* pVal);
};

[
    uuid(A353FD49-E94E-42CE-A599-8FB121807900),
    version(1.0),
    helpstring("MyTimeKeeper 1.0 Type Library")
]
library MyTimeKeeperLib
{
    importlib("stdole2.tlb");
    [
        uuid(50CEDB6E-475E-4D29-B314-ABDDF27A31CA),
        helpstring("MyTimeKeeper Class")
    ]
    coclass MyTimeKeeper
    {
        [default] interface IMyTimeKeeper;
        interface ITimeTeller;
    };
};
```

Интерфейс `IMyTimeKeeper` содержит метод `GetEmployeeWorkHours()`, который возвращает количество часов, отработанных служащим, исходя из задаваемого кода служащего. Интерфейс `ITimeTeller` включает нередактируемое свойство `CurrentTime`, которое возвращает текущее время.

Приведем теперь определение класса `СMyTimeKeeper`:

```

// Определение карты, которая будет сохранять код служащего и
// количество часов, отработанных сегодня этим служащим
typedef map<long, float> EmployeeWorkHoursMap;

class ATL_NO_VTABLE CMyTimeKeeper
{
public:
    CComObjectRootEx<CComSingleThreadModel>,
    public CComCoClass<CMyTimeKeeper, &CLSID_MyTimeKeeper>,
    public ISupportErrorInfo,
    public IDispatchImpl<IMyTimeKeeper, &IID_IMyTimeKeeper,
        " &LIBID_TIMEKEEPERLib>,
    public IDispatchImpl<ITimeTeller, &IID_ITimeTeller,
        &LIBID_TIMEKEEPERLib>
{
public:
    CMyTimeKeeper(){}
    HRESULT FinalConstruct();

    // Здесь располагаются макросы, создаваемые мастером.
    // Для краткости опущены.

    BEGIN_COM_MAP(CMyTimeKeeper)
        COM_INTERFACE_ENTRY(IMyTimeKeeper)
        COM_INTERFACE_ENTRY2(IDispatch, IMyTimeKeeper)
        COM_INTERFACE_ENTRY(ISupportErrorInfo)
        COM_INTERFACE_ENTRY(ITimeTeller)
    END_COM_MAP()

private:
    EmployeeWorkHoursMap m_mapEmployeeWorkHours;

public:
    // Интерфейс ISupportsErrorInfo
    STDMETHOD(InterfaceSupportsErrorInfo)(REFIID riid);

    // Методы интерфейса IMyTimeKeeper
    STDMETHOD(GetEmployeeWorkHours)(/*[in]*/ long lEmpCode,
        /*[out, retval]*/ float* pVal);

    // Методы интерфейса ITimeTeller
    STDMETHOD(get_CurrentTime)(/*[out, retval]*/ BSTR* pVal);
};

```

Для выполнения необходимой инициализации мы подменяем метод FinalConstruct(). Добавим в экземпляр карты EmployeeWorkHoursMap коды сотрудников и количество отработанных ими часов.

```

HRESULT CMyTimeKeeper::FinalConstruct()
{
    // Добавляем в экземпляр карты EmployeeWorkHoursMap коды сотрудников
    // и количество отработанных ими часов
    m_mapEmployeeWorkHours[1001] = 8.0;
    m_mapEmployeeWorkHours[1002] = 8.5;
    m_mapEmployeeWorkHours[1003] = 7.5;
    m_mapEmployeeWorkHours[1004] = 6.0;

    // Добавляем часы, отработанные другими сотрудниками
    return S_OK;
}

```

Реализация метода IMyTimeKeeper::GetEmployeeWorkHours() ищет в заранее заполненной карте заданный код сотрудника. Если такой код существует, метод

возвращает количество часов, отработанных сотрудником. В противном случае выдается исключительная ситуация с кодом ошибки TK_E_EMPLOYEE_CODE_NOT_FOUND, а также соответствующее описание ошибки с помощью функции AtlReportError().

Функция AtlReportError() предоставляет расширенную информацию об ошибке, используя объект ошибки, который реализует интерфейсы ICreateErrorInfo и IErrorInfo. Функция использует интерфейс ICreateErrorInfo для занесения информации об ошибке в объект ошибки и позволяет клиентам получать эту информацию с помощью интерфейса IErrorInfo. Ниже приведен код метода GetEmployeeWorkHours():

```
STDMETHODIMP CMvTimeKeeper::GetEmployeeWorkHours(long lEmpCode,
                                                    float* pWorkedHours)
{
    // Проверка, является ли указатель действительным
    if(NULL == pWorkedHours)
        return E_POINTER;

    // Объявляем итератор для карты, содержащей информацию о сотрудниках
    EmployeeWorkHoursMap::iterator DOS;

    // Попытка найти сотрудника по его коду
    DOS = m_mapEmployeeWorkHours.find(lEmpCode);

    // Найдено ли совпадение?
    if(DOS != m_mapEmployeeWorkHours.end())
    {
        // Присваиваем возвращаемому значению количество часов, отработанных
        // сотрудником
        *pWorkedHours = DOS->second;
    }
    else
    {
        // II 0! Мы не нашли сотрудника по заданному коду
        return AtlReportError( uuidof(MvTimeKeeper),
                               OLESTR("Specified employee code does not exist"),
                               uuidof(IMvTimeKeeper), TK_E_EMPLOYEE_CODE_NOT_FOUND);
    }

    // Возвращаем состояние
    return S_OK;
}
```

Код ошибки TK_E_EMPLOYEE_CODE_NOT_FOUND должен быть обязательно определен путем добавления в файл IDL строки, которая обеспечивает экспорт кода ошибки в заголовочный файл, генерируемый компилятором MIDL:

```
cpp_quote("#define TK_E_EMPLOYEE_CODE_NOT_FOUND 0x80040001L")
```

Реализация свойства CurrentTime интерфейса ITimeTeller также очень проста. Данное свойство возвращает только текущее время:

```
STDMETHODIMP CMvTimeKeeper::GetCurrentTime(BSTR* pbstrCurrTime)
{
    // Проверка, является ли указатель действительным
    if(NULL == pbstrCurrTime)
        return E_POINTER;

    // Получение локального времени
    SYSTEMTIME sysTime;
    GetLocalTime(&sysTime);

    // Форматирование текущего времени
    TCHAR strCurrentTime[255] = {0};
    sprintf(strCurrentTime, "%d-%d-%dT%d:%d:%d",
            sysTime.wYear, sysTime.wMonth, sysTime.wDay,
```

```

        sysTime.wHour, sysTime.wMinute, sysTime.wSecond));

// Присвоение возвращаемому значению текущего времени
CComBSTR bstr_CurrentTime(st_CurrentTime);
*pbstr_CurrentTime = bstr_CurrentTime.Detach();

// Возвращение состояния
return S_OK;
}

```

И это все, что нужно было сделать! Теперь скомпилируем проект. Скоро мы покажем, как можно заставить управляемое приложение C++ общаться с нашим COM-компонентом. Код проекта, естественно, можно найти на Web-сайте, где представлены все загружаемые файлы, используемые в настоящей книге.

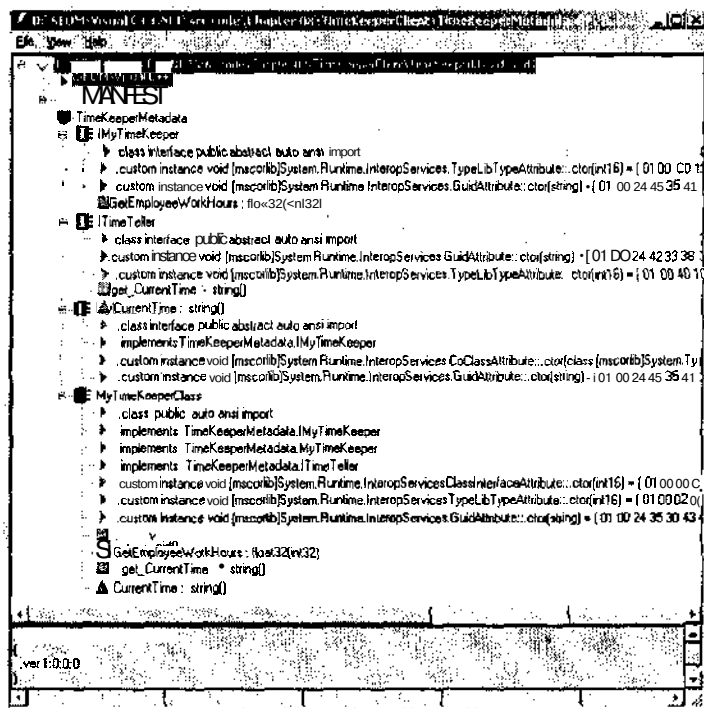
Утилита импорта библиотек типов TLBIMP

Если нужно, чтобы код .NET использовал COM-компонент, следует в первую очередь создать сборку для управляемой оболочки, содержащей метаданные, на основе информации о типах, представленной в библиотеке типов компонента. Набор средств разработки .NET Framework SDK включает инструмент, который выполняет данную работу. Это утилита импорта библиотеки типов. Для ее активизации необходимо набрать в командной строке следующий текст:

```
> tlbimp TimeKeeper.tlb /out:TimeKeeperMetadata.dll
```

Используя библиотеку типов компонента TimeKeeper.tlb, эта команда генерирует сборку TimeKeeperMetadata.dll, в которой находится оболочка, содержащая метаданные.

Воспользуемся утилитой ILDasm, чтобы просмотреть классы сборки TimeKeeperMetadata.dll, содержащие метаданные, и увидеть, как утилита tlbimp преобразовала данные из библиотеки типов в информацию о типах .NET:



ГЛЯДЯ на этот рисунок, можно сделать ряд важных выводов:

- а Утилита импорта библиотеки типов поместила все сгенерированные файлы в пространство имен `TimeKeeperMetadata`. По умолчанию это имя совпадает с именем сборки, в которой находится оболочка, содержащая метаданные. Однако можно создать пространство имен по своему выбору, используя опцию командной строки `/namespace` при запуске утилиты `tlbimp`.
- Интерфейсы `IMyTimeKeeper` и `ITimeTeller` импортированы в сборку, содержащую оболочку, как управляемые интерфейсы и сохраняют свои первоначальные имена.
- ❑ Когда `coclass` импортируется в сборку, средство импорта библиотеки типов генерирует *два* управляемых класса-оболочки:
 - Абстрактный класс, ассоциированный с тем же идентификатором `GUID`, что и интерфейс по умолчанию `coclass COM`. Когда утилита `tlbimp` генерирует имя этого класса, она использует имя `coclass`. В нашем примере классу присваивается имя `MyTimeKeeper`. Он наследует от интерфейса по умолчанию `IMyTimeKeeper`, но не обеспечивает реализацию методов интерфейса.
 - Конкретный класс, наследующий от абстрактного класса и реализующий все интерфейсы, поддерживаемые `coclass`. Реализации методов этого класса вызывают оболочку `RCW`, которая играет роль прокси `COM`-компонента. Когда утилита импорта библиотеки типов генерирует имя данного класса, она добавляет к имени `coclass` строку `Class`. В нашем примере указанный класс имеет имя `MyTimeKeeperClass`. Для создания `COM`-компонента из управляемого кода необходимо создать экземпляр этого класса.
 - Метод `GetEmployeeWorkHours()` и свойство `CurrentTime` перечислены как открытые члены интерфейсов `IMyTimeKeeper` и `ITimeTeller` соответственно. Причем оба интерфейса реализуются классом `MyTimeKeeperClass`. Утилита `tlbimp` преобразует типы параметров этих методов, используемые в `COM`, в эквивалентные управляемые типы. В частности, метод `GetEmployeeWorkHours()` принимает тип данных `long`, который трансформируется в управляемый тип `Int32`. Видно также, что параметр типа `float` в методе `GetEmployeeWorkHours()`, помеченный как `[out, retval]`, преобразован в возвращаемое значение метода.

Раннее связывание управляемого кода C++ с компонентом **COM**

Итак, на основе библиотеки типов `COM` мы создали сборку, где содержатся управляемые классы-оболочки, включающие метаданные. Разработаем теперь управляемое приложение C++, которое будет создавать и вызывать методы `COM`-компонента. Откроем *новый* проект `Managed C++ Application` и назовем его `TimeKeeperClient`.

Вначале нужно импортировать в наше управляемое приложение C++ информацию о типах, представленную метаданными, которые содержатся в классах-оболочках. Как и при импорте любой другой сборки, это делается с помощью директивы `#using`:

```
#using <mscorlib.dll>
#using "TimeKeeperMetadata.dll"
```

Далее необходимо добавить в код две директивы `using namespace`. Одна из них дает доступ к пространству имен `TimeKeeperMetadata`, которое создано средством импорта библиотеки типов. С помощью второй директивы мы импортируем пространство `System::Runtime::Interopservices`, функциональность которого часто будет использоваться в дальнейшем:

```
using namespace System;
using namespace System::Runtime::Interopservices;
using namespace TimeKeeperMetadata;
```

Для создания экземпляра COM-компонента `MyTimeKeeper` необходимо создать экземпляр класса `MyTimeKeeperClass`, который определен в метаданных оболочки, хранящейся в созданной нами сборке. Чтобы обратиться к методам и свойствам компонента COM, нужно просто вызвать соответствующие методы и свойства объекта `MyTimeKeeperClass`:

```
MyTimeKeeperClass* pTimeKeeper = new MyTimeKeeperClass;

int nEmployeeCode = 1002;
float sWorkHours = pTimeKeeper->GetEmployeeWorkHours(nEmployeeCode);
Console::WriteLine($"Employee {0} has worked today for {1} hours",
    nEmployeeCode.ToString(), sWorkHours.ToString());

Console::WriteLine($"The Time teller says: {0}", pTimeKeeper->CurrentTime);
```

Поскольку класс `MyTimeKeeperClass` реализует оба интерфейса `IMyTimeKeeper` и `ITimeTeller`, можно вызвать любое свойство или метод, принадлежащие к какому-либо из этих интерфейсов, используя указатель на экземпляр класса `MyTimeKeeperClass`. Это делается так же, как при создании и вызове методов объекта C++. В глубине, на внутренних уровнях, оболочка RCW выполняет большой объем работы, позволяющий: реализовывать подсчет ссылок на объект COM, опрашивать интерфейсы, поддерживаемые COM-объектом, осуществлятьmarshaling данных между управляемым и неуправляемым кодом, извлекать расширенную информацию об ошибках из COM-объекта и др. В результате всего этого магическим образом реализуется взаимодействие с COM.

Семантика метода ***Unknown::QueryInterface*** в управляемой среде C++

Для динамической проверки того, поддерживает ли объект COM конкретный интерфейс (как это делается в COM с помощью метода `Unknown::QueryInterface()`) можно воспользоваться операторами `dynamic_cast` или `__try_cast`, попытавшись преобразовать объект к нужному интерфейсному типу. Если операция выполняется успешно, значит вызов метода `QueryInterface()` также оказался успешным, и, следовательно, объект поддерживает нужный интерфейс. В противном случае можно сделать вывод, что вызов метода `QueryInterface()` закончился неудачей. Если применяется оператор `__try_cast` и код не может выполнить преобразование типа, выдается исключительная ситуация `System::InvalidCastException`. Исходя из этого, можно судить о том, был ли успешным вызов метода `QueryInterface()`:

```
IArbitrary* pIArbitrary = 0;
try
{
    ITimeTeller* pITimeTeller = __try_cast<ITimeTeller*>(pTimeKeeper);
    Console::WriteLine($"The Time teller says: {0}", pTimeKeeper->CurrentTime);

    pIArbitrary = __try_cast<IArbitrary*>(pTimeKeeper);
}
catch (InvalidCastException* pCastExcep)
```

```

{
    Console.WriteLine($"Invalid cast exception message: {0}",
        pCastExcep->Message);
}

```

ПОЛНЫЙ КОД проекта TimeKeeperClient, включая расширенную функциональность обработки ошибок, можно загрузить с Web-сайта издательства Wrox. Перед запуском управляемого приложения C++ необходимо поместить сборку TimeKeeperMetadata.dll в тот же самый каталог, где находится исполняемый файл.

Детерминированное освобождение ссылок на СОМ-компоненты со стороны управляемого приложения C++

Когда управляемое приложение создает компонент СОМ, среда CLR генерирует оболочку RCW, которая играет роль управляемого представителя СОМ-компонента, а время жизни RCW полностью контролируется CLR. Однако, поскольку оболочка RCW является фактическим клиентом СОМ-компонента, она отслеживает ссылки на СОМ-компонент (используя классический механизм подсчета ссылок СОМ) до тех пор, пока сборщик мусора не вызовет метод FinalizeO оболочки RCW. Когда запускается метод FinalizeO, оболочка RCW вызывает метод IUnknown::Release компонента СОМ, освобождая выделенную для этого компонента память. Таким образом, время жизни компонента СОМ не детерминировано. Он будет оставаться в памяти до тех пор, пока сборщик мусора не удалит оболочку RCW.

Однако встречаются ситуации, когда СОМ-компоненты занимают ценные ресурсы, такие как дескрипторы файлов, соединения с базами данных и др., которые должны быть своевременно освобождены. В таких случаях можно явным образом заставить оболочку RCW освободить свои ссылки на компонент СОМ и таким образом удалить его из памяти. Это позволяет сделать статический метод ReleaseComObject() класса System::Runtime::InteropServices::Marshal.

Например, следующая строка кода в нашем управляемом клиентском приложении C++ явным образом сообщает оболочке RCW, что она должна освободить свои ссылки на СОМ-компонент MyTimeKeeper:

```

Marshal::ReleaseComObject(pTimeKeeper);

```

Если компонент СОМ взаимодействует с управляемым приложением, лучше попытаться освободить ресурсы, занимаемые этим компонентом, когда они больше не нужны, а не полагаться на неявное освобождение этих ресурсов при уничтожении компонента.

Позднее связывание управляемого кода C++ с компонентом СОМ

В предыдущем разделе мы ссылались из управляемого приложения C++ на сгенерированную оболочку, содержащую метаданные, тем самым реализуя раннее связывание компонента. Эта модель связывания обеспечивает проверку типов во время компиляции и лучшую производительность. Однако бывают случаи, когда нет информации о типе, необходимой для раннего связывания, либо может возникнуть необходимость создать конкретный СОМ-компонент "на лету". Если нужно получить доступ к компоненту СОМ, который поддерживает только чистый диспетчерский интерфейс, то мы ограничены моделью позднего связывания. В этих случаях для решения задач можно использовать отражение .NET.

Для этого необходимо знать идентификаторы СОМ-компонента CLSID или ProgID. Имея такую информацию, можно использовать статические методы GetTypeFromProgID() или GetTypeFromCLSID() класса System::Type, которые позволяют получить информацию об управляемом типе из СОМ-компонента. Ниже приведен пример применения этих методов для извлечения данных о типе из СОМ-компонента MyTimeKeeper:

```
Type* pTypeTimeKeeper = Type::GetTypeFromProgID(S"TimeKeeper.MyTimeKeeper");
```

```
Type* pTypeTimeKeeper = Type::GetTypeFromCLSID(
System::Guid(S"{22077040-DBB2-43E5-A325-2A852C0DF143}"));
```

После получения информации о типе можно создать экземпляр COM-компонента с помощью статического метода `CreateInstance()` класса `System::Activator`:

```
Object* pObjTimeKeeper = Activator::CreateInstance(pTypeTimeKeeper);
```

Для вызова метода или свойства компонента следует использовать метод `InvokeMember()` объекта `System::Type`, созданного ранее на основе идентификаторов `ProgID` или `CLSID`. Ниже представлен прототип функции `InvokeMember()`:

```
Object* InvokeMember(String*, BindingFlags, Binder*, Object*, Object* []);
```

Эта функция принимает пять параметров. Первый из них задает имя метода или свойства. Второй представляет собой перечисление `System::Reflection::BindingFlags` и указывает, вызывается ли метод или задается/возвращается значение свойства. Ниже представлены наиболее употребительные значения перечисления `BindingFlags`:

Действие	Используемые флаги связывания
Вызов метода	<code>BindingFlags::InvokeMethod</code>
Получение значения свойства	<code>BindingFlags::GetProperty</code>
Установка значения свойства	<code>BindingFlags::SetProperty</code>

Третий параметр — это указатель на экземпляр объекта `Binder`, который может быть использован для выбора между множественными перегруженными функциями и приведением типов. Данный параметр применяется редко. Обычно вместо него подставляется значение по умолчанию, которое указывается с помощью статического свойства `DefaultBinder` класса `System::Type`.

Четвертый параметр функции представляет собой указатель на экземпляр типа `Object`, который является целью вызова метода. Иными словами, этот параметр задает объект, чей метод или свойство мы собираемся использовать. В качестве данного параметра следует передавать указатель типа `Object`-, который возвращается методом `Activator::CreateInstance()`.

Наконец, пятый параметр — это массив указателей типа `Object`. Данный массив содержит упакованные значения параметров метода (если вызывается метод) либо упакованное значение свойства (когда задается свойство).

Следующий код вызывает метод `GetEmployeeWorkHours()`, определенный в интерфейсе по умолчанию `IMyTimeKeeper` COM-компонента `MyTimeKeeper`:

```
int nEmployeeCode = 1002;
Object* nObjArgsArr[] = new Object*[1];
pObjArgsArr[0] = __box(nEmployeeCode);

Object* pRetVal = pTypeTimeKeeper->InvokeMember(S"GetEmployeeWorkHours",
BindingFlags::InvokeMethod, Type::DefaultBinder,
pObjTimeKeeper, pObjArgsArr);
```

Метод `InvokeMember()` выдает указатель на объект, который содержит упакованное значение, возвращаемое методом, или упакованную величину свойства. Необходимо распаковать данный указатель, чтобы получить фактическую величину. Для этого следует использовать оператор `dynamic_cast` или `__try_cast`. Возвращаемое значение метода `GetEmployeeWorkHours()` принадлежит к типу данных `float`. Поэтому указатель на объект следует распаковывать, используя управляемый тип `Single`:

```
float sEmpWorkHours = dynamic_cast<Single>>(pRetVal);
Console.WriteLine("Employee {0} has worked today for {1} hours",
    employeeCode.ToString(), sEmpWorkHours.ToString());
```

Полный код данной версии клиентского приложения находится в папке MyTime-Keeper ClientLate загружаемого исходного кода.

Передача в приложения .NET уведомлений о событиях компонента COM

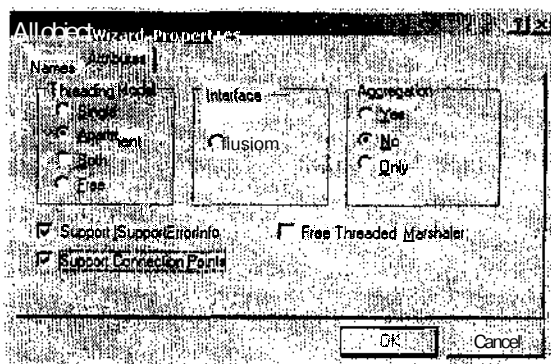
Парадигма программирования на основе интерфейсов, принятая в COM, позволяет организовывать двунаправленные коммуникации между компонентом и его клиентом. Это дает возможность клиенту зарегистрировать указатель на интерфейс своего приемника события для получения уведомлений о событии от компонента-источника. Компонент может затем использовать этот указатель на интерфейс с целью обратного вызова клиентского приемника события, который обрабатывает уведомление о событии. Когда клиент более не хочет получать такие уведомления, он может отменить регистрацию. **Точки подключения COM (connection points)** определяют стандартный протокол программирования для регистрации (и отмены регистрации) интерфейса приемника события, которому компонент посылает обратный вызов.

С другой стороны, в мире .NET уведомления о событиях передаются через делегатов. В данном разделе мы покажем, как управляемое приложение C++ может подписываться на уведомления о событиях и получать их от неуправляемых компонентов COM, являющихся источниками событий.

Создание COM-компонента, иницилирующего события, с помощью библиотеки ATL 3.0

Мы кратко опишем создание COM-компонента — источника событий. На этот раз с помощью COM-объекта мы смоделируем термостат. Объект будет ставить в известность своих подписчиков о том, что температура вышла за пределы установленного диапазона.

В интегрированной среде разработки Visual Studio 6.0 с помощью мастера ATL COM AppWizard создадим новый проект ThermostatApp, выбрав опцию Executable. Добавим в этот проект новый простой объект ATL COM с именем Thermostat (позже мы покажем, как превратить его в одноэлементный объект, чтобы все клиенты могли связываться с одним и тем же экземпляром). Нужно выбрать опцию Support Connection Points (Поддержка точек подключения), чтобы мастер добавил исходящий интерфейс в компонент Thermostat:



Компонент Thermostat реализует входящий интерфейс IThermostat, а также посылает уведомления о событиях через свой исходящий интерфейс _IThermostatEvents. Интерфейс IThermostat поддерживает два метода: SetCurrentTemperature() и GetCurrentTemperature():

```
interface IThermostat : IDispatch
```

```
{
    [id(1), helpstring("method SetCurrentTemperature")]
    HRESULT SetCurrentTemperature([in] float fTemperature);

    [id(2), helpstring("method GetCurrentTemperature")]
    HRESULT GetCurrentTemperature([out, retval] float* pTemperature);
};
```

Исходящий интерфейс _IThermostatEvents поддерживает два типа уведомлений. Уведомление первого типа отправляется в случае, когда температура превышает верхний допустимый предел, а второй тип означает, что температура оказалась меньше нижнего установленного предела. Эти ситуации соответственно представлены методами уведомлений о событиях OnExceededUpperThreshold и OnExceededLowerThreshold():

```
dispinterface _IThermostatEvents
```

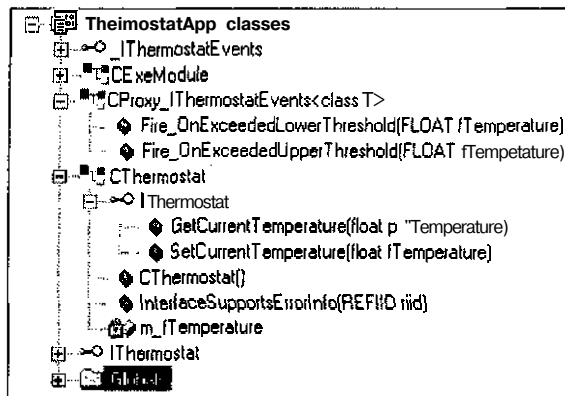
```
{
    properties:
    methods:
    [id(1), helpstring("method OnExceededUpperThreshold")]
    void OnExceededUpperThreshold([in] float fTemperature);

    [id(2), helpstring("method OnExceededLowerThreshold")]
    void OnExceededLowerThreshold([in] float fTemperature);
};
```

Обратимся к определению coclass в файле IDL. Интерфейс _IThermostatEvents помечен атрибутом [source], который указывает, что интерфейс является исходящим:

```
coclass Thermostat
{
    [default]
    [default, source] dispinterface _IThermostatEvents;
};
```

Откомпилируем проект, чтобы сгенерировать библиотеку типов. Это позволит использовать исходящий интерфейс в библиотеке типов и создать с помощью мастера **Implement Connection Point Wizard** прокси-класс для отправки уведомлений о событиях. Щелкнем правой кнопкой мыши на классе CThermostat в окне Class View и выберем в открывшемся контекстном меню команду **Implement Connection Point** (Создать точку подключения). Затем выберем из списка интерфейс _IThermostatEvents и щелкнем кнопку **ОК** для генерации вспомогательного прокси-класса. Класс CThermostat происходит от этого класса, поэтому он может вызывать соответствующие вспомогательные методы Fire_EventHandler() с целью инициализации событий для своих подписчиков.



Реализация метода `SetCurrentTemperature()` проверяет, находится ли температура в допустимых пределах. Если это не так, данная реализация инициирует событие для своих подписчиков с помощью вспомогательных методов `Fire_EventHandler()`, генерируемых мастером **Implement Connection Point Wizard**:

```
// Задаем пороговые значения для температуры
#define MAXIMUM_TEMPERATURE_THRESHOLD 80.0
#define MINIMUM_TEMPERATURE_THRESHOLD 20.0
```

```
CThermostat : CThermostat()
{
    // Установка начальной температуры
    m_fTemperature = 50.0;
}
```

```
STDMETHODIMP CThermostat::SetCurrentTemperature(float fTemperature)
{
    if(fTemperature > MAXIMUM_TEMPERATURE_THRESHOLD)
    {
        // Инициализация события, указывающего, что температура превысила
        // верхний предел
        Fire_OnExceededUpperThreshold(fTemperature);
    }
    else if(fTemperature < MINIMUM_TEMPERATURE_THRESHOLD)
    {
        // Инициализация события, которое указывает, что температура
        // оказалась ниже нижнего предела
        Fire_OnExceededLowerThreshold(fTemperature);
    }
    else
    {
        // Температура находится в заданных пределах. Поэтому мы просто
        // устанавливаем ее значение.
        m_fTemperature = fTemperature;
    }

    // Возвращаем состояние
    return S_OK;
}
```

Реализация метода `GetCurrentTemperature()` просто возвращает текущее значение температуры, поддерживаемое термостатом:

```
STDMETHODIMP CThermostat::GetCurrentTemperature(float *pfTemperature)
```

```

{
    // Проверяем, является ли указатель действительным
    if (NULL == pfTemperature)
        return E_POINTER;
    // Возвращаем текущую температуру
    *pfTemperature = m_fTemperature;
    return S_OK;
}

```

Для того чтобы все клиенты использовали один экземпляр COM-компонента Thermostat, создаем одноэлементный объект, добавляя следующий макрос ATL в определение класса CThermostat:

```
DECLARE_CLASSFACTORY_SINGLETON(CThermostat)
```

Скомпилируем проект. В следующем разделе мы увидим, как с помощью управляемого приложения C++ можно подписаться на уведомления о событиях, генерируемых компонентом Thermostat.

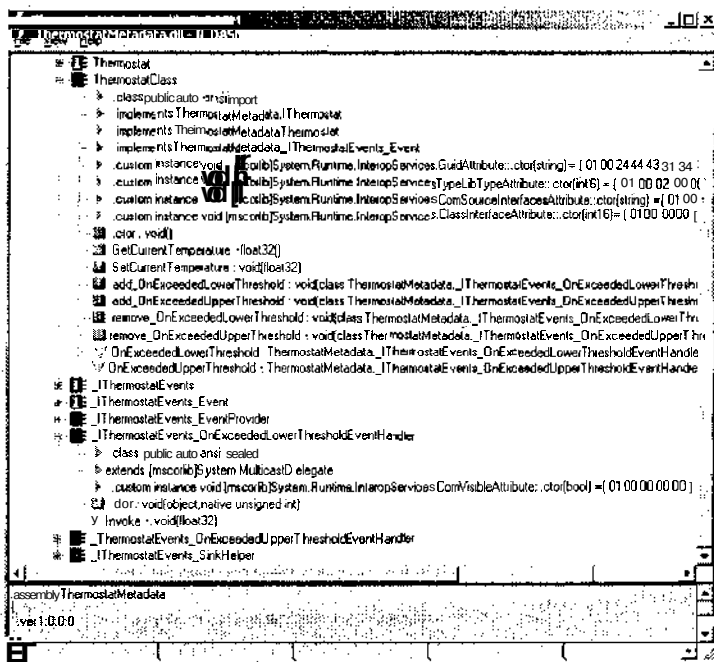
Использование управляемого приложения C++ для получения уведомлений о событиях от компонента COM

Мы разработаем управляемое приложение C++, которое подписывается на уведомления о событиях, генерируемых COM-компонентом Thermostat. Вначале создадим управляемое приложение C++ и назовем его ThermostatWatcher.

В первую очередь нужно активизировать утилиту импорта библиотеки типов для генерации оболочки, содержащей метаданные. При этом в командной строке зададим библиотеку типов компонента Thermostat:

```
> tlbimp ThermostatApp.tlb /out:ThermostatMetadata.dll
```

Как и ранее, откроем сборку, содержащую оболочку, с помощью утилиты ILDasm. Здесь мы увидим более сложный набор типов:



Видно, что входящий интерфейс `IThermostat` и исходящий интерфейс `_IThermostatEvents` импортированы сюда как управляемые интерфейсы, причем их имена сохранены. Все методы исходящего интерфейса `_IThermostatEvents` конвертированы в соответствующие классы, содержащие многоцелевые делегаты (*multicast delegates*). Каждому из этих классов присваивается имя согласно следующему образцу: `ИмяИсходящегоИнтерфейса_ИмяМетодаEventHandler`. В нашем примере для методов исходящего интерфейса `OnExceededLowerThreshold()` и `OnExceededUpperThreshold()` созданы два класса, включающих делегаты. Имена этих классов соответственно имеют вид: `_IThermostatEvents_OnExceededLowerThresholdEventHandler` и `_IThermostatEvents_OnExceededUpperThresholdEventHandler`. Сигнатуры методов исходящего интерфейса преобразованы в методы `Invoke()`, которые также присутствуют в классах, содержащих делегаты. Эти методы имеют такие же сигнатуры, а вместо неуправляемых параметров они принимают эквивалентные управляемые типы. Например, метод `void OnExceededLowerThreshold([in] float fTemperature)` в исходящем интерфейсе компонента импортирован в класс многоцелевого делегата, где представлен методом `void invoke([in] float32 fTemperature)`.

В сборке также сгенерирован управляемый интерфейс `_IThermostatEvents_Event`. Он содержит члены-события, представляющие каждый из типов многоцелевых делегатов, упомянутых выше. События импортируются с теми же именами, что и имена методов исходящего интерфейса. В рассматриваемом примере события имеют имена `OnExceededLowerThreshold` и `OnExceededUpperThreshold`.

На верхнем уровне всей этой схемы генерируются два вспомогательных класса, обеспечивающих соединение точек подключения `COM` и механизма обработки событий .NET, основанного на использовании делегатов:

D Класс, которому присваивается имя в соответствии с правилом:

`ИмяИсходящегоИнтерфейса_EventProvider`. В нашем случае это класс `_IThermostatEvents_EventProvider`. Он реализует методы `add_XXX()` и `remove_XXX()` для добавления и удаления делегатов, связанных с событиями. Члены экземпляра этого класса, создаваемые на основе классов `Runtime::InteropServices::UCOMIConnectionPointContainer` и `Runtime::InteropServices::UCOMIConnectionPoint` используются для взаимодействия с `RCW` с целью передачи уведомлений от `COM`-компонента обработчикам событий в управляемом коде, использующем делегаты. Описываемый класс также поддерживает список `ArrayList`, включающий вспомогательные объекты-стоки (см. следующий пункт перечисления), необходимые для отслеживания обработчиков событий, которые подписались на уведомления компонента `COM`.

- Вспомогательный класс-приемник, именуемый согласно образцу `ИмяИсходящегоИнтерфейса_SinkHelper`. В рассматриваемом нами примере имя этого класса принимает вид: `_IThermostatEvents_SinkHelper`. Каждый экземпляр данного класса кэширует cookie-индикатор, возвращаемый оболочкой `RCW` после успешного вызова метода `COM`-компонента `IConnectionPoint::Advise()`. Экземпляры этих классов (а следовательно, и cookie-индикаторы) перечислены в списке `ArrayList`, который поддерживается классом `_IThermostatEvents_EventProvider`.

Как мы видели ранее, `coclass` импортируется утилитой `tlbimp` в виде двух управляемых классов-оболочек. В нашем примере это следующие классы:

- Абстрактный класс `Thermostat`, ассоциированный с тем же идентификатором `GUID`, что и входящий интерфейс `coclass` по умолчанию (`IThermostat`). Данный

абстрактный класс наследует от входящего интерфейса `IThermostat` и исходящего интерфейса `_IThermostatEvents`.

- D Конкретный класс `ThermostatClass`, реализующий абстрактный класс `Thermostat` и интерфейс `_IThermostatEvents_Event`. Этот класс нужно инициализировать, чтобы создать экземпляр COM-компонента. Можно использовать члены-события данного класса, унаследованные от интерфейса `_IThermostatEvents_Event`, для подписки на уведомления о событиях. При создании управляемого приложения C++, которое подписывается на события, генерируемые COM-компонентом `Thermostat`, необходимо связать экземпляры делегатов, ссылающихся на наши методы обработки событий, с событиями, поддерживаемыми классом `ThermostatClass`. В то время как в событиях используются операторы `+=` или `-=` для добавления или удаления делегатов, связанных с этими событиями, в классе `_IThermostatEvents_EventProvider` применяются соответственно методы `add_XXX()` и `remove_XXX()`.

Мы получили представление о классах-оболочках, генерируемых утилитой импорта библиотеки типов. Теперь используем эти классы для получения управляемым приложением C++ уведомлений о событиях от COM-компонента.

Для подписки на уведомления о событиях, генерируемых COM-компонентом, необходимо создать экземпляр делегата, представляющего метод уведомления. Данный делегат будет ссылаться на метод, обрабатывающий уведомления о событиях. Поэтому в качестве параметра нужно передать конструктору делегата указатель на функцию обработчика события. Экземпляр делегата следует связать с событием. Рассмотрим теперь более подробно этапы реализации описанной процедуры.

Вначале добавим в код ссылку на сборку `ThermostatMetadata` с помощью директивы `#using`:

```
#using "ThermostatMetadata.dll"
```

Затем создадим новый управляемый класс C++ с именем `ThermostatNotificationHandler`, включающий методы обработки уведомлений о событиях. Этот класс состоит из двух статических методов: `OnLowerThreshold()` и `OnUpperThreshold()`, каждый из которых является обработчиком уведомлений о событиях, инициируемых компонентом `Thermostat`:

```
__gc class ThermostatNotificationHandler
{
public:
    static void OnLowerThreshold(float fLowTemperature)
    {
        Console.WriteLine(S"Below lower threshold : {0}",
            fLowTemperature.ToString());
    }
    static void OnUpperThreshold(float fHighTemperature)
    {
        Console.WriteLine(S"Above upper threshold : {0}",
            fHighTemperature.ToString());
    }
};
```

Для того чтобы избежать кистевого туннельного синдрома и необходимости печатать слишком длинные имена делегатов, сгенерированных утилитой импорта библиотеки типов, используем спецификатор `typedef` и введем более короткие имена:

```
typedef _IThermostatEvents_OnExceededLowerThresholdEventHandler
    LOW_TEMPERATURE_HANDLER;
typedef _IThermostatEvents_OnExceededUpperThresholdEventHandler
    HIGH_TEMPERATURE_HANDLER;
```

После этого создадим экземпляр делегата LOW_TEMPERATURE_HANDLER, который ссылается на функцию обработки уведомления о событии. Для этого передадим указатель на данную функцию в конструктор делегата:

```
LOW_TEMPERATURE_HANDLER* pLowHandler = new LOW_TEMPERATURE_HANDLER(
    0, &ThermostatNotificationHandler::OnLowerThreshold);
```

Теперь остается лишь связать этот делегат с событием, добавив его в список делегатов события. Ниже показано, как это делается применительно к делегату LOW_TEMPERATURE_HANDLER и событию OnExceededLowerThreshold:

```
ThermostatClass* pThermostat = new ThermostatClass;
pThermostat->OnExceededLowerThreshold += pLowHandler;
```

Этот код эффективно связывает уведомление OnExceededLowerThreshold, посылаемое COM-компонентом Thermostat, с методом обработки ThermostatNotificationHandler::OnLowerThreshold().

Если больше не нужно получать уведомления о событии, то можно использовать оператор -= для того, чтобы удалить связь между делегатом и событием:

```
pThermostat->OnExceededLowerThreshold -= pLowHandler;
```

• "За сценой" оператор += вызывает метод add_OnExceededLowerThreshold() объекта ThermostatClass, который представлен метаданными оболочки, помещенными в сборку. При этом используется метод Advise() класса Runtime::InteropServices::UCOMIConnectionPoint для передачи инструкции оболочке RCW о том, что она должна вызвать метод IConnectionPoint::Advise() компонента COM, который возвращает cookie-индикатор, идентифицирующий подписчика. Когда оболочка RCW возвращает управляемому коду этот индикатор, он сохраняется членом m_dwCookie класса _IThermostatEvents_SinkHelper.

Аналогично, оператор -= вызывает метод remove_OnExceededLowerThreshold() объекта ThermostatClass. В данном случае запускается метод Unadvise() класса Runtime::InteropServices::UCOMIConnectionPoint, и при этом указанному методу передается cookie-индикатор, который удален из члена m_dwCookie класса _IThermostatEvents_SinkHelper. В результате оболочка RCW вызывает метод компонента COM IConnectionPoint::Unadvise(), что ликвидирует связь подписчика с уведомлением о событии.

Как видно из этого примера, управляемое приложение C++ подписывается на уведомления о событиях и получает их с помощью делегатов, тогда как компоненты COM для передачи уведомлений используют точки подключения. Как приложения .NET, так и компоненты COM продолжают использовать свои собственные, присущие им механизмы обработки событий, а оболочка RCW обеспечивает необходимые каналы взаимодействия между этими механизмами, связывая их воедино. Как обычно, полный исходный код для данного примера можно найти на Web-сайте издательства Wrox: www.wrox.com.

Работа приложения .NET с классом коллекций COM

При разработке любой нетривиальной объектной модели в COM обычно требуется представить группу объектов как коллекцию. Предположим, что пользователю нужно дать возможность просматривать книги в библиотеке. Для этого можно смоделировать объект Library как коллекцию, содержащую объекты Book (книга) и позволяющую пользователю перебирать отдельные книги.

Популярные объектные модели, такие как ADO, CDO и др., интенсивно используют коллекции. Например, в модели ADO объекты `Fields` и `Parameters` соответственно представляют собой коллекции объектов `Field` и `Parameter`. Если говорить о средствах программирования, то COM поддерживает такие интерфейсы, как `IEnumVARIANT`, который позволяют работать с коллекциями элементов типа `VARIANT`. Более того, ряд библиотек, например ATL, облегчает разработку коллекций COM, предоставляя вспомогательные классы, которые реализуют методы, необходимые для работы с коллекциями.

В настоящем разделе мы рассмотрим создание коллекции с помощью COM, а затем покажем, как легко использовать эту коллекцию в управляемом приложении C++.

Реализация коллекции компонентом COM с помощью библиотеки ATL 3.0.

Разработаем COM-компонент с именем `WroxBookCollection`, который моделирует коллекцию книг. Используя Visual Studio 6, создадим новый проект ATL с помощью мастера ATL COM Wizard и выберем опцию по умолчанию `Dynamic Link Library (DLL)`. Добавим в проект новый, простой объект ATL COM, назовем его `WroxBookCollection` и присвоим ему те же атрибуты, что и COM-компоненту `MyTimeKeeper`, рассмотренному выше.

Типичный компонент COM, моделирующий коллекцию, поддерживает следующие нередактируемые свойства:

- `Count` — возвращает количество элементов в коллекции.
- `Item` — позволяет клиентам получать элемент из коллекции по заданному индексу. Идентификатору `DISPID` этого свойства присваивается нулевое значение, которое представлено константой `DISPID_VALUE`. Это значение `DISPID` позволяет клиентам (например, Visual Basic) использовать данное свойство по умолчанию, когда при вызове метода имя свойства не задается явно.
- `D_NewEnum` — возвращает указатель интерфейсного типа на объект, который поддерживает интерфейс `IEnumVARIANT`, позволяющий клиенту перебирать элементы используемой коллекции. Префикс в виде символа подчеркивания запрещает отображать данное свойство в средствах просмотра библиотеки типов. Идентификатору `DISPID` для данного свойства присваивается значение -4, представляемое константой `DISPID_NEWENUM`. Данное значение позволяет контроллерам, например в Visual Basic, использовать синтаксис `For Each... Next` для перебора элементов коллекции.

Откроем файл IDL и добавим в интерфейс `IWroxBookCollection` объекта `WroxBookCollection` следующие методы:

```
interface IWroxBookCollection : IDispatch
{
    [propget, id(1), helpstring("property Count")]
    HRESULT Count([out, retval] long *pVal);

    [propget, id(DISPID_NEWENUM),
    helpstring("property NewEnum"), restricted]
    HRESULT NewEnum([out, retval] LPUNKNOWN* pVal);

    [propget, id(DISPID_VALUE), helpstring("property Item")]
    HRESULT Item([in] long lIndex, [out, retval] VARIANT* pVal);
};
```

Атрибут `[restricted]`, присоединяемый к свойству `_NewEnum`, не позволяет клиентам получать прямой доступ к этому свойству.

Мы не будем здесь писать реализацию этих трех методов нашего компонента. Вместо этого приспособим для наших целей вспомогательный класс ATL `ICollectionOnSTLImpl<>`. Для реализации `ICollectionOnSTLImpl<>` необходимо задать политику копирования, указывающую, с какой точностью элементы коллекции должны возвращаться клиенту, когда тот запрашивает их. Поскольку коллекция `WroxBookCollection` будет содержать только заглавия книг, необходимо определить такую политику, чтобы строки, представляющие эти заглавия, копировались в параметр `Variant`, передаваемый через метод `copy()`.

Определим политику копирования с помощью класса `DeepCopyBooks`:

```
// Класс, определяющий политику копирования элементов коллекции
class DeepCopyBooks
{
public:
    static HRESULT copy(VARIANT* pVarDest, BSTR* pbstrBookTitle)
    {
        // Присваивание значения переменной CComVariant
        CComVariant varBookTitle(*pbstrBookTitle);

        // Копирование заглавия книги в переменную типа VARIANT
        return varBookTitle.Detach(pVarDest);
    }

    static void Init(VARIANT* pVar)
    {
        VariantInit(pVar);
    }

    static void Destroy(VARIANT* pVar)
    {
        VariantClear(pVar);
    }

    // Определение вектора STL для хранения коллекции книг
    typedef vector<CComBSTR> WROXBOOKS_COLLECTION_VECTOR;

    // Определение переиспользуемого COM на основе этого вектора
    typedef CComOnSTLInplaceVariant<IID_IWroxBookCollection,
        DeepCopyBooks, WROXBOOKS_COLLECTION_VECTOR> VarEnum;

    // Вспомогательный класс коллекций для контейнеров на основе STL
    typedef ICollectionOnSTLImpl<IWroxBookCollection, WROXBOOKS_COLLECTION_VECTOR,
        VARIANT, DeepCopyBooks, VarEnum> IWroxBookCollectionImpl;
```

Осталось лишь сделать так, чтобы класс `CWroxBookCollection` наследовал от конкретной реализации шаблона `ICollectionOnSTLImpl`, для которой мы с помощью спецификатора `typedef` ввели обозначение `IWroxBookCollectionImpl`:

```
class ATL_NO_VTABLE CWroxBookCollection :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CWroxBookCollection, &CLSID_WroxBookCollection>,
public ISupportErrorInfo,
public IDispatchImpl<IWroxBookCollectionImpl, &IID_IWroxBookCollection>
{
    HRESULT FinalConstruct()
```

```
// Остальной стандартный код для краткости опущен.
}
```

Наконец, добавим в коллекцию наши любимые книги издательства Wrox. Член `m_coll` класса `ICollectionOnSTLImpl` представляет контейнер для коллекции. В нашем случае этот контейнер является экземпляром вектора `STL`, содержащего строки `CCComBSTR` (с помощью спецификатора `typedef` мы ввели для вектора имя `WROXBOOKS_COLLECTION_VECTOR`):

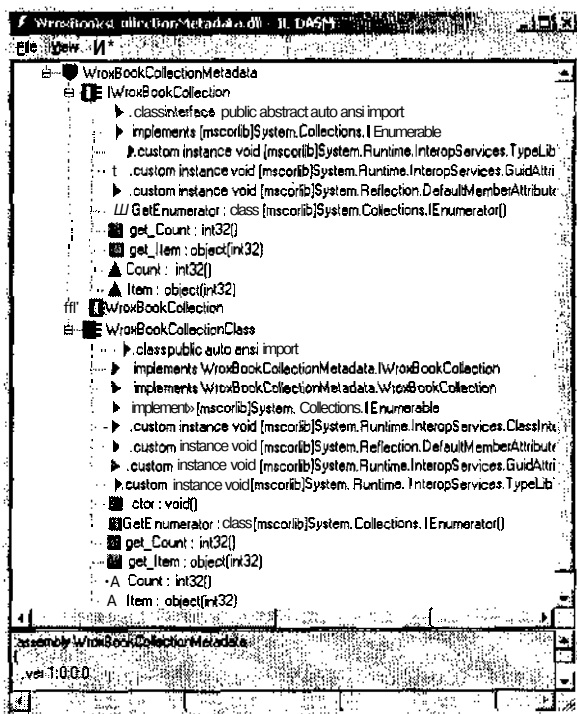
```
HRESULT CWroxBookCollection::FinalConstruct()
```

```
{
    // Добавляем книги в коллекцию
    m_coll.push_back(CComBSTR(T("Professional C++ programming")));
    m_coll.push_back(CComBSTR(T("Professional Visual C++ .NET")));
    m_coll.push_back(CComBSTR(T("Professional C# Web Services")));
    m_coll.push_back(CComBSTR(T("Professional ASP.NET")));
    return S_OK;
}
```

На этом завершается создание COM-компонента `WroxBookCollection`. Теперь посмотрим, как можно использовать нашу коллекцию в управляемом приложении C++.

Использование управляемого приложения C++ Алгоритмы с коллекцией COM

После создания сборки `WroxBookCollectionMetadata.dll` с помощью утилиты импорта библиотеки типов воспользуемся утилитой `ILDasm` и выведем на экран содержимое данной сборки:



Посмотрим, что показывает дизассемблер на этот раз. Интерфейс `IWroxBookCollection` импортирован как управляемый интерфейс с тем же самым именем. Также импортированы специфические для коллекции свойства интерфейса. Два из трех этих свойств, `Count` и `Item`, сохранены без видимых преобразований (если не считать замены неуправляемых типов данных их управляемыми эквивалентами). Однако свойство `_NewEnum` заменено методом `GetEnumerator()`, которое возвращает интерфейс `System::Collections::IEnumerator`, представляющий перечислитель коллекции. Интерфейс `IWroxBookCollection` также реализует интерфейс `System::Collections::IEnumerable`, позволяющий клиентам перебирать элементы соответствующей коллекции.

Интерфейсы `IEnumerator` и `IEnumerator` являются первичными объектами, которые делают возможным перебор элементов коллекции в .NET. Если клиент хочет знать, поддерживает ли объект перечисление элементов, он может получить такую информацию, проверив, имеет ли данный объект интерфейс `IEnumerator`. Если объект поддерживает интерфейс `IEnumerator`, тогда его метод `IEnumerator::GetEnumerator()` позволяет получить перечислитель, дающий возможность перебирать элементы коллекции.

Как обычно, `class` компонента `WroxBookCollection` импортирован в форме двух управляемых классов. Во-первых, имеется абстрактный класс `WroxBookCollection`, наследующий от интерфейса по умолчанию `IWroxBookCollection`. Во-вторых, в сборке присутствует конкретный класс `WroxBookCollectionClass`, реализующий абстрактный класс `WroxBookCollection` и обеспечивающий реализацию его методов. Для создания СОМ-компонента `WroxBookCollection` необходимо создать экземпляр класса `WroxBookCollectionClass`. Все методы данного класса внутренним образом вызывают оболочку `RCW`, которая, в свою очередь, вызывает коллекцию компонента СОМ.

Метод `GetEnumerator()` класса `WroxBookCollectionClass` взаимодействует с оболочкой `RCW`, чтобы представить коллекцию СОМ, основанную на применении интерфейса `IEnumerator`, как коллекцию .NET, использующую интерфейс `IEnumerator`. Чтобы такое преобразование оказалось возможным, используется класс `System::Runtime::InteropServices::CustomMarshalers::EnumeratorToEnumVariantMarshaler`.

Для того чтобы использовать СОМ-объект, вначале создадим новое управляемое приложение C++ с именем `TechieBookworm`. После добавления в код ссылки на сборку, которая содержит оболочки с метаданными, можно приступить к работе с коллекцией. Для начала получим количество книг с помощью свойства `Count` класса `WroxBookCollectionClass`:

```
WroxBookCollectionClass* pColl = new WroxBookCollectionClass;
void nNumberOfBooks = pColl->Count;
Console::WriteLine(S"Number of books: {0}", nNumberOfBooks.ToString());
```

Для получения объекта, находящегося в коллекции на конкретной позиции, необходимо воспользоваться свойством `Item`:

```
Console::WriteLine(S"The second book is: {0}", pColl->Item[2]);
```

Здесь основанием индекса является *единица*, а не *нуль*, как мы привыкли. Метод `pColl->Item[1]` возвращает самый первый элемент коллекции.

Для получения перечислителя коллекции применим метод `GetEnumerator()`:

```
IEnumerator pEnum = pColl->GetEnumerator();
```

Вначале перечислитель помещается *перед* первым элементом коллекции. Поэтому, чтобы получить к нему доступ, нужно вызвать метод `MoveNext()`. После этого

метод `MoveNext()` можно вызывать многократно до тех пор, пока не будет достигнут конец коллекции и метод не возвратит значения `false`. При перемещении по коллекции можно использовать свойство `Current` для получения элемента, указываемого перечислителем. Это свойство возвращает указатель типа `System.Object`, который можно привести к базовому строковому указателю с помощью методов `dynamic_cast` или `__try_cast`:

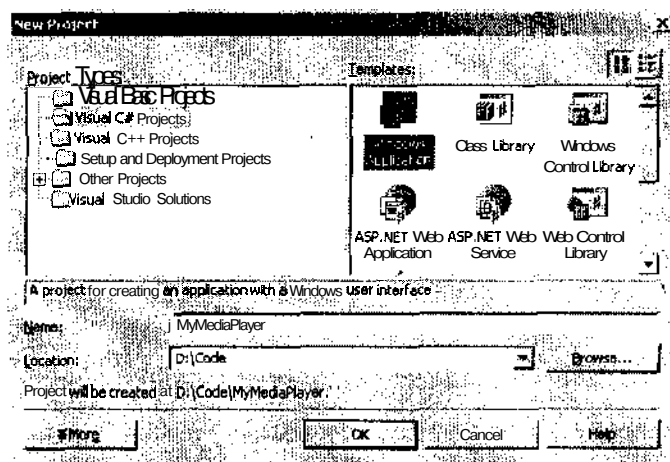
```
while(pEnum->MoveNext())
{
    Object* pObjBookTitle = pEnum->Current;
    String* pstrTitle = dynamic_cast<String*>(pObjBookTitle);
    Console.WriteLine(pstrTitle);
}
```

Полный исходный код управляемого приложения C++ `TechieBookWorm` помещен в папку с таким же именем, как и в загружаемом коде для данной книги.

Применение элементов управления ActiveX в приложениях.NET

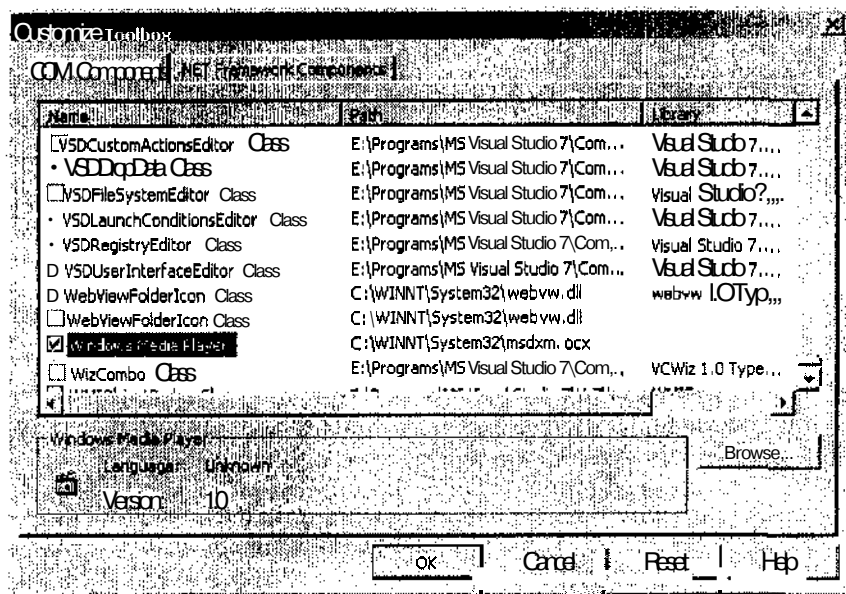
Процедура добавления элементов управления ActiveX в приложения Windows Forms довольно проста. Вместо разработки тяжеловесных объектов с самого начала мы используем для целей демонстрации один из элементов управления, ActiveX, который уже имеется на нашей машине, например проигрыватель мультимедиа Windows Media Player (WMP). Если проигрыватель установлен на компьютере, его можно найти в файле `Панка_Windows\System32\msdxm.ocx`.

К моменту написания данной книги среда Visual Studio .NET IDE не поддерживала создание приложений Windows Forms с помощью управляемого кода C++. По-видимому, предполагается, что программисты будут использовать C# или Visual Basic .NET для разработки приложений, основанных на формах Windows. Мы так и поступим, используя созданное с помощью C# приложение Windows Forms для добавления в него элемента управления ActiveX WMP. Начнем с создания нового приложения Visual C# Windows Application, которое назовем `MyMediaPlayer`:

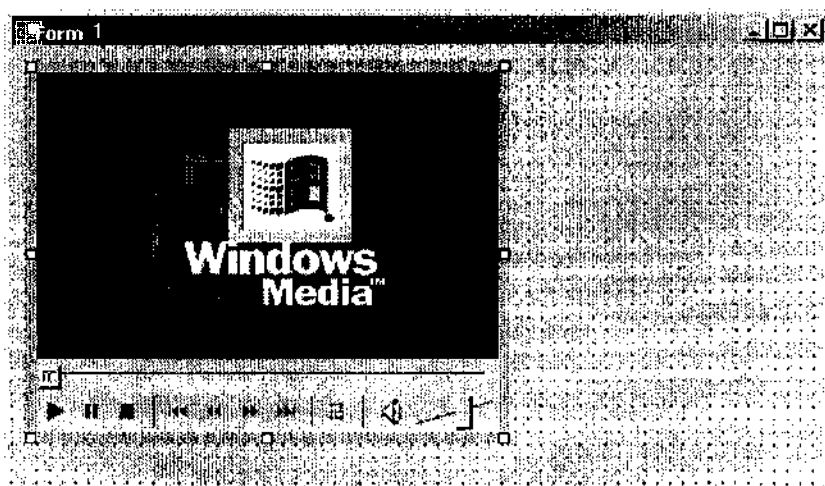


Далее используем команду меню `Tools | Customize Toolbox (Инструменты | Настройка панели инструментов)`. Если проигрыватель WMP зарегистрирован на

компьютере, он должен появиться в списке элементов управления на вкладке **COM Components** (Компоненты COM). Установим флажок рядом с элементом списка **Windows Media Player** и нажмем кнопку **OK**:



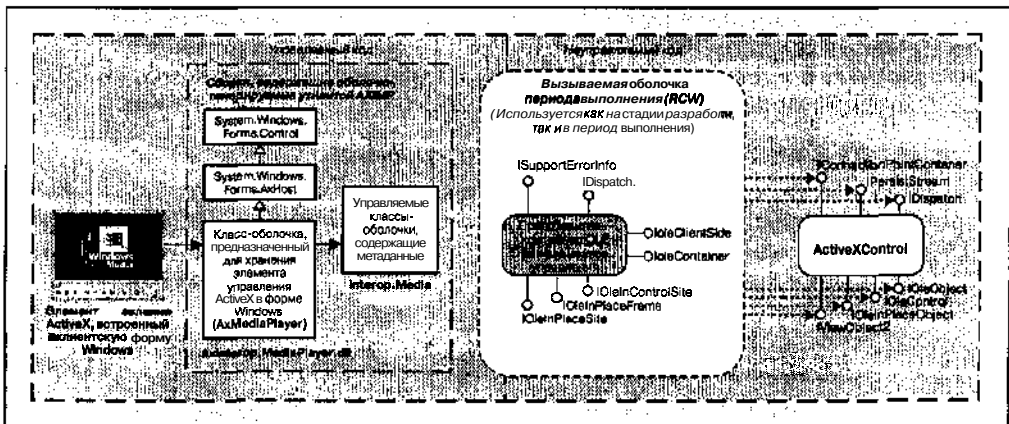
Теперь можно выбрать элемент управления WMP на панели инструментов и перетящить его на рабочую поверхность дизайнера форм Windows:



После этого среда Visual Studio .NET сама позаботится о генерации нужных оболочек, содержащих метаданные, на основе библиотеки типов элемента управления ActiveX, а также о добавлении в проект ссылок на сборки, где находятся эти оболочки. Visual Studio .NET использует утилиту импорта ActiveX (aximp.exe) для генерации двух оболочек:

- а Сборка `AxInterop.MediaPlayer.dll` включает в себя элемент ActiveX для того, чтобы тот выглядел, как элемент управления .NET Windows Forms. На формах Windows элементы управления могут размещаться только в том случае, если они прямо или косвенно наследуют от класса `System::Windows::Forms::Control`. Поэтому утилита импорта ActiveX генерирует управляемый класс-оболочку `AxMediaPlayer`, наследующий от класса `System::Windows::Forms::AxHost`, который, в свою очередь, происходит от `System::Windows::Forms::Control`. Библиотечный класс `System::Windows::Forms::AxHost` выполняет преобразование, необходимое для представления элемента ActiveX в среде .NET как обычного элемента управления Windows Forms.
- б Сборка `Interop.MediaPlayer.dll` включает управляемые классы-оболочки, содержащие информацию об управляемых типах, которая относится к свойствам, методам и событиям, поддерживаемым элементом управления ActiveX WMP. Эта сборка функционально похожа на те сборки, содержащие оболочки, что генерируются утилитой импорта библиотеки типов, которая использовалась в наших предыдущих примерах. Однако форма Windows не работает с данной сборкой непосредственно, а вместо этого использует сборку `AxInterop.MediaPlayer.dll`, которая, в свою очередь, с помощью `Interop.MediaPlayer.dll` общается с оболочкой RCW элемента управления ActiveX. Оболочка RCW создается средой CLR как на стадии разработки, так и в период выполнения. Она вызывает элемент управления ActiveX, чтобы замкнуть цепочку.

Следующий рисунок показывает, какую роль совместно с RCW играют эти две сборки (куда входят оболочки, содержащие метаданные) в том, чтобы представить элемент ActiveX как элемент управления Windows Forms при его использовании формами Windows:

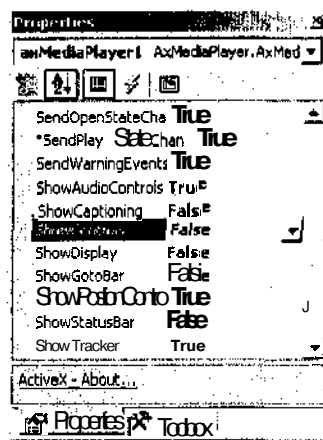


Методы и аксессоры/модификаторы свойств класса-оболочки `AxMediaPlayer` вызывают свои аналоги в классе `MediaPlayerClass`, принадлежащем к сборке `Interop.MediaPlayer.dll`. Реализация данного класса направляет вызов в оболочку RCW, которая вызывает соответствующий метод или получает/задает значение свойства в элементе управления ActiveX. Эта оболочка также выполняет согласование механизма обработки событий на основе делегатов, применяемого в формах Windows, и механизма элемента управления ActiveX, где обработка событий выполняется с использованием точек подключения. Оболочка RCW реализует все интерфейсы,

которые должен реализовывать любой контейнер элементов управления ActiveX. Благодаря этому элемент ActiveX пребывает в уверенности, что он действительно помещен в контейнер элементов управления ActiveX.

Свойства периода разработки

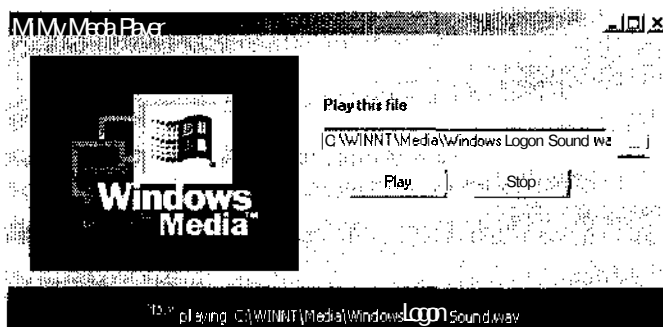
Свойства периода разработки для элемента управления ActiveX можно изменять, используя окно свойств (Properties) в среде Visual Studio .NET. Это окно позволяет исследовать элемент ActiveX с помощью классов-оболочек, содержащих метаданные, и модифицировать его свойства на этапе создания приложения. Например, можно изменить свойство ShowControls, установив его значение равным False:



В результате вид проигрывателя мультимедиа на форме Windows в режиме разработки немедленно изменится, так как исчезнет панель управляющих элементов {media control bar}. Более того, поскольку рассматриваемый нами элемент управления сохраняет постоянным свойство ShowControls, значение этого свойства, заданное на стадии разработки, останется тем же самым и в период выполнения.

Свойства и методы периода выполнения

Рассмотрим теперь, как вызываются методы и свойства элемента управления в период выполнения. Добавим на форму кнопки Play и Stop, которые соответственно вызывают методы Play() и Stop() проигрывателя мультимедиа. Далее мы используем свойство этого элемента управления FileName для того, чтобы задавать проигрываемый фрагмент:



Ниже приведен код методов обработки события Click для кнопок Play и Stop:

```
private void buttonPlay_Click (object sender, System.EventArgs e)
{
    try
    {
        // Установка пути для медиа-файла, который мы хотим проигрывать
        axMediaPlayer.FileName = textBoxFileName.Text;

        // Начало проигрывания фрагмента
        axMediaPlayer.Play;
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

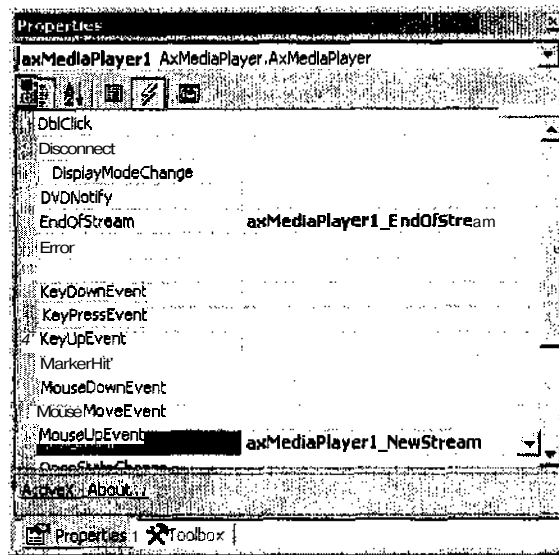
private void buttonStop_Click(object sender, System.EventArgs e)
{
    try
    {
        // Остановка проигрывания элмента
        axMediaPlayer.Stop();

        // Обновление сообщения в строке состояния
        labelStatus.Text = "Last played media: " + m_strCurrentFile;
    }
    catch(Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}
```

Как видим, вызов методов или свойств элемента управления ActiveX выполняется достаточно просто. Благодаря классам в сборке, содержащей оболочки с метаданными, эта процедура выглядит так, будто мы вызываем методы или получаем/задаем значения свойств управляемого класса.

Обработка событий

Обсудим, как можно получать уведомления о событиях от элемента управления ActiveX WMP. Зарегистрируем методы обработки для событий NewStream и EndOfStream. Первое из них инициируется, когда элемент управления начинает загружать фрагмент медиа-данных, а второе вызывается по окончании проигрывания записи. Чтобы просмотреть в Visual Studio .NET оба этих события, источником которых служит элемент управления, щелкнем на значке молнии в окне Properties для элемента управления WMP:



В результате двойного щелчка мышью на событии NewStream дизайнер форм Visual Studio .NET сгенерирует метод обработки события axMediaPlayer_NewStream() (в соответствии с соглашением об именах `ИмяЭлементаУправления_ИмяСобытия`). Повторим эту же процедуру для события EndOfStream. Если теперь обратиться к коду формы и посмотреть на метод InitializeComponent(), можно увидеть, что делегаты, ссылающиеся на методы обработки событий, связаны с соответствующими событиями:

```
this.axMediaPlayer.NewStream +=
    new System.EventHandler(this.axMediaPlayer_NewStream);

this.axMediaPlayer.EndOfStream +=
    new AxMediaPlayer1.MediaPlayerEvents.EndOfStreamEventHandler(
        this.axMediaPlayer_EndOfStream);
```

Здесь делегат `System:EventHandler`, ссылающийся на метод обработки события `axMediaPlayer_NewStream()`, связан с событием `NewStream`, а делегат `_MediaPlayerEvents_EndOfStreamEventHandler`, который ссылается на обработчик `axMediaPlayer_EndOfStream()`, присоединен к событию `EndOfStream`. Когда приложение получает событие `NewStream`, оно обновляет сообщение, выдаваемое строкой состояния в нижней части формы и отображающее текущий проигрываемый фрагмент. При получении события `EndOfStream` приложение обновляет сообщение строки состояния, которая теперь выводит последний исполненный фрагмент. Приведем методы обработки для обоих этих событий:

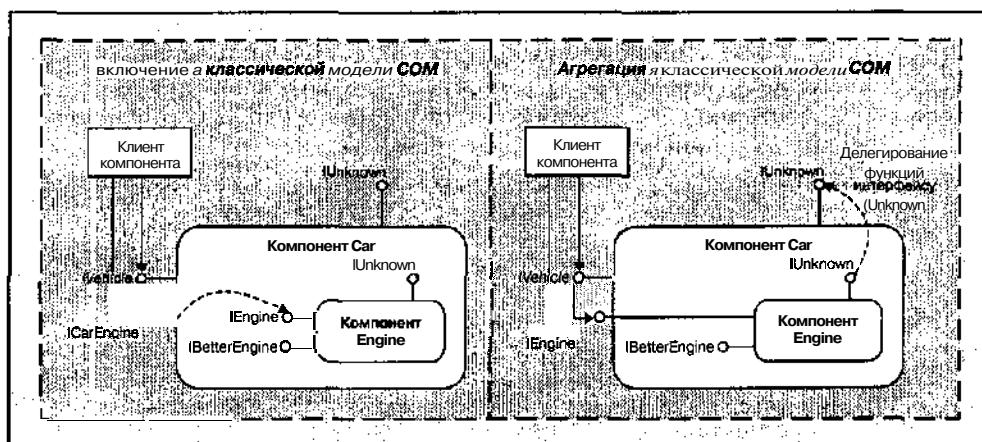
```
private void axMediaPlayer_NewStream(object sender, System.EventArgs e)
{
    labelStatus.Text = "Now playing: " + m_strCurrentFile;
}

private void axMediaPlayer_EndOfStream(object sender,
    AxMediaPlayer1.MediaPlayerEvents.EndOfStreamEvent e)
{
    labelStatus.Text = "Last played media: " + m_strCurrentFile;
}
```

Вручную мы добавили очень мало кода, однако элемент управления ActiveX уютно чувствует себя на форме Windows и ведет себя так, будто он является элементом управления .NET Windows Forms. Классы-оболочки, содержащие метаданные, и слой RCW реализуют описанную интеграцию, поддерживая иллюзию, что каждый из них взаимодействует с родственными себе объектами. Компании могут сохранить свои существующие элементы управления ActiveX, помещая их в контейнеры .NET, такие как формы Windows, и при этом планировать стратегию миграции к более мощным собратьям элементов ActiveX, т.е. к элементам управления .NET Windows Forms.

Применение в управляемом коде моделей повторного использования для COM-компонентов

Одна из основных целей компонентного подхода к разработке ПО — обеспечить возможность повторного использования существующих компонентов приложениями или другими компонентами. В мире COM один компонент может многократно использовать службы другого компонента через **включение (containment)** или **агрегацию (aggregation)**:



Данная диаграмма показывает, как компонент Car повторно использует реализацию компонента Engine на основе модели включения. В этой модели внешний компонент (Car) непосредственно не показывает клиенту интерфейсы внутреннего компонента (Engine). Вместо этого он просто делегирует вызовы включаемому экземпляру компонента Engine.

В модели агрегации компонент Car предоставляет один или несколько интерфейсов компонента Engine непосредственно клиенту. Клиент может напрямую общаться с внутренним компонентом. В случае агрегации внутренний компонент (Engine) делегирует подсчет ссылок и семантику опроса интерфейса внешнему компоненту (Car).

Управляемые компоненты .NET могут повторно использовать COM-компоненты через наследование или включение, но не через агрегацию. Рассмотрим пример, иллюстрирующий оба сценария. Мы создадим два COM-компонента: Vehicle (Транспортное средство) и Engine (Двигатель). Затем в среде Visual C++ .NET разработаем управляемый компонент Car (Автомобиль), повторно использующий COM-компоненты Vehicle и Engine, соответственно через наследование и включение:



С помощью мастера ATL COM AppWizard создадим в Visual C++ 6.0 новый проект Automotive, выберем опции по умолчанию и добавим в проект два новых COM-объекта: **Vehicle** и **Engine**. Компонент **Vehicle** будет предоставлять два метода: `StartVehicle()` и `StopVehicle()` через свой интерфейс **IVehicle**:

```

[id(1), helpstring("method StartVehicle")] HRESULT StartVehicle([out, retval] BSTR* pbstrStatus);

[id(2), helpstring("method StopVehicle")] HRESULT StopVehicle([out, retval] BSTR* pbstrStatus);

STDMETHODIMP CVehicle::StartVehicle(BSTR* pbstrStatus)
{
    if(NULL == pbstrStatus) return E_POINTER;

    CComBSTR bstrStartMsg(_T("From Classic COM Vehicle Start"));
    *pbstrStatus = bstrStartMsg.Detach();
    return S_OK;
}

STDMETHODIMP CVehicle::StopVehicle(BSTR* pbstrStatus)
{
    if(NULL == pbstrStatus) return E_POINTER;

    CComBSTR bstrStopMsg(_T("From Classic COM Vehicle Stop"));
    *pbstrStatus = bstrStopMsg.Detach();
    return S_OK;
}

```

COM-компонент **Engine** с помощью своего интерфейса **IEngine** открывает два метода: `MovePiston()` и `VaporizeFuel()`:

```

[id(1), helpstring("method MovePiston")] HRESULT MovePiston([out, retval] BSTR* pbstrStatus);

[id(2), helpstring("method VaporizeFuel")] HRESULT VaporizeFuel([out, retval] BSTR* pbstrStatus);

```

```

    CEngine::MovePiston(BSTR* pbstrStatus)
    {
        if (pbstrStatus == NULL) return E_POINTER;

        CComBSTR bstrMovePistonMsg(_T("From ... .. Piston"));
        bstrMovePistonMsg.Detach();
        return S_OK;
    }

    STDMETHODIMP CEngine::VaporizeFuel(BSTR* pbstrStatus)
    {
        if (NULL == pbstrStatus) return E_POINTER;

        CComBSTR bstrVaporizeMsg(_T("From Classic COM: Vaporize Fuel"));
        bstrVaporizeMsg.Detach();
        return S_OK;
    }

```

Как и ранее, на основе библиотеки типов компонента с помощью утилиты `tlbimp` создадим сборку `AutomotiveMetadata.dll`, куда входит оболочка, содержащая метаданные. Затем приступим к разработке управляемого приложения C++, которое назовем `ReuseManagedClient`. Это приложение позволит протестировать обе модели повторного использования.

Управляемый компонент C++ `Car` наследует от представляющего COM-компонент `Vehicle` класса-оболочки, содержащего метаданные. Для этого мы применяем вездесущий синтаксис наследования C++. В модели включения с использованием компонента `Engine` компонент `Car` кэширует указатель на класс-оболочку с метаданными, относящимися к `Engine`, а затем создает новый экземпляр компонента `Engine` в своем конструкторе или перед первым использованием компонента (отложенная загрузка):

```

__gc class Car : public VehicleClass
{
private:
    EngineClass* m_pEngine;
public:
    Car()
    {
        m_pEngine = new EngineClass;
    }

    // Остальной код
};

```

Добавим в класс `Car` новый метод, чтобы продемонстрировать, как он будет повторно использовать код COM-компонентов `Engine` и `Vehicle`:

```

void StartCar()
{
    Console.WriteLine(StartVehicle());
    Console.WriteLine(m_pEngine->MovePiston());
    Console.WriteLine(m_pEngine->VaporizeFuel());
    Console.WriteLine($"Managed C++ Started Car");
}

```

Функция `StartCar()` вызывает метод `StartVehicle()`, что приводит к вызову соответствующей реализации метода в COM-компоненте `Vehicle`, от которого происходит класс `Car`. После этого объект `Car` использует включаемый экземпляр оболочки,

содержащей метаданные COM-компонента Engine, для вызова методов этого компонента MovePiston() и VaporizeFuel(). Когда необходимо вызвать метод COM, классы-оболочки с метаданными и оболочка RCW обеспечивают переходы между управляемым и неуправляемым кодом. Это позволяет управляемому коду повторно использовать классический код COM через наследование и включение, как будто мы работаем с другим управляемым классом C++.

Сейчас подходящий момент для того, чтобы открыть и просмотреть полный исходный код проектов Automotive и ReuseManagedClient (если вы не сделали этого до сих пор).

Теперь посмотрим, как можно в управляемом классе C++ Car подменить методы StartVehicle() и StopVehicleO компонента COM:

```
__gc class Car : public VehicleClass
{
// Остаток кода реализации класса Car

public:
    // Подмена реализации метода базового класса COM
    // VehicleClass::StartVehicle
    String StartVehicle()
    {
        // Заменяем реализацию COM-компонента Vehicle
        // на реализацию из Managed C++
        return S[Overridden] ManagedC++ Vehicle::Start";
    }

    // Подмена реализации метода базового класса COM VehicleClass::StopVehicle
    String StopVehicle()
    {
        return S[Overridden] ManagedC++ Vehicle::Stop";
    }
}
```

После добавления этих двух методов в класс Car любые обращения к методам StartVehicle() и StopVehicleO приведут к вызову методов управляемого объекта Car, а не функций COM-компонента Vehicle. Если необходимо явно вызвать реализацию COM-компонента Vehicle, то можно использовать оператор разрешения контекста C++ применительно к оболочке VehicleClass, содержащей метаданные. Подменяющая реализация класса Car может также обеспечивать дополнительное выполнение кода до или после вызова реализации базового класса.

Такого поведения и следовало ожидать, и читатель может поинтересоваться, зачем мы останавливаемся на данном вопросе. Здесь мы должны его предостеречь. Методы базового класса, реализация которых обеспечивается классическим COM-компонентом, нельзя подменять поодиночке. Иными словами, если мы хотим подменить один метод, следует также подменить все остальные методы. Если этого не сделать, приложение будет откомпилировано, однако во время выполнения мы получим исключительную ситуацию System::TypeLoadException.

Это утверждение можно проверить, если временно закомментировать подменяющую реализацию метода StopVehicleO в классе Car и оставить лишь реализацию, где подменяется метод StartVehicle(). Попытавшись запустить приложение, мы получим следующее сообщение об ошибке:

System.TypeLoadException: Types extending from COM objects should override all methods of an interface implemented by the base COM class

(Типы, происходящие от объектов COM, должны подменять все методы интерфейса, реализованного базовым классом COM).

Управляемые приложения C++ позволяют реализовать обычное наследование C++ и синтаксис включения для повторного использования кода COM-компонента. Эти приложения могут также в случае необходимости использовать управляемый код для обеспечения дополнительной функциональности и применения средств, предлагаемых библиотекой классов .NET Framework. Таким образом, имея в виду многократное использование кода, можно сказать, что управляемые компоненты позволяют получить наилучшие результаты как в управляемой, так и в неуправляемой среде.

Управляемые потоки и апартamentы COM

В среде COM неуправляемый поток, вызывающий COM-компонент, может входить в однопоточный апартмент (single-threaded apartment, STA) или в многопоточный апартмент (multi-threaded apartment, MTA). Клиент COM в неуправляемой среде использует один из следующих операторов вызова для информирования COM о том, что поток данного клиента входит в STA:

```
CoInitialize(NULL);
CoInitializeEx(0, COINIT_APARTMENTTHREADED);
```

Аналогичным образом клиент COM информирует о том, что его поток входит в MTA, с помощью такого оператора вызова:

```
CoInitializeEx(0, COINIT_MULTITHREADED);
```

В управляемых приложениях перед вызовом COM-компонентов потоки также могут объявлять свои предпочтения, указывающие типы апартментов COM. По умолчанию управляемый поток выбирает MTA. Однако можно использовать свойство ApartmentState класса System::Threading::Thread, чтобы задать тип апартмента COM для управляемого потока. Ниже приведены возможные значения перечисления ApartmentState:

Значение ApartmentState	Описание
STA	Поток входит в STA.
MTA	Поток входит в MTA.
Unknown	Значение ApartmentState не установлено. По умолчанию поток входит в MTA.

Приведем пример тривиального управляемого приложения C++, которое демонстрирует установку свойства Apartment State для управляемого потока:

```
(fusing <mscorlib.dll>

using namespace System;
using namespace System::Threading;

int _tmain(void)
{
    // Получаем указатель на текущий поток
    Thread* pThread = Thread::CurrentThread;
    // Получаем текущее состояние апартмента
    ApartmentState enumState = pThread->ApartmentState;
```

```

    Object* pObject = __box(enumState);
    Console.WriteLine($"Default apartment state: {OK}. {pObject->ToString()}");

    // Задаем состояние апартаментов STA
    pThread->ApartmentState = ApartmentState::STA;

    // Получаем новое состояние апартаментов
    enumState = pThread->ApartmentState;
    pObject = __box(enumState);

    Console.WriteLine($"Current apartment state: {O}", pObject->ToString());

    // Код создания COM-компонента...

} return 0;

```

Для того чтобы установка свойства ApartmentState привела к желаемому результату, ее необходимо выполнить *до того*, как управляемый поток создаст компонент COM. Описываемое средство поможет избежать проблем, связанных с появлением дополнительных заглушек при взаимодействии между несовместимыми апартаментами.

Заставить основной поток приложения войти в STA можно также, пометив точку входа в приложение атрибутом `STAThread`:

```

[STAThread]
int _tmain(void)
{
    // Поток вошел в STA
    // Код, создающий COM-компонент
    return 0;
}

```

Однако в этом случае следует проявить осторожность, поскольку данный атрибут имеет смысл применять, только если приложение не использует библиотеку периода выполнения C CRT. Иначе атрибут будет введен в фактическую точку входа приложения, находящуюся внутри CRT, и не окажет влияния на основной поток управляемого приложения.

Подробная информация об управляемых потоках в приложениях .NET представлена в главе 5.

Работа COM-ориентированного кода C++ с управляемыми компонентами C++

Мы рассмотрели использование компонентов COM в управляемых приложениях C++. Теперь обсудим противоположную ситуацию, когда неуправляемые COM-ориентированные клиенты могут работать с управляемыми компонентами C++.

Неуправляемые клиенты COM не могут непосредственно использовать управляемый компонент. Этим клиентам необходимо предоставить специфичную для COM информацию о типах, какую они обычно получают из библиотеки типов COM. Нужно также опубликовать информацию (идентификаторы компонента CLSID, ProgID и т.д.), которая обеспечит связывание неуправляемого клиента COM с управляемым компонентом. Проблема состоит в том, что управляемый компонент ничего не знает о COM и не имеет никакого отношения к типам COM и информации о связывании. Каким же образом неуправляемый COM-клиент может работать с управляемым компонентом?

Для этого необходимо экспортировать данные о типах управляемого компонента в библиотеку типов COM. Чтобы среда исполнения COM могла найти эти данные, необходимо также обеспечить запись в реестр всей информации о связывании. К счастью, для создания библиотеки типов COM на основе атрибутов и данных о типах управляемого компонента платформа .NET Framework предоставляет две утилиты: утилиту **экспорта библиотеки типов (type library exporter, файл tlbexp.exe)** и утилиту **регистрации сборок (assembly registration utility, файл regasm.exe)**. Утилита regasm.exe также добавляет записи в реестр, что позволяет среде исполнения COM выполнять связывание с компонентом.

Неуправляемое приложение COM может использовать библиотеку типов, генерируемую с помощью утилит tlbexp/regasm, для создания управляемого компонента таким же образом, как и для создания COM-компонента. На основе записей в реестре среда исполнения COM активизирует файл mscoree.dll (работающий в среде .NET CLR), который далее использует хранящуюся в реестре информацию для загрузки сборки, содержащей управляемый компонент. Среда CLR также создает **вызываемую оболочку COM (COM callable wrapper - CCW)**, которая играет роль посредника между COM и управляемым компонентом, представляя .NET-компонент неуправляемому клиенту как обычный компонент COM. Для каждого экземпляра управляемого компонента создается отдельная оболочка CCW.

Управляемый компонент живет по законам CLR и удаляется сборщиком мусора, работающим в среде исполнения .NET. CCW обслуживает всю семантику COM, используемую неуправляемым клиентом (подсчет ссылок, запросы о поддерживаемых интерфейсах, позднее связывание с помощью интерфейса IDispatch и т.д.). Оболочка CCW также выполняет маршрутизацию данных между управляемым и неуправляемым кодом. Время жизни CCW контролируется неуправляемым клиентом, и оболочка уничтожается, если количество ссылок на нее становится равным нулю. Это происходит, когда клиент освобождает последнюю ссылку на данную оболочку. Одновременно CCW перестает ссылаться на управляемый компонент, позволяя сборщику мусора .NET выполнить очистку.

Далее мы покажем, насколько просто можно добиться, чтобы неуправляемые клиенты COM работали с управляемыми компонентами C++. Мы также обсудим, каким образом неуправляемый код C++, работающий с точками подключения, обрабатывает события управляемого компонента C++, который использует механизм делегатов. Затем мы рассмотрим атрибуты, позволяющие задать способ представления управляемого интерфейса или класса C++ в среде COM.

Создание компонента .NET с помощью управляемого кода C++

Разработаем управляемый компонент C++, который имитирует две функции, выполняемые панелью приборов автомобиля. Создадим новую управляемую библиотеку классов C++, присвоим ей имя Dashboard и определим интерфейс IDashboard, содержащий все методы и свойства, которые должен иметь компонент Dashboard. Этот интерфейс реализуется в классе управляемого компонента:

```
namespace Dashboard
{
    public __gc __interface IDashboard
    {
        __property float get_Speed();
        __property void set_Speed(float value);
        __property float get_GallonsConsumed();
    }
}
```

```

    __property void set_GallonsConsumed(float value);

    float computeMilesPerGallon(int riMiles);
};
}

```

Можно добавить методы и свойства непосредственно в класс компонента (не используя прямое наследование от интерфейса) и при этом открыть их для неуправляемых клиентов COM, однако такой подход не рекомендуется. Хорошим стилем программирования является объединение в интерфейсе всех методов и свойств, после чего можно создать управляемый класс, реализующий этот интерфейс.

Управляемый интерфейс IDashboard предоставляет два свойства: Speed (Скорость) и GallonsConsumed (Расход топлива), а также метод computeMilesPerGallon() (Пробег в милях на один галлон). Определим управляемый класс компонента Dashboard и реализуем в нем интерфейс IDashboard:

```

[ClassInterface(ClassInterfaceType::None)]
public __gc __not_s`Dasfiboaru!' public IDashboard
{
private:
    float m_sCurrentSpeed;
    float m_sGallonsConsumed;
    bool m_bFuelTankEmpty;

public:
    Dashboard(): m_sCurrentSpeed(20),
                 m_sGallonsConsumed(2),
                 m_bFuelTankEmpty(false)
    {
    }

    // Остальные методы и свойства...
};

```

Класс Dashboard помечен атрибутом System::Runtime::InteropServices::ClassInterface, который можно использовать для предоставления утилитам tlbexp и regasm информации о том, каким образом мы хотим открыть в COM управляемый класс компонента. Атрибут указывает, как должна определяться секция coclass в IDL, когда управляемый класс будет экспортироваться в библиотеку типов COM. Позже мы подробно обсудим различные возможные значения атрибута ClassInterface и рассмотрим влияние каждого из них на создаваемую библиотеку типов. В настоящий момент зададим значение параметра атрибута ClassInterfaceType::None.

Класс Dashboard должен быть открытым, а кроме того, иметь открытый конструктор по умолчанию. Это требование обязательно, если нужно, чтобы класс мог работать с клиентами COM.

Реализуем в классе Dashboard методы и свойства, наследуемые от интерфейса IDashboard. Начнем с реализации методов get/set свойства Speed:

```

__property float get_Speed()
{
    return m_sCurrentSpeed;
}

__property void set_Speed(float value)
{
    if (value < 0.0)
    {
        throw new DashboardException(S`Speed cannot be set to a negative value, "');
    }
}

```

```

    if (value > 150.0)
    {
        throw new DashboardException("Oops! Your engine has gone kaput.");
    }
    if (!m_bFuelTankEmpty)
    {
        throw new DashboardException("Your fuel tank is empty.");
    }

    m_sCurrentSpeed = value;

    // Проверяем, не превышена ли скорость
    checkSpeed();
}

private:
    void checkSpeed()
    {
        if (m_sCurrentSpeed > 75.0)
        {
            // Сообщаем водителю, что эта скорость небезопасна
            // Позже мы покажем, как инициализировать события
        }
    }
}

```

Метод доступа `get_Speed()`, принадлежащий к свойству `Speed`, возвращает текущую скорость, которая сохраняется членом `m_sCurrentSpeed`. Метод `set_Speed()`, изменяющий значение свойства `Speed`, проверяет, допустимо ли вводимое значение. Если да, то члену `m_sCurrentSpeed` присваивается значение, передаваемое методу. Если же попытаться присвоить свойству `Speed` отрицательное или слишком большое значение, метод `set_Speed()` выдаст исключение, используя класс `System::ApplicationException`:

```

// Определение исключительной ситуации, специфической для компонента
// Dashboard
public: __gc class DashboardException : public ApplicationException
{
public:
    DashboardException(String* pstrExceptionMsg) :
        ApplicationException(pstrExceptionMsg)
    {
    };
};

```

Класс `DashboardException`, наследующий от `System::ApplicationException`, применяется для генерации специфических для приложения исключительных ситуаций в случае некорректности данных или в соответствии с бизнес-логикой. Наша программа может выдавать исключительную ситуацию `DashboardException` с описанием ошибки. Код ошибки, связанной с исключительной ситуацией этого типа, имеет вид: `COR_E_APPLICATION (0x80131600)`.

Теперь напомним реализацию методов `get` и `set` свойства `GallonsConsumed`:

```

__property float get GallonsConsumed()
{
    return m_sGallonsConsumed;
}

```



```

__property void set_GallonsConsumed(float value)
{
    // Здесь помещается код проверки корректности данных.
    // Для краткости опущен.

    m_sGallonsConsumed = value;

    // Проверим, достаточно ли в баке бензина.
    checkLowFuel();
}

private:
void checkLowFuel()
{
    // Предполагаем, что емкость бака 20 галлонов. Предупреждаем
    // водителя, когда в баке остается не более трех галлонов.
    if(m_sGallonsConsumed >= 17,0)
    {
        // Водителю выдается предупреждение, что в баке мало топлива.
        // Остальной код будет обсуждаться позже, при рассмотрении
        // генерации событий,
    }
}

```

Методы `get_GallonsConsumed()` и `set_GallonsConsumed()` позволяют соответственно получать и изменять значение свойства `GallonsConsumed`. Методы `snackSpeed()` и `checkLowFuel()`, которые являются закрытыми членами класса, проверяют, находятся ли в установленных пределах значения свойств `Speed` и `GallonsConsumed`. Если результат проверки отрицателен, генерируются события. В следующем разделе мы обсудим, как компонент `Dashboard` может отправлять уведомления о событиях.

Реализация метода `computeMilesPerGallon()` возвращает эффективность использования топлива транспортным средством, рассчитывая количество миль, пройденных на одном галлоне бензина:

```

float computeMilesPerGallon(int nMiles)
{
    // Выполнение проверки данных. Для краткости код опущен.
    return static_cast<float>(nMiles)/m_sGallonsConsumed;
}

```

На этом мы пока прекратим добавлять функциональность в компонент `Dashboard`. Имеющегося кода вполне достаточно для того, чтобы его можно было вызвать из неуправляемого клиентского приложения COM и продемонстрировать, как оба типа кода работают вместе. Для разнообразия добавим наш компонент в GAC. Поэтому прежде, чем нажать кнопку `Build`, перейдем в командную строку и создадим новый ключ:

```
> sn -k keypair.snk
```

Затем откроем файл `AssemblyInfo.cpp` и изменим атрибут `AssemblyKeyFile`:

```
using namespace System.Reflection;
[assembly:AssemblyKeyFileAttribute("keypair.snk")];
```

Наконец, скомпилируем проект, чтобы получить сборку `Dashboard.dll`, и введем команду, которая помещает эту сборку в GAC:

```
> gacutil -i Dashboard.dll
```

Работа неуправляемого приложения с компонентами .NET

Для использования управляемого компонента в неуправляемом COM-совместимом приложении необходимо в первую очередь экспортировать сборку управляемого компонента в библиотеку типов COM, понятную для COM-клиента.

Утилита экспорта библиотек типов и утилита регистрации сборок

Платформа .NET Framework предоставляет две утилиты, позволяющие создать библиотеку типов из управляемой сборки:

- Утилиту **экспорта библиотеки типов (type library exporter)**, файл `tlbexp.exe`, которая генерирует библиотеку типов COM на основе управляемой сборки. Создаваемая таким образом библиотека содержит специфическую для COM информацию о типах, соответствующих управляемым типам в сборке.
- Утилиту **регистрации сборок (assembly registration utility)**, файл `regasm.exe`, позволяющую генерировать библиотеку типов COM из управляемой сборки и регистрировать необходимые управляемые типы, содержащиеся в сборке, делая в реестре записи, специфические для COM.

Можно создать библиотеку типов на основе управляемой сборки `Dashboard.dll` с помощью такой команды:

```
> tlbexp Dashboard.dll /out:Dashboard.tlb
```

Однако намного проще воспользоваться утилитой регистрации сборок, поскольку при этом за один прием не только генерируется библиотека типов, но и выполняется COM-регистрация. Введем следующий текст в командную строку:

```
> regasm Dashboard.dll /tlb:Dashboard.tlb
```

Эта команда выполняет следующие операции:

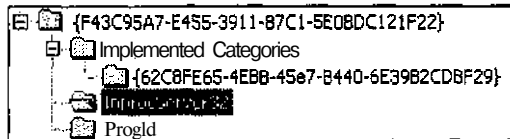
- О Создание библиотеки типов `Dashboard.tlb`, где представлена специфическая для COM информация о типах, относящаяся к открытым управляемым типам, которые содержатся в сборке `Dashboard.dll`.
- Регистрация имеющихся в сборке `Dashboard.dll` открытых управляемых типов (таких, как классы и интерфейсы) для работы в COM путем добавления соответствующих записей в реестр.

Просмотрим записи, которые утилита `regasm` помещает в реестр, чтобы открыть для COM управляемый компонент. Откроем библиотеку типов `Dashboard.tlb`, запустив программу просмотра объектов OLE/COM (`oleview.exe`), найдем в определении `coclass` атрибут `uuid` и скопируем оттуда идентификатор компонента CLSID:

```
[
  uuid(F43C95A7-E455-3911-87C1-5E0BDC121F22),
  version(1.0),
  custom(0F21F359-AB84-41E8-9A78-36D110E6D2F9),
  "DashboardNS.Dashboard")
]
coclass Dashboard(
  interface _Object;
  [default] interface Dashboard;
);
```

Получив идентификатор компонента CLSID, найдем его в реестре. Он должен находиться под ключом CLSID в ветке HKEY_CLASSES_ROOT.

Всякий раз, когда запускается средство экспорта библиотеки типов или утилита регистрации сборок, генерируются различные идентификаторы CLSID. В следующем разделе мы покажем, как можно связать определенный GUID с идентификатором CLSID нашего компонента.



Как видно из ключа разработанного нами компонента Implemented Categories, данный компонент относится к категории .NETCategory. В этом можно убедиться, перейдя в ключ HKEY_CLASSES_ROOT\Component Categories\{62C8FE65-4EBB-45e7-B440-6E39B2CDBF29}.

Щелчком на ключе InProcServer32 под идентификатором компонента CLSID, чтобы отобразить информацию, которую утилита regasm включила в дополнение к тем стандартным записям, какие вносят в реестр типичные COM-серверы. Мы сразу же заметим, что записи Assembly, Class и RuntimeVersion не добавляются обычными COM-серверами и используются для предоставления сборке mscoree.dll дополнительной информации об управляемой сборке и типах управляемого компонента:

Name	Type	Data
(Default)	REG_SZ	C:\WINNT\System32\mscoree.dll
Assembly	REG_SZ	Dashboard, Version=1.0.765.29658, Culture=neutral, PublicKeyToken=null
Class	REG_SZ	Dashboard.Dashboard
Runtime Version	REG_SZ	v1.0.3705
ThreadingModel	REG_SZ	Both

Управляемые компоненты, открытые для COM, объявляют свою потоковую модель как 'both' (оба), что видно из записи ThreadingModel. В COM компонент, применяющий модель потоков 'both', всегда входит в апартмент вызывающего потока. По существу, управляемые компоненты аналогичны использующим модель 'both' компонентам COM, которые могут работать с маршалером свободных потоков.

Когда COM-компонент, применяющий модель 'both', использует маршалер свободных потоков, указатель на интерфейс этого компонента может передаваться другим апартаментам в том же самом процессе без явного маришallingа между апартаментами. Принимающий апартамент напрямую получает указатель на интерфейс компонента без вмешательства прокси.

В случае всех управляемых компонентов, открытых для COM, запись (Default) в ключе InProcServer32 определяет путь к сборке mscoree.dll. Среда исполнения COM загружает эту сборку и вызывает метод DLLGetClassObject(), чтобы получить фабрику классов COM-объекта. Затем сборка mscoree.dll использует ключи реестра Assembly и Class для загрузки нашей сборки и создания управляемого компонента. В этот момент mscoree.dll также создает вызываемую оболочку COM (CCW), которая является представителем COM для управляемого компонента. Наконец, сборка mscoree.dll возвращает указатель IClassFactory оболочки CCW на среду исполнения

COM и завершает свою работу. С этого момента неуправляемый клиент непосредственно взаимодействует с оболочкой CCW, которая, в свою очередь, вызывает управляемый компонент.

Использование управляемого компонента C++

На основе проекта Visual C++ 6.0 MFC создадим диалоговое приложение, вызывающее компонент Dashboard. Вначале необходимо импортировать библиотеку типов COM с помощью директивы компилятора `#import`. Поскольку библиотека `Dashboard.tlb` содержит ссылки на некоторые из специфических типов .NET, нужно импортировать библиотеку типов `mscorlib.tlb`, которая предоставляется платформой .NET Framework и содержит соответствующую информацию о собственных типах CLR:

```
#import "mscorlib.tlb"
#import "Dashboard.tlb" rename namespace( "DashboardNS" )
using namespace DashboardNS
```

Возможно, вам придется добавить путь к файлу `mscorlib.tlb` в список включаемых каталогов. На машине автора этот файл находится в папке `Папка_Windows\Microsoft. Net\Framework\v1.0.3705`

Добавим экземпляр интеллектуального указателя-оболочки на интерфейс `IDashboard` в качестве закрытого члена класса диалога. Такие указатели генерируются с помощью директивы `<import`:

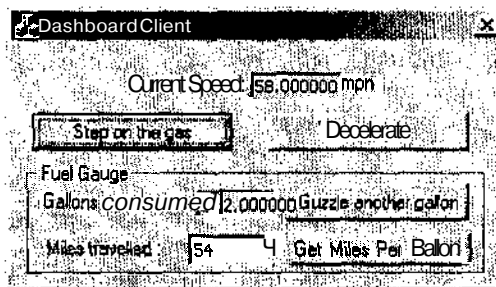
```
class CDashboardClientDlg ; public CDialog
{
    // Макросы диалога MFC и карты сообщений для краткости опущены.
```

```
private:
    IDashboardPtr m_spDashboard;
};
```

В методе нашего приложения `InitInstance()` необходимо инициализировать COM, вызвав функции `AfxOleInit()` или `CoInitialize(NULL)`:

```
bool CDashboardClientApp::InitInstance()
{
    AfxOleInit();
```

```
    // Код инициализации, генерируемый мастером MFC для краткости опущен.
```



Добавим в окно диалога кнопки и текстовые поля (как на рисунке), чтобы можно было взаимодействовать с компонентом `Dashboard`. В функции-члене `OnInitDialog()` класса диалога создадим экземпляр компонента `Dashboard`, после чего получим

текущие значения свойств `Speed` и `GallonsConsumed`. Затем обновим статические текстовые поля в диалоговом окне, задав для них полученные значения свойств:

```

BOOL CDashboardClientDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    // Код инициализации диалога. Для краткости опущен,

    // Создание экземпляра компонента Dashboard
    HRESULT hr = m_spDashboard.CreateInstance(__uuidof(Dashboard));
    // Успешно ли создан компонент?
    if (SUCCEEDED(hr))
    {
        // Получаем текущую скорость и расход топлива
        float fSpeed = m_spDashboard->Speed;
        float fGallonsConsumed = m_spDashboard->GallonsConsumed;

        // Отображаем скорость и расход топлива
        m_strSpeed.Format(_T("%f"), fSpeed);
        m_strGallonsConsumed.Format(_T("%f"), fGallonsConsumed);
        UpdateData(FALSE);
    }

    return TRUE;
}

```

Кнопка `Step on the gas` (Нажать на акселератор) увеличивает значение скорости (`Speed`) на 5 миль в час. Кнопка `Decelerate` (Тормозить) уменьшает значение скорости на ту же самую величину. Ниже приведен код обработчика события `ON_BN_CLICKED` для первой кнопки:

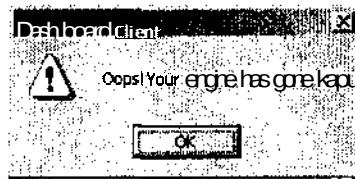
```

void CDashboardClientDlg::OnAccelerate()
{
    try
    {
        if (m_spDashboard != NULL)
        {
            // Увеличиваем скорость на 5 миль в час
            m_spDashboard->Speed = m_spDashboard->Speed + 5;

            // Отображаем скорость в диалоговом окне
            m_strSpeed.Format(_T("%f"), m_spDashboard->Speed);
            UpdateData(FALSE);
        }
    }
    catch (_com_error& e)
    {
        AfxMessageBox(e.Description());
    }
}

```

Будем многократно нажимать на кнопку `Step On the Gas`, чтобы увеличить скорость. Если попытаться превысить барьер 150 миль в час, метод `set_Speed()` компонента `Dashboard` выдаст исключительную ситуацию `DashboardException`. Эта исключительная ситуация преобразуется оболочкой `CCW` в значение `HRESULT`, которое затем возвращается клиенту `COM`. Оболочка `CCW` создает объект ошибки, который предоставляет расширенную информацию об ошибке клиенту `COM` через его интерфейс `IErrorInfo`.



Обработчик `ON_BN_CLICKED` для кнопки **Guzzle another gallon** (Потратить еще один галлон) увеличивает значение свойства `GallonsConsumed` (Расход топлива) на один галлон:

```
void CDashboardClientDlg::OnGuzzleAnotherGallon()
{
    try
    {
        if(m_spDashboard != NULL)
        {
            // Заставим транспортное средство израсходовать
            // еще один галлон топлива
            m_spDashboard->GallonsConsumed += 1;

            // Обновим строку, количество израсходованных галлонов
            m_strGallonsConsumed.Format(_T("%f"),
                                       m_spDashboard->GallonsConsumed);
            UpdateData(FALSE);
        }
    }
    catch(_com_error& e)
    {
        AfxMessageBox(e, Description());
    }
}
```

Видно, что значение свойства `GallonsConsumed` изменяется при нажатии кнопки **Guzzle another gallon**. Попытаемся увеличить данное значение, чтобы оно превысило 20 галлонов, и вновь получим от компонента `Dashboard` исключительную ситуацию `DashboardException`, указывающую на этот раз, что бензобак пуст.

Метод обработки `ON_BN_CLICKED` для кнопки **Get Miles Per Gallon** (Провер в милях на один галлон) показывает эффективность использования топлива транспортным средством:

```
void CDashboardClientDlg::OnGetMilesPerGallon()
{
    try
    {
        if(m_spDashboard != NULL)
        {
            // Получаем пробег в милях
            UpdateData(TRUE);
            long lMilesTraveled = atoi((LPCTSTR)m_strMilesTraveled);
            // Рассчитываем эффективность использования топлива транспортным
            // средством
            float fMilesPerGallon =
                m_spDashboard->computeMilesPerGallon(lMilesTraveled);
            CString strMilesPerGallon;
```

```

        strMilesPerGallon.Format(_T("Fuel efficiency is %f miles/gallon"),
                                fMilesPerGallon);
        AfxMessageBox(strMilesPerGallon);
    }
}
catch(_com_error& e)
{
    AfxMessageBox(e.Description());
}
}

```

Реализовать использование клиентами COM управляемых компонентов исключительно легко. Для этого нужно лишь с помощью утилиты `regasm` сгенерировать библиотеку типов COM на основе управляемой сборки и зарегистрировать данную сборку. В результате неуправляемый клиент может использовать библиотеку типов для создания и вызова методов управляемого компонента, как если бы он работал с компонентом COM. Оболочка CCW реализует всю функциональность, необходимую для соединения двух типов кода.

Получение уведомлений о событиях, иницируемых управляемым компонентом C++

Вернемся к разработанному ранее управляемому компоненту `Dashboard` и сделаем его источником некоторых событий. После этого обсудим, как эти события могут обрабатываться неуправляемым приложением.

Компонент `Dashboard` будет инициировать два события. Первое из них — `OnSafeSpeedLimitExceeded` должно наступать, когда скорость превысит допустимый предел 75 миль в час. Второе событие — `OnLowFuelWarning` будет генерироваться в том случае, если в баке емкостью 20 галлонов останется не более трех галлонов бензина.

Добавление событий в управляемый компонент

Начнем с определения исходящего интерфейса `IDashboardEvents`. Добавим в него методы, представляющие события `OnSafeSpeedLimitExceeded` и `OnLowFuelWarning`. Сделаем `IDashboardEvents` диспетчерским интерфейсом, чтобы с ним мог работать широкий спектр клиентов. Для этого необходимо пометить данный интерфейс атрибутом `InterfaceType` с параметром `ComInterfaceType::InterfaceIsIDispatch`. Это определение должно располагаться вне пространства имен `Dashboard`, которое было создано Visual C++ .NET.

```

[InterfaceType(ComInterfaceType::InterfaceIsIDispatch)]
public __gc interface IDashboardEvents
{
    void OnSafeSpeedLimitExceeded(float sCurrentSpeed);
    void OnLowFuelWarning();
};

```

В разделе "Задание формы экспорта управляемого интерфейса C++ в библиотеку типов COM" мы обсудим другие возможные значения, которые может принимать этот атрибут, и их влияние на определяемый интерфейс.

Для приведенного выше определения управляемого интерфейса `IDashboardEvents` соответствующая секция в определении IDL, которая связана с создаваемой при помощи `tlbexp/regasm` библиотекой типов, имеет вид:

```
[
    uuid(2A02102C-FB9E-306F-9FF3-47232B5119D3),
    version(1.0),
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9}, "IDashboardEvents")
]
dispinterface IDashboardEvents
{
    methods:
        [id(0x60020000)] void OnSafeSpeedLimitExceeded([in] single sCurrentSpeed);
        [id(0x60020001)] void OnLowFuelWarning();
}
```

Эта информация получена с помощью средства просмотра объектов OLE/COM. Далее необходимо объявить делегата для каждого метода исходящего интерфейса. Делегаты, соответствующие методам `OnSafeSpeedLimitExceeded()` и `OnLowFuelWarning()`, которые передают уведомления о событиях, должны иметь вид:

```
__delegate void SpeedLimitDelegate(float sCurrentSpeed);
__delegate void LowFuelWarningDelegate();
```

Теперь нужно объявить события, представляемые этими делегатами, как открытые члены класса компонента `Dashboard`:

```
public __gc class Dashboard : public IDashboard
{
public:
    __event SpeedLimitDelegate* OnSafeSpeedLimitExceeded;
    __event LowFuelWarningDelegate* OnLowFuelWarning;

    // Остальной код реализации класса
};
```

Все выглядит довольно просто, поэтому самое время сделать очередное важное предупреждение. *Событие, которое мы объявляем, должно иметь то же имя, что и метод исходящего интерфейса, представляемый типом делегата события.* Например, имя события, соответствующего делегату `SpeedLimitDelegate`, должно совпадать с именем метода исходящего интерфейса `OnSafeSpeedLimitExceededO`, который представляет данный делегат.

Если указанные имена не совпадают, при наступлении события управляемого компонента мы получим ошибку `DISP_E_UNKNOWNNAME` из сборки `mscorlib.dll`. Причина здесь заключается в том, что для получения идентификатора `DISPID` метода исходящего диспетчерского интерфейса, который реализуется объектом-получателем (неуправляемым кодом), оболочка `CCW` использует метод `IDispatch::GetIDsOfNames()` с именем иницируемого события.

Наконец, для указания классу `Dashboard`, что `IDashboardEvents` будет его исходящим интерфейсом, необходимо пометить данный класс атрибутом `ComSourceInterfaces`. Первый параметр атрибута содержит полностью квалифицированное имя типа интерфейса, иницирующего события. Это также позволяет передавать в мета-данные для средств генерации библиотеки типов информацию о поддержке компонентом исходящего интерфейса:

```
[
    ComSourceInterfaces("IDashboardEvents"),
    ClassInterface(ClassInterfaceType::None)
]
public __gc class Dashboard : public IDashboard
```



```
{
    // Остальной код реализации класса
}
```

В соответствии с этим определением управляемого класса секция coclass в определении IDL для генерируемой библиотеки типов имеет вид:

```
coclass Dashboard
{
    interface _Object;
    [default] interface IDashboard;
    [default, source] dispinterface IDashboardEvents;
}
```

В определении IDL для библиотеки типов, генерируемой с помощью `tibexp/re-gasm`, интерфейс `IDashboardEvents` стал исходящим интерфейсом по умолчанию компонента `Dashboard`. Это показывает атрибут `[default, source]`.

Генерация событий управляемым компонентом

Прежде чем работать с событиями, нужно связать с ними всех делегатов. При этом следует убедиться, что член события не является нулевым. Для генерации события необходимо использовать член события и вызвать его как обычный метод. Это приводит к вызову делегатов, связанных с событием, которые затем передают данное событие для обработки приемникам, подписавшимся на уведомления:

```
void checkSpeed()
{
    if(m_sCurrentSpeed > 75.0)
    {
        if(OnSafeSpeedLim Exceeded) X
        {
            OnSafeSpeedLimitExceeded(m_sCurrentSpeed);
        }
    }
}
```

Когда пользователь устанавливает значение свойства `Speed`, вызывается метод `CheckSpeed()`, который проверяет, не превышен ли предел скорости 75 миль в час. Если он превышен, вызывается событие `OnSafeSpeedLimitExceeded`. Аналогично, когда пользователь задает свойство `GallonsConsumed`, вызывается метод `CheckLowFuel()`. Этот метод выполняет проверку условия, что в баке емкостью 20 галлонов осталось не менее 3 галлонов топлива. Если результат проверки оказывается отрицательным, генерируется событие `OnLowFuelWarning`:

```
void checkLowFuel()
{
    if(m_sGallonsConsumed >= 17.0)
    {
        if(OnLowFuelWarning)
        {
            OnLowFuelWarning();
        }
    }
}
```

Поскольку мы добавили поддержку уведомлений о событиях в компонент `Dashboard`, необходимо зарегистрировать в GAC обновленную сборку `Dashboard.dll`. Для обновления номера версии сборки нужно изменить атрибут `AssemblyVersionAttribute`

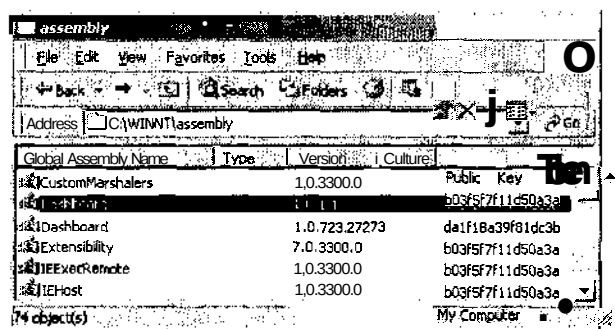
в файле AssemblyInfo.cpp. Установим номера ревизии и компоновки равными 1 в информации о версиях, которая имеет формат: *СтаршаяВерсия. МладшаяВерсия. НомерКомпоновки.Ревизия*:

```
[assembly:AssemblyVersionAttribute("1.0.1.1")];
```

После компоновки сборки зарегистрируем ее в GAC, используя ту же команду, что и ранее:

```
> gacutil -i Dashboard.dll
```

Открыв GAC, убедимся, что там присутствуют обе версии сборки Dashboard.dll:



Наконец, в связи с добавлением исходящего интерфейса и изменением определения `class`, заново создадим библиотеку типов, которая будет содержать обновленную информацию о типах, присутствующих в нашей сборке. Кроме того, перерегистрируем типы в COM. Для этого в командной строке DOS введем следующий текст:

```
> regasm Dashboard.dll /tlb:Dashboard.tlb
```

Получение неуправляемым приложением C++ уведомлений о событиях управляемого кода C++

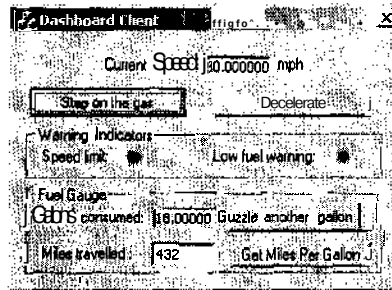
Добавим в клиентское приложение MFC код подписки на уведомления о событиях управляемого компонента Dashboard. Полный текст приложения с учетом этих изменений можно найти в папке загружаемого кода DashboardClient.

Вначале внесем небольшое изменение в директиву `(import, добавив туда опцию named_guids`. Это делается в связи с тем, что нам нужен идентификатор GUID библиотеки типов компонента Dashboard для создания приемника события в клиентском приложении. Опция `named_guids` заставляет компилятор генерировать в файле с расширением `.tlh` значения GUID для библиотеки типов, классов и интерфейсов.

```
#import "Dashboard.tlb" rename namespace("DashboardNS"), named_guids
```

На этом этапе необходимо перекомпоновать приложение MFC, чтобы директива `#import` сгенерировала обновленные оболочки для новой библиотеки типов.

Затем добавим в созданный ранее диалог панель индикатора предупреждения, а также некоторые статические элементы управления, включая два графических элемента, которые будут зажигать красные предупреждающие сигналы. Эти сигналы будут видны всякий раз, когда приложение получит соответствующее уведомление о событии от компонента Dashboard. Так мы сможем отобразить информацию о том, что водитель превысил скорость или в баке недостаточно топлива:



Для создания объекта-приемника, реализующего исходящий интерфейс `IDashboardEvents`, добавим в проект объект ATL COM с именем `DashboardSink`, используя мастер ATL Object Wizard. В результате этой операции в наш проект MFC будет добавлена поддержка ATL. Теперь нужно, чтобы объект `DashboardSink` реализовал интерфейс `IDashboardEvents`, а затем необходимо связать данный объект-приемник, являющийся подписчиком, с уведомлениями о событиях компонента `Dashboard`. Ниже перечислены этапы этого процесса:

1. Создадим класс `CDashboardSink`, производный от специализации шаблона `IDispatchImpl<>` для диспетчерского интерфейса `IDashboardEvents`. Шаблонный класс `IDispatchImpl` освобождает нас от работы по реализации методов интерфейса `IDispatch` в интерфейсе `IDashboardEvents`:

```
#define IDC_DISP_EVENT_ID 1

class ATL_NO_VTABLE CDashboardSink :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CDashboardSink, &CLSID_DashboardSink>,
public ISupportErrorInfo,
public IDispatchImpl<IDashboardSink, &IID_IDashboardSink,
&LIBID_DashboardClientLib>,
public IDispatchImpl<IDC_DISP_EVENT_ID, CDashboardSink,
&IID_IDashboardEvents, &LIBID_DashboardNS, 1, 0>
{
    // Остальной код определения класса
}
```

2. Создадим карту приемника события, которая связывает `DISPID`-идентификатор события с конкретным обработчиком в классе `CDashboardSink`. Для этого нужно знать `DISPID`-идентификаторы методов `OnSafeSpeedLimitExceeded()` и `OnLowFuelWarning()` в интерфейсе `IDashboardEvents`. Если открыть библиотеку типов `Dashboard.tlb` с помощью средства просмотра объектов OLE/COM, мы увидим, что методам `OnSafeSpeedLimitExceeded()` и `OnLowFuelWarning()` соответственно присвоены идентификаторы `DISPID 0x60020000` и `0x60020001`. Ниже представлен код приемника события, который передает идентификаторы события `DISPID` соответствующим методам обработки, реализованным в классе `CDashboardSink`:

```
BEGIN_SINK_MAP(CDashboardSink)
    SINK_ENTRY_EX(IDC_DISP_EVENT_ID, DIID_IDashboardEvents, 0x60020000,
OnSafeSpeedLimitExceeded)
    SINK_ENTRY_EX(IDC_DISP_EVENT_ID, DIID_IDashboardEvents, 0x60020001,
OnLowFuelWarning)
END_SINK_MAP()
```

3. Добавим обработчики событий *OnLowFuelWarning* и *OnSafeSpeedLimitExceeded* в класс *CDashboardSink*. Эти обработчики используются для того, чтобы делать видимыми статические элементы управления диалога, которые отображают индикаторы предупреждения красного цвета. Класс *CDashboardSink* первоначально должен сохранить дескрипторы окон для этих статических элементов управления. Поэтому создадим метод *SetWarningLights()*. Диалог приложения MFC при своей загрузке будет вызывать этот метод, передавая ему дескрипторы окон для статических элементов управления, которые отображают рисунки. Эти дескрипторы затем сохраняются в переменных-членах и позже используются при получении уведомлений о событиях:

```
void __stdcall OnSafeSpeedLimitExceeded(float fCurrentSpeed)
{
    ShowWindow(m_hwndSpeedWarningLight, SW_SHOW);
}

void __stdcall OnLowFuelWarning()
{
    ShowWindow(m_hwndLowFuelWarningLight, SW_SHOW);
}

void SetWarningLights(HWND hwndSpeedWarningLight, HWND hwndLowFuelWarningLight)
{
    m_hwndSpeedWarningLight = hwndSpeedWarningLight;
    m_hwndLowFuelWarningLight = hwndLowFuelWarningLight;
}
```

Осталось лишь подписаться на уведомления о событиях компонента *Dashboard*. Ниже перечислены необходимые для этого этапы связывания объекта *DashboardSink*:

1. Добавим в класс диалога MFC переменную-член типа

*CComObject<CDashboardSink>**:

```
class CDashboardClientDlg : public CDialog
{
    // Код, генерируемый мастером MFC. Для краткости опущен.

private:
    // Интеллектуальный указатель-оболочка для интерфейса IDashboard
    IDashboardPtr m_spDashboard;

    // Объект-приемник событий компонента Dashboard
    CComObject<CDashboardSink>* m_pDashboardSink;
};

CDashboardClientDlg::CDashboardClientDlg(CWnd* pParent /*= NULL*/):
    CDialog(CDashboardClientDlg::IDD, pParent)
{
    // Код, генерируемый мастером MFC. Для краткости опущен.

    // Инициализация указателя
    m_pDashboardSink = new CComObject<CDashboardSink>();
}
```

2. В методе диалога *OnInitDialog()* создадим экземпляр объекта-получателя и вызовем его метод *DispEventAdvise()*, которому в качестве параметра передадим указатель на компонент *Dashboard*, инициирующий события.

В результате будет вызван метод `IConnectionPoint::Advise()` CCW-оболочки компонента `Dashboard` и реализуется связывание объекта `DashboardSink` уведомлениями о событиях компонента `Dashboard`:

```

BOOL CDashboardClientDlg::OnInitDialog()
{
    // Код диалога, генерируемый мастером MFC. Для краткости опущен.

    // В начале скрываем индикаторы предупреждения. Они должны быть видны
    // при получении соответствующих событий от компонента Dashboard.
    m_ctrlSpeedIndicator.ShowWindow(SW_HIDE);
    m_ctrlLowFuelIndicator.ShowWindow(SW_HIDE);

    // Создание экземпляра компонента Dashboard
    HRESULT hr = m_spDashboard.CreateInstance(__uuidof(Dashboard));

    // Успешно ли создан экземпляр?
    if(SUCCEEDED(hr))
    {
        // Создание экземпляра объекта-получателя, который будет принимать
        // уведомления о событиях от компонента Dashboard.
        hr = CComObject<CDashboardSink>::CreateInstance(&m_pDashboardSink);
        if(FAILED(hr)) AfxMessageBox(_T("Sink object creation failed"));
        if(m_pDashboardSink)
        {
            // Увеличиваем количество ссылок
            m_pDashboardSink->AddRef();

            // Передаем дескрипторы окон для статических элементов
            // управления диалогом получателю уведомлений о событиях,
            // В результате при получении диалогом событий
            // от компонента Dashboard индикаторы предупреждения будут
            // отображаться на экране.
            m_pDashboardSink->SetWarningLights(m_ctrlSpeedIndicator.m_hWnd,
                                              m_ctrlLowFuelIndicator.m_hWnd);

            // Подписка на уведомления о событиях компонента Dashboard
            hr = m_pDashboardSink->DispEventAdvise(m_spDashboard);
            if(FAILED(hr)) AfxMessageBox(_T("Failed in ICP: Advise"));
        }

        // Получение текущей скорости и количества израсходованного топлива
        float fSpeed = m_spDashboard->Speed;
        float fGallonsConsumed = m_spDashboard->GallonsConsumed;

        // Отображение скорости и количества израсходованного топлива
        m_strSpeed.Format(_T("%f"), fSpeed);
        m_strGallonsConsumed.Format(_T("%f"), fGallonsConsumed);
        UpdateData(FALSE);
    }

    return TRUE;
}

```

3. В деструкторе класса диалога вызовем метод `DispEventUnadvise()` объекта `DashboardSink`, чтобы удалить связь между этим объектом и компонентом `Dashboard`, инициирующим события. При этом вызывается метод `IConnectionPoint::Unadvise()` CCW-оболочки компонента `Dashboard`:

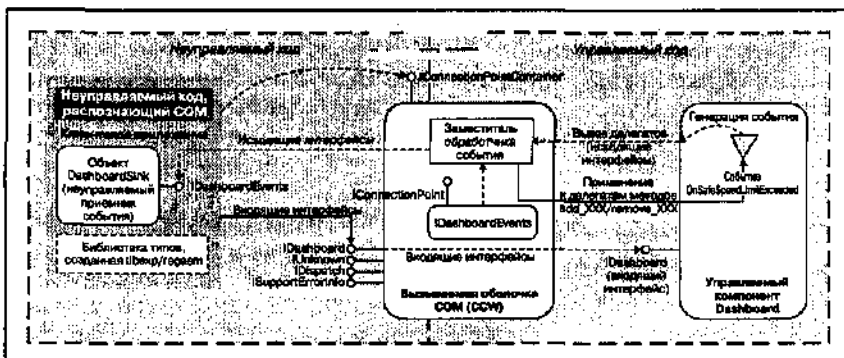
```

CDashboardClientDlg::~CDashboardClientDlg()
{
    if(m_pDashboardSink)
    {
        // Удаляем связь между получателем и источником события
        if(m_pDashboardSink->m_dwEventCookie != 0xFEFEFEFE)
        {
            m_pDashboardSink->DispEventUnadvise(m_spDashboard);
        }
        // Освобождаем ссылку
        m_pDashboardSink->Release();
        m_pDashboardSink = NULL;
    }
}

```

Как видим, довольно просто сделать так, чтобы неуправляемый клиент мог получать уведомления о событиях от управляемого компонента C++ (по крайней мере, это ненамного сложнее, чем работа с неуправляемым источником события). На внутренних уровнях оболочка CCW синтезирует интерфейсы, такие как `IConnectionPointContainer`, требуемые протоколом обработки событий с использованием точек подключения. CCW создает также реализацию интерфейса `IConnectionPoint` для каждого исходящего интерфейса. Эти реализации обнаруживаются неуправляемым клиентом с помощью метода `IConnectionPointContainer::FindConnectionPoint()`.

Реализации методов `IConnectionPoint::Advise()` и `IConnectionPoint::Unadvise()` добавляют и удаляют делегатов, ссылающихся на создаваемых оболочкой CCW заместителей обработчиков события управляемого компонента. Когда управляемый компонент инициирует событие, эти заместители, принадлежащие к оболочке CCW, вызываются с помощью делегатов, связанных с событием. Заместители транслируют событие неуправляемым приемникам, подписанным на уведомления о событии. Приведенный ниже рисунок демонстрирует "с высоты птичьего полета" взаимодействие между неуправляемым клиентом и управляемым компонентом-источником события:



С точки зрения неуправляемого клиента, нет никаких признаков того, что он получает уведомления о событиях от управляемого компонента C++. Поэтому разработчику нет необходимости разбираться в тонкостях технологии, склеивающей механизм делегатов .NET с механизмом точек подключения COM. Оболочка CCW делает все необходимое для соединения этих несопоставимых механизмов работы с событиями, обеспечивая взаимодействие между управляемым и неуправляемым кодом.

Размещение элементов управления форм Windows в неуправляемых контейнерах

Ранее мы показали, насколько проста процедура добавления неуправляемого элемента управления ActiveX на форму .NET Windows Form. Возникает вполне естественный вопрос, справедливо ли обратное, т.е. можно ли размещать элементы управления Windows Forms в неуправляемых контейнерах, таких как формы Visual Basic 6.0, диалоги MFC и т.д. К сожалению, Microsoft решила, что, хотя это в принципе осуществимо, попытка реализовать такую возможность породит множество проблем, связанных с разработкой ПО, а спрос на подобную технологию будет невысоким. Поэтому официально Microsoft не поддерживает размещение элементов управления Windows Forms в неуправляемых контейнерах.

К моменту написания книги единственным неуправляемым контейнером, куда можно добавлять элементы управления Windows Forms, являлся обозреватель Internet Explorer (IE) версии 5.x или выше. Чтобы сделать это, нужно просто использовать тег `<object>`, как при размещении элементов ActiveX на страницах HTML:

```
<html>
<title>Hosting Windows Forms controls in IE 5.x</title>
</head>
<body>
<object id="ClockCtrl1" classid="clsid:ClockNS.ClockCtrl1"
program="ClockControl.exe" value="Tick Tock">
</object>
</body>
</html>
```

Атрибут `classid` в теге `<object>` предоставляет имя сборки и полностью квалифицированное имя типа для элемента управления Windows Forms в формате `URL#Сборка#Полностью_Квалифицированное_Имя_Типа_Элемента_Управления`. Значение атрибутов `id`, `width` и `height` остается прежним, а тег `<param>`, как и следовало ожидать, позволяет задавать величину свойств элемента управления.

Для тех, кому нужно размещать элементы управления Windows Forms в других неуправляемых контейнерах, сообщим, что в данной области существует неофициальная деятельность. В настоящий момент Morgan Skinner предоставляет информацию о том, как это можно сделать, на сайте www.codeproject.com/netcomponents/exposingdotnetcontrols.asp

Задание формы экспорта управляемого класса C++ в библиотеку типов COM

Здесь мы обсудим использование атрибутов в управляемом классе C++ для указания того, в каком виде этот класс должен предстать перед COM. Атрибут `ClassInterface` позволяет выбрать, какое определение `coclass` будет сгенерировано при экспорте управляемого компонента C++ в библиотеку типов COM.

В примере с компонентом `Dashboard` был определен интерфейс `IDashboard`, содержащий свойства и методы, которые мы хотели бы открыть для COM. Для реализации этого интерфейса использовался класс `Dashboard`. Можно добавлять методы и свойства непосредственно в класс `Dashboard` вместо того, чтобы явным образом использовать наследование и реализацию интерфейса `IDashboard`. При этом все методы и свойства по-прежнему будут доступны неуправляемым COM-клиентам. Покажем, почему такой подход запрещен.

Атрибут *ClassInterface*

Рассмотрим следующий управляемый класс C++:

```
[ClassInterface(ClassInterfaceType::AutoDual)]
public __gc class SportsCar
{
public:
    SportsCar() {}
    void GoVroom() {}
};
```

Класс `SportsCar` не наследует явно от какого-либо интерфейса. Он имеет открытый метод `GoVroom()`. Когда класс `SportsCar` экспортируется в библиотеку типов COM помощью утилит `tlbexp` или `regasm`, создается интерфейс по умолчанию `_SportsCar` с именем, составленным из имени класса и префикса в виде символа подчеркивания. Такой интерфейс называется **интерфейсом класса (class interface)**. Он содержит методы класса `System::Object` (от которого неявно наследует управляемый класс) в дополнение к другим открытым методам управляемого класса, таким как `GoVroom()`:

```
[
    odl,
    uuid(BFC2A389-5927-3097-AD19-810CE1693396),
    hidden,
    dual,
    nonextensible,
    oleautomation,
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
        "RaceTrackNS.SportsCar")
]
interface _SportsCar : IDispatch
{
    [id(00000000), project]
    HRESULT ToString([out, retval] BSTR* pRetVal);
    [id(0x60020001)]
    HRESULT Equals([in] VARIANT obj, [out, retval] VARIANT_BOOL* pRetVal);
    [id(0x60020002)]
    HRESULT GetHashCode([out, retval] long, pRetVal);
    [id(0x60020003)]
    HRESULT GetType([out, retval] Type* pRetVal);
    [id(0x60020004)]
    HRESULT GoVroom();
};

[
    uuid(AB0BCF4B-7D3A-33B2-A3F3-D7E958139E46),
    version(1.0),
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
        "RaceTrackNS.SportsCar")
]
coclass SportsCar
{
    [default] interface _SportsCar;
    interface _Object;
};
```

Один из недостатков такого подхода заключается в следующем. Поскольку методы управляемого класса помещаются непосредственно в интерфейс по умолчанию, это может создать для неуправляемых клиентов массу проблем, связанных

с конфликтом версий. Представьте себе, что может случиться, если кто-то добавит еще один открытый метод в управляемый класс, не подозревая, что существуют неуправляемые клиенты, которые также используют этот компонент. Когда обновленный компонент предоставляется неуправляемым клиентам через COM, новый метод становится частью интерфейса класса (`_SportsCar`), не позволяя клиентам использовать старый компонент из-за изменений в таблице `vtable`. Проблемы возникают даже у неуправляемых клиентов, применяющих позднее связывание и использующих кэширование идентификаторов `DISPID` (полученных через определение интерфейса), поскольку данные идентификаторы могут быть перетасованы или сгенерированы заново после добавления в класс новых методов.

Поздно связанные клиенты, использующие метод `IDispatch::GetIDsOfNames()` для динамического получения `DISPID`-идентификаторов, продолжают работу, так как они будут иметь новые идентификаторы. Очевидно, что управляемые клиенты также будут работать. Однако из соображений безопасности предоставление управляемого компонента объектам COM через интерфейс класса указанного компонента обычно не рекомендуется, и следует избегать использования атрибута `ClassInterface` с параметром `ClassInterfaceType::AutoDual`. Ниже представлены возможные значения параметра `ClassInterfaceType`:

Перечисление <code>ClassInterfaceType</code>	Управляемый класс будет экспортироваться в COM как...
<code>AutoDual</code>	Класс, интерфейс которого по умолчанию представляет собой интерфейс класса и который поддерживает как раннее, так и позднее связывание.
<code>AutoDispatch</code>	Класс, интерфейс которого по умолчанию представляет собой интерфейс класса и который поддерживает только позднее связывание. Такое же поведение по умолчанию имеют неуправляемые классы, экспортируемые в COM, если они не используют атрибут <code>ClassInterface</code> .
<code>None</code>	Класс, интерфейс которого по умолчанию реализуется явно.

Проведем следующий эксперимент, задав в нашем управляемом классе C++ значение атрибута `ClassInterface` равным `ClassInterfaceType::AutoDispatch`:

```
[ClassInterface(ClassInterfaceType::AutoDispatch)]
public __gc class SportsCar
{
public:
    SportsCar() {}
    void GoVroom() {}
};
```

Здесь также не используется явное наследование от управляемого интерфейса. Вместо этого управляемый класс непосредственно предоставляет свои методы и свойства. Когда этот управляемый класс экспортируется в COM, секции `coclass` и `interface` в определении IDL для генерируемой библиотеки типов выглядят следующим образом:

```
[
    uuid{713BE9EF-8A61-3A6D-90CA-1F203F56532E},
    version(1.0),
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9}, "RaceTrackNS.SportsCar")
]
coclass SportsCar
```

```

[default] interface _SportsCar
interface _Object:
{
    odl,
    uuid(F8DBBC03-868D-307C-86B4-36C23EB7E8D3),
    hidden,
    dual,
    oleautomation,
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
        "RaceTrackNS.SportsCar")
}
interface _SportsCar : IDispatch
{
    ...
};

```

На этот раз определение интерфейса класса (`_SportsCar`) в файле IDL не содержит ни одного метода. Это означает, что раннее связывание невозможно. Управляемый класс C++ может использоваться неуправляемым клиентом COM только через позднее связывание. Очевидное следствие такого подхода заключается в том, что больше не возникает неприятностей с версиями, связанных с интерфейсом класса. Можно добавлять методы и свойства в класс `SportsCar`, и новые управляемые клиенты будут работать с ними. Когда же старые неуправляемые клиенты воспользуются методом `IDispatch::GetIDsOfNames()` для получения идентификаторов DISPID, они получат обновленные значения этих идентификаторов и смогут продолжить работу со старыми методами и свойствами.

Оба описанных выше подхода имеют свои недостатки. В идеале следует явным образом собрать в одном интерфейсе все методы, которые мы хотим предоставить клиентам, а управляемый класс C++ должен наследовать от данного интерфейса и реализовывать его методы. Затем нужно отменить создание интерфейса класса, задавая значение `ClassInterfaceType: None` для параметра атрибута `ClassInterface` (именно так мы поступили ранее, когда экспортировали в COM компонент `Dashoccard`):

```

public __gc interface IFastVehicle
{
    void GoVroom();
};

[ClassInterface(ClassInterfaceType: None)]
public __gc class SportsCar : public IFastVehicle
{
public:
    SportsCar() {}
    void GoVroom() {}
};

```

В интерфейсе `IFastVehicle` мы объединили методы, с которыми должны работать неуправляемые клиенты, а затем реализовали наследование от этого интерфейса в классе `SportsCar`. Кроме того, приведенный код отменяет создание интерфейса класса, поскольку класс `SportsCar` помечен атрибутом `ClassInterface` с параметром `ClassInterfaceType: None`.

Посмотрим на соответствующие секции IDL в генерируемой библиотеке типов:

```

[
    odl,

```

```

        uuid(7F02B6D5-C469-3A36-85A3-E256E996F321),
        version(1.0),
        guid(
            oleautomation,
            custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
                "RaceTrackNS.IFastVehicle")
        ]
    }
}

```

```

interface IFastVehicle : IDispatch
{
    [id(0x60020000)] HRESULT GoVroom();
};

```

```

[
    uuid(2D98AFB1-A19C-3374-A4F0-4751DEFC509E),
    version(1.0),
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
        "RaceTrackNS.SportsCar")
]

```

```

coclass SportsCar
{
    interface __object
    [default] interface IFastVehicle;
};

```

Интерфейсом по умолчанию компонента `SportsCar` является интерфейс `IFastVehicle`, который позволяет использовать как раннее, так и позднее связывание. Кроме того, добавление новых методов и свойств в класс `Dashboard` не нарушит работу существующих неуправляемых клиентов COM до тех пор, пока сохраняется контракт интерфейса `IFastVehicle`.

Атрибуты *ProgId* и *Guid*

Атрибут `ClassInterface` может оказаться очень полезным, когда необходимо выбрать, в каком виде будет экспортирован в COM управляемый класс. Мы приведем краткий обзор двух других атрибутов, которые позволяют задавать значения `ProgID` для экспортируемых в COM управляемых классов и величины `GUID` для управляемых классов и интерфейсов, которые должны работать с COM.

По умолчанию утилиты `tlbexp` и `regasm` присваивают регистрируемому в COM управляемому классу идентификатор `ProgID`, совпадающий с полностью квалифицированным именем типа этого класса. Однако можно изменить идентификатор `ProgID` управляемого класса C++, пометив его атрибутом `ProgId`:

```

[
    ClassInterface(ClassInterfaceType::None),
    ProgId("AcmeCompany.SportsCar")
]
public __gcclass SportsCar : public IFastVehicle
{
    // Остаток кода определения класса
};

```

Утилиты `tlbexp` и `regasm` генерируют также произвольное значение `GUID` для идентификатора компонента `CLSID` всякий раз, когда заново компилируется управляемый компонент C++. Однако для интерфейсов величина `IID` не меняется от компоновки к компоновке, если сигнатуры методов в интерфейсе остаются прежними и туда не добавляются новые методы. В случае же, когда изменяется определение

интерфейса, генерируется новый идентификатор IID. Чтобы сохранить имеющиеся идентификаторы CLSID или IID при выполнении новой компоновки или изменении определения интерфейса, можно задать указанные идентификаторы по нашему выбору. Для этого нужно пометить управляемый класс или интерфейс C++ атрибутом Guid:

```
[
    ClassInterface(ClassInterfaceType::None),
    Guid("67881D41-FCFF-421b-9D47-699C5DFF5E3D")
]
public __gc class SportsCar : public IFastVehicle
{
    // Остальной код определения класса
};
```

Задание формы экспорта управляемого интерфейса C++ в библиотеку типов COM

Управляемый интерфейс C++ по умолчанию экспортируется в COM как дуальный интерфейс. Однако может возникнуть необходимость представить его в COM как диспетчерский или как специализированный интерфейс. Это позволяет сделать атрибут `InterfaceType`, который принимает одно из следующих значений перечисления `ComInterfaceType`:

Перечисление <code>ComInterfaceType</code>	Управляемый интерфейс будет экспортироваться в COM как...
<code>InterfaceIsDual</code>	Дуальный интерфейс
<code>InterfaceIsDispatch</code>	Чистый диспетчерский интерфейс
<code>InterfaceIsUnknown</code>	Специализированный интерфейс

Используем каждое из этих значений в качестве параметра атрибута `InterfaceType` и посмотрим, как при этом меняется определение IDL в генерируемой библиотеке типов:

1. Пометим управляемый интерфейс C++ атрибутом `InterfaceType` с параметром `ComInterfaceType::InterfaceIsDual`:

```
[InterfaceType(ComInterfaceType::InterfaceIsDual)]
public __gc interface IVehicle
{
};
```

Этот управляемый интерфейс C++ экспортируется в COM как дуальный интерфейс, что видно из атрибута `[dual]` в IDL-определении интерфейса:

```
[
    odl,
    uuid(77E5CCF0-53BD-36DE-9346-E8C2BEC521F6),
    version(1.0),
    dual,
    oleautomation,
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
        "VehicleNS.IVehicle")
]
interface IVehicle : IDispatch
{
};
```

```
};
```

Если атрибут `InterfaceType` не используется, средство экспорта библиотеки типов и утилита регистрации сборок передают интерфейс в COM как дуальный.

2. Теперь присоединим к управляемому интерфейсу C++ атрибут `InterfaceType` с параметром `ComInterfaceType::InterfaceIsDispatch`

```
[InterfaceType(ComInterfaceType::InterfaceIsDispatch)]
public __gc__ interface IVehicle
{
    // ...
};
```

Такой управляемый интерфейс C++ экспортируется в COM как чистый диспетчерский интерфейс. Это следует из определения IDL:

```
[
    uuid(77E5CCF0-53BD-36DE-9346-E8C2BEC521F6),
    version(1.0),
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
        "VehicleNS.IVehicle")
]
dispinterface IVehicle
{
    properties:
};
```

3. Наконец, пометим управляемый интерфейс C++ атрибутом `InterfaceType` с параметром `ComInterfaceType::InterfaceIsUnknown`

```
[InterfaceType(ComInterfaceType::InterfaceIsUnknown)]
public __gc__ interface IVehicle
{
    // ...
};
```

В результате управляемый интерфейс C++ экспортируется в COM как специализированный интерфейс `IUnknown`, что демонстрирует IDL-определение интерфейса:

```
[
    odl,
    uuid(77E5CCF0-53BD-36DE-9346-E8C2BEC521F6),
    version(1.0),
    oleautomation,
    custom({0F21F359-AB84-41E8-9A78-36D110E6D2F9},
        "VehicleNS.IVehicle")
]
interface IVehicle: IUnknown
{
    // ...
};
```

Как видим, атрибут `InterfaceType` может быть весьма полезным, когда необходимо задать ту форму, в какой интерфейс C++ будет экспортирован в COM.

ГЛАВА 9



COM-программирование с помощью ATL

В настоящей главе мы оставим в стороне .NET-программирование и обсудим то новое, что предлагает последняя версия C++ для программистов ATL и COM. При том повышенном интересе (и шумихе) вокруг .NET и управляемых расширений следует отметить огромную работу, проделанную для улучшения существующих средств "неуправляемого" языка C++. Библиотека ATL подверглась существенной переработке. Теперь имеется версия ATL 7.0, которая поддерживает новые атрибуты C++, облегчающие создание COM-объектов.

Может возникнуть вопрос, почему после появления управляемого языка C++ и .NET по-прежнему сохраняется значительный интерес к ATL-программированию. Для этого есть две веские причины:

- ▣ Необходимость поддержки унаследованных приложений
- Спрос на новые COM-компоненты

До появления .NET практически любое приложение, разработанное Microsoft, имело в своей основе COM. Использование общих элементов управления в Visual Basic, встраивание одних документов Office в другие, применение Web-интерфейса Outlook — все эти технологии используют COM как средство, соединяющее вместе различные детали. Фактически, многие классы в библиотеке классов .NET Framework — это не более, чем оболочки вокруг вызовов функций API и компонентов COM. Отсюда следует, что неизбежно воздействие на производительность системы, связанное с использованием .NET. Оно может быть незначительным, и со временем приложения .NET, возможно, будут работать даже быстрее, чем обычные исполняемые файлы. Однако если мы сейчас хотим создавать облегченные компоненты, имеющие небольшой размер и максимальную эффективность, то по-прежнему будем иметь дело с COM и ATL. И в перспективе в течение определенного времени эти технологии будут иметь большое значение.

Начнем с краткого обзора новых возможностей ATL. Не вдаваясь в подробности упомянутых нововведений, мы разработаем несколько небольших проектов, которые должны дать представление о том, как работает ATL 7.0. Вначале без использования новых атрибутов создадим простой внутрипроцессный сервер, который позволит продемонстрировать изменения мастеров и генерируемого ими кода. Затем повторим данный процесс, но на этот раз с применением атрибутов, в результате чего будет видно, насколько они упрощают работу. После этого мы рассмотрим некоторые классы-утилиты ATL и разработаем демонстрационный компонент COM с использованием ряда новых средств.

Мы предполагаем, что читатель уже знаком с ATL 3.0, и сосредоточимся на новшествах ATL 7.0.

Что нового в ATL

Ниже перечислены наиболее важные новые особенности ATL 7.0:

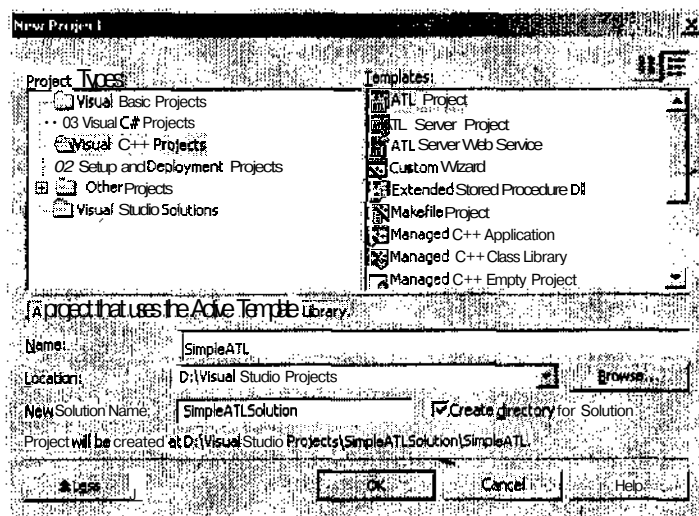
- **Программирование для COM с использованием атрибутов.** — В главе 4 мы показали, как работают атрибуты. Они значительно упрощают написание кода для объектов COM, реализуя дополнительный уровень препроцессорной обработки (гораздо более мощный, чем препроцессор C++). Атрибуты способны предоставить достаточно информации о компонентах компилятору C++ для того, чтобы исключить необходимость создания отдельного IDL-файла и позволить лучше отделить спецификацию объектов от кода их реализации.
- **ATL Server**, обеспечивающий в ATL поддержку написания расширений ISAPI и Web-служб. — ATL Server не предназначен для разработки компонентов COM. Это совершенно новая библиотека, которая служит для создания высокопроизводительных Web-приложений.
- **Лучшая интеграция с MFC.** — Например, некоторые классы (такие, как CPoint и CSize) переписаны и теперь совместно используются ATL и MFC, позволяя библиотеке MFC извлекать выгоду из облегченного подхода ATL. Следовательно, теперь стало гораздо легче соединять в одном проекте ATL и MFC.
- **Изменения в структуре классов ATL.** — Например, класс CComModule исключен и заменен более мощными классами, такими как CAtlDllModule и CAtlExeModule.

Просматривая документацию ATL, мы обнаружим большое количество новых **служебных классов**, относящихся к таким сферам, как работа с файлами, преобразование данных и коллекции. Некоторые классы окон, которые ранее были реализованы в официально не поддерживаемой библиотеке WTL (Windows Template Library), теперь нашли свое место в ATL 7.0. Программисты, работающие со строками, вздохнут с облегчением, узнав, что, благодаря новому классу CAtlString, ATL имеет теперь столь же мощные средства обработки строк, как и MFC. К моменту написания книги библиотека WTL не была обновлена, однако имеются указания на то, что Microsoft работает над новой, более мощной версией WTL для использования с ATL 7.0.

ATL Server является предметом обсуждения в следующей главе.

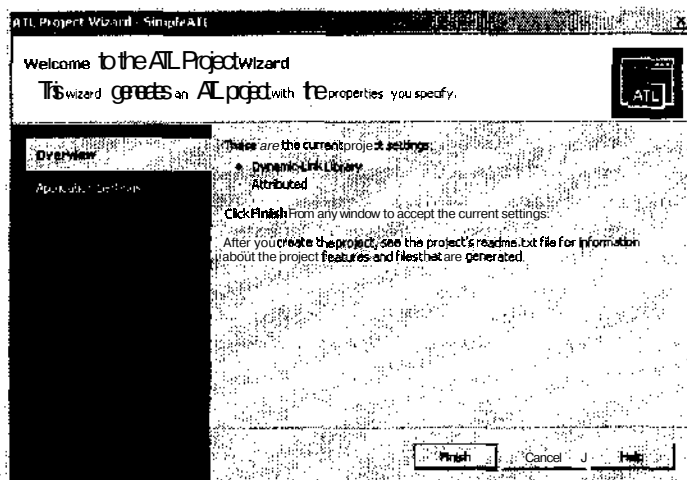
Простой проект ATL 7.0

Начнем с разработки базового объекта с помощью ATL. Пока мы не будем пользоваться атрибутами, поскольку структура классов ATL определенным образом изменилась и применение атрибутов может сделать эти изменения не столь наглядными. Откроем Visual Studio .NET и создадим новый проект ATL Project:

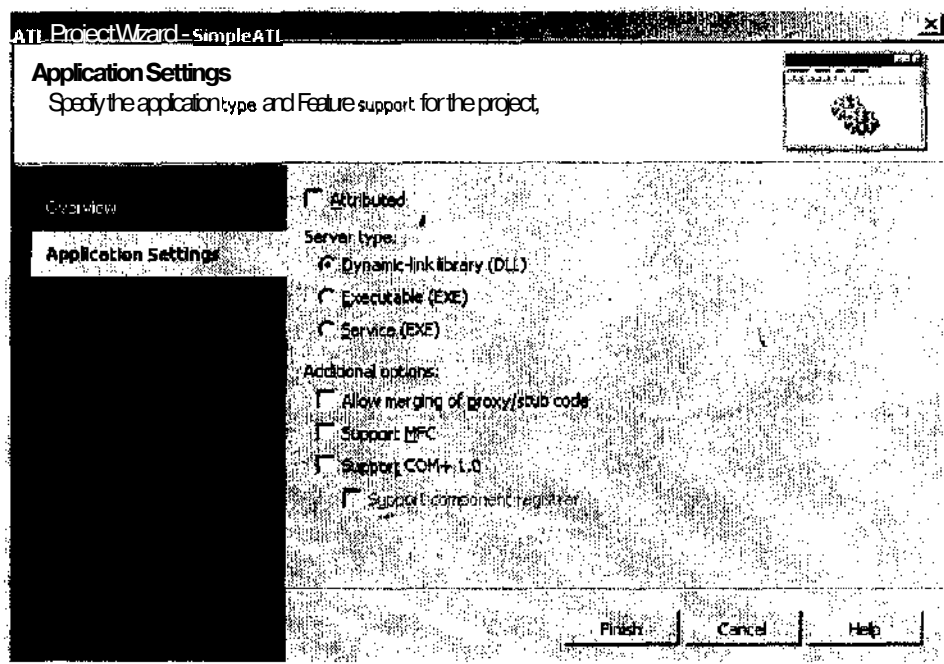


Мы назвали проект *SimpleATL*, а решение - *SimpleATLSolution*. Мы создали отдельный каталог для решения, поэтому можем поместить клиентский проект в тот же самый каталог.

Как и в случае других мастеров, пользовательский интерфейс в диалоговых окнах Visual Studio .NET, применяемых для создания проектов ATL, выглядит совсем по-другому. Окно, изображенное ниже, сообщает, что по умолчанию будет создан внутрипроцессный компонент COM, использующий атрибуты:



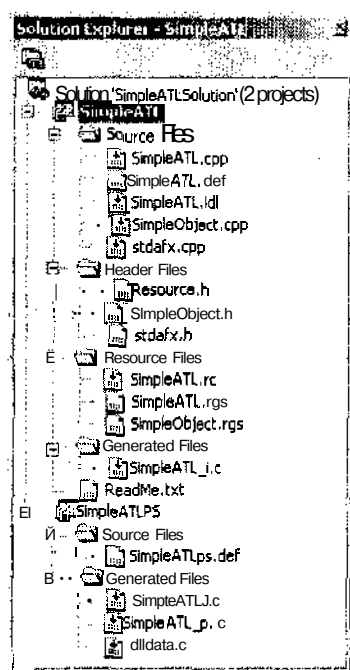
ЕСЛИ создается компонент DLL, включающий атрибуты, нужно просто нажать кнопку Finish. Мы же щелкнем на вкладке Application Settings (Установки приложения) и сбросим флажок Attributed (Использовать атрибуты):



Здесь присутствуют все опции, знакомые пользователям ATL 3.0. Окно позволяет создать DLL, исполняемый файл или службу. При работе с проектом DLL можно выбрать добавление в DLL кода прокси и заглушек, (что облегчает распространение компонента), а также поддержку MFC. Старая опция, позволяющая работать с MTS, заменена эквивалентом, активизирующим поддержку COM+ 1.0. Если мы собираемся работать с COM+, мы можем выбрать поддержку редактора реестра для компонентов (component registrar). Как и в случае ATL 3.0, эти дополнительные опции отсутствуют в проектах исполняемых файлов и служб.

Мы не будем использовать атрибуты в данном проекте, но, вообще говоря, установка флажка в соответствующем поле облегчает разработку компонентов. Однако применение атрибутов не поддерживается при работе с MFC или в случае добавления кода прокси/заглушек. Поэтому, если нужно использовать эти возможности, следует выбрать старый стиль проекта.

Посмотрим теперь, что получится после нажатия кнопки Finish. Проводник решений (Solution Explorer) показывает, что генерируемые файлы выглядят в основном также, как в ATL 3.0:



Visual Studio .NET облегчает работу, отделяя уже существующие файлы от тех, что генерируются при компиляции (файл SimpleATL_i.c в проекте SimpleATL, а также файлы SimpleATL_i.c, SimpleATL_p.c и dlldata.c в проекте прокси/заглушек SimpleATLPS). Однако в целом большая часть файлов очень похожа на код ATL 3.0. Основные различия здесь заключаются в следующем:

D В файл stdafx.h включено несколько новых макросов.

о Файл SimpleATL.cpp, содержащий основной исходный код проекта, использует некоторые новые классы ATL.

Обратимся к основному файлу SimpleATL.cpp. Этот файл содержит код реализации для модуля в целом и вызывается, в частности, при загрузке и выгрузке DLL:

```
#include "stdafx.h"
#include "resource.h"
#include "SimpleATL.h"

class CSimpleATLModule : public CAtlDllModuleT<CSimpleATLModule>
{
public:
    DECLARE_LIBID(LIBID_SimpleATLLib)
    DECLARE_REGISTRY_APPID_RESOURCEID(
        IDR_SIMPLEATL, "{98B48404-4E51-4057-A23E-56E99B58F738}")
};

CSimpleATLModule AtlModule;
// Точка входа для DLL
extern "C" __declspec(dllexport) DllMain(
    HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
```

```

1
// Instance
return _AtlModule.DllMain(dwReason, lpReserved);

// Определяем, может ли OLE выгрузить DLL
STDAPI DllCanUnloadNow(void)
{
    return _AtlModule.DllCanUnloadNow();
}

// Возвращаем фабрику классов для создания объекта запрашиваемого типа
STDAPI DllGetClassObject(REFCLSID rclsid, IID riid, LPVOID ppv)
{
    return _AtlModule.DllGetClassObject(rclsid, riid, ppv);
}

// Используем DllRegisterServer для добавления записей в системный
// реестр
STDAPI DllRegisterServer(void)
{
    // Регистрация объекта, библиотеки типов и всех интерфейсов
    // библиотеки типов
    HRESULT hr = _AtlModule.DllRegisterServer();
    return hr;
}

// Использование DllUnregisterServer для удаления записей из
// системного реестра
STDAPI DllUnregisterServer(void)
{
    HRESULT hr = _AtlModule.DllUnregisterServer();
    return hr;
}

```

Этот код определяет класс `CSimpleATLModule`, который происходит от шаблонного класса `CAtlDllModuleT<>`. При этом используется обычный трюк ATL, когда производный класс передается как параметр шаблона базовому классу. `CAtlDllModuleT<>` — это новый шаблон, заменивший класс `CComModule<>` предыдущих версий ATL. Можно видеть, что знакомые нам по ATL 3.0 макросы `DECLARE_LIBID` и `DECLARE_REGISTRY_APPID_RESOURCEID` для определения GUID включены в класс `CSimpleATLModule`.

Класс `CComModule` реализовывал средства управления набором компонентов, содержащихся в заданном DLL- или EXE-файле. В проектах ATL 3.0 глобальный экземпляр этого класса всегда определялся в основном CPP-файле проекта:

```

// Фрагмент кода из унаследованного проекта ATL 3.0
CComModule _Module;

```

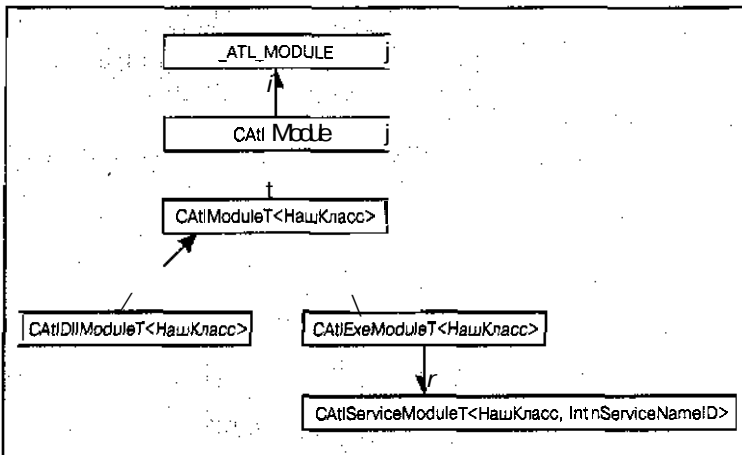
Те же возможности предоставляет теперь глобальный экземпляр нового шаблонного класса:

```

CSimpleATLModule _AtlModule;

```

Класс `CComModule` теперь устарел, и в новой версии ATL он заменен несколькими, более специализированными классами — по одному на каждый тип проекта ATL. Соответствующая этим классам иерархия имеет вид:



Такая замена выполнена с целью структурирования базового кода ATL и лучшей реализации объектно-ориентированного подхода. Мы не будем вдаваться в детали классов этой иерархии, однако из рисунка видно, что три основных шаблона соответствуют трем основным типам COM-компонентов, которые можно создавать. Это **внутрипроцессные**, **внепроцессные** и **сервисные** компоненты. Здесь не показаны остальные аналогичные классы, используемые при создании других типов проектов ATL, таких как приложения Windows. Корневой тип `_ATL_MODULE` представляет собой простую структуру C. Специализированные классы они могут решать некоторые задачи, для выполнения которых ранее мастеру явным образом приходилось добавлять код в проект.

Для иллюстрации преимуществ такого подхода рассмотрим код для точки входа в DLL в файле `SimpleATL.cpp`:

```

// Точка входа для DLL
extern "C" BOOL WINAPI DllMain(
    _ININSTANCE hInstance, _DWORD dwReason, LPVOID lpReserved)
{
    hInstance;
    return _AtlModule.DllMain(dwReason, lpReserved);
}

```

Соответствующий код для проекта ATL 3.0 выглядел бы следующим образом:

```

// Это унаследованный код проекта ATL 3.0!
extern "C" BOOL WINAPI DllMain(
    _ININSTANCE hInstance, _DWORD dwReason, LPVOID /*lpReserved*/)
{
    if(dwReason == DLL_PROCESS_ATTACH)
    {
        _Module.Init(ObjectMap, hInstance, &LIBID_ATLCLASSICDLLLib);
        DisableThreadLibraryCalls(hInstance);
    }
    else if(dwReason == DLL_PROCESS_DETACH)
        _Module.Term();
    return TRUE; // ok
}

```

Преимущества для разработчика в первом случае очевидны: наш проект упростился, поскольку код перенесен из проекта C++ в стандартные файлы ATL.

Теперь обратимся к новому файлу `stdafx.h`. Мы не будем приводить здесь полной распечатки этого файла, поскольку он в основном определяет символы препроцессора, чье назначение достаточно ясно видно из самого кода. Однако данный файл содержит две строки, отражающих новые ключевые особенности ATL. Так, в конце файла имеется строка:

```
using namespace ATL;
```

Определения и классы ATL 7.0 содержатся в пространстве имен ATL. Приведенная строка лишний раз свидетельствует о том, что использование пространств имен для библиотек является хорошим стилем программирования, поскольку снижает риск возникновения конфликтов имен.

Предварительно в файле определен символ препроцессора `-- _ATL_NO_AUTOMATIC_NAMESPACES`:

```
#define _ATL_NO_AUTOMATIC_NAMESPACES
```

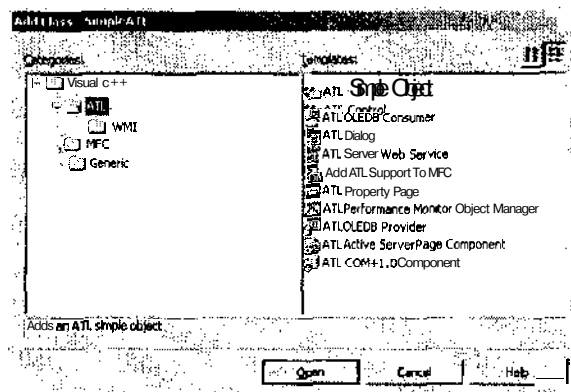
Данный символ определяется наряду с добавлением следующего фрагмента кода в заголовочный файл `atlbase.h`:

```
// Код из файла atlbase.h
#ifndef _ATL_NO_AUTOMATIC_NAMESPACES
using namespace ATL;
#endif // !_ATL_NO_AUTOMATIC_NAMESPACES
```

Все это имеет следующий смысл. По умолчанию пространство имен ATL добавляется в файл `atlbase.h`. Однако наш файл `stdafx.h` определяет макрос, который не позволяет этого сделать. Поэтому мы включаем команду `using namespace` в наш собственный файл, откуда при необходимости ее можно удалить. Определив символ `_ATL_NO_AUTOMATIC_NAMESPACES` в файле `SimpleATL.cpp`, мы сохраняем контроль над использованием пространства имен ATL и можем удалить команду `using namespace`, если возникнут какие-либо конфликты имен. Конечно, это не самый изящный способ, но он работает.

Добавление компонента в проект

Чтобы добавить компонент в проект, щелкнем на этом проекте правой кнопкой мыши в проводнике Solution Explorer и выберем в контекстном меню команду `Add (Добавить)`, а затем `Add Class...` (Добавить класс). В результате откроется диалоговое окно, в котором необходимо указать, какой тип объекта мы хотим добавить. Данное окно имеет новый вид, но большинство опций те же самые, что и в диалоге `Add ATL Object` инструментария Visual Studio 6.0:



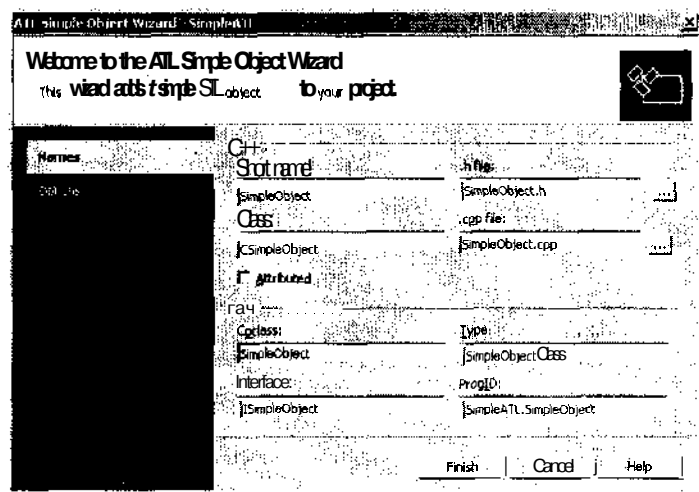
По сравнению с Visual Studio 6.0 мы обнаружим в этом окне следующие дополнения:

- ☐ Windows Management Instrumentation (Инструментальные средства управления средой Windows — WMI). Если требуется написать провайдер событий или провайдер экземпляров WMI, соответствующие встроенные средства позволяют генерировать необходимый для этого базовый код ATL.
- ☐ Менеджер объектов ATL Performance Monitor Object Manager.
- ☐ ATL Server Web Service (Web-служба, использующая ATL Server).

Мы обсуждаем изменения структуры ATL и не будем рассматривать эти типы проектов. Выберем проект Simple Object, чтобы сгенерировать код для нового класса COM.

Поясним для читателей, что WMI представляет собой технологию, позволяющую выполнять опрос и модификацию конфигураций аппаратных средств и локальной машины, используя промышленные стандарты. С помощью WMI можно осуществлять загрузку драйверов либо извлечение компакт-дисков из приводов CD или DVD, не используя код, специфический для данного устройства. В некоторых отношениях WMI можно рассматривать как эквивалент Active Directory, который используется для локальной машины и аппаратных средств, а не для домена Windows.

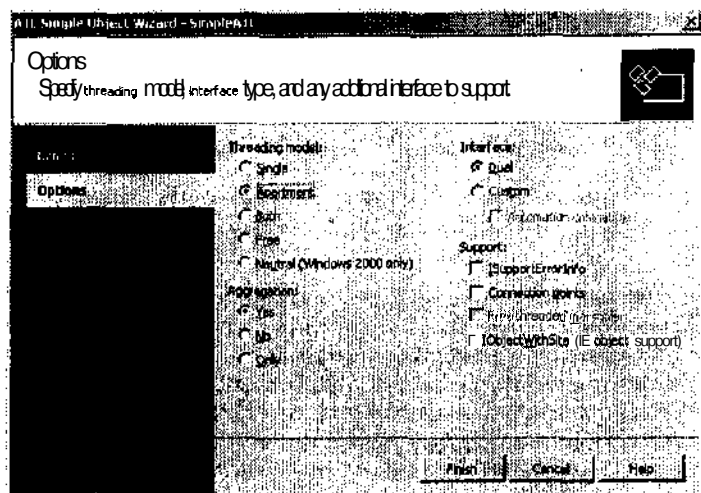
После выбора типа объекта мы увидим на экране диалоговое окно, предлагающее указать дополнительную информацию о данном объекте:



И вновь это окно не сильно отличается от своего предшественника в Visual Studio 6.0. Оно также имеет две вкладки, позволяющие задавать имена и выполнять другие установки. Назовем наш объект SimpleObject. Наиболее важная новая особенность вкладки Names (Имена) — это флажок, который дает возможность присваивать объекту атрибуты. Даже несмотря на то, что мы не используем атрибуты для проекта в целом, все еще остается возможность упростить код для данного конкретного объекта. Однако обратное неверно: если указать, что проект будет иметь атрибуты, то в таком же режиме будут создаваться и все объекты. В настоящий

момент мы хотим лишь проиллюстрировать изменения ATL и поэтому не будем устанавливать флажок.

Вкладка Options также имеет пару дополнений:



Теперь можно выбрать модель нейтрального апартамента при условии, что наш объект будет работать в Windows 2000 или в более поздней операционной системе (такой, как Windows XP). Нейтральный апартамент похож на многопоточный, однако обеспечивает большую гибкость. Если объект существует в нейтральном апартаменте, к нему в любой момент времени может получить доступ только один поток. Однако данный объект не привязан к какому-либо конкретному потоку. Это означает, что по завершении одного вызова объекта тот становится доступным для следующего вызова со стороны другого потока. Объекты, использующие нейтральную модель, не должны быть безопасными для потока в течение вызова метода, но при этом они не зависят от последовательных вызовов, исходящих от одного и того же потока.

Есть также возможность выбрать поддержку интерфейса IE (Опция `IObjWithSite`).

Для нашего объекта выберем апартаментную модель потоков. Хотя при этом генерируется код, похожий на тот, что создавался ATL 3.0, здесь имеется важное отличие: отсутствует карта объекта. Код файла `SimpleObject.h` выглядит следующим образом:

```
#pragma once
#include "resource.h"           // Основные символы

#include "SimpleATL.h"

// Класс CSimpleObject

class ATL_NO_VTABLE CSimpleObject :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CSimpleObject, &CLSID_SimpleObject>,
public IDispatchImpl<ISimpleObject, &IID_ISimpleObject,
    &LIBID_SimpleATLLib, /*wMajor =*/ 1, /*wMinor =*/ 0>
```

```

{
public:
    CSimpleObject()
    {
    }

DECLARE_REGISTRY_RESOURCEID(IDR_SIMPLEOBJECT)

BEGIN_COM_MAP(CSimpleObject)
    COM_INTERFACE_ENTRY(ISimpleObject)
    COM_INTERFACE_ENTRY(IDispatch)
END_COM_MAP()

DECLARE_PROTECT_FINAL_CONSTRUCT()

```

```
HRESULT FinalConstruct()
```

```
{
    return S_OK;
}
```

```
void FinalRelease()
```

```
public:
};
```

```
OBJECT_ENTRY_AUTO(__uuidof(SimpleObject), CSimpleObject)
```

В этом коде выделены операторы, которые существенно отличаются от используемых в ATL 3.0. На самом деле методы `FinalConstruct()` и `FinalRelease()` не являются новыми. Они были определены в ATL 3.0 как рекомендованные конструкции, в которые следовало помещать код инициализации и очистки. Однако в ATL 7.0 по умолчанию мастер добавляет в класс пустые версии этих методов.

Макрос `OBJECT_ENTRY_AUTO` более интересен. В результате его использования наш класс добавляется в карту объекта ATL, которая обеспечивает регистрацию класса и гарантию того, что фабрика классов будет иметь всю необходимую информацию для инициализации объекта. В ATL 3.0 для аналогичных манипуляций использовался макрос `OBJECT_ENTRY`, который помещался в основной .cpp-файл проекта.

```

// Унаследованный код ATL 3.0, генерируемый мастером. Определяет карту
// объекта CSimpleClassic
BEGIN_OBJECT_MAP(ObjectMap)
OBJECT_ENTRY(CLSID_SimpleClassic, CSimpleClassic)
END_OBJECT_MAP()

```

В ATL 7.0 использование `OBJECT_ENTRY` и связанных с ним макросов не рекомендуется. Они будут работать, но лучше не включать их в код. Макрос `OBJECT_ENTRY_AUTO` предоставляет достаточно информации для автоматического конструирования карты объекта. Хотя при этом в программу и не добавляются новые особенности, генерируемый мастером код упрощается. Это оставляет меньше места для ошибок, возникающих при ручном редактировании кода.

Если мы хотим, чтобы объект нельзя было создать с помощью метода `CoCreateInstance()`, необходимо заменить `OBJECT_ENTRY_AUTO` макросом `OBJECT_ENTRY_NON_CREATABLE_EX_AUTO`:

```
OBJECT_ENTRY_NON_CREATABLE_EX_AUTO(__uuidof(SimpleObject), CSimpleObject)
```


Добавление методов

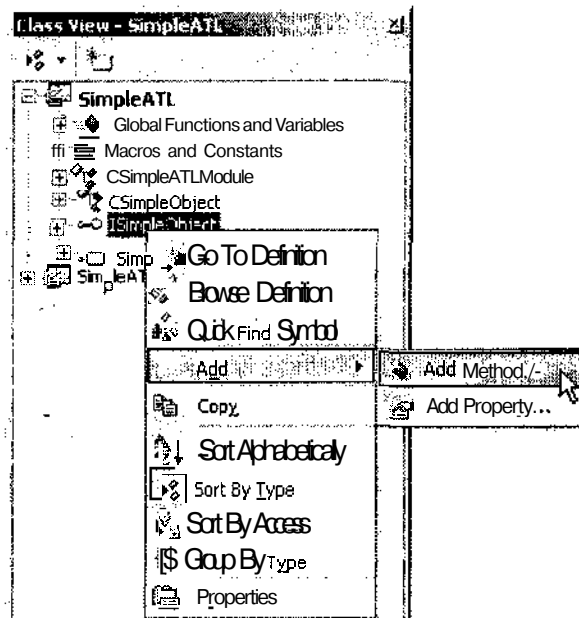
Закончим создание нашего простого объекта и добавим методы в интерфейс `ISimpleObject` класса `SimpleObject`. Объявим один метод и одно нередактируемое свойство:

```
public:
    STDMETHOD(Square)(LONG value, LONG* result);
    STDMETHOD(get_ObjectString)(BSTR* pVal);
```

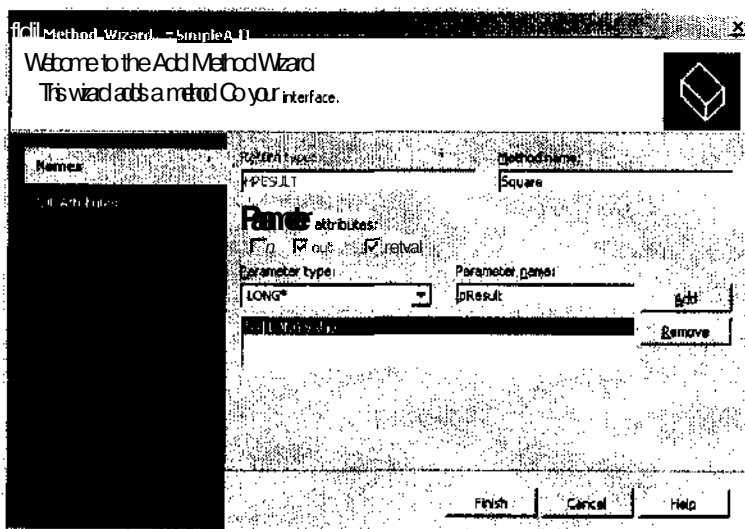
Метод `Square()` будет возводить в квадрат передаваемое ему целое число. Так, получив в качестве параметра 4, метод выдаст значение 16. Свойство `ObjectString()` является нередактируемым и позволит получить строковое значение, относящееся к объекту. В случае простого тестового объекта это будет жестко закодированная строка, содержащая текст "This is a simple object" (Это простой объект).

В данном случае генерируемый код не представляет большого интереса, поскольку он не изменился по сравнению с версией ATL 3.0. Однако диалоговое окно, предназначенное для добавления новых методов и свойств, существенно улучшено. Больше не нужно вручную вводить список параметров метода. Набор флажков, список и текстовые поля позволяют разработчику задавать необходимые параметры, не заботясь о соблюдении всех синтаксических правил.

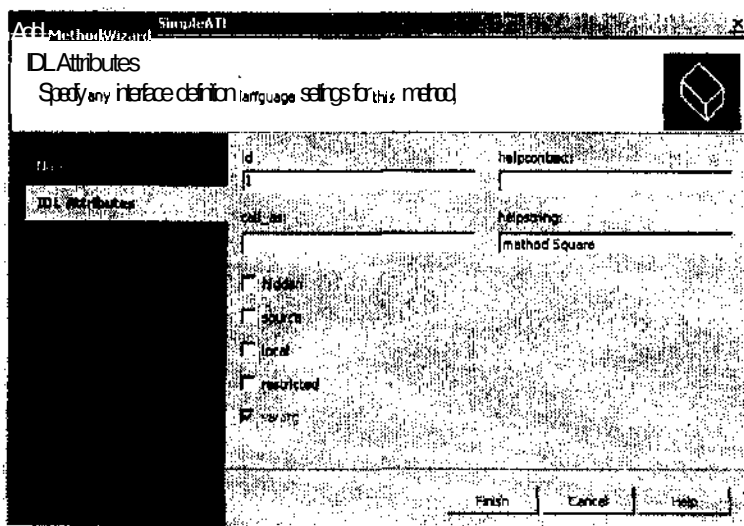
Можно вывести на экран это диалоговое окно, щелкнув мышью на нужном интерфейсе в окне просмотра классов (Class View) и выбрав команду `Add Method` (Добавить метод) или `Add Property` (Добавить свойство). Необходимо щелкать именно на интерфейсе (как показано на рисунке), а не на классе `C++`.



Команда `Add Method` открывает следующий диалог:



Здесь мы добавляем параметр [out, retval] LONG* pResult. Вторая вкладка этого диалога позволяет выбирать различные атрибуты IDL:



Если подобным образом воспользоваться мастером для добавления свойства, мы получим следующие объявления функций:

```
public:
    STDMETHOD(Square)(LONG value, LONG* pResult);
    STDMETHOD(get_ObjectString)(BSTR* pVal);
```

Теперь необходимо реализовать эти функции. Вначале определим переменную в классе CSimpleObject, сохраняющую жестко закодированную строку и обеспечивающую ее корректную инициализацию:

```
private:
    CComBSTR m_bstrObjectString;
public:
    HRESULT FinalConstruct()
    {
        m_bstrObjectString = L"This is a simple ATL object";
        return S_OK;
    }
```

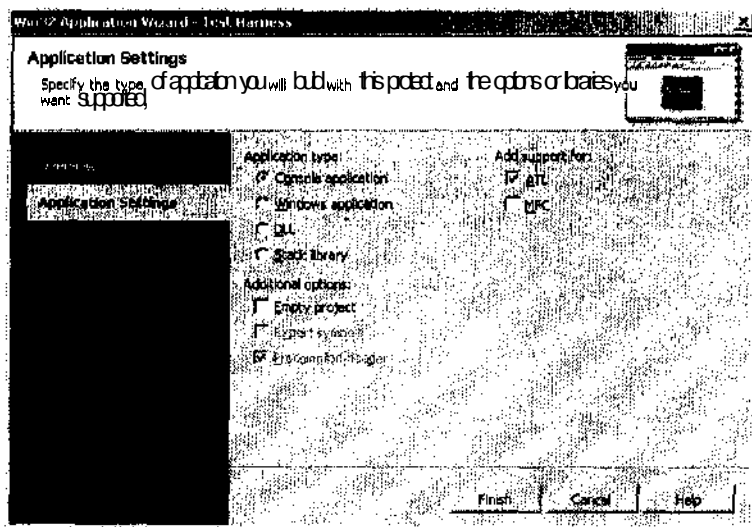
Реализация класса содержится в файле SimpleObject.cpp:

```
STDMETHODIMP CSimpleObject::Square(LONG value, LONG* pResult)
{
    *pResult = value * value;
    return S_OK;
}

STDMETHODIMP CSimpleObject::get_ObjectString(BSTR* pVal)
{
    m_bstrObjectString.CopyTo(pVal);
    return S_OK;
}
```

Этот проект можно компилировать. Однако прежде, чем приступить к его тестированию, необходимо создать клиента COM, который бы выполнял данную функцию. С помощью Visual Studio .NET добавим новый проект в наше решение, выбрав тип проекта C++ Win32 и присвоив ему имя Test Harness.

При этом нужно выбрать установки, создающие консольное приложение с поддержкой ATL:



Затем нужно указать, что это будет первый загружаемый проект в решении. Щелчком правой кнопкой мыши на проекте в Solution Explorer и выберем в открывшемся контекстном меню команду Set As Startup Project (Установить как первый загружаемый проект). После этого добавим выделенный код в новый файл TestHarness.cpp:

```

#include "stdafx.h"
#include <iostream>
#include "SimpleATL\SimpleATL.h"
#include "SimpleATL\SimpleATL.ilc"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
    CoInitialize(NULL);
    CComPtr<ISimpleObject> spObject;
    CoCreateInstance(CLSID_SimpleObject, NULL, CLSCTX_INPROC_SERVER,
        IID_ISimpleObject, reinterpret_cast<void*>(&spObject));

    long value = 5;
    long result = 0;
    spObject->Square(value, &result);
    cout << "value is " << value << " squared is " << result << endl;

    CComBSTR bstr;
    spObject->get_ObjectString(&bstr);
    CW2AEX<64> pszObjectString(bstr);
    cout << "object string is "
        << static_cast<char*>(pszObjectString) << endl;

    spObject = NULL;
    CoUninitialize();
    return 0;
}

```

Первая часть добавляемого кода обеспечивает включение необходимых заголовочных файлов. Мы используем файл `iostream`, чтобы осуществлять чтение/запись на консоль. Кроме того, в программу включаются заголовочные файлы, содержащие GUID и определения интерфейсов для нового компонента `SimpleObject`, содержащегося в проекте `SimpleATL`.

Основная часть кода в функции `_tmain()` — это стандартный клиентский код ATL, инициализирующий компонент и вызывающий его методы. Вначале вызывается метод `Square()`, а затем свойство `ObjectString`, и результаты обеих операций выводятся на экран.

Для простоты мы не проверяем возвращаемые значения `HRESULT`. Т, однако в реальном проекте это необходимо делать.

Читателю может быть незнаком один оператор, выполняющий преобразование строки из формата Unicode `BSTR` в строку ANSI для передачи в оператор вывода `cout`:

```
CW2AEX<64> pszObjectString(bstr);
```

Библиотека ATL 7.0 включает новые средства преобразования строк (которые сделали устаревшими знакомые нам старые макросы `OLE2T`, `T2OLE` и др.). Эти средства представляют собой шаблоны и основываются на новых классах преобразования строк, обеспечивая большую гибкость и определенное повышение производительности. В частности, теперь отсутствует риск переполнения стека, если используются новые классы в циклах, как это было в случае макросов. Шаблоны имеют еще и то преимущество, что они не требуют присутствия в коде макроса `USES_CONVERSION` (также устаревшего). Выбранный нами макрос `CW2AEX<n>` преобразует строки Unicode в ANSI и используется тогда, когда есть уверенность, что число символов в строке не превысит `n`. В противном случае следует выбрать макрос `CW2A` (о других новых средствах преобразования строк см. ниже).

Теперь код можно запускать на исполнение и тестировать. Мы увидим, что методы объекта успешно вызываются.

Создание проекта, использующего атрибуты

В главе 4 мы обсуждали программирование с применением атрибутов, сосредоточившись на новых атрибутах .NET. Синтаксис новых COM-ориентированных атрибутов, по существу, идентичен тому, что используется атрибутами .NET. Однако при этом процессы, происходящие "за сценой", оказываются различными. Некоторые атрибуты .NET включают дополнительную информацию в метаданные создаваемой сборки. Другие же оказывают влияние на сам процесс компиляции. Например, атрибут **StructLayout** изменяет расположение в памяти управляемой структуры. С другой стороны, атрибуты, предназначенные для работы с COM и ATL, заставляют компилятор C++ вызывать **провайдер атрибутов (attribute provider)**, который проверяет атрибут и на основе результатов проверки модифицирует исходный код, прежде чем тот будет передан компилятору.

Сам исходный код проекта при этом остается неизменным. Как и при использовании символов препроцессора, мы, в отличие от компилятора, не видим изменений в своем исходном коде.

Атрибуты являются более мощным средством, чем символы препроцессора C++, с которыми их можно сравнить. Во-первых, они распознаются отладчиком. Во-вторых, символы препроцессора C++ можно заменять лишь самым элементарным образом путем простой подстановки текста в то место файла, куда входят эти символы. С другой стороны, провайдер атрибутов может проверять весь проект и вставлять или модифицировать любой код там, где это необходимо, а также учитывать параметры, передаваемые в атрибуты. Например, при создании класса, который должен быть компонентом COM, а не обычным классом C++, можно просто пометить его атрибутом `coclass`. Провайдер атрибутов обнаружит это и модифицирует код нашего проекта, в частности, создав файл **IDL** (если тот отсутствует) или же изменив содержимое данного файла (если он уже имеется в проекте). Таким образом, при компиляции класс будет наделен соответствующими бинарными характеристиками, необходимыми для COM-компонента.

Преимущество подобного подхода заключается в том, программист имеет более простой код для редактирования, а работать с этим кодом значительно удобнее, поскольку атрибуты отделяют описание выполняемой операции от самого кода. Так, если созданный класс не должен быть `coclass`, то внести в код соответствующие изменения исключительно легко. Достаточно лишь удалить атрибут `coclass` из определения класса. Без использования атрибута пришлось бы вручную удалять весь соответствующий код ATL и редактировать **IDL-файл**. Итак, возможно, самым большим преимуществом атрибутов является то, что они избавляют от необходимости вручную редактировать файл **IDL**. Вместо этого определения `coclass` и интерфейсов содержатся в файле C++.

В настоящем разделе мы обсудим влияние атрибутов на исходный код. Однако этот материал не может служить полным справочным руководством по применению всех атрибутов. Для получения информации, относящейся к новым атрибутам и классам ATL, следует обратиться к документации MSDN. Ниже мы обсудим разработку проекта, содержащего фактически тот же самый COM-класс, который был создан в предыдущем примере, но на этот раз применим атрибуты. Вначале с помощью Visual Studio .NET создадим новый проект C++ ATL и присвоим ему имя `SimpleAttributedATL`. Когда откроется соответствующее окно, сохраним опцию по умолчанию `Attributed` для проекта **DLL**.

Если теперь посмотреть на содержимое файла SimpleAttributedATL.cpp, мы увидим менее знакомый код:

```

#include "stdafx.h"
#include "resource.h"

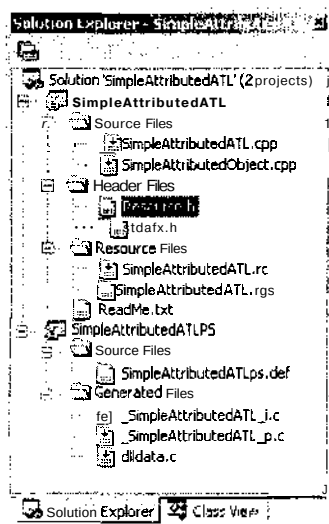
// Атрибут module автоматически реализует точки входа DllMain,
// DllRegisterServer и DllUnregisterServer
[ module(exe, uuid = "{B968C2A2-FC8F-45CF-BA18-8A550C532B4B}",
    name = "SimpleAttributedATL",
    helpstring = "Simple Attributed ATL 1.0 Type Library",
    resource_name = "IDR_SIMPLEATTRIBUTEDATL") ];

```

И это весь файл SimpleAttributedATL.cpp! Как частично отмечено в комментарии кода, атрибут module будет развернут для создания базового блока library в файле IDL и всего кода реализации DLL, необходимого для размещения в нем СОМ-компонента. Иными словами, мы получаем определение класса на основе шаблона CAtlModuleT<>, определение глобального экземпляра этого класса, а также реализации DllMain(), DllCanUnloadNow(), DllGetClassObject(), DllRegisterServer() и DllUnregisterServer(), т.е. все то, что мы получили в первом созданном нами проекте.

Атрибут module принимает несколько параметров. Первый из них указывает тип проекта и может принимать значения dll, exe или service. Следующий параметр представляет собой GUID (параметр uuid), который станет идентификатором GUID соответствующей библиотеки типов. Если опустить этот параметр, идентификатор GUID будет сгенерирован автоматически. Обычно нужно присвоить имя библиотеке типов, а код, генерируемый мастером, создаст по умолчанию строку helpstring и идентификатор для ресурсов проекта. Существует большое количество других параметров, которые можно ввести вручную, в частности lcid, helpcontext и version. Большинство этих атрибутов прямо соответствуют атрибутам IDL, имеющим те же самые имена и назначения.

Отметим одно важное обстоятельство. Если просмотреть наш проект с помощью Solution Explorer, то среди созданных файлов мы не найдем файла IDL. Он будет сгенерирован позже с помощью атрибутов. Он даже не включается в список генерируемых файлов, поскольку его создает не Visual Studio .NET, а провайдер атрибутов:



Продолжим разработку проекта и добавим в него компонент COM. Откроем нужный мастер и выберем простой объект с атрибутами, присвоив ему имя SimpleAttributedObject(). В результате будут сгенерированы пустой файл исходного кода и заголовочный файл с определением нового класса CSimpleAttributedObject. В заголовочный файл помещается следующий код:

```
#pragma once

#include "resource.h" // основные символы
// Интерфейс ISimpleAttributedObject

object
{
    uuid("7E5C9DF6-39D7-46FD-8BD5-0C68881B4A5B"),
    dual, helpstring("ISimpleAttributedObject Interface"),
    pointer_default(unique)
}
interface ISimpleAttributedObject : IDispatch
{
};

// Класс CSimpleAttributedObject

[
    coclass,
    threading("apartment"),
    vi_progid("SimpleAttributedATL.SimpleAttributedObj"),
    progid("SimpleAttributedATL.SimpleAttributedObj.1"),
    version(1.0),
    uuid("5FFA75B6-EC61-4340-A06D-1918A56EE3C9"),
    helpstring("SimpleAttributedObject Class")
]

class ATL_NO_VTABLE CSimpleAttributedObject :
public CComObject<ATL::CComObjectBase>
{
public:
    CSimpleAttributedObject()
    {
    }

    DECLARE_PROTECT_FINAL_CONSTRUCT()

    HRESULT FinalConstruct()
    {
        return S_OK;
    }

public:
    void FinalRelease()
    {
    }
};
```

Этот код существенно отличается от рассмотренного нами ранее аналогичного кода для объекта, не использующего атрибуты. Здесь четко видно разделение определений класса и интерфейса. Интерфейс ISimpleAttributedObject нашего класса определяется с помощью нового ключевого слова „interface“:

```
[
    object,
    uuid("7E5CFDF6-39D7-46FD-8BD5-0C6B881B4A5B"),
    dual, helpstring("ISimpleAttributedObject Interface"),
    pointer_default(unique)
]
__interface ISimpleAttributedObject : IDispatch
{
};
```

Ключевое слово `__interface` имеет тот же эффект, что и `struct`, однако в первом случае компилятор использует семантику интерфейсов. Например, все методы интерфейса будут интерпретироваться как чисто виртуальные независимо от любого применяемого синтаксиса типа `"=0"`, и компилятор выдаст ошибку, если будет написана любая реализация.

Определение интерфейса сопровождается несколькими атрибутами, имеющими то же самое назначение, что и их эквиваленты в IDL. Так, `uuid` задает идентификатор GUID интерфейса, а `dual` указывает, что интерфейс должен происходить от `IDispatch`. Атрибут `object` здесь несколько сбивает с толку, поскольку так же, как в IDL, он указывает на объявление интерфейса COM, а не старого интерфейса DCE. Наличие данного атрибута заставляет провайдер атрибутов поместить определение интерфейса в файл IDL (предварительно создав такой файл, если он отсутствует).

Определение класса, представляющего компонент, теперь значительно упростилось. Функциональность класса COM в ATL достигается за счет того, что класс происходит от нескольких шаблонных базовых классов ATL, включая `CComObjectRootEx<>` и `CComCoClass<>`. Однако в нашем коде класс явно наследует только от интерфейса `ISimpleAttributedObject`:

```
[
    coclass,
    threading("apartment"),
    vi_progid("SimpleAttributedATL.SimpleAttributedObj"),
    progid("SimpleAttributedATL.SimpleAttributed0.1"),
    version(1.0),
    uuid("5FFA75B6-EC61-4340-A06D-1918A56EE3C9"),
    helpstring("SimpleAttributedObject Class")
]
class ATL_NO_VTABLE CSimpleAttributedObject :
    public ISimpleAttributedObject
{
};
```

Атрибут `coclass`, включенный в определение класса, заставляет провайдер атрибутов модифицировать код данного класса, чтобы тот представлял COM-компонент с параметрами, указанными в остальных атрибутах. Провайдер атрибутов автоматически сгенерирует код, гарантирующий, что наш класс будет происходить от нужных классов ATL, а также добавит карту COM для интерфейсов, реализуемых классом. Именно поэтому определение класса явно не содержит карты COM в отличие от предыдущего проекта, где не применялись атрибуты. Оставшиеся атрибуты в основном похожи на своих собратьев, используемых в файле IDL. Может оказаться неясным лишь назначение атрибута `vi_progid`, который определяет идентификатор `ProgID`, независимый от версии.

Наконец, воспользуемся мастерами для добавления в класс тех же методов, что были реализованы в предыдущем примере. В данном случае новый код, добавленный к определению класса, будет идентичен коду предыдущего проекта:


```
public:
    STDMETHOD(Square)(LONG value, LONG* pResult);
    STDMETHOD(get_ObjectString)(BSTR* pVal);
```

Добавим вручную такие же реализации этих методов. Единственное отличие нового кода от предыдущего состоит в том, что теперь в объекте используется слегка модифицированная строка:

```
HRESULT FinalConstruct()
{
    m_bstrObjectString = L"This is a simple Attributed ATL object";
    return S_OK;
}

private;
    CComBSTR m_bstrObjectString;
```

Когда в класс добавляются эти методы, мастер автоматически вносит соответствующие изменения в определение интерфейса:

```
[
    object,
    uuid("7E5CFDF6-39D7-46FD-8BD5-0C6B881B4A5B"),
    dual, helpstring("ISimpleAttributedObject Interface"),
    pointer_default(unique)
]
interface ISimpleAttributedObject : IDispatch
{
    [id(1), helpstring("takes the square of a number")]
    HRESULT Square([in] LONG value, [out, retval] LONG* pResult);
    [propget, id(2), helpstring("property ObjectString")]
    HRESULT ObjectString([out, retval] BSTR* pVal);
};
```

Это гарантирует, что корректное определение интерфейса будет передано через файл IDL в библиотеку типов и в код прокси/заглушек.

Теперь проект можно компилировать. Как и ранее, добавим в решение отдельный проект, содержащий код тестирования. Не считая другого имени интерфейса и новых идентификаторов GUID, данный код в основном идентичен коду предыдущего примера. Единственное отличие нового варианта кода от старого заключается в способе преобразования строк. Поскольку теперь длина строки в объекте превосходит 64 байта (когда она представлена как строка Unicode), мы не можем больше использовать макрос `CW2AEX<64>` для преобразования ее в строку ANSI.

Конечно, можно задать в макросе `CW2AEX<>` любую необходимую длину строки (например, записать `CW2AEX<100>`). Однако для демонстрации дополнительных возможностей ATL воспользуемся атрибутом `CW2A`, который может работать со строками произвольного размера за счет определенных издержек производительности:

```
CComBSTR bstr;
spObject->get_ObjectString(&bstr);
CW2A pszObjectString(bstr);
cout << "object string is: " « static_cast<char*>(pszObjectString)
      « endl;
```

Как обычно, полный код этого примера вместе с другим исходным кодом для данной главы можно найти в загружаемом архиве на Web-сайте издательства Wrox.

Преобразование DLL-файла в исполняемый файл

В данном разделе мы продемонстрируем применение атрибутов для преобразования с EXE-сервер класса SimpleAttributedObject, созданного в нашем последнем примере для выполнения функций DLL-сервера. Без использования атрибутов это потребовало бы существенной модификации проекта. Однако благодаря атрибутам единственное изменение, которое нужно внести в код C++ для создания исполняемого COM-сервера, заключается в следующем:

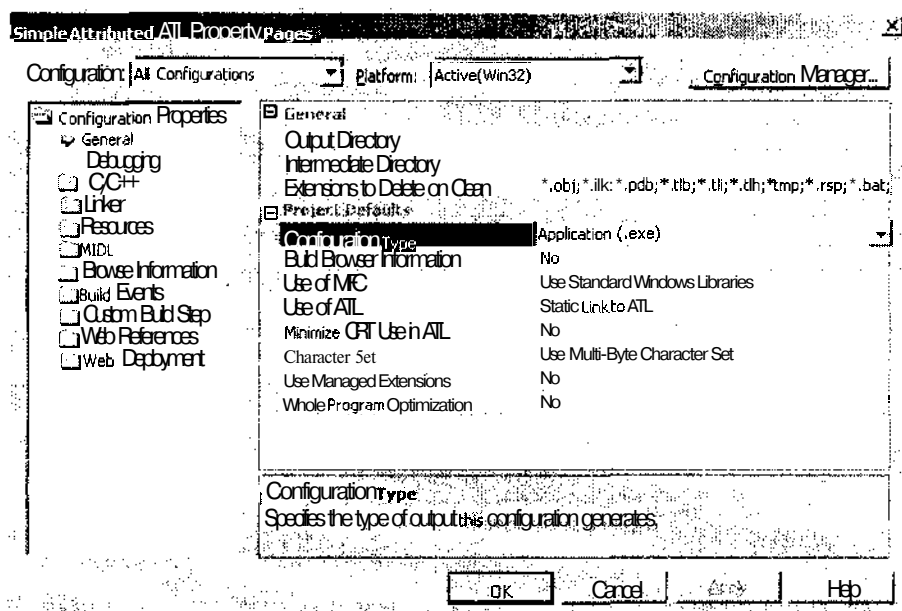
//Файл SimpleAttributedATL.cpp: Реализация экспорта кода в DLL

```
((include "stdafx.h"
#include "resource.h"
```

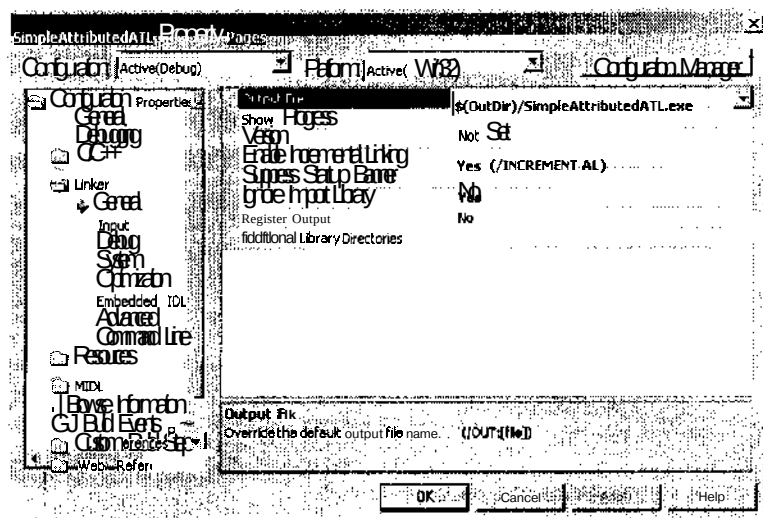
```
// Атрибут module автоматически реализует точки входа DllMain,
// DllRegisterServer и DllUnregisterServer
```

```
[ module(exe, uuid = "{9DF208A2-F3BE-4F38-9C5C-54BBF9568234}",
    name = "SimpleAttributedATL",
    helpstring = "SimpleAttributedATL 1.0 Type Library",
    resource_name = "IDR_SIMPLEATTRIBUTEDATL") ];
```

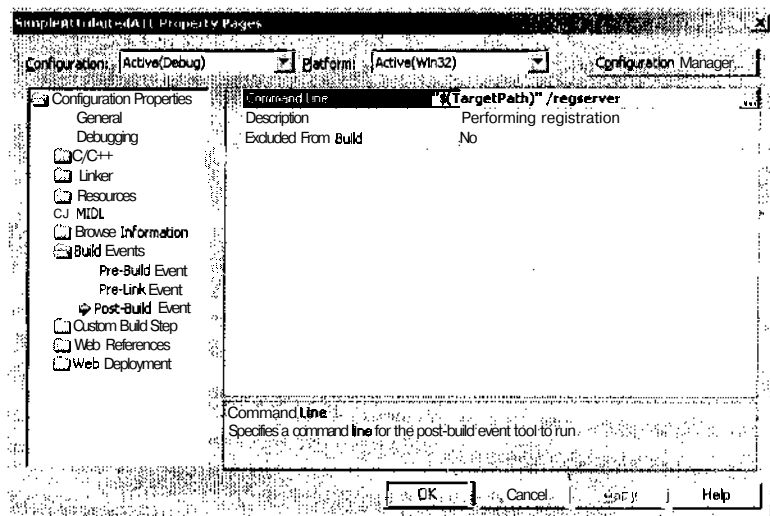
Кроме того, необходимо изменить опции компилятора. Откроем окно Properties для проекта SimpleAttributedATL, выберем в списке Configuration (Конфигурация) элемент All Configurations (Все конфигурации), а также поменяем тип проекта Dynamic Library (Динамическая библиотека) на Application (Приложение):



Затем нужно настроить установки компоновщика так, чтобы выходной файл имел корректное имя SimpleAttributedATL.exe вместо SimpleAttributedATL.dll:



Следует также изменить этап, выполняемый после компоновки, на котором регистрируется COM-компонент. В случае DLL на этом этапе запускается утилита `regsvr32 /s /c "S(TargetPath)"`. Однако исполняемый файл саморегистрируется, получая ключ `regserver`. Поэтому в окне свойств необходимо указать строку: `"S(TargetPath)" /regserver`.



После этих изменений можно скомпилировать и зарегистрировать компонент как исполняемый сервер. Остается лишь изменить код проекта, предназначенного для тестирования, чтобы он вызывал нужный тип компонента:

```
int _tmain(int argc, _TCHAR* argv[])
{
    CoInitialize(NULL);
    CComPtr<ISimpleAttributedObject> spObject;
```

```
CoCreateInstance(CLSID_CSimpleAttributedObject, NULL, CLSCTX_LOCAL_SERVER,
IID_ISimpleAttributedObject, reinterpret_cast<void*>(&spObject));
```

Теперь все решение готово к компиляции и запуску. Но, поскольку мы не изменили идентификаторы GUID при создании нового решения, нельзя одновременно зарегистрировать компоненты DLL и EXE. Регистрация одного из них автоматически приведет к отмене регистрации другого. Если мы запустим один пример, а потом захотим вернуться обратно, то необходимо заново выполнить полную компоновку данного компонента, включая повторную регистрацию. В противном случае возникает риск получения некорректных записей в реестре для компонента.

Новые классы ATL

В данном разделе дается обзор некоторых новых классов, доступных для программистов ATL. Эти классы происходят из двух источников:

- ❑ Классы, которые были частью MFC, но теперь реализованы заново в облегченной форме. Они больше не зависят от файлов DLL, входящих в MFC, и поэтому могут быть использованы в проектах MFC или ATL. Microsoft рассматривает их как классы, совместно используемые библиотеками MFC и ATL.
- ❑ Классы, добавленные в ATL. Они включают классы динамических массивов, оболочки вокруг определенных объектов Windows (таких как файлы, дескрипторы и ключи реестра), а также классы преобразования строк.

В обоих случаях список классов оказывается весьма обширным, поэтому мы ограничим обсуждение некоторыми наиболее важными дополнениями. Обращаясь к первому пункту перечисления, выделим важнейшие классы, совместно используемые ATL и MFC:

- ❑ Простые классы (CPoint, CSize, CRect и др.), которые представляют собой оболочки для соответствующих структур Windows C,
- ❑ Классы, предназначенные для работы с датой и временем, такие как CTime и CTimeSpan.
- D Строковый шаблон CStringTo и соответствующие определения, введенные с помощью спецификатора typedef. Эти классы обеспечивают расширенные возможности манипулирования строками.

Ниже в этой главе мы проиллюстрируем классы CTime и CTimeSpan при разработке компонента, демонстрирующего использование классов-утилит. Такие классы, как CPoint, являются очень простыми и не будут обсуждаться. Мы рассмотрим более подробно новые классы, предназначенные для работы со строками.

Новые строковые классы

В течение длительного времени библиотека MFC включает класс CString, который (в зависимости от опций компилятора) представляет строки Unicode или ANSI. Этот класс имеет огромное количество функций-членов для работы со строками, а также перегруженные версии операторов, делающих синтаксис слияния строк очень простым. С другой стороны, ATL содержит класс CComBSTR, который разработан специально для того, чтобы заключать в оболочку тип BSTR, используемый инфраструктурой COM для представления строк. Хотя класс CComBSTR значительно упрощает такие операции, как создание и уничтожение строк BSTR, он

не содержит многих общих методов, используемых для манипулирования строками. Другим претендентом на первенство в этом соревновании является строковый класс STL, однако с точки зрения ATL он имеет тот недостаток, что требует связывания с CRT. Этот класс также использует своеобразный синтаксис, который далеко не все разработчики считают удобным.

Для того чтобы включить в новую версию ATL расширенные средства обработки строк без связывания с большой библиотекой, Microsoft полностью переписала реализацию класса CString. Вместо того чтобы размещаться в DLL-файлах библиотеки MFC, класс CString теперь базируется на шаблоне CStringTO (этот шаблон определяется в заголовочном файле `atlstr.h` и реализует все функции класса CString без использования багажа MFC). Шаблон CStringTO принимает два параметра. Первый из них представляет собой базовый символьный тип (`char` для ANSI, `wchar_t` для Unicode или `TCHAR`, чтобы реализовать зависимость типа от опций компилятора). Вторым параметром является класс, указывающий, где расположены ресурсы и будет ли объект CStringTO использовать библиотеку CRT:

```
template<typename BaseType, class StringTraits>
class CStringT : public CStringT<BaseType>
```

Хотя шаблон CStringTO предоставляет все необходимое, вряд ли кому понравится объявлять шаблон всякий раз, когда нужно использовать строку. Понимая это, Microsoft с помощью спецификатора `typedef` ввела определения двух других классов, основанных на CStringTO. В проектах, использующих MFC, как и прежде, доступен класс CString. Пользователи ATL могут работать с классом CAtlString.

Класс CAtlString задает параметры шаблона CStringTO, которые обеспечивают поддержку символов `TCHAR` и исключают зависимость от CRT, как это и должно быть в проекте ATL. Хотя это выглядит несколько неожиданным, в нашем коде можно просто объявить переменную типа CAtlString, что даст нам строковый объект с широким набором методов, позволяющих манипулировать строками. В проекте ATL можно работать с любым из двух классов: CAtlString или CString. При этом в нашем распоряжении будут одни и те же методы, однако на основе документации можно сделать вывод, что класс CString лучше использовать в проектах MFC. В настоящей главе рассматривается класс CAtlString.

Для использования класса CAtlString (или CString) в проекте ATL необходимо с помощью директивы `#include` включить в проект файл `atlstr.h`. Соответствующий класс включает методы, позволяющие выполнять слияние, форматирование в стиле `printf` и преобразование строк, а также сравнение и поиск подстрок. Мы продемонстрируем некоторые из этих операций в заключительном примере настоящей главы.

Классы преобразования строк

Библиотека ATL содержит шесть новых шаблонных классов, предназначенных для преобразования строк в различные форматы: `CA2AEX<>`, `CA2WAEX<>`, `CA2WEX<>`, `CA2WEX<>` и `CA2WEX<>`. Все эти шаблоны принимают один параметр, который указывает количество байт, используемых для хранения преобразуемой строки. Однако чаще применяются переопределения с помощью ключевого слова `typedef`, основанные на данных классах. Рассмотрим пример. Предположим, что имеются следующие переменные:

```
char* pszA = "Hello";
TCHAR* pszT = _T("Hello");
wchar_t* pszW = L"Hello";
```

С учетом этого объявим новые переменные и выполним преобразование типов и процессе создания переменных:

```
CA2A convertStrA(pszW); // Преобразование типа wchar_t в char
```

```
... CA2CT convertStrT(pszA); // Преобразование типа char в константу TCHAR
... CT2W convertStrW(pszT); // Преобразование типа TCHAR в wchar_t
```

Здесь мы инициализируем переменные типов, представленных классами преобразования. В первой строке кода объявляется переменная типа `CW2A`, и в результате создается объект, содержащий буфер со строкой **"Hello"**, которая представляет собой набор элементов `char` (символов ANSI). Объект инициализируется путем передачи ему строки формата `wchar_t`. Если же нужна строка ANSI, можно получить ее из буфера, используя такой метод:

```
... char* pszRes = convertStrA;
```

Данный код создает указатель `pszRes` на буфер внутри `convertStrA`, однако копирования буфера при этом не происходит. Такое же поведение имеет место и в случае других типов преобразования.

Приведенный код демонстрирует лишь пример возможных преобразований типов. Однако здесь можно проиллюстрировать способ построения имен классов преобразования, определенных с помощью спецификатора `typedef`. В начале имени стоит буква `C`. Затем следует буква, указывающая преобразуемый тип (`A`, `T` или `W`). Далее идет символ `'2'`, после которого помещается буква, обозначающая результирующий тип (вновь `A`, `T` или `W`). Если целевая переменная является константой, перед последним символом ставится буква `C`. Таким образом, класс `CW2CT` будет преобразовывать строку `wchar_t` в строковую константу `TCHAR`.

Изящество рассматриваемого способа преобразования состоит в том, что благодаря использованию экземпляров классов память, в которой хранится буфер, будет автоматически освобождаться после выхода экземпляра из области видимости:

```
... CA2CT convertStrT(pszA); // Преобразование типа char в тип TCHAR
... const TCHAR* pszResT = convertStrT;

// Переменная convertStrT находится в области видимости,
// поэтому можно использовать указатель pszResT.
}
// Переменная convertStrT вышла из области видимости. Память будет
// освобождена, больше нельзя использовать указатель pszResT, так как он
// может указывать на мусор.
```

Это означает, что можно выполнять преобразования внутри цикла, не рискуя переполнить стек. Классы используют стек для строк, не превышающих определенную длину (по умолчанию 128 байт, что достаточно для большинства случаев). Однако, если строка оказывается слишком длинной, классы резервируют пространство в куче с помощью метода `malloc()`.

Если точно не известно, какова будет длина строки, следует применять простые определения, использующие `typedef`, как в приведенном выше коде. Однако, когда есть уверенность, что строка поместится в буфер определенного размера, можно использовать другие определения `typedef` (или непосредственно сам класс), чтобы уменьшить размер буфера. Для этого нужно добавить символы `EX` в соответствующее имя, определенное с помощью `typedef`, а также задать параметр шаблона, указывающий число байт памяти, используемое в буфере. Следующий код выполняет то же самое преобразование, что и в предыдущем примере, однако размер буфера в каждом случае теперь устанавливается равным 64.

```
... CW2AEX<64> convertStrA(pszW); // Преобразование типа wchar_t в char
... CA2CTEX<64> convertStrT(pszA); // Преобразование типа char в TCHAR
... CT2WEX<64> convertStrW(pszT); // Преобразование типа TCHAR в wchar_t
```

Тем, кто привык работать со старыми макросами ATL 3.0, следует обратить внимание на одно обстоятельство. Приведенный ниже код является синтаксически

правильным и будет откомпилирован. Он в достаточной степени похож на то, что мы делали в ATL 3.0. Однако мы настоятельно рекомендуем не употреблять подобные конструкции с новыми определениями, использующими `typedef`:

```
char* pszResult = CW2A(L"Hello");
```

Данный код вводит в заблуждение. Он выглядит так, словно выполняется прямое преобразование и результат передается в переменную `pszResult`. Но фактически результат помещается во временный объект `CW2A`, который выйдет из области видимости при переходе к следующей строке кода. Очень скоро переменная `pszResult` будет указывать на мусор. Приведенная строка кода полностью эквивалентна следующему фрагменту, делающему проблему области видимости более понятной:

```
char* pszResult;
{
    CW2A temp(L"Hello");
    pszResult = temp;
}
```

При работе с новыми определениями классов, использующими спецификатор `typedef`, лучше применять конструкции, продемонстрированные нами вначале. При таком методе более очевидно, что объявляется временный объект, выполняющий преобразование и сохраняющий преобразованную строку.

Пример проекта, использующего классы-утилиты

Теперь мы рассмотрим создание небольшого проекта, иллюстрирующего использование атрибутов и некоторых новых классов-утилит ATL. Разработаем исполняемый COM-компонент `UtilitiesComponent`, предназначенный для решения некоторых вспомогательных задач, в частности:

- ❑ Вычисление даты следующего (за сегодняшним) буднего дня и вывод результата в виде строки. Если метод будет вызван в воскресенье, понедельник, вторник, среду или четверг, он выведет завтрашнюю дату. Вызвав метод в пятницу или в субботу, мы получим дату следующего понедельника.
- ❑ Удаление из строки всех лишних пробелов. Это означает, что будут удалены все пробелы в начале или в конце строки и группы последовательных пробелов внутри строки заменены одним пробелом. В нашем случае под пробелами имеются в виду символы пробела, табулятора и возврата каретки.

Комбинация этих двух методов в одном компоненте может показаться несколько странной, однако она очень хорошо иллюстрирует применение новых классов: `CAtlString` и `CTime`.

Создадим наш новый проект `ATLUtilities`, используя шаблон `ATL Project` в `Visual Studio .NET` и выбрав в качестве типа проекта `DLL-сервер`, использующий атрибуты. Затем воспользуемся обычными мастерами для создания компонента `UtilitiesComponent`, после чего добавим в интерфейс компонента `IUtilitiesComponent` методы со следующими сигнатурами:

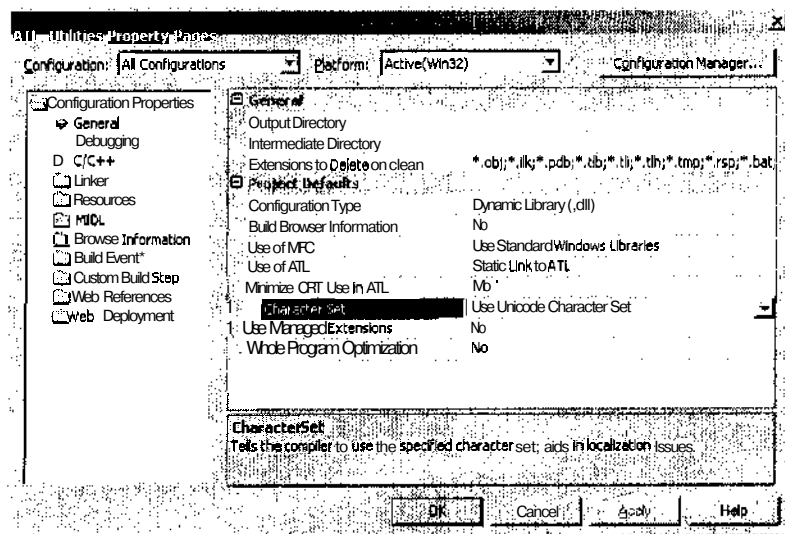
```
STDMETHOD(TrueTrim)(BSTR bstrInput, BSTR* pbstrResult);
STDMETHOD(get_NextWeekday)(BSTR* pVal);
```

Прежде чем приступить к модификации кода, созданного мастером, внесем в проект одно изменение. При работе с ATL 3.0 среда `Visual Studio 6.0` давала возможность выбора конфигурации для проекта ATL, включая опции `Debug` (Отладка), `Release` (Распространение), `Unicode Debug` (Отладка, формат `Unicode`) и

Unicode Release (Распространение, формат Unicode). Если нужно было использовать в строках формат Unicode, а не набор символов MBCS, мы соответственно могли выбрать подходящий режим компоновки.

Visual Studio .NET не предоставляет такого выбора. В данном случае имеются лишь две конфигурации: Debug и Release, которые по умолчанию используют строки MBCS. Если нужен формат Unicode, для этого необходимо явным образом изменить свойства проекта. Поскольку мы будем работать со строками, а тип `BSTR` в COM всегда соответствует формату Unicode, наша задача существенно упростится, если внести в проект такое изменение. Класс `CAtlString` может работать со строками Unicode или MBCS, в зависимости от установок проекта. Однако, если сохранить режим MBCS, это приведет к потере производительности из-за необходимости выполнять преобразование форматов между `CAtlString` и `BSTR`.

Для того чтобы компонент всегда использовал строки Unicode, откроем диалоговое окно свойств проекта. Найдем папку General Configuration Properties (Общие установки конфигурации), выберем в списке Configuration элемент AN Configurations (Все конфигурации) и изменим установку Character Set (Набор символов), указав формат Unicode:



Теперь можно приступить к написанию кода. После того, как был создан компонент и с помощью мастера в него добавлены методы, напомним следующий дополнительный код реализации метода `TrueTrim()`:

```
STDMETHODIMP UtilitiesComponent::TrueTrim(BSTR bstrInput, BSTR* pbstrResult)
{
    const CAtlString csInput = bstrInput;
    CAtlString csResult;
    CAtlString csTemp;
    int pos = 0;
    while(!csTemp = csInput.Tokenize(_T(" \n\t"), pos)).IsEmpty())
    {
        if(!csResult.IsEmpty())
            csResult += _T(" ");
        csResult += csTemp;
    }
}
```



```

    *pbstrResult = csResult.AllocSysString();
    return S_OK;
}

```

Этот метод показывает, насколько просто манипулировать строками при использовании класса `CAtlString`. Основное удобство в данном случае состоит в использовании метода `CAtlString::Tokenize()`, разбивающего строку на части в местах вхождения пробелов, а также перегруженной версии оператора суммирования-присваивания `CAtlString::operator+=()` для слияния строк.

Для понимания того, что происходит в цикле `while`, необходимо разобраться, как работает метод `Tokenize()`. Этот метод принимает два параметра:

- Строку, содержащую набор символов-меток, в местах расположения которых будет разбиваться исходная строка
- Позицию начала поиска в строке `CAtlString`

Когда вызывается метод `Tokenize()`, он ищет часть строки, ограниченную одним или несколькими символами (метками), и отделяет эту часть. Метод возвращает полученную подстроку, а второй параметр обновляется и указывает на новую позицию, начиная с которой будет выполняться следующая операция `Tokenize()`. Данный процесс проще всего проиллюстрировать на примере.

Если в качестве меток используются символы пробела, табуляции или возврата каретки (как в нашем коде) и вызывается метод `Tokenize()` для строки `" \t First Second\nThird"`, то возвращаемая величина будет равна "First", а значение начальной позиции увеличится до 13, указывая первую букву слова "Second". При втором вызове метода `Tokenize()` будет выдано значение "Second", после чего начало поиска переместится в 20-ю позицию. В результате третьего вызова возвращаемая подстрока окажется равной "Third", а начальная позиция будет установлена в конце строки. Наконец, вызвав метод в четвертый раз, мы получим пустую строку. Поскольку именно это возвращаемое значение используется нами в качестве условия, проверяемого в цикле `while`, код выйдет из данного цикла.

В теле цикла `while` создается строка, не содержащая лишних пробелов, путем слияния подстрок, извлеченных из исходной строки. После этого метод копирует результат в передаваемую строку `BSTR` и возвращает управление.

Теперь обратимся ко второму методу, реализующему `нередактируемое` свойство `NextWeekday`:

```

STDMETHODIMP CUtilitiesComponent::get_NextWeekday(BSTR* pVal)
{
    CTime currentTime = CTime::GetCurrentTime();
    int iDay = currentTime.GetDayOfWeek();
    int nDaysTillNextWeekday = 1;
    if(iDay == 6)
        nDaysTillNextWeekday = 2;
    else if(iDay == 5)
        nDaysTillNextWeekday = 3;
    CTimeSpan daysTillNextWeekday(nDaysTillNextWeekday, 0, 0, 0);
    currentTime += daysTillNextWeekday;
    CAtlString result = currentTime.Format(T("%A, %d %B %Y"));
    (static_cast<CComBSTR>(result)).CopyTo(pVal);
    return S_OK;
}

```

Данное свойство иллюстрирует классы `CTime` и `CTimeSpan`, используемые как в ATL, так и в MFC. Вначале мы получаем текущее время, применяя статический метод `CTime::GetCurrentTime()`, а затем определяем сегодняшний день недели с помощью метода `CTime::GetDayOfWeek()`. Последний возвращает целое число, причем 1 обозначает воскресенье, 2 — понедельник и т.д.

Далее полученное целое число используется для создания экземпляра *класса* `CTimeSpan`, представляющего количество дней, отделяющих сегодняшний день от следующего буднего дня (`weekday`). Для дней недели с воскресенья по четверг это число равно 1, в случае пятницы оно составляет 3, а для воскресенья равно 2. Затем мы прибавляем величину `CTimeSpan` к текущей дате и форматируем результирующий объект `CTime` как строку, применив метод `CTime::Format C`. Этот метод использует спецификаторы формата, позволяющие точно указать, как должна быть отформатирована строка. Мы применяем спецификаторы `%A` для дня недели, `%d` для дня месяца, `%B` для названия месяца и `%Y` для четырехзначного представления года (полный список спецификаторов можно найти в документации MSDN). Получив эту строку, мы преобразуем и копируем ее в строку `BSR`, которую можно возвратить клиенту.

Необходимо также добавить в код компонента еще две директивы `#include` для заголовочных файлов ATL, содержащих определения используемых нами классов:

```
ttinclude<atlstr.h>
#include<atltime.h>
```

Теперь компонент можно компилировать, однако нужно еще создать клиента для его тестирования. Добавим новый проект в текущее решение и создадим консольное приложение `Win32` с именем `UtilitiesTestHarness`, не забывая подключить поддержку ATL. Затем добавим следующий код в файл `UtilitiesTestHarness.cpp`:

```
#include "stdafx.h"

<include ".\ATLUtilities\ATLUtilities.h"
#include ".\ATLUtilities\ATLUtilities_1.c"

int _tmain(int argc, _TCHAR* argv())
{
    CoInitialize(NULL);
    CComPtr<IUtilitiesComponent> spObject;
    CoCreateInstance(CLSID_UtilitiesComponent, NULL, CLSCTX_INPROC_SERVER,
        IID IUtilitiesComponent, reinterpret_cast<void*>(&spObject));
    CComBSTR bstrNextWeekday;
    spObject->get_NextWeekday(&bstrNextWeekday);
    CW2A pszNextWeekday(bstrNextWeekday);
    cout << "Next weekday is " << static_cast<char*>(pszNextWeekday) << endl;
    CComBSTR bstrTestString = L"hello \n there ";
    CComBSTR bstrResult;
    spObject->TrueTrim(bstrTestString, &bstrResult);
    CW2A pszTestString(bstrTestString);
    cout << "Test string is >>" << static_cast<char*>(pszTestString);
    cout << " " << endl;
    CW2A pszResult(bstrResult);
    cout << "Result of TrueTrim is >" << static_cast<char*>(pszResult);
    cout << " " << endl;

    spObject = 0;
    CoUninitialize();
    return 0;
}
```

Запуск этого кода приводит к следующему (успешному) результату:

```
Next weekday is Monday, 14 January 2002
(Следующий будний день - понедельник, 14 января 2002 г.)
Test String is >> hello
there <<
Result of TrueTrim is >hello there<<
```

ГЛАВА 10



Введение в ATL Server

ATL Server — это набор обычных классов C++, предоставляемых Microsoft для того, чтобы облегчить создание высокопроизводительных Web-приложений и Web-служб. Программисты, которым приходилось разрабатывать Web-приложения путем создания библиотек DLL с использованием расширения ISAPI, согласятся, что эта технология обладает большими преимуществами. Она основана на ATL и потому значительно упрощает процесс разработки, ATL Server поддерживает такие возможности и средства, как работа с очередями сообщений, криптография, SMTP Mail, регулярные выражения, работа с пулами объектов, кэширование и мониторинг производительности. Более того, ATL Server включает также средства, облегчающие отладку наиболее сложных Web-приложений.

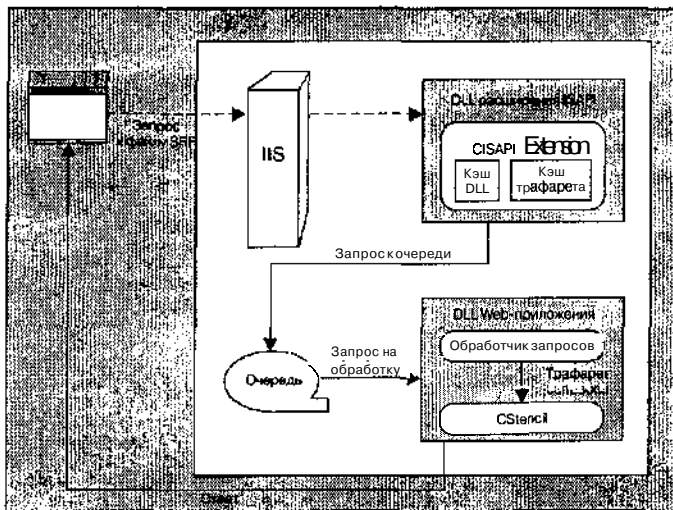
В данной главе мы покажем, что представляет собой ATL Server, рассмотрим его работу, обсудим использование данной технологии для построения Web-приложений и приведем примеры, иллюстрирующие ее возможности. В следующей главе мы сосредоточимся на применении ATL Server для создания Web-служб.

Архитектура

Модель программирования ATL Server основывается на создаваемых пользователем маркерах, которые во время выполнения заменяются вызовами методов. Маркеры помещаются в **файл отклика сервера (server response file - SRF)**, который также называется **трафаретом (stencil)**. Сами методы пишутся на C++ и реализуются в **DLL Web-приложения (Web DLL)**. Включая в себя смесь статических данных и динамических маркеров, SRF-файл создает тем самым очень простой механизм разработки шаблонов и генерации Web-контента в период выполнения.

По существу, ATL Server имеет сходство с такими технологиями, как ASP: запрос пользователя на получение Web-страницы служит непосредственной причиной выполнения некоторых операций на стороне сервера, в результате чего генерируются нужные данные в формате HTML. Однако ATL Server использует код C++, который компилируется на стадии разработки, а затем размещается на сервере, что обеспечивает очень высокую производительность.

ATL Server размещается поверх Internet Information Server (IIS). На следующем рисунке изображены элементы архитектуры ATL Server и в упрощенном виде представлена схема обработки запросов:



Архитектура ATL Server включает в себя библиотеку **DLL расширения ISAPI** (одну на виртуальный каталог IIS), одну или несколько **DLL Web-приложения**, а также один или более **SRF-файлов**. Гибкость такой схемы заключается в том, что в одном приложении можно создать несколько библиотек Web DLL, каждая из которых реализует определенный логический элемент функциональности. Аналогичным образом архитектура ATL Server позволяет создавать множественные файлы **SRF** для удовлетворения требований пользовательских интерфейсов приложения.

Процедура, проиллюстрированная на рисунке, запускается, когда клиент посылает запрос HTTP к размещенному на IIS-сервере файлу SRF (иногда запрос адресуется библиотеке DLL Web-приложения). Сервер передает запрос библиотеке DLL расширения ISAPI. Данная библиотека либо считывает SRF-файл и направляет соответствующий запрос обработчику в нужную DLL Web-приложения, либо проверяет строку запроса, чтобы передать параметр обработчику непосредственно в DLL. В обоих случаях запросы к Web DLL немедленно помещаются в очередь, освобождая потоки IIS для получения других клиентских запросов. Запросы, находящиеся в очереди, предаются соответствующим обработчикам, когда те становятся доступными.

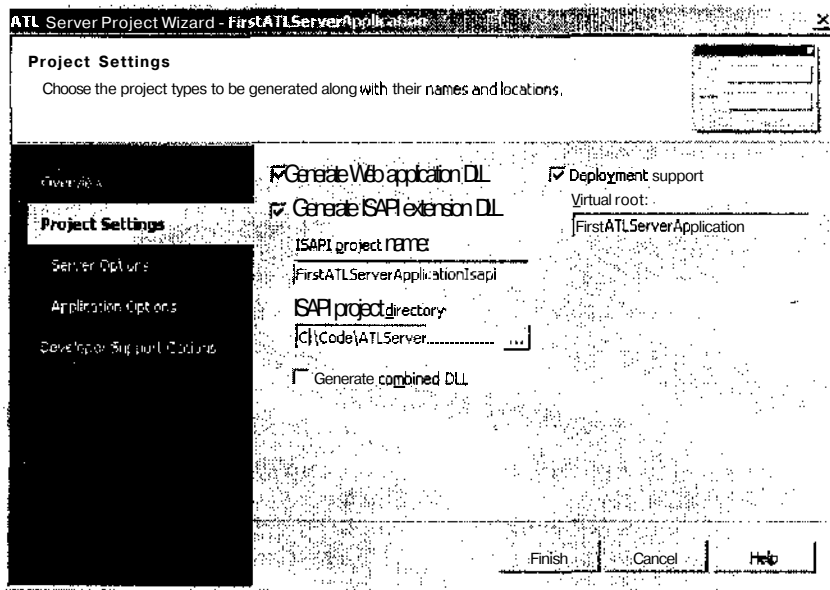
Читатели, знакомые с написанием расширений ISAPI на Visual C++, возможно, ожидали, что вся функциональность приложения будет определена в ISAPI DLL. Однако в технологии ATL Server библиотека ISAPI DLL используется для получения запросов от клиента и для передачи ему ответов. ISAPI DLL передает сами запросы обработчику, реализованному в Web DDL, и фактически играет роль коммуникационного канала между IIS и библиотекой DDL Web-приложения.

Разработка простого приложения ATL Server

Вначале познакомимся с мастером приложений ATL Server (ATL Server Wizard), который мы используем для создания простого Web-приложения. Позже мы добавим в приложение наш собственный код, однако прежде обсудим код, создаваемый по умолчанию, и посмотрим, какую информацию можно из него извлечь.

Создание проекта

Откроем Visual Studio .NET и создадим новый проект Visual C++ с помощью мастера ATL Server Project Wizard (в загружаемом коде этот проект называется `FirstATLServerApplication`). В диалоговом окне мастера раздел **Project Settings** (Установки проекта) имеет следующий вид:



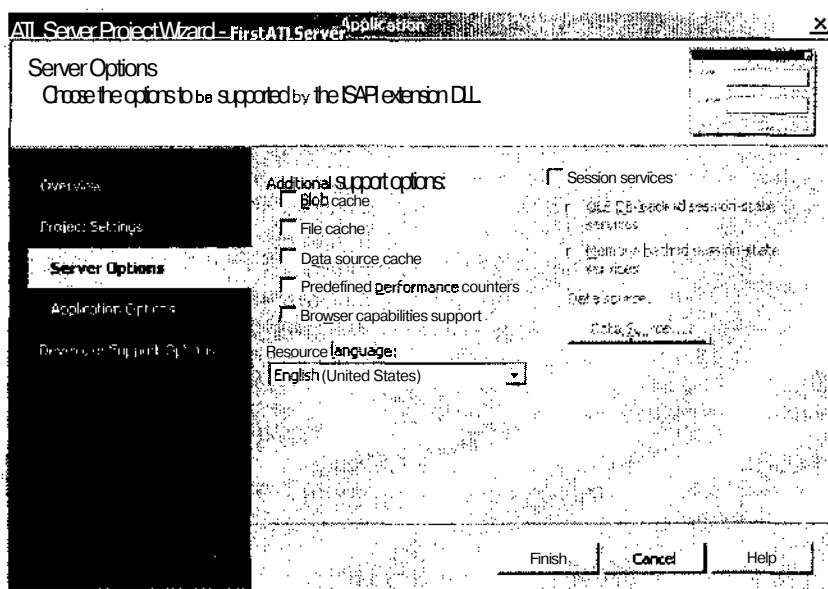
Представленные здесь установки, задаваемые по умолчанию, указывают, что расширение ISAPI и DLL Web-приложения будут создаваться отдельно. Можно отключить ту или другую опцию при желании создать множественные Web DLL в рамках одного и того же приложения ATL Server.

Хотя, как правило, библиотеки Web DLL и ISAPI DLL разрабатываются отдельно, можно создать единую DLL, включающую функциональность обеих библиотек. Это делается для того, чтобы уменьшить время загрузки и снизить требования к памяти либо просто для упрощения процесса развертывания. Однако следует иметь в виду, что в результате может пострадать эффективность приложения. Как бы то ни было, для реализации подобной возможности следует установить флажок **Generate Combined DLL** (Создать объединенную DLL).

Поддержка развертывания (Deployment Support) по умолчанию разрешена, хотя ее также можно отключить и активизировать позже с помощью страницы свойств Web Deployment (Развертывание в Web). Установка данной опции означает, что создаваемые DLL будут автоматически регистрироваться на машине в процессе компоновки проекта. Окно редактирования Virtual Root (Виртуальный каталог) позволяет задать виртуальный каталог IIS, в который будут установлены файлы проекта (конечно, если на машине установлен сервер IIS). Если виртуальный каталог отсутствует, средство развертывания автоматически добавит его в общую папку IIS.

Кроме базовых функций, определяемых установками проекта, ISAPI DLL поддерживает многие дополнительные возможности, конфигурируемые в разделе **Server Options** (Опции сервера). В обсуждаемых ниже приложениях мы затронем

только установки, выполняемые мастером по умолчанию. Тем не менее имеет смысл кратко познакомиться с дополнительными опциями.



Ниже перечислены возможности, которые могут быть добавлены к функциональности DLL расширения ISAPI:

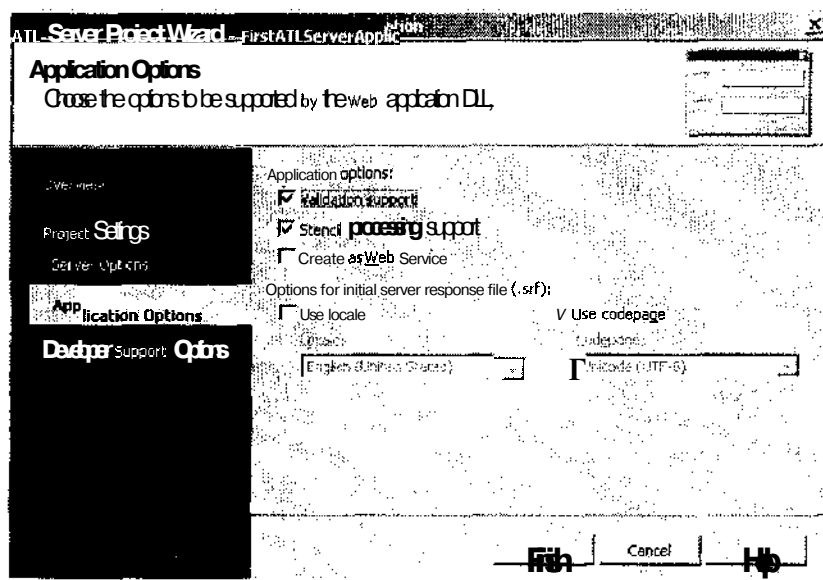
- ☐ Кэш больших двоичных объектов (**Blob cache**) — обеспечивает эффективное хранение и извлечение произвольных двоичных данных. Если запрос к Web DLL или ее ответ требуют работы с большими объемами двоичных данных, установка описываемой опции позволяет улучшить производительность приложения, поскольку система помещает такие данные в кэш.
- ☐ Файловый кэш (**File cache**) — служит для эффективного хранения и получения файлов и данных из файлов. Если в процессе общения с пользователями приложение работает с файловыми объектами, данная опция за счет кэширования соответствующих операций помогает улучшить производительность приложения.
- ☐ Кэш источников данных (**Data source cache**) — позволяет кэшировать соединения с источниками данных OLE DB. Когда приложение использует такой источник данных, выбор этой опции также повышает производительность приложения.
- ☐ Стандартные счетчики производительности (**Predefined performance counters**) — используются в ISAPI DLL для мониторинга производительности.
- ☐ Определение возможностей браузера (**Browser capabilities support**) — реализует в ISAPI DLL службу определения возможностей браузера. С помощью этой службы Web-сервер, в частности, может установить, поддерживает ли браузер Java, работу с фрагментами данных cookies, языки сценариев и т.д.

Выбор опции **Session Services** (Сеансовые службы) облегчает управление состоянием сеансов в случае множественных запросов, поскольку специфичные для

клиентов данные сохраняются в переменных сеанса. Можно использовать базу данных для сохранения информации о сеансе. Когда флажок установлен, предоставляется возможность задать две опции:

- Сохранение состояния сеанса в OLE DB (OLE DB-backed session-state). - При этом информация о сеансе запоминается источником данных OLE DB. Естественно, выбирая эту опцию, необходимо задать источник данных. Запись информации о сеансе в базу данных защитит эту информацию в случае выхода сервера из строя и позволит сохранять ее в течение длительного времени, возможно, до следующего сеанса с клиентом (вспомним о покупках через Amazon.com, где наша карточка сохраняется до следующего подключения к сайту). Однако хранение информации о сеансе в базе данных может ухудшить время отклика по сравнению со случаем выбора второй опции.
- а Сохранение состояния сеанса в памяти (Memory-backed session-state). — В этом случае информация о сеансе хранится в памяти. Это может ускорить извлечение данных о сеансе, однако не позволяет защитить их в случае выхода сервера из строя.

Как нетрудно догадаться, выбор раздела Application Options (Опции приложения) открывает список возможностей, которые могут быть реализованы в библиотеке DLL Web-приложения:



Ниже представлены опции раздела Application Options:

- Опция Validation Support (Проверка корректности данных) — добавляет в DLL код проверки данных, содержащихся в параметрах запроса и в переменных формы. При выборе этой опции мастер включает в код приведенное ниже определение. Метод ValidateAndExchange() вызывается первым в любой коммуникационной последовательности между ISAPI DLL и Web DLL:

```

HTTP_CODE ValidateAndExchange()
{
    // TODO: Здесь следует поместить любой код инициализации
    // и проверки данных

    // Установка типа информации (content-type)
    ra_HttpResponse.SetContentType("text/html");

    return HTTP_SUCCESS;
}

```

Из этой реализации метода мы получаем доступ к объектам запроса (request object) и ответа (response object), которые служат для коммуникации с пользователями приложения. Здесь можно выполнить любую проверку данных и инициализацию, какую мы сочтем необходимой.

- Опция Stencil Processing Support (Поддержка обработки трафарета). — В результате ее установки в исходный код включается метод, позволяющий продемонстрировать использование атрибута tag_name. Если выбрать данную опцию, в приложение добавится следующий блок кода:

```

protected:
    // Пример использования маркера замены с процессором трафарета
    [tag_name(name = "Hello")]
    HTTP_CODE OnHello(void)
    {
        m_HttpResponse << "Hello World!";
        return HTTP_SUCCESS;
    }

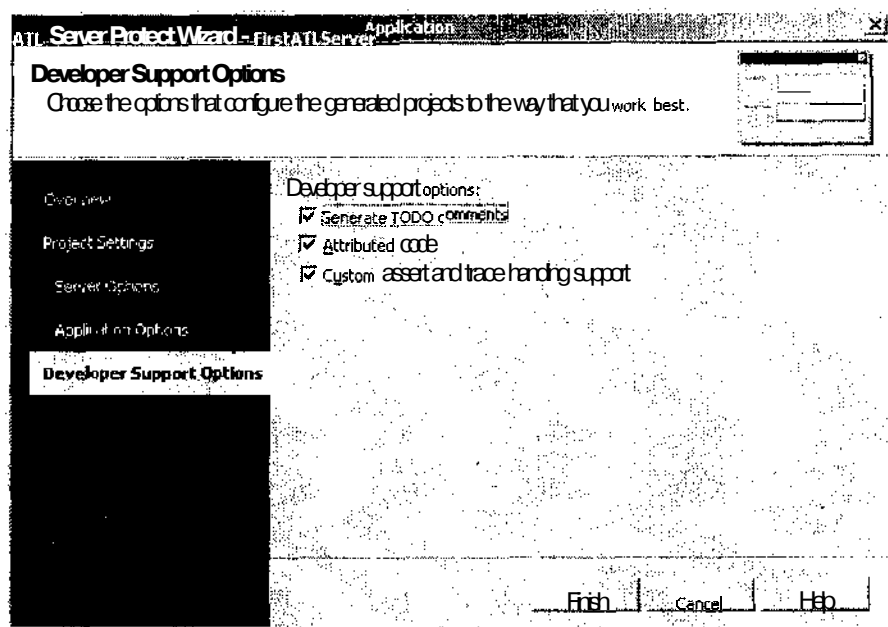
```

Данный метод представляет собой прототип для методов замены, используемых процессором трафарета (stencil processor). Например, если включить в SRF-файл маркер Hello, то во время выполнения он будет заменен вызовом метода OnHello().

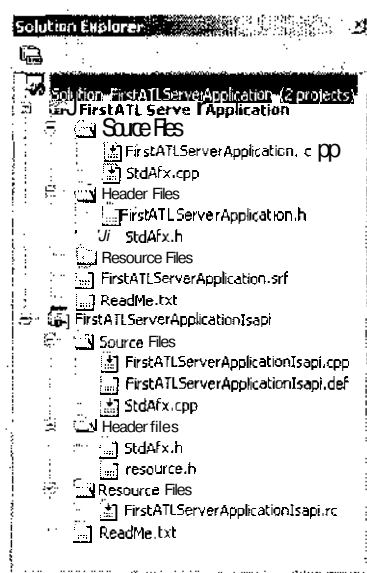
- Опция Create as Web Service (Создать как Web-службу). — Если установлен этот флажок, мастер генерирует "скелетный" код для Web-службы (о Web-службах подробнее см. в следующей главе). В противном случае создается код для стандартного обработчика запросов.

Заключительные установки на экране позволяют задавать опции для первоначально создаваемого трафарета. Если установить флажок Use locale (Использовать местную специфику), в генерируемый SRF-файл будет добавлен выбранный маркер, характеризующий местные особенности. При установленном флажке Use codpage (Использовать кодовую страницу) в создаваемый SRF-файл добавится выбранный маркер кодовой страницы. Кодовая страница определяет рабочий набор символов. Если файл SRF содержит символы, не относящиеся к набору ANSI, необходимо указать нужную кодовую страницу.

Наконец, раздел Developer Support Options (Опции поддержки разработчика) позволяет добавлять (или, скорее, удалять, поскольку все эти опции задаются по умолчанию) комментарии "to do", использование атрибутов в коде и поддержку средств трассировки:



СМЫСЛ ЭТИХ ОПЦИЙ понятен без лишних объяснений. Например, последняя опция позволяет добавлять в проект поддержку трассировки и операторов ASSERT, которые дают возможность выполнять отладку с помощью средства WebDbg, помещенного в папку Tools инсталляции Visual C++. В нашем проекте мы сохраним опции, установленные по умолчанию, поэтому для создания файлов проекта щелкнем на кнопке Finish. Откроем проводник Solution Explorer. Он будет заполнен списком генерируемых файлов:



Кроме обычных файлов Visual C++, здесь находятся `FirstATLServerApplication.cpp` и одноименный заголовочный файл с расширением `.h`, которые определяют Dll Web-приложения. Далее, `FirstATLServerApplicationIsapi.cpp` и соответствующий ему `.h`-файл определяют ISAPI DLL. Наконец, `FirstATLServerApplication.srf` представляет собой файл отклика сервера.

Генерируемый код

Теперь кратко обсудим код, созданный мастером, а также выполняемые этим кодом операции и их смысл. В контексте нашего обсуждения наиболее важными являются два файла: `FirstATLServerApplication.h` и `FirstATLServerApplication.srf`. Вначале рассмотрим первый из них.

Файл *FirstATLServerApplication.h*

Представленный ниже заголовочный файл генерируется мастером. Этот файл содержит определение класса `CFirstATLServerApplicationHandler`, который будет использоваться для обработки траффера. Методы этого класса могут обеспечивать проверку корректности вводимых параметров и создавать выходной код HTML, который заменяет маркеры в SRF-файле:

```
#pragma once

[request_handler("Default")]
class CFirstATLServerApplicationHandler
{
private:
    // Здесь размещаются закрытые члены
protected:
    // Сюда следует поместить защищенные члены
public:
    // Здесь размещаются открытые члены
    HTTP_CODE ValidateAndExchange()
    {
        // TODO: Здесь следует поместить любой код инициализации
        // и проверки данных
        // Установка типа информации (content-type)
        m_HttpResponse.SetContentType("text/html");
        return HTTP_SUCCESS;
    }
protected:
    // Пример использования маркера замены с процессором траффера
    [tagname(name="Hello")]
    HTTP_CODE OnHello(void)
    {
        m_HttpResponse << "Hello World!";
        return HTTP_SUCCESS;
    }
};
```

Присоединение атрибута `request_handler` к классу `CFirstATLServerApplicationHandler` делает последний классом обработчика запросов ATL Server, способным обрабатывать запросы HTTP. Благодаря применению атрибута указанный класс происходит от класса ATL `CRequestHandlerT<>`, определение которого имеет вид:

```
template <class THandler,
class ThreadModel = CComSingleThreadModel,
class TagReplacerType = CHtmlTagReplacer<THandler>>
class CRequestHandlerT : public TagReplacerType,
public CComObjectRootEx<ThreadModel>,
public IRequestHandlerImpl<THandler>
```

Это базовый класс для всех обработчиков запросов, и можно подменять его методы для получения дополнительной функциональности, реализующей инициализацию. Данный класс обеспечивает доступ к контексту сервера и ссылается на объекты запроса и ответа, используя два важных члена:

```
CHttpRequest m_HttpRequest;
CHttpResponse m_HttpResponse;
```

Класс CRequestHandlerTo определяет, в частности, метод ValidateAndExchange(), который подменяется по умолчанию. С помощью данного метода можно инициализировать члены класса, присваивая им величины, взятые из объекта m_HttpRequest. Можно также выполнить проверку корректности данных на стороне сервера. Другой обычно подменяемый метод, CCheckValidRequest(), имеет следующую сигнатуру:

```
HTTP_CODE CCheckValidRequest (O_
```

Этот метод может быть реализован для базовой проверки корректности запроса и будет вызван перед сравнительно дорогостоящей инициализацией объекта запроса. Отсюда следует, что нельзя работать с переменной m_HttpRequest в коде метода CCheckValidRequest(), хотя можно использовать объект m_HttpResponse для отправки ответа клиенту.

На основе параметра атрибута request_handler (в нашем случае равного "Default") файл трафарета определяет класс обработчика, методы которого используются для обработки маркеров, содержащихся в этом файле. Поскольку мы выбрали опцию поддержки трафарета, мастер включил в код описанный нами выше метод OnHello() с атрибутом tag_name. Обычно методы, определяемые подобным образом, имеют доступ к объектам m_HttpRequest и m_HttpResponse, и в любом реальном приложении сюда могут быть добавлены некоторые дополнительные операции. Эти методы позволяют извлекать данные из объекта запроса, выполнять с ними определенные вычисления, форматировать результат и затем возвращать его клиенту.

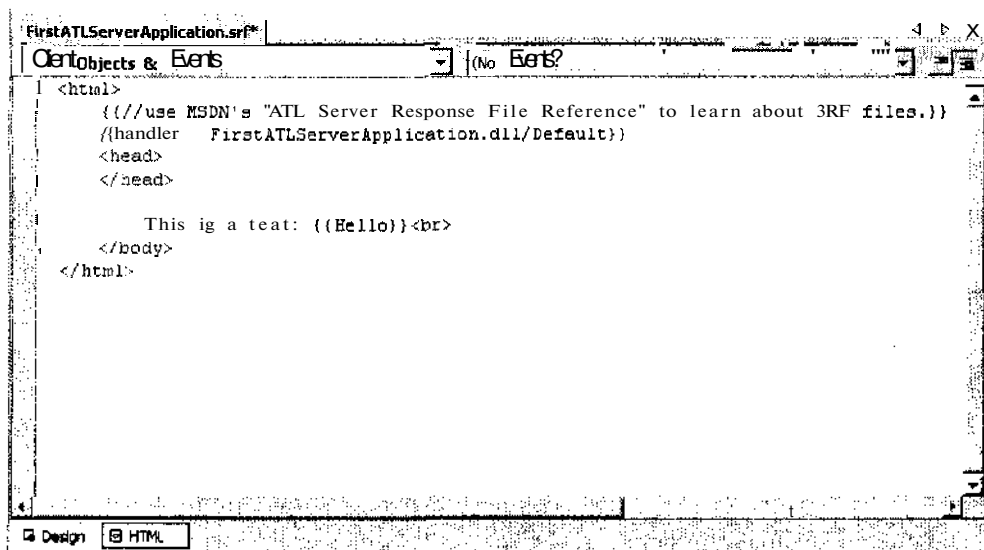
Клиенту можно вернуть код состояния, показывающий, успешно ли выполнены операции в указанных методах. Например, значение HTTP_SUCCESS соответствует успешно завершенной операции, а код HTTP_FAIL показывает, что операция не выполнена. Существует также несколько промежуточных состояний, о которых можно прочитать в документации.

Файл *First.ATLServerApplication.srf*

Другой создаваемый мастером файл, с которым нам следует познакомиться, — это файл трафарета. По умолчанию в него включается весьма краткая информация:

```
{// Используйте раздел MSDN "ATL Server Response File Reference"}
{// для знакомства с файлами SRF.}
{{handler FirstATLServerApplication.dll/Default}} This is a test: {{Hello}}
```

Это не единственный способ просмотра файла. В левом нижнем углу окна, отображающего представленный текст, находятся две кнопки: **Design** и **HTML**. Если щелкнуть кнопку **HTML**, мы увидим более знакомую картину:



В этом коде двойные фигурные скобки '{' и '}' обозначают маркеры, которые будут обрабатываться библиотекой Web DLL. В приведенном фрагменте кода содержатся три маркера, однако первый из них представляет собой простой комментарий, как показывает двойная косая черта в стиле C++. Второй маркер более интересен, поскольку он определяет класс обработчика, связанного с файлом трафарета, а также атрибуты, присоединенные к данному классу. При этом указывается название библиотеки, содержащей класс. В нашем случае задается класс `CFirstATLServerApplicationHandler` в библиотеке `FirstATLServerApplication.dll`.

Если клиент пошлет запрос к файлу трафарета, то в клиентском браузере будет показан статический текст "This is a test: ". Далее отобразится содержимое, представленное маркером `Hello`, т.е. результат вызова метода `OnHello()`. Маркер вызывает метод обработчика и выполняет код C++, который динамически загружает в объект ответа соответствующий код HTML.

В более сложных файлах SRF может быть несколько маркеров:

```
{{handler aaa.dll/Default}} This is an example: {({TagA})} {({TagB})}
```

При обработке такого файла маркеры `TagA` и `TagB` будут заменены HTML-кодом, который генерируется методами, связанными с маркерами при помощи атрибута `tag_name`. Если автор Web-страницы задает ошибочное имя маркера, это имя в неизменном виде будет напечатано в браузере, поскольку для данного маркера не существует метода замещения.

Файлы трафарета позволяют использовать условные выражения и программные циклы. Например, следующий код демонстрирует применение условного оператора `if`. Если метод, связанный с маркером `Available`, возвращает код успешного выполнения операции (`HTTP_SUCCESS`), то статический текст, определенный в блоке `if`, будет передан клиенту. Можно также писать вложенные условные операторы с помощью конструкций `else-elseif`, используя похожий синтаксис:

```
{{if Available}}
The resource is available
{{endif}}
```

Следующий сегмент кода иллюстрирует применение блока `while`. Обработчик выполняет цикл до тех пор, пока метод, связанный с маркером `GetNextItem`, не возвратит код невыполненной операции (`HTTP_S_FALSE`). При каждой прогонке цикла оценивается маркер `DisplayItem` и генерируемое им содержимое отображается в клиентском браузере:

```
{{while GetNextItem И  
  {{ DisplayItem }}  
{{endwhile}}
```

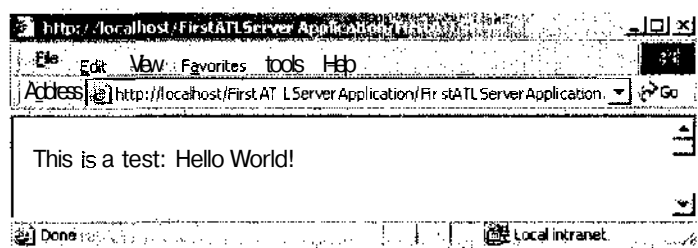
В обсуждаемом нами проекте файл, созданный мастером, возвращает клиенту следующий код HTML:

```
<html>  
  <head>  
  </head>  
  <body>  
    This is a test: Hello World!<br>  
  </body>  
</html>
```

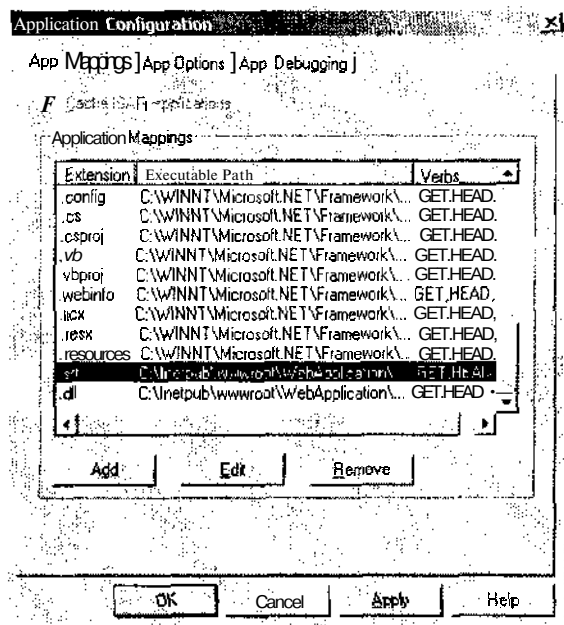
Компоновка, развертывание и запуск приложения

Когда решение компоуется в Visual Studio .NET IDE, на финальной стадии этого процесса в виртуальный каталог IIS копируются три важных файла: `FirstATLServerApplicationIsapi.dll`, `FirstATLServerApplication.srf` и `FirstATLServerApplication.dll`. При выполнении копирования сервер IIS прекращает свою работу, а затем перезагружается.

После этого можно запустить приложение, просто введя соответствующий указатель URL в Web-браузер. В результате открывается SRF-файл, который, в свою очередь, загружает библиотеку ISAPI DLL. Последняя выполняет загрузку библиотеки DLL Web-приложения, вызывает соответствующий обработчик команд и заменяет маркер `Hello` в файле SRF кодом HTML, который генерируется библиотекой Web DLL для передачи клиенту. Результат выглядит следующим образом:



Если на машине не установлен сервер IIS или в процессе копирования возникает ошибка, проект не будет развернут. С другой стороны, Visual Studio .NET позволяет в любой момент модифицировать код DLL, заново развернуть его и протестировать, не покидая IDE. В процессе компоновки `.srf`- и `.dll`-файлы регистрируются на сервере IIS, к чему легко убедиться с помощью менеджера IIS:



После того как проект будет полностью протестирован, его можно развернуть на работающем сервере. Наилучший способ сделать это — создать проект Web Setup Project (Проект развертывания для Web) в Visual Studio .NET. Для получения информации об этом и других мастерах, предназначенных для разработки проектов установки и развертывания, следует обратиться к онлайн-о документации.

Модификация кода

В данном разделе мы выйдем за пределы базового проекта и изменим приложение ATL Server так, что оно будет возвращать таблицу налоговых ставок для всех штатов, находящихся в нашей базе данных. Откроем заголовочный файл `FirstATLServerApplication.h`, принадлежащий к библиотеке Web DLL, и добавим следующее объявление в раздел `private` класса `CFirstATLServerApplicationHandler`:

```
class CATLMap<LPSTR, double> map;
```

Здесь шаблонный класс `CATLMap<>` используется для создания карты, элементы которой состоят из строковых указателей, представляющих код штата, и чисел с плавающей точкой, указывающих ставку налога. Затем карта инициализируется в созданном ранее методе класса `ValidateAndExchange()`:

```
HTTP_CODE ValidateAndExchange()
```

```
{
    map.SetAt("IL", 6.5);
    map.SetAt("CA", 11.0);
    map.SetAt("FL", 12.0);
    map.SetAt("NY", 8.5);
    map.SetAt("OL", 7.5);
    map.SetAt("MA", 4.5);
    map.SetAt("QH", 12.5);
    map.SetAt("NJ", 5.5);
}
```

```

    map.SetAt("UT", 5.0);
    map.SetAt("CO", 14.0);

    // Установка типа информации (content-type)
    m_HttpResponse.SetContentType("text/html");

    return HTTP_SUCCESS;
}

```

Далее создадим новый метод OnTaxRateTableO и присоединим к нему атрибут tag_name:

```

[tag_name(name="TaxRateTable")]
HTTP_CODE OnTaxRateTable(void)
{
    LPSTR Str;
    double taxRate;

    // Отображаем налог
    m_HttpResponse << "<html>";
    m_HttpResponse << "<body bgcolor=#C0C0C0>";
    m_HttpResponse << "<p align=center><b><u>TAX DETAILS</u></b></p>";
    m_HttpResponse << "<table align= center border=1 width=53%>";
    m_HttpResponse << "<tr>";
    m_HttpResponse << "<td width=50% align=center><b>STATE CODE</b></td>";
    m_HttpResponse << "<td width=50% align=center><b>TAX RATE %</b></td>";
    m_HttpResponse << "</tr>";

    // Получение доступа к таблице коллекции
    POSITION pos := map.GetStartPosition();
    while(pos != NULL)
    {
        map.GetNextAssoc(pos, StateCode, taxRate);

        // Форматирование данных о ставках налога для вывода
        CString strRate;
        strRate.Format(_T("% 21f\n"), taxRate);

        m_HttpResponse << "<tr>";
        m_HttpResponse << "<td width=50% align=center>";
        m_HttpResponse << StateCode;
        m_HttpResponse << "</td>";
        m_HttpResponse << "<td width=50% align=right>";
        m_HttpResponse << strRate;
        m_HttpResponse << "</td>";
        m_HttpResponse << "</tr>";

    }

    m_HttpResponse << "</table>";
    m_HttpResponse << "</html>";

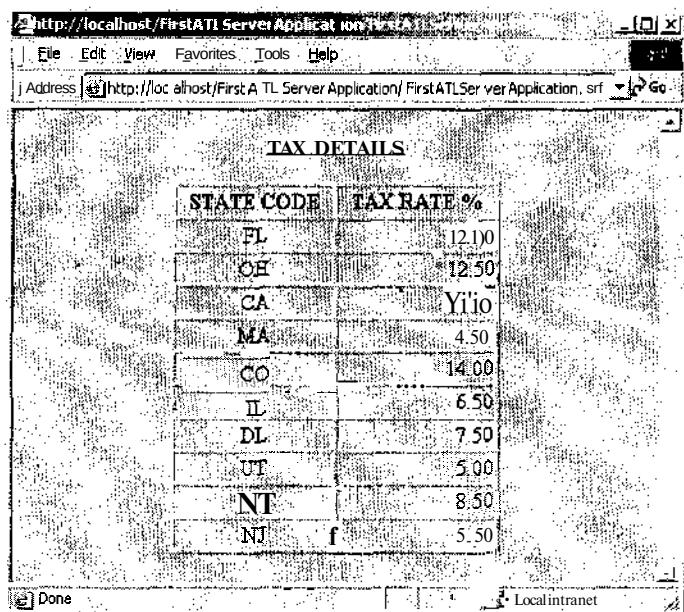
    return HTTP_SUCCESS;
}

```

Обнаружив маркер TaxRateTable в SRF-файле, библиотека DLL расширения ISAPI вызовет метод OnTaxRateTableO. Этот метод считывает содержимое карты и создает таблицу HTML для передачи клиенту. Очевидно, для того, чтобы приложение заработало, необходимо внести изменение в файл SRF, добавив туда маркер TaxRateTable:

```
{{handler FirstATLServerApplication.dll/Default}} {{TaxRateTable}}
```

При успешном выполнении описанных операций на экран будет выведена следующая таблица:



The screenshot shows a web browser window with the address bar displaying 'http://localhost/FirstATLServerApplication/FirstATLServerApplication.srf'. The browser's menu bar includes 'File', 'Edit', 'View', 'Favorites', 'Tools', and 'Help'. The main content area displays a table titled 'TAX DETAILS' with two columns: 'STATE CODE' and 'TAX RATE %'. The table contains the following data:

STATE CODE	TAX RATE %
FL	12.10
OH	12.50
CA	Y110
MA	4.50
CO	14.00
IL	6.50
DL	7.50
UT	5.00
NI	8.50
NJ	5.50

The browser's status bar at the bottom shows 'Done' and 'Localintranet'.

Создание гостевой книги с помощью ATL Server

В данном разделе мы обсудим создание законченного приложения ATL Server, использующего многие новые возможности библиотеки ATL. Это приложение может быть установлено на любом Web-сайте для сбора информации о посетителях в целях маркетинга. Приложение использует форму HTML для ввода пользователями сведений, которые сохраняются в базе данных после проверки на стороне сервера корректности вводимой информации. Кроме того, приложение посылает пользователям по электронной почте уведомления о регистрации.

Создадим в Visual Studio .NET новый проект ATL Server и назовем его GuestBook (Гостевая книга). Сохраним все установки, задаваемые мастером по умолчанию. В результате в решении будут созданы два проекта:

- Проект GuestBook, содержащий код DLL Web-приложения
- Проект GuestBookIsapi, включающий код ISAPI DLL

Мастер также генерирует SRF-файл, который мы изменим для того, чтобы создать пользовательский интерфейс.

Модификация файла SRF

ПОЛНОСТЬЮ ЭТОТ файл может быть загружен с Web-сайта издательства Wrox вместе с остальным кодом для данной главы. Как и ранее, файл SRF включает элементы HTML-дизайна формы. Поэтому необходимо добавить сюда элемент `form` вместе с операцией `'submit'`, которая будет реализовывать запрос к файлу:


```
<form id="Form1"
  action="http://localhost/GuestBook/GuestBook.srf" method="post">
</form>
```

Когда клиент обращается к SRF-файлу, вызывается метод, определенный в классе обработчика. Этот метод проверяет корректность данных, вводимых пользователем, и выводит на форму любые сообщения об ошибках. При первом вызове формы проверка не должна проводиться, поскольку форма пуста. Чтобы реализовать такое поведение, создадим скрытую переменную, добавив приведенный ниже HTML-код в файл SRF (позже мы покажем, как эта переменная используется в классе обработчика):

```
<input type="hidden" name="bNotFirst" value="0" size="1">
```

Создадим форму, перетаскивая мышью элементы управления с панели элементов Toolbox. Ниже показано, как выглядит файл SRF в режиме Design (Дизайн):

В начале файла маркер `handler` указывает имя библиотеки DLL Web-приложения и определяет класс, где содержатся запрашиваемые обработчики. Прежде всего необходимо проверить состояние, возвращаемое маркером `NotValidating`, чтобы установить, требуется ли проверка корректности данных в полях ввода. При первом вызове SRF-файла введенные пользователем данные отсутствуют и маркер `NotValidating` возвращает значение `true`. После этого для пользователя открывается пустая форма. Затем форма использует маркер `GetControls` с соответствующими значениями параметров для отображения вводимых пользователем данных в каждом поле.

В предложении `else` (т.е. когда форма вызывается во второй раз и содержит введенную информацию) мы сохраняем значения полей ввода в базе данных и посылаем пользователю уведомление. Маркер `StoreData` выполняет первую из этих операций, а `SendMail` — вторую.

Реализация обработчиков

После создания формы нужно реализовать различные обработчики в файле `GuestBook.h`. Изменим заголовочный файл, добавив туда следующие объявления закрытых переменных:

```
Private:
    CString Guest_Email;
    CString Guest_FName;
    CString Guest_LName;
    bool valid_input;
    CString errorMsg;
```

Маркер *GetControls*

Напишем теперь код для методов, связанных с маркерами, которые включены в трафарет. Начнем с метода, соответствующего маркеру `GetControls`:

```
[tag_name(name = "GetControls")]
HTTP_CODE OnGetControls(char* controlName)
{
    if(!strcmp(controlName, "Profession", 10))
    {
        m_HttpResponse << "<select name=\"Profession\">\n";
        for(int i = 0; i < (sizeof(professions) / sizeof(char*)); i++)
        {
            m_HttpResponse << "    values=\"";
            professions(i) << "\">";
            professions(i) << "</option>\n";
        }
        m_HttpResponse << "</select>";
        return HTTP_SUCCESS;
    }
    else
    {
        m_HttpResponse << "<input name=\""; << controlName << "\">";
        return HTTP_SUCCESS;
    }
}
```

В этом методе вначале проверяется входной параметр, который обозначает имя элемента управления. Исходя из этого имени, создается соответствующий элемент управления HTML для пользователя. Массив `profession`, на который ссылается код, определен во внешнем файле `ControlData.h`.

Маркер *StoreData*

Маркер `StoreData` сохраняет значения полей ввода в базе данных. Ниже представлен код, соответствующий этому маркеру:

```
[tag_name(name = "StoreData")]
HTTP_CODE OnStoreData(void)
{
    // Возвращаем значение true тогда и только тогда, когда подтверждена
    // корректность данных в каждом поле формы
    CDataSource ds;
    CCommand<CAccessor<CGuest>> cmd;
    CSession session;

    HRESULT hr = ds.OpenFromInitializationString(L"Provider=
Microsoft.Jet.OLEDB.4.0; User ID=Admin; Data Source=C:\\GuestDB.mdb");
```

```

if(SUCCEEDED(hr))
{
    if(SUCCEEDED(session.Open(ds)))
    {
        CString FirstName(
            CA2CT(m_HttpRequest.FormVars.Lookup("FirstName")));
        CString LastName(
            CA2CT(m_HttpRequest.FormVars.Lookup("LastName")));
        CString Email(
            CA2CT(m_HttpRequest.FormVars.Lookup("Email")));
        CString Profession(
            CA2CT(m_HttpRequest.FormVars.Lookup("Profession")));

        char str[15];
        char buffer[3];
        char Item[50];
        char Interests[500];

        strcpy(Interests, "Interests: ");
        for(int i = 1; i < 9; i++)
        {
            strcpy(str, "chkInterest");
            strcat(str, " ");
            strcat(str, buffer);
            strcpy(Item, "");

            strcpy(Item, m_HttpRequest.FormVars.Lookup(str));

            if(strcmp(Item, "") != 0)
            {
                if(strcmp(Interests, "Interests: ") != 0)
                    strcat(Interests, " ");
                strcat(Interests, Item);
            }
        }

        CString SQLString("INSERT INTO Guest_Table VALUES(");
        SQLString += FirstName;
        SQLString += " ";
        SQLString += LastName;
        SQLString += " ";
        SQLString += Email;
        SQLString += " ";
        SQLString += Profession;
        SQLString += " ";
        SQLString += Interests;
        SQLString += " )";

        // Выполнение запроса на добавление записей
        hr = cmd.Open(session, SQLString);
        m_HttpResponse << "<FONT color='#ff0033' size='6'>";
        m_HttpResponse << "Congratulations!";
        m_HttpResponse << "You have registered successfully.</FONT>";
        Guest_Email = CA2CT(m_HttpRequest.FormVars.Lookup("Email"));
        Guest_FName = CA2CT(m_HttpRequest.FormVars.Lookup(
            "FirstName"));
        Guest_LName = CA2CT(m_HttpRequest.FormVars.Lookup(
            "LastName"));
    }
}

```

```

    }
    else
    {
        m_HttpResponse << "<FONT color=#ff0033>Error at server.
    }
}
else
{
    m_HttpResponse << "<FONT color=#ff0033>Error at server.</FONT>";
}
return HTTP_SUCCESS;
}

```

Здесь вначале объявлены переменные, представляющие источник данных, команду и сеанс, а затем открывается источник данных (для конкретной системы необходимо правильно задать строку соединения). Затем для обновления базы данных создается строка SQL путем считывания значений переменных формы. При этом используется свойство `FormVars` объекта `CHttpRequest`. Далее выполняется запрос и отправляется уведомление пользователю.

Маркер SendMail

Во время выполнения маркер `SendMail` в файле `SRF` заменяется выходными данными метода `OnSendMail()`:

```

[tag_name(name = "SendMail")]
HTTP_CODE OnSendMail(void)
{
    CString Message; ("Congratulations!");
    Message += Guest_FName;
    Message += " ";
    Message += Guest_LName;
    Message += ", You have registered successfully.";

    CMessage msg;
    msg.SetSender("abcom@vsnl.com");
    msg.SetSenderName(Guest_FName);
    msg.AddRecipient(Guest_Email);
    msg.AddCc("abcom@vsnl.com");
    msg.SetSubject("Confirmation");
    msg.AddText(Message);

    CSMTCPConnection connection;
    if(!connection.Connect("202.54.1.1"))
    {
        m_HttpResponse << "not";
        return HTTP_SUCCESS_NO_PROCESS;
    }

    if(!connection.SendMessage(msg))
    {
        m_HttpResponse << "Failed to send message!";
        return HTTP_SUCCESS_NO_PROCESS;
    }

    m_HttpRequest.Deletefiles();
    return HTTP_SUCCESS;
}

```

Вначале этот метод составляет сообщение пользователю. Затем создается объект `CMimeType` и задаются его свойства. Далее устанавливается соединение с почтовым сервером, для чего вызывается метод `Connect()` объекта `CSMTPConnection` (здесь необходимо задать правильный указатель URL для конкретного почтового сервера). Наконец, мы отправляем сообщение, вызвав метод `SendMessage()` объекта `Connection`.

Маркер *NotValidating*

Метод, соответствующий маркеру `NotValidating`, имеет вид:

```
[tag_name(name = "NotValidating")]
HTTP_CODE_OnNotValidating(void)
{
    // Возвращаем значение true тогда и только тогда, когда подтверждена
    // корректность данных в каждом поле формы
    if(!valid_input)
    {
        if((m_HttpRequest.FormVars.Lookup("bNotFirst"))){
            m_HttpResponse < "<FONT color= #ff0033' size='3'"
            < errorMsg < "</FONT>"
            return HTTP_SUCCESS;
        }
        return HTTP_S_FALSE;
    }
}
```

Этот код проверяет состояние флага `valid_input`, для которого в методе `ValidateAndExchange()` устанавливается значение `false`, если данные в каком-либо из полей ввода не удовлетворяют установленным критериям. Поскольку мы не хотим вывести сообщение об ошибке, когда форма вызывается в первый раз, наш код проверяет наличие скрытой переменной `bNotFirst` в объекте запроса.

Метод *ValidateAndExchange*

Наконец, обратимся к реализации метода `ValidateAndExchange()`, где проверяется корректность данных в каждом из полей ввода на основе заранее установленных правил. Код проверки данных в полях `FirstName` и `LastName` приведен ниже:

```
m_HttpRequest.FormVars.Validate("FirstName", (LPCSTR*)NULL, 1, 20,
                                &CValidate);
- if(!CValidate.ParamsOK() || !atlRegExp.Match(
    CA2CT(m_HttpRequest.FormVars.Lookup("FirstName")), &atlRematchContext))
{
    valid_input = false;
    lstrcpy(errorMsg, "First name is required and may only contain letters
    and spaces, and length should not exceed 20.");
    return HTTP_SUCCESS;
}

m_HttpRequest.FormVars.Validate("LastName", (LPCSTR*)NULL, 1, 20,
                                &CValidate);
if(!CValidate.ParamsOK() || !atlRegExp.Match(
    CA2CT(m_HttpRequest.FormVars.Lookup("LastName")), &atlRematchContext))
{
    valid_input = false;
    lstrcpy(errorMsg, "Last name is required and may only contain letters
    and spaces, and length should not exceed 20.");
    return HTTP_SUCCESS;
}
```

Остальные поля проверяются аналогичным образом (подробности см. в полном тексте программы).

Компоновка и запуск проекта

После компоновки проекта можно запустить его с помощью Internet Explorer, ввести некоторые данные в форму HTML и нажать кнопку Submit. Все поля ввода будут проверены созданной нами библиотекой DLL Web-приложения. Если проверка подтвердит корректность введенной информации, то она запишется в таблицу базы данных, а клиенту будет отправлено уведомление по электронной почте.

Другие возможности ATL Server

До сих пор мы рассматривали особенности ATL Server, связанные исключительно с получением динамически генерируемого HTML-кода. Кроме того, ATL Server предоставляет классы, обеспечивающие организацию потоковых пулов, кэширование и мониторинг производительности. Все эти возможности мы сейчас обсудим.

Организация потоковых пулов

Библиотека ATL Server содержит класс `CThreadPool`, позволяющий организовать пул рабочих потоков. Функции этого класса реализуются поверх высокоэффективных портов завершения операций ввода-вывода (IO completion ports), поддерживаемых Windows NT. Объявление класса `CThreadPool` выглядит следующим образом:

```
template <class Worker, class ThreadTraits = DefaultThreadTraits>
class CThreadPool : public IThreadPoolConfig
```

Для каждого потока, принадлежащего к пулу, в стеке размещается экземпляр создаваемого пользователем класса потока `Worker`. Метод `Initialized` этого класса вызывается при создании объекта, а метод `Terminated` — при его уничтожении. Метод `Execute()` реализует функциональность, т.е. выполняет необходимую работу.

Система создает набор рабочих потоков и сохраняет их в рамках пула. Независимый поток выполняет мониторинг состояния всех ожидающих операций, находящихся в очереди потокового пула. Когда поступает сигнал о некотором событии, один из потоков, принадлежащих к пулу, выбирается для исполнения функции обратного вызова. Результатом такой организации является эффективное использование ресурсов: если пул содержит ограниченное количество потоков, приложение будет способно обслуживать очень большое число запросов.

Если поток не должен ждать внешнего события и предназначен только для реализации бизнес-логики, он все равно может работать в рамках потокового пула. Для этого класс `CThreadPool` предоставляет метод `QueueUserWorkItem()`.

Кэширование

Интерфейс ATL Server API предоставляет разработчику приложения большие возможности для управления кэшированием в процессе сетевых коммуникаций. Ниже перечислены некоторые наиболее часто используемые для этого классы:

- `CFileCache` — реализует кэш путем записи файлов на диск, а их имена сохраняются в памяти в отдельном списке. Когда в соответствии с установленными нами правилами истекает срок хранения файла, помещенного в кэш, данный файл удаляется.
- `CStencilCache` — с помощью данного класса организуется кэш трафаретов, где хранятся заранее обработанные файлы трафаретов, что сокращает время отклика.

- а `CGlobalCache` — обеспечивает функциональность, необходимую для кэширования больших двоичных объектов. Данные добавляются в кэш и удаляются оттуда с помощью методов `Add()` и `RemoveEntry()` соответственно. Для получения числа элементов, помещенных в кэш, применяется метод `GetCurrentEntryCount()`.
- `CMemoryCache` — может применяться для сохранения в памяти элементов разных типов.

Мониторинг производительности

Для получения информации о производительности приложений, созданных с помощью ATL Server, можно использовать интерфейс мониторинга производительности Win32 API, а также средства ATL, обеспечивающие мониторинг производительности. Последняя технология является мощным средством отладки и протоколирования. Она позволяет выполнять тонкую настройку Web-приложений и требует выполнения двух условий:

- Контролируемое приложение должно определять значения параметров, характеризующих производительность, и открывать к ним доступ.
- Приложение, выполняющее мониторинг, должно иметь возможность находить и извлекать указанные данные.

Во втором случае вместо создания отдельного приложения, обеспечивающего мониторинг, можно использовать консоль контроля производительности Windows (Windows Performance Console). Выполнить второе условие, т.е. организовать данные, характеризующие производительность, позволяют следующие средства ATL:

- Менеджер объектов контроля производительности (performance object manager)
- Объект контроля производительности (performance object)
- Счетчик производительности (performance counter)

Менеджер объектов контроля производительности предоставляет приложению, выполняющему мониторинг производительности, объекты двух других типов и управляет доступом к ним. Объекты контроля производительности и счетчики размещаются в совместно используемых областях памяти, конфигурируемых менеджером.

Объект контроля производительности может содержать некоторые или все данные, характеризующие производительность, которые предоставляются приложением. Каждому такому объекту ставится в соответствие имя, удобное для восприятия человеком. Выполняющее мониторинг приложение, такое как Windows Performance Console, может применять эти имена при создании пользовательского интерфейса. Один объект контроля производительности может быть совместно использован несколькими экземплярами приложения, или одно приложение способно создать несколько экземпляров для обеспечения более специфического мониторинга.

Счетчик производительности представляет собой отдельный элемент данных о производительности приложения. Один объект контроля производительности может содержать несколько счетчиков, каждый из которых имеет имя, удобное для чтения. Данные счетчика хранятся в совместно используемой памяти, и доступ к ним должен осуществляться только через безопасный для потока код.

ГЛАВА 11



Создание Web-служб с помощью ATL Server

Изменение Интернета происходит в направлении развития инфраструктуры, ориентированной на предоставление услуг. Люди используют компьютеры для чтения электронной почты, получения информации о биржевых котировках, для совершения покупок, оплаты счетов и просто для того, чтобы найти лучший китайский ресторан в городе. В разных обстоятельствах для получения доступа к таким услугам потребитель может выбрать PDA, мобильный телефон, настольный компьютер и т.д. Выражаясь техническим языком, можно сказать, что пользователи нуждаются в переходе от традиционного ПО к программам, основанным на службах, доступ к которым может быть реализован с помощью широкого спектра клиентских устройств и других служб.

Существующие технологии, такие как DOOM, CORBA, RMI и RPC, способны обеспечить предоставление услуг через Web. Однако приложения, основанные на этих технологиях, ориентированы на сильно связанные распределенные компоненты, а стандарт коммуникаций между различными технологиями отсутствует. Из-за этого исключительно сложно добиться взаимодействия двух различных служб друг с другом.

Мы определим Web-службу как модульное приложение, которое поддерживает самоописание и может быть опубликовано в сети, обнаружено и вызвано из любой точки Web. Функциональность Web-службы должна быть доступна через стандартные протоколы Web. Можно утверждать, что Web-службы реализуют новый этап развития распределенных вычислений, имея в виду слабо связанные сервисы, обменивающиеся информацией с помощью XML и HTTP, а не компоненты, взаимодействующие на основе двоичного стандарта. Появление XML дало разработчиком формат, позволяющий описывать информацию в структурированной форме, несущей смысловую нагрузку, и именно это создаст основу для Web-служб. Клиенты могут получать доступ к Web-службам и обмениваться данными с ними независимо от типа приложения, платформы или языка.

Преимущества Web-служб

Мы уже отмечали, что очень важно обеспечить взаимодействие различных Web-приложений. Спрашивается: а действительно ли это настолько необходимо? Однозначный ответ на этот вопрос в определенной степени зависит от специфики работы. Потребность в указанном взаимодействии возникает, если нужно обмениваться документами с другими пользователями и открыть им доступ к службам. Например, возможны такие ситуации:

- а Агентство недвижимости должно дать возможность клиентам запрашивать информацию о последних ценах и направлять онлайн-заказы. Если создать необходимое для этого приложение в виде Web-службы, то к нему можно будет получить доступ не только с настольного компьютера, но и с мобильного телефона, что значительно расширит круг клиентов.
- ❑ Организаторы аукциона захотят принимать от участников онлайн-предложения цены, а также позволить им обмениваться контрактными документами с другими участниками аукциона. И вновь интерфейс Web-службы поможет обеспечить взаимодействие между приложениями, работающими на разных платформах и созданными с помощью различных технологий.
- ❑ Изготовителю продукции имеет смысл опубликовать каталог своих предложений на Web-сайте и предоставить клиенту возможность просматривать его, выполнять поиск нужного продукта, получать справочную информацию и размещать заказы, используя предоставленный интерфейс.
- Страховая компания захочет получать иски от партнерских страховых фирм с целью их быстрого урегулирования. Все это требует обеспечения взаимодействия, которого можно достичь, создав Web-службу.

Как же все это должно работать? Прежде чем клиент воспользуется Web-службой, он должен знать о ее существовании и о том, где ее найти. Отсюда следует, что владелец Web-службы должен опубликовать информацию о предоставляемых услугах в некоторой стандартной информационной системе. Далее, для того чтобы клиент мог воспользоваться Web-службой, необходимо, чтобы последняя каким-либо образом описывала свои методы, параметры, типы данных и т.д.

Поиск информации о Web-службах

После создания Web-службы необходимо опубликовать информацию о ней, чтобы клиенты, партнеры по бизнесу и другие пользователи могли ее найти. Для этой цели создан новый стандарт, называемый **Universal Description, Discovery, and Integration** (Универсальное описание, обнаружение и интеграция — **UDDI**), который определяет общее хранилище данных, где могут быть зарегистрированы Web-службы. Более 250 компаний в настоящее время поддерживают проект UDDI, позволяющий любой организации опубликовать информацию о своих Web-службах и дающий возможность сторонним организациям обнаружить эти Web-службы и воспользоваться ими. Образец спецификации UDDI в настоящее время доступен на Web-сайте www.uddi.org.

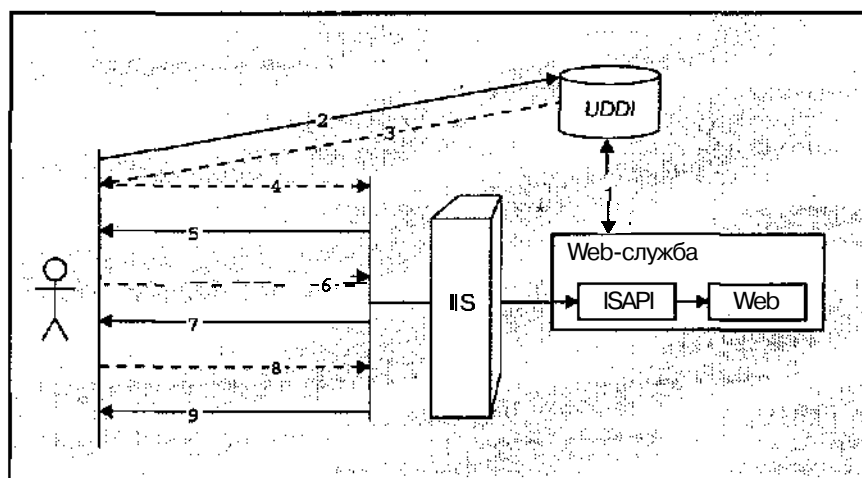
Это хранилище содержит как описательные, так и технические сведения о Web-службах, а также контактную информацию о тех организациях, которые их

предлагают. Там имеется средство локализации области поиска для заданного сегмента бизнеса и/или для конкретной географической области.

В связи с Web-службами чаще всего говорят о двух технологиях, основанных на XML: одна из них называется **Simple Object Access Protocol** (Простой протокол доступа к объектам — **SOAP**) и представляет собой стандарт, описывающий использование XML для осуществления вызова методов между распределенными объектами. Вторая технология носит имя **Web Services Description Language** (Язык описания Web-служб — **WSDL**). Она описывает Web-службы на языке определяемых ими методов в той форме, которая понятна клиентам, посылающим запросы.

Web-службы и ATL Server

Разумеется, у читателя уже возник вопрос, как написать Web-службу. Нужно ли вручную кодировать эти SOAP и WSDL? К счастью, средства Visual Studio .NET делают вместо нас основной объем работы, поэтому создание Web-службы ни в коей мере не является таким сложным, как это может показаться. Архитектура Web-служб, созданных на основе ATL Server, представлена на следующем рисунке. После первого знакомства с ним мы опишем его детали:



1. На первом этапе мы создаем Web-службу и разворачиваем ее в реестре UDDI. Здесь мы обеспечиваем ссылку на Web-службу вместе с бизнес-описанием, которое используется для определения ее категории. Идея состоит в том, что, если кто-то будет заниматься поиском услуг, относящихся, например, к путешествиям, он может быстро и легко сузить область поиска до рамок указанной категории.
2. Заинтересованный клиент ищет Web-службу в реестре.
3. После нахождения нужной службы клиент выбирает ее и получает на нее ссылку. Это будет указатель URL, в котором имя целевого файла оканчивается на `.disco`, например: `http://www.aucocom.com/TravelDetails.disco`
4. Клиент просматривает этот файл, чтобы узнать первичный адрес Web-службы.
5. Клиент получает путь к Web-службе в формате XML, например:

```
<?xml version="1.0"?>
<discovery xmlns="http://schemas.xmlsoap.org/disco/">
  <contractRef xmlns="http://schemas.xmlsoap.org/disco/sc1"
    ref="http://www.abcom.com/Travel/
      TravelDetails.dll?Handler=GenTravelWSO1"
    docRef="http://www.abcom.com/Travel/TravelDetails.htm"/>
</discovery>
```

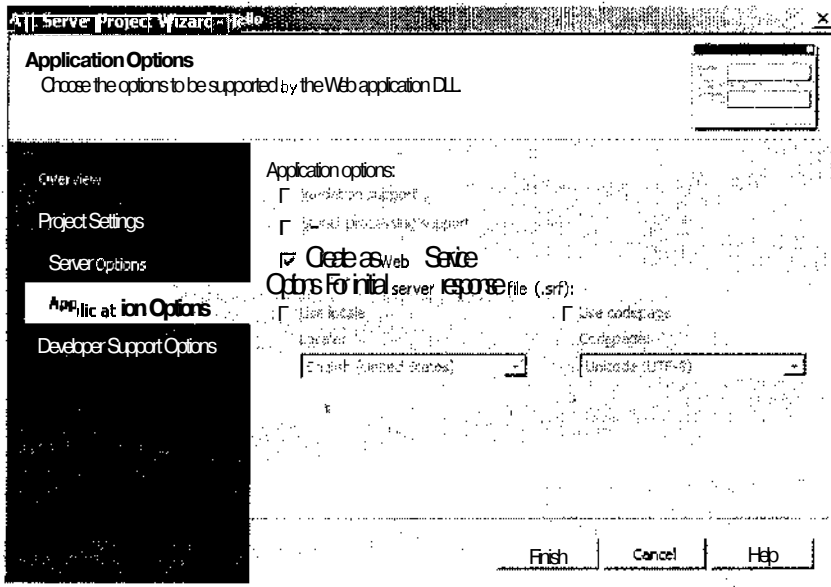
6. Клиент находит ссылку на службу (указатель URL, содержащий текст "Handler"; см. выше), чтобы получить файл описания Web-службы.
7. Клиент получает описание Web-службы и изучает его. Указанный файл содержит описание методов, включая их имена, количество и типы параметров, а также типы возвращаемых величин. Клиент использует эту информацию для вызова службы и предоставления ей корректных данных. Если понадобится, клиенты могут разрабатывать интерфейсы для доступа к Web-службе.
8. Клиент вызывает Web-службу. Этот запрос приходит на сервер IIS, который (на основе URL и указанного расширения) передает его в соответствующую библиотеку DLL расширения ISAPI. Последняя, в свою очередь, определяет, какой обработчик указан в запросе, и передаст запрос нужной библиотеке DLL Web-приложения. Web DLL передает данный запрос объекту C++. Различие между Web-службой и Web-приложением возникает на конечной стадии. В Web-службе объект C++ способен кодировать и декодировать данные SOAP. Необходимый для этого программный код добавляется компилятором, когда тот анализирует атрибут `soap_handler`.
9. Клиент получает ответ на свой запрос в формате XML.

Создание Web-службы

В данном разделе мы обсудим создание простой Web-службы с помощью ATL Server и рассмотрим приложение "Hello World", которое генерируется мастером. Анализируя этапы разработки приложения и изучая генерируемый код, мы получим хорошее представление о том, как работает описываемая технология. Позднее в этой главе мы рассмотрим создание более сложного приложения "Stock Ticker" (Биржевой аппарат), которое может использоваться фондовым брокером для предоставления клиенту последних котировок и приема онлайн-заказов.

Создание проекта

Создадим новый проект в Visual Studio .NET, выбрав из списка тип проекта ATL Server Web Service (Проект создания Web-службы с помощью ATL Server). Этот мастер фактически не отличается от мастера ATL Server Project Wizard (см. предыдущую главу) за исключением того, что опция Create as Web Service (Создать как Web-службу) выбирается здесь по умолчанию. Назовем проект Hello и нажмем кнопку ОК. Поскольку мы собираемся сохранить для проекта установки по умолчанию, нажмем кнопку Finish, но перед этим сделаем небольшую паузу и обратим внимание на то, как выбор проекта создания Web-службы ограничивает набор опций:



"Ограничения" здесь определяются тем, что выходные данные проекта будут представлены в формате SOAP. В проекте отсутствует файл SRF, и любые связанные с ним опции становятся излишними. Если теперь щелкнуть кнопку Finish, мастер создаст, в частности, следующие файлы:

- Hello.cpp и Hello.h.

Эти файлы содержат класс обработчика запросов ATL Server, конфигурация которого зависит от опций, задаваемых в окне мастера.

- Hello.disco.

Данный файл включает информацию, которая позволяет открывать доступ к Web-службе через Web-серверы, поддерживающие протокол DISCO (discovery — обнаружение). Этот файл необходим при создании клиентов для Web-службы.

Файл Hello.h

Рассмотрим генерируемый мастером заголовочный файл Hello.h. Прежде всего этот файл определяет пространство имен HelloService, внутри которого размещаются все остальные объявления:

```
namespace HelloService
{
    // Остальной код
}
```

Далее здесь объявляется интерфейс IHelloService, содержащий Web-метод (т.е. метод, который использует SOAP для коммуникаций с клиентом) с именем HelloWorld(). Данный метод принимает один параметр типа BSTR и возвращает параметр того же типа:

```
[ uuid("F7CE2FE0-09E6-4DE4-8C05-94CF01E2BB9D"), ... ]
```

```

    object
  ]
  interface IHelloService
  {
    [id(1)] HRESULT HelloWorld([in] BSIR bstrInput,
                              [out, retval] BSIR *bstrOutput);
  };

```

После этого объявляется класс CHelloService вместе с двумя атрибутами (request_handler и soap_handler):

```

[
  7- request_handler(name="Default" sdl="GenHelloWSDL"),
    soap_handler(
      name="HelloService",
      namespace="urn:HelloService",
      protocol="soap"
    )
]

```

Атрибут request_handler

Атрибут request_handler указывает, что класс представляет собой обработчик запросов ATL Server. Данный класс реализовывает Web-метод, определенный перед этим в интерфейсе, а объекты класса используются для обработки запросов клиентов. В отличие от примера, рассмотренного в главе 10, атрибут имеет второй параметр — sdl, указывающий имя генерируемого компилятором обработчика "GenHelloWSDL", который возвращает клиенту WSDL-данные. Это значит, что мы получим WSDL-данные, если введем следующий указатель URL в браузер:

http://ServerName/Hello.dll?Handler= GenHelloWSDL

Параметр sdl имеет смысл лишь тогда, когда класс объявляется вместе с атрибутом soap_handler.

Атрибут soap_handler

Поскольку Web-службы используют протокол SOAP для коммуникаций с клиентом, любые методы, предоставляемые клиенту как Web-сервисы, должны поддерживать этот протокол. Такая поддержка обеспечивается за счет присоединения атрибута soap_handler к классу, содержащему указанные методы. После объявления класса с этим атрибутом в проект автоматически добавляется код, необходимый для выполнения преобразования между SOAP и сигнатурой, определенной для метода.

Полное определение атрибута SOAP имеет вид:

```

[soap_handler(name,
              namespace,
              protocol,
              style,
              use
            )]

```

Параметр name задает имя, которое будет использовать клиент, чтобы ссылаться на Web-службу. Если не указать этот параметр, в проект вместо него будет добавлено имя, полученное путем добавления слова Service к имени класса. Параметр namespace указывает пространство имен XML, используемое при идентификации службы, методов и типов данных. Данный параметр также является необязательным, и если он опущен, то заменяется именем класса.

Мы вернемся к обсуждению этого параметра, когда будем рассматривать компоновку Web-службы.

Третий параметр задает протокол, используемый для доступа к Web-службе. В настоящее время поддерживается только протокол SOAP, и именно он включается мастером в код.

Параметр Style является необязательным и указывает стиль операций, выполняемых Web-службой. Он может принимать значения document или rpc. Последнее значение задается по умолчанию, и именно оно необходимо при написании Web-службы. Наконец, параметр use определяет, должны ли кодироваться части сообщений WSDL. Данный параметр может принимать только два значения: literal и escaped; при этом второе значение также задается по умолчанию.

Вернемся к файлу Hello.h, в конце которого помещается сам класс обработчика, включающий генерируемую реализацию единственного метода интерфейса:

```
class CHelloService : public IHelloService
{
public:
    [soap_method]
    HRESULT HelloWorld(/*[in]*/ BSTR bstrInput,
                      /*[out, retval]*/ BSTR *bstrOutput)
    {
        CComBSTR bstrOut(L"Hello ");
        bstrOut += bstrInput;
        bstrOut += L"!";
        *bstrOutput = bstrOut.Detach();

        return S_OK;
    };
};
```

Как и следовало ожидать, генерируемая реализация очень проста. Она лишь считывает значение параметра [in], добавляет его к строке "Hello" и возвращает модифицированную строку вызывающему методу, используя параметр out.

Атрибут soap_method

Описываемый нами метод объявляется с атрибутом soap_method, который указывает, что метод будет обмениваться данными с клиентом по протоколу SOAP. Ниже приведен синтаксис атрибута, где необязательный параметр name задает WSDL-имя метода:

```
[soap_method(name)]
```

Если атрибут присоединен к методу, к этому методу будет добавлен весь код, необходимый для обработки запросов/ответов в формате SOAP. Когда клиент вызывает данный метод, используя протокол SOAP, выполняется такая последовательность действий:

- Извлекаются параметры, включенные в запрос SOAP.
- а Параметры преобразуются в соответствующие типы данных C++.
- Вызывается метод, куда передаются эти преобразованные параметры.
- Создается ответ клиенту в формате XML.
- Ответ возвращается клиенту.

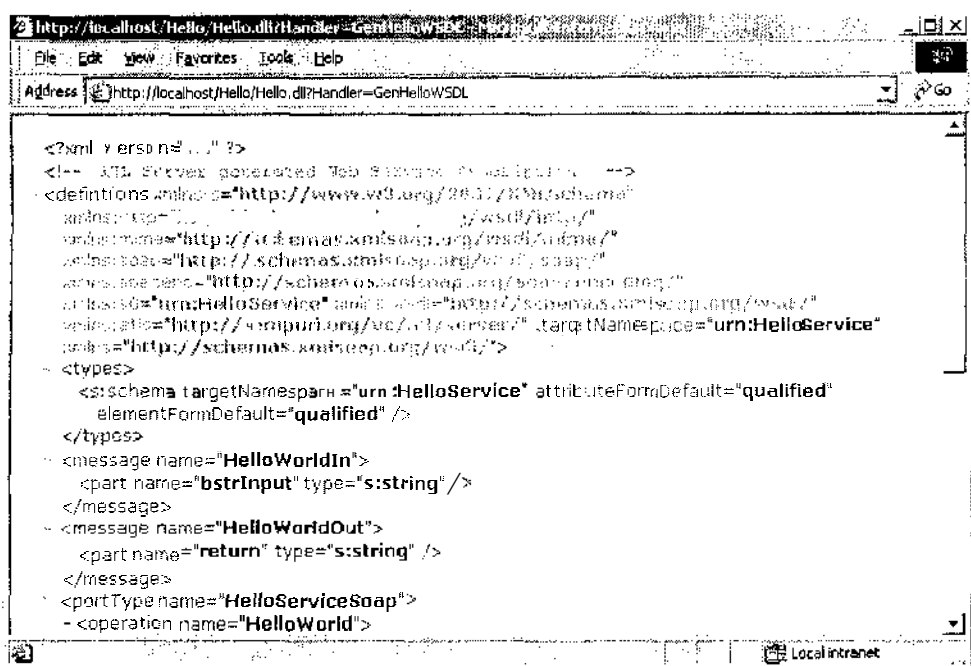
Разумеется, все это происходит "молча". Благодаря ATL Server нам не приходится пошевелить и пальцем, чтобы реализовать описанный процесс в приложении.

Компоновка проекта

При компоновке этого решения в Visual Studio .NET среда разработки создаст необходимые для проекта библиотеки DLL и запишет их либо в корневой каталог Web-сервера (IIS), либо в папку, определенную нами в окне Project Settings (Установки проекта). В результате библиотеки DLL, относящиеся к проекту, будут возвращены, а Web-служба — готова к использованию.

Запуск проекта

Для запуска приложения из Visual Studio .NET необходимо нажать комбинацию клавиш **<Ctrl> + F5**, как и в случае проекта любого другого типа. При этом генерируется код WSDL и выводится в окна браузера:



Код WSDL описывает методы SOAP, определенные в классе `CHelloService`. Этот код включает имена методов, типы параметров и возвращаемых величин, а также содержит другую информацию. В начале выходных данных представлены импортируемые пространства имен. Например, пространство имен, определенное атрибутом класса `soap_handler`, отображено в следующей строке:

```
xmlns:s0="urn:HelloService"
```

После определения пространств имен и типов объявляются два сообщения (входящее `HelloWorldIn` и исходящее `HelloWorldOut`):

```
<message name="HelloWorldIn">
  <part name="bstrInput" type="s:string" />
</message>
<message name="HelloWorldOut">
  <part name="return" type="s:string" />
</message>
```

Элемент `portType` описывает службу `HelloWorld`. Входящее и исходящее сообщения, используемые операцией `HelloWorld`, помечаются двумя соответствующими атрибутами:

```
<portType name="HelloServiceSoap">
  <operation name="HelloWorld">
    <input message="s0:HelloWorldIn" />
    <output message="s0:HelloWorldOut" />
  </operation>
</portType>
```

Далее элемент `binding` определяет транспортный механизм как HTTP и стиль данных как `rpc`. Тело блока данных SOAP кодируется с использованием стиля кодировки, задаваемого атрибутом `encodingStyle`:

```
<binding name="HelloServiceSoap" type="so:HelloServiceSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="rpc" />
  <operation name="HelloWorld">
    <soap:operation soapAction="#HelloWorld" style="rpc" />
    <input>
      <soap:body use="encoded" namespace="urn:HelloService"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </input>
    <output>
      <soap:body use="encoded" namespace="urn:HelloService"
        encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </output>
  </operation>
</binding>
```

В конце выходных данных размещается код WSDL, описывающий Web-метод `HelloService` в элементе `service`:

```
<service name="HelloService">
  <port name="HelloServiceSoap" binding="s0:HelloServiceSoap">
    <soap:address
      location="http://localhost/Hello/Hello.dll?Handler=Default" />
  </port>
</service>
```

Web-служба использует атрибут `Handler` в указателе URL. Для этого атрибута задано значение `Default`, совпадающее со значением аргумента, передаваемого атрибуту `request_handler`, с которым объявляется наш класс.

Простой клиент Web-службы

Теперь нужен лишь клиент, который будет работать с Web-службой. Когда клиентское приложение получает доступ к Web-службе, оно обнаруживает ее, получает на нее ссылку и использует функциональность, предоставляемую этой службой. При этом клиент может принадлежать к одному из следующих типов:

- ☐ Управляемое или неуправляемое приложение C++
- ☐ Приложение на основе браузера
 - КомпонентCOM
- ☐ Другая Web-служба

Мы создадим консольное приложение `Win32`, которое будет клиентом нашей простой Web-службы `Hello`. Разработка данного приложения включает следующие этапы:

- Создание консольного приложения
- Создание Web-ссылки на нашу службу
 - Написание клиентского кода, создающего объект типа Web-службы
- D Вызов Web-метода с использованием указанной ссылки
 - Компиляция и запуск клиентской программы

Создание проекта

Создадим в Visual Studio .NET новый проект Visual C++ Win32 и назовем его HelloClient. В окне мастера выберем компоновку консольного приложения и добавим поддержку ATL. Этот шаг необходим, поскольку Web-ссылка, которую мы добавим, будет зависеть от заголовочных файлов ATL Server. Щелкнем на кнопке Finish для генерации файлов проекта.

Нашему клиентскому приложению нужен прокси-класс для доступа к Web-службе. Этот класс будет играть роль заглушки для Web-службы и содержать код, обеспечивающий преобразование параметров Web-метода в формат SOAP и обратно. Microsoft предоставляет для этой цели утилиту под названием **генератор прокси** (**proxy generator**, файл `sproxy.exe`). Эта утилита запускается из командной строки, принимает в качестве параметра URL Web-службы и генерирует заголовочный файл, который содержит прокси-класс, действующий на стороне клиента:

```
> sproxy /out:Hello.h http://localhost/Hello/
Hello.dll?Handler=GenHelloWSDL
```

Задаваемый нами здесь указатель URL встраивается в генерируемый заголовочный файл прокси-класса. Поэтому важно создать такой класс с указанием реального адреса Web-службы.

Анализ генерируемого заголовочного файла

Если открыть заголовочный файл `Hello.h`, созданный в результате предыдущих действий, мы увидим следующее объявление класса:

```
template <typename TClient = CSoapSocketClientT<>>
class CHelloServiceT: public TClient, public CSoapRootHandler
```

Конструктор класса использует заданный нами обработчик Default:

```
CHelloServiceT(ISAXMLReader *pReader = NULL)
: TClient(_T("http://localhost/Hello/Hello.dll?Handler=Default"))
```

Метод `HelloWorld()` объявляется в теле класса:

```
template <typename TClient>
inline HRESULT CHelloServiceT<TClient>::HelloWorld(
    _BSTR bstrInput,
    _BSTR* __retval)
```

Метод `HelloWorld()` принимает параметр типа `BSTR` и возвращает указатель `BSTR*`. Реализация данного метода вызывает метод `InitializeSOAP()`, чтобы сделать возможной обработку SOAP для запросов и ответов:

```
HRESULT __atlssoap_hr = InitializeSOAP(NULL);
```

а следующий оператор выполняет подготовку ответа в методе:

```
__atlssoap_hr = GenerateResponse(GetWriteStream());
```

Сам же метод вызывается путем вызова метода `SendRequest()`:

```
... __atlssoap_hr = SendRequest(T("SOAPAction: \"#HelloWorld\"\\r\\n"));
```

Наконец, программа получает возвращаемое значение с помощью оператора присваивания:

```
*__retval = __params.__retval;
```

Написание клиентского кода

Откроем сгенерированный файл `stdafx.h` и добавим туда такие объявления:

```
// В целях тестирования отменяем установку интервала ожидания
// для клиента. Величина этого интервала по умолчанию равна 10000 мс.
#define ATL_SOCKET_TIMEOUT INFINITE

// Минимальные системные требования: Windows 98; Windows NT 4.0
// или более поздняя ОС
#define WIN32_WINDOWS 0x0410
```

По умолчанию значение `ATL_SOCKET_TIMEOUT` составляет 10 с, после чего истекает время ожидания и соединение через сокет закрывается. Мы делаем это значение равным `INFINITE`, отменяя установку клиентского интервала ожидания, что может быть полезно при тестировании приложения. В реальных условиях изменять эту константу обычно не требуется.

В случае платформы Windows минимальные системные требования, необходимые для использования Web-служб, — это Win 98, Win NT 4.0 или более поздние версии ОС. Указанные требования задаются во втором определении приведенного фрагмента кода. Естественно, включение в данное определение конкретной константы не означает, что с Web-службой могут работать только Windows-клиенты. Любой клиент, способный генерировать запросы SOAP, получает возможность использовать эту Web-службу.

Добавим в файл `HelloClient.cpp` директиву `#include Hello.h`, а затем напишем новую функцию `Display()`, которая будет использовать Web-службу для вывода сообщений на консоль:

```
void Display()
{
    CComBSTR bstr(20);

    // Создание экземпляра прокси-класса
    HelloService obj;
    HRESULT hr = obj.HelloWorld(CComBSTR(L"Bill"), &bstr.m_str);
    CString result(bstr.m_str);
    printf(result);
}
```

Вначале мы объявляем переменную для передачи ее методу Web-службы, затем получаем ссылку на Web-службу и после этого вызываем метод `HelloWorld()`. Наконец, полученный результат выводится на консоль. Ниже представлен код функции `main()`, вызывающей метод `Display()`:

```
int main(int argc, TCHAR* argv[])
{
    if (SUCCEEDED(CoInitialize(NULL)))
    {
        Display();
        CoUninitialize();
    }
    return 0;
}
```

ЕСЛИ все будет работать нормально, на экране мы увидим сообщение:

Hello Bill!

Анализ клиентского кода

При запуске кода происходит следующее. Строка

```
HelloService::CHelloService obj;
```

создает экземпляр прокси-класса, описанный выше. Затем клиент вызывает метод `HelloWorld()` данного объекта:

```
HRESULT hr=obj.HelloWorld(CComBSTR(L"BILL"), &bstr_m_str);
```

С помощью этого метода строка `Bill` приводится к типу `String`, определенному в типах данных XML Schema. Затем вызов метода инкапсулируется в запрос SOAP, отправляемый серверу. Сервер выполняет анализ запроса, извлекает оттуда параметр и вызывает метод объекта, который возвращает результат в строковом формате C++. Этот результат преобразуется в строку XML на стороне сервера, прежде чем вновь выполняется инкапсуляция в данные SOAP. Затем данные отправляются клиенту. Клиент анализирует полученный ответ в виде данных SOAP, извлекает оттуда результирующую строку в формате XML, преобразует ее в формат C++ и возвращает результат клиентскому коду C++.

Проект службы биржевых котировок

Теперь обсудим пример реальной Web-службы, используемой фондовым брокером. Брокер может позволить зарегистрированным пользователям получать последние биржевые котировки для конкретного выпуска акций и размещать онлайн-заказы.

Напишем Web-службу, которая определяет интерфейс для получения курса акций и размещения заказа. Последовательность вызовов в приложении будет следующей. Вначале клиент указывает интересующий его выпуск акций и получает последние котировки с Web-сайта брокера. Затем клиент размещает заказ на покупку или продажу пакета акций через предоставляемый интерфейс. Наконец, заказ сохраняется в базе данных брокера на сервере для локального просмотра.

Создание базы данных

Для службы фондового брокера необходима база данных. Создадим ее и включим туда три таблицы: `Stock` (Акции), `OrderDetails` (Заказы) и `Login` (Учетные записи). Для демонстрационных целей мы использовали Microsoft Access, однако читатель может выбрать любую СУБД.

Таблица `Stock`

Таблица `Stock` содержит последние цены для каждого выпуска акций, которые котируются на бирже. Она имеет следующую структуру:

Имя поля	Тип данных
<code>stockSymbol</code>	Text
<code>stockPrice</code>	Currency

Здесь поле `stockSymbol` представляет выпуск акций, а поле `stockPrice` — последние биржевые котировки. Если Web-служба брокера обеспечивает онлайн-овое

совершение сделок, эта таблица может обновляться в режиме реального времени и предоставлять последнюю цену покупателя для каждого выпуска акций. Вначале необходимо внести в эту таблицу несколько записей для тестирования (описываемая база данных содержится в загружаемом коде).

Таблица *OrderDetails*

Таблица *OrderDetails* сохраняет заказы, полученные от клиента. Ее структура имеет следующий вид:

Имя поля	Тип данных
OrderID	Number
UserID	Text
StkSymbol	Text
StkPrice	Currency
buysell	Yes/No
Quantity	Number
Date	Date/Time
TotalAmount	Currency

Здесь поле *OrderID* является первичным ключом таблицы и представляет идентификатор полученного заказа. В остальных полях соответственно содержатся следующие данные: *UserID* — идентификатор пользователя; *StkSymbol* — выпуск акций, выбранный для совершения сделки; *StkPrice* — запрашиваемая цена сделки; *buysell* — тип сделки; *Quantity* — количество акций; *Date* — время размещения заказа; *TotalAmount* — полная стоимость сделки.

Таблица *Login*

Таблица *Login* сохраняет такую информацию о пользователях, как: *Name* (Имя), *UserID* (Идентификатор пользователя) и *Password* (Пароль). Ее структура такова:

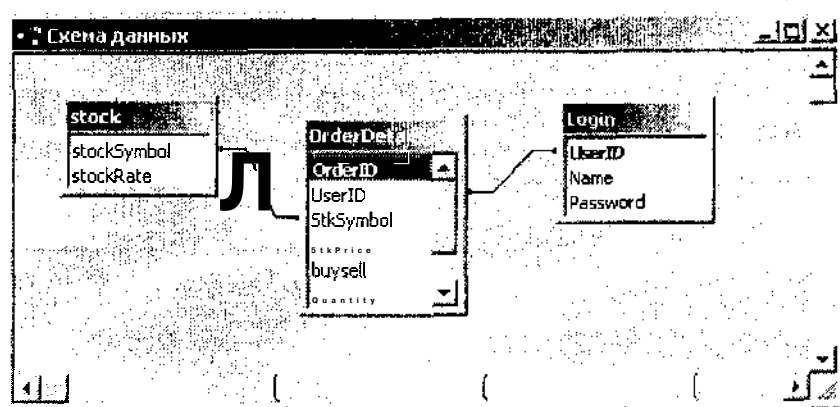
Имя поля	Тип данных
UserID	Text
Password	Text
Name	Text

Связи

Связи между таблицами изображены на снимке экрана, помещенном в начале следующей страницы.

Создание проекта

Вновь создадим проект ATL Server Web Service. Присвоим проекту имя *StockQuote* и сохраним все опции по умолчанию. Даже разработку столь сложного проекта можно начать, оставив неизменными имеющиеся установки. Поэтому щелкнем на кнопке *Finish*, чтобы сгенерировать файлы проекта.



Написание кода

В этом проекте мы удалим создаваемый по умолчанию метод Web-службы Hello() и добавим четыре новых метода:

- getStockQuoteO
- placeOrder()
- ValidateUser()
- AddMemberInfo()

Добавление объявлений методов

Метод getStockQuoteO позволит клиентам Web-службы получать последние биржевые котировки для заданного выпуска акций:

```
[id(1)] HRESULT getStockQuote([in] BSTR strSymbol,
                                [out, retval] float* stockRate);
```

Этот метод принимает строковый параметр, указывающий тот выпуск акций, для которого запрашивается последняя цена покупателя. Метод возвращает вызывающей функции текущую биржевую цену с помощью параметра stockRate.

Объявление метода placeOrder() имеет вид:

```
[id(2)] HRESULT placeOrder([in] BSTR strUserID,
                            [in] BSTR strStockSymbol,
                            [in] float fStockRate,
                            [in] BSTR strBuySell,
                            [in] int Quantity,
                            [out, retval] BSTR strMessage);
```

Данный метод содержит несколько входных параметров:

- strUserID — задает идентификатор пользователя.
- strStockSymbol — указывает выбранный выпуск акций для совершения сделки.
- fStockRate — содержит запрашиваемую цену.
- strBuySell — определяет вид операции: продажа или покупка
- Quantity — задает количество акций.

Сведения об успешной или неудачной попытке размещения заказа у брокера возвращаются клиенту с помощью выходного параметра `strMessage`.

Далее следует объявление метода `ValidateUser()`:

```
[id(3)] HRESULT ValidateUser([in] BSTR strUserID,
                             [in] BSTR strPassword,
                             [out, retval] BSTR* StrResult);
```

Метод имеет два входных параметра:

- `strUserID` — указывает идентификатор пользователя.
- `strPassword` — содержит пароль пользователя.

Сведения о корректности или некорректности этих данных возвращаются клиенту через выходной параметр `StrResult`.

Наконец, приведем объявление метода `AddMemberInfo()`:

```
[id(4)] HRESULT AddMemberInfo([in] BSTR strUserID,
                              [in] BSTR strPassword,
                              [in] BSTR Name,
                              [out, retval] BSTR* strResult);
```

В этом методе имеется три входных параметра:

- `strUserID` — указывает идентификатор пользователя.
- `strPassword` — указывает пароль пользователя.
- а `Name` — задает имя пользователя.

Информация об успешной или неудачной попытке регистрации пользователя возвращается клиенту с помощью выходного параметра `strResult`.

Реализация методов

Следующая задача заключается в реализации Web-методов. Поскольку код для всех этих четырех методов является достаточно громоздким, мы остановимся лишь на первых двух. Исходный код остальных двух методов читатель может проанализировать самостоятельно. Начнем с метода `getStockQuote()`. Прежде всего нужно открыть его для клиентов как метод SOAP, т.е. присоединить к нему атрибут `soap_method`:

```
[soap_method]
```

Далее код метода считывает входной параметр и создает строку SQL для выбора записи в таблице `Stock` (Акции):

```
HRESULT getStockQuote(BSTR strSymbol, float* stockPrice)
{
    CString symbol(bstrSymbol);

    // Создание строки запроса
    CString SQLString("SELECT * FROM stock where stockSymbol=");
    SQLString += symbol;
    SQLString += " ";
}
```

Затем добавим в программу соединение с источником данных и откроем это соединение:

```
CDataSource ds;
CCommand<CAccessor<CStock>> cmd;
```

```

CSession session;

// Соединение с базой данных StockDB.mdb
HRESULT hr = ds.OpenFromInitializationString(L"Provider=
Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data Source=C:\\StockDB.mdb");

// Проверка установки соединения
if(SUCCEEDED(hr))
{
    if(SUCCEEDED(session.Open(ds)))
    {

```

После этого откроем набор записей и получим цену акций для запрашиваемого пакета, переходя к первой записи:

```

hr = cmd.Open(session, SQLString);
if (SUCCEEDED(hr))
{
    cmd.MoveNext();

    // Получаем цену акций для заданного выпуска
    *stockPrice = cmd.m_stockPrice;
}

```

Если программа не работает, как требовалось, возвращаем соответствующую ошибку:

```

else
{
    *stockPrice = -1; // Запрос не выполнен должным образом
}
else
{
    *stockPrice = -2; // Невозможно открыть источник данных
}
else
{
    *stockPrice = -3; // Невозможно установить соединение
}
return S_OK;
}

```

Приведем теперь полный код реализации метода placeOrder(), которая похожа на приведенную выше реализацию метода getStockQuote():

```

[soap_method]
HRESULT placeOrder(BSTR strUserID,
                  BSTR strStockSymbol,
                  float fStockPrice,
                  VARIANT_BOOL BuySell,
                  int Quantity,
                  BSTR* strMessage)
{
    CDataSource ds;
    CCommand<CAccessor<COrder>> > cmd;
    CCommand<CAccessor<COrderID>> > cmd1;
    CSession session;

    HRESULT hr = ds.OpenFromInitializationString = (L"Provider=

```

```

Microsoft.Jet.OLEDB.4.0;User ID=Admin;Data Source=C:\\StockDB.mdb");
if(SUCCEEDED(hr))
{
    if(SUCCEEDED(session.Open(ds)))
    {
        CString stockSymbol(strStockSymbol);
        CString UserID(strUserID);
        long order_ID= 0;
        CString str("SELECT MAX(OrderID) FROM OrderDetails");
        hr= cmd1.Open(session, str);
        if (SUCCEEDED(hr))
        {
            cmd1.MoveNext();
            order_ID= cmd1.m_OrderID;

            // Форматирование даты
            char date[17];
            strdate(date);

            // Расчет и форматирование стоимости сделки
            float Amount= Quantity * fStockPrice;
            CString strAmount;
            strAmount.Format(_T("%.2f\\n"), Amount);

            // Форматирование цены акций
            CString tempPrice;
            tempPrice.Format(_T("%.2f\\n"), fStockPrice);

            char tempQty[20];
            char tempOrderID[20];
            char tempDate[17];
            char tempBuySell[5];
            wsprintf (tempQty, "%d", Quantity);
            wsprintf (tempOrderID, "%d", order_ID + 1);
            wsprintf (tempDate, "%s", date);
            wsprintf (tempBuySell, "%d", BuySell);

            CString SQLString("INSERT INTO OrderDetails VALUES(");
            SQLString += tempOrderID;
            SQLString += ",";
            SQLString += UserID;
            SQLString += ",";
            SQLString += stockSymbol;
            SQLString += ",";
            SQLString += tempPrice;
            SQLString += ",";
            SQLString += tempBuySell;
            SQLString += ",";
            SQLString += tempQty;
            SQLString += ",";
            SQLString += tempDate;
            SQLString += ",";
            SQLString += strAmount;
            SQLString += ")";

            // Выполнение запроса на добавление записей
            hr= cmd.Open(session, SQLString);

```



```

    *strMessage = L"Thank you for placing your order with us!";
}
else
{
    *strMessage = L"Session failed";
}
else
{
    *strMessage = L"Connection failed";
}

return S_OK;
}

```

Здесь строка SQL использует различные входные параметры метода как значения полей в записи. Код открывает соединение с базой данных, устанавливает нужные свойства и добавляет запись. Сведения об успешной или неудачной попытке выполнения операции возвращаются в вызывающий метод через выходной параметр. В реальной ситуации мы бы объединили подобный код в одну транзакцию.

Определение соответствия с базой данных

Предположим, мы написали реализации методов `ValidateUser()` и `AddMemberInfo()`. Однако прежде, чем компоновать проект, необходимо установить соответствие между переменными C++ и столбцами таблиц базы данных, чтобы ее мог использовать класс `CCommand<>`. Создадим заголовочный файл `CStock.h`, который должен содержать описание соответствия для таблицы `Stock`, и добавим в этот файл следующий код:

```

class CStock
{
public:
    // Элементы данных
    float m_stockPrice;
    TCHAR m_stockSymbol[51];

    BEGIN_COLUMN_MAP(CStock)
        COLUMN_ENTRY(2, m_stockPrice)
        COLUMN_ENTRY(1, m_stockSymbol)
    END_COLUMN_MAP()
};

```

Далее создадим еще один заголовочный файл `COrder.h`, чтобы задать соответствие между кодом C++ и таблицей `OrderDetails`. Добавим в этот файл приведенное ниже определение. Класс `COrder` обеспечивает указанное соответствие для выполнения операций вставки (INSERT) в таблицу `OrderDetails`:

```

class COrder
{
public:
    // Элементы данных
    VARIANT_BOOL m_buySell;
    LONG m_Quantity;
    TCHAR m_StkSymbol[20];
    float m_TotalAmount;
    int m_OrderID;
    TCHAR m_UserID[50];
};

```

```

DATE m_Date;
float m_StkPrice;

// Схема соответствия столбцов
BEGIN_COLUMN_MAP(COrder)
    COLUMN_ENTRY(1, m_OrderID)
    COLUMN_ENTRY(2, m_UserID)
    COLUMN_ENTRY(3, m_StkSymbol)
    COLUMN_ENTRY(4, m_StkPrice)
    COLUMN_ENTRY(5, m_BuySell)
    COLUMN_ENTRY(6, m_Quantity)
    COLUMN_ENTRY(7, m_Date)
    COLUMN_ENTRY(8, m_TotalAmount)
END_COLUMN_MAP()
};

```

Класс COrderID, определяемый в том же заголовочном файле, необходим для получения результатов запроса "SELECT MAX(OrderID) FROM OrderDetails" в методе PlaceOrder():

```

class COrderID
{
public:
    // Элементы данных
    int m_OrderID;

    // Схема соответствия столбцов
    BEGIN_COLUMN_MAP(COrderID)
        COLUMN_ENTRY(1, m_OrderID)
    END_COLUMN_MAP()
};

```

Класс CLogin, определенный в файле CLogin.h, задает соответствие для запроса INSERT, который используется с целью добавления в базу данных информации об участнике торгов:

```

class CLogin
{
public:
    // Элементы данных
    TCHAR m_Name[51];
    TCHAR m_Password[51];
    TCHAR m_UserID[51];

    BEGIN_COLUMN_MAP(CLogin)
        COLUMN_ENTRY(1, m_UserID)
        COLUMN_ENTRY(2, m_Password)
        COLUMN_ENTRY(3, m_Password)
    END_COLUMN_MAP()
};

```

Наконец, класс CUserID определяет соответствие для запроса SELECT, выполняемого при входе пользователя в систему брокера:

```

class CUserID
{
public:
    // Элементы данных
    TCHAR m_UserID[51];

```

```

TCHAR m_Password[51];

BEGIN_COLUMN_MAP(CUserID)
    COLUMN_ENTRY(1, m_UserID)
    COLUMN_ENTRY(1, m_Password)
END_COLUMN_MAP()
);

```

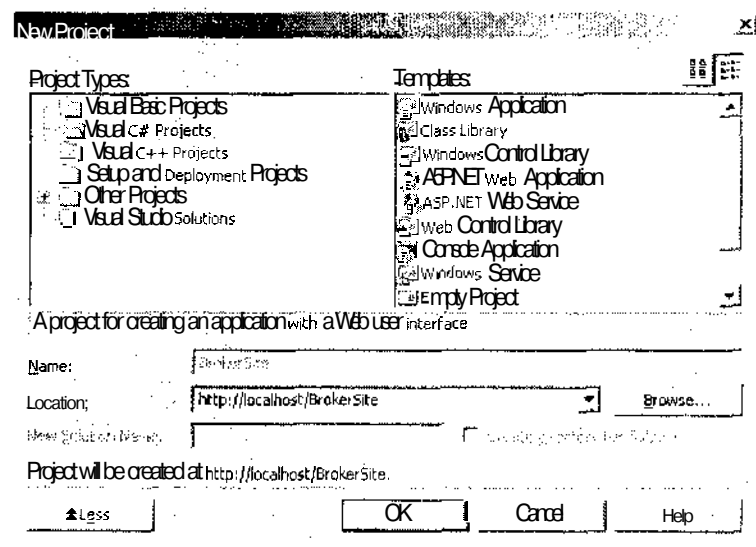
Все три заголовочных файла, представленных выше, должны быть указаны в директивах `#include`, помещенных в начало файла `StockQuote.h`, чтобы эти заголовочные файлы были доступны методам, определенным в `StockQuote.p`. Это подходящий момент для проверки загружаемых файлов. Прежде чем двигаться дальше, необходимо убедиться, что все находится на своих местах.

Компоновка и тестирование Web-службы

Скомпилируем проект для создания библиотек DLL. Протестируем Web-службу, запустив ее в Visual Studio .NET. При этом мы увидим генерируемый код WSDL, который описывает различные методы, предоставляемые Web-службой (аналогичный код в уменьшенном масштабе мы видели при запуске Web-службы ранее в данной главе).

Разработка клиентского приложения

Для тестирования Web-службы воспользуемся еще раз возможностью взаимодействия языков программирования и создадим на C# Web-приложение ASP.NET (ASP.NET Web Application). Выберем эту опцию в диалоге New Project (Новый проект), назовем проект `BrokerSite` и разместим его по адресу: `http://localhost`. После этого еще раз нажмем кнопку OK для создания файлов проекта:



Создание пользовательского интерфейса

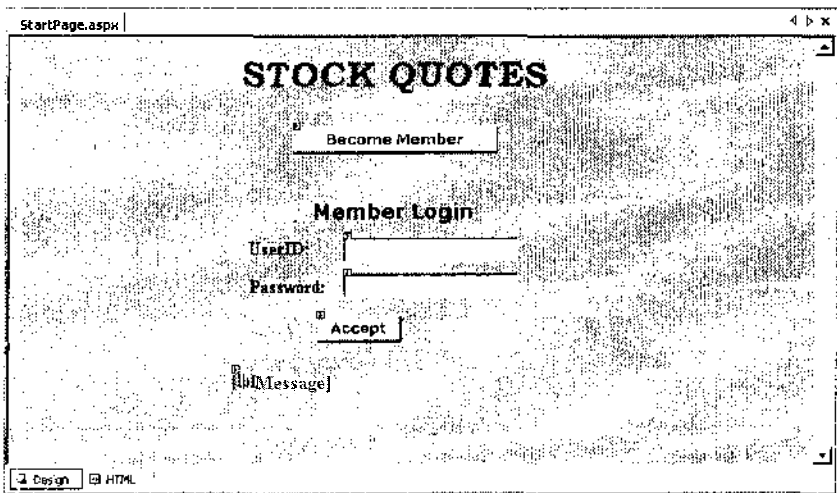
Пользовательский интерфейс нашего клиентского приложения будут поддерживать три файла:

- ❑ StartPage.aspx — предназначен для открытия начальной страницы.
- ❑ Registration.aspx — используется для регистрации участника торгов.
- ❑ ServicePage.aspx — создастся для заполнения бланка заказа и его передачи.

Форма **StartPage.aspx**

Мастер автоматически генерирует в проекте один ASPX-файл с именем Web-Form1.aspx. Начнем с того, что поменяем имя этого файла на StartPage.aspx. Поскольку данный файл будет отображать стартовую страницу приложения, необходимо сконфигурировать ее соответствующим образом. Щелкнем правой кнопкой мыши па имени формы в Solution Explorer и выберем пункт меню Start Page (Начальная страница).

Далее следует с помощью дизайнера изменить файл StartPage.aspx, перетащив на форму Web-компоненты с панели элементов управления. При этом необходимо создать следующий интерфейс:

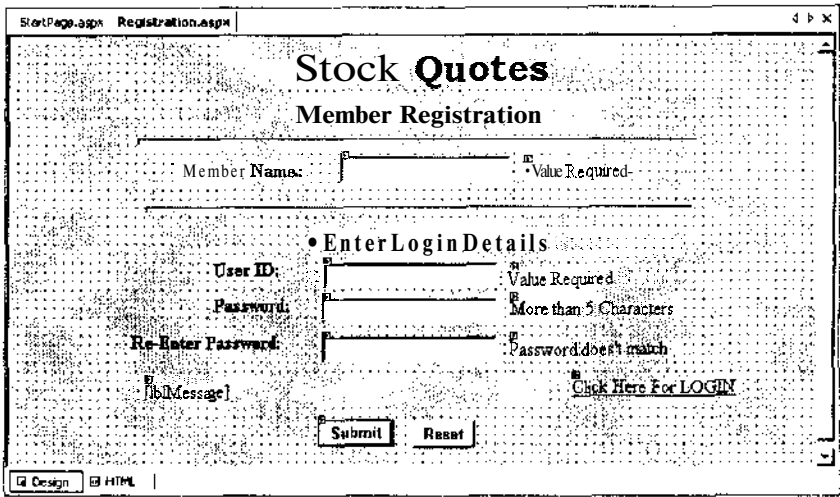


Для этого нужно добавить на форму, представляющую файл StartPage.aspx, такие Web-компоненты:

	Элемент управления	Тип	Свойства
Метка	lblTitle1	HTML	Text: STOCK QUOTES
	lblTitle2	HTML	Text: Member Login
	lblUserID	HTML	Text: User ID
	lblPassword	HTML	Text: Password
	lblMessage	Элемент управления Web	
Кнопка	btnNewMember	Элемент управления Web	Text: Become Member, ID: btnNewMember
	DtnAccept	Элемент управления Web	Text: AccedT, ID: DtnAccept
	txtUserID	Элемент управления Web	ID: txtUserID
	txtPassword	Элемент управления Web	ID: txtPassword
		Элемент управления Web	

Форма **Registration.aspx**

Создадим теперь форму для регистрации участника торгов, которая будет открываться нажатием кнопки **Become Member** (Стать участником). Для создания формы воспользуемся командой меню **Project | Add Web Form** (Проект | Добавить Web-форму). Интегрированная среда разработки откроет диалоговое окно, где можно задать имя формы. Выберем имя **Registration.aspx**. Web-форма появится в окне **Solution Explorer**. Ниже показан пользовательский интерфейс, который должна предоставлять эта форма:



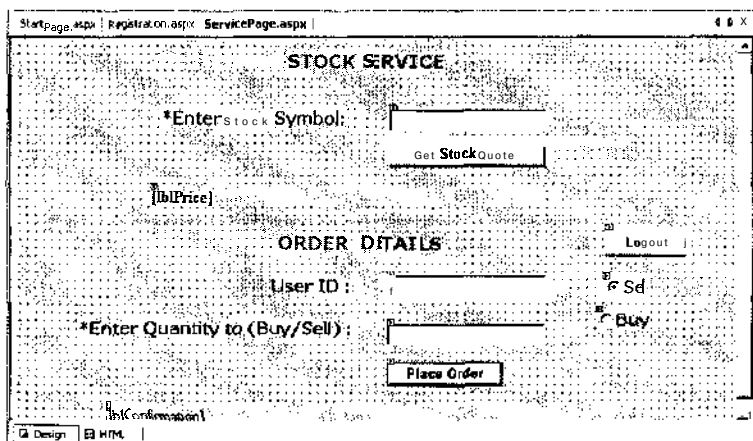
Для получения такого интерфейса необходимо добавить на Web-форму следующие элементы управления:

	Элемент управления	Тип	Свойства
Метка	lblTitle1	HTML	Text: Stock Quotes
	lblTitle2	HTML	Text: Member Registration
	lblMemberName	HTML	Text: Member Name
	lblTitle3	HTML	Text: Enter Login Details
	lblUserID	HTML	Text: User ID
	lblPassword	HTML	Text: Password
	lblRePassword	HTML	Text: Re-enter password
	lblMessage	Элемент управления Web	
Гипер-ссылка		Элемент управления Web	Text: Click Here For LOGIN, ID: HomePage, Visible: false
Кнопка	btnSubmit	Элемент управления Web	Text: Submit, ID: btnSubmit
	btnReset	HTML	Text: Reset, ID: btnReset
Текстовое поле	txtName	Элемент управления Web	ID: txtName
	txtUserID	Элемент управления Web	ID: txtUserID

	Элемент управления	Тип	Свойства
Текстовое поле	txtPassword	Элемент управления Web	ID: txtPassword
	txtRePassword	Элемент управления Web	ID: txtRePassword
Компонент проверки корректности данных (validator)	RequiredFieldValidator	Элемент управления Web	ID: ValidateFName, ErrorMessage: Value Required, ControlToValidate: txtName ID: ValidateUserID, ErrorMessage: Value Required, ControlToValidate: txtUserID
	CompareValidator	Элемент управления Web	ID: ValidateRePwo, ErrorMessage: Password doesn't match, ControlToValidate: txtRePassword, ControlToCompare: txtPassword
	RegularExpressionValidator	Элемент управления Web	ID: ValidatePassword, ErrorMessage: More than 5 Characters, ControlToValidate: txtPassword, Validation Expression: "[\w{5}]{(\w)*}"

Форма ServicePage.aspx

Наконец, создадим третью форму ServicePage.aspx, которая открывается только после успешного входа пользователя в систему брокера. С помощью этой страницы пользователь может просматривать последние цены на акции, указав конкретный выпуск акций. Страница также позволяет участникам торгов размещать заказы на покупку или продажу акций. Форма должна выглядеть так, как показано на снимке экрана:

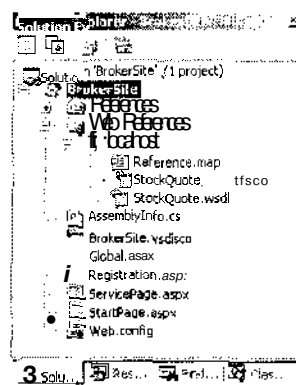


Ниже приведен список элементов управления для этой формы:

	Элемент управления	Тип	Свойства
Метка	lblTitle1	HTML	Text: STOCK SERVICE
	lblStockSymbol	HTML	Text: *Enter Stock Symbol
	lblTitle2	HTML	Text: ORDER DETAILS
	lblUserID	HTML	Text: User ID
	lblQuantity	HTML	Text: -Enter- Quant: Ly to (Buy/Sell)
	lblPrice	Элемент управления Web	
	lblConfirmation	Элемент управления Web	
Кнопка	btnGetStockQuote	Элемент управления Web	Text: Get Stock Quote, ID: btnGetStockQuote
	btnLogout	Элемент управления Web	Text: Logout, ID: btnLogout
	btnPlaceOrder	Элемент управления Web	Text: Place Order, ID: btnPlaceOrder
Текстовое поле	txtSymbol	Элемент управления Web	ID: txtSymbol
	txtUserID	Элемент управления Web	ID: txtUserID, Enable: false
	txtQuantity	Элемент управления Web	ID: txtQuantity
Кнопка-переключатель	rbtnSell	Элемент управления Web	GroupName: rbtnGroup, ID: rbtnSell, Checked: true
	rbtnBuy	Элемент управления Web	GroupName: rbtnGroup, ID: rbtnBuy

Добавление ссылки на DLL Web-службы

В проводнике Solution Explorer щелкнем правой кнопкой мыши папке проекте и выберем команду меню Add Web Reference (Добавить Web-ссылку). Зададим указатель URL для Web-службы (<http://localhost/StockQuote/StockQuote.disco>) и щелкнем на кнопке Add Reference, чтобы добавить в проект ссылку на библиотеку DLL Web-службы. В результате мастер создаст папку `localhost` и запишет туда несколько файлов. Эти файлы показаны на рисунке:



В отличие от файлов `Reference map`, `StockQuote disco` и `StockQuote.wsdl`, файл `StockQuote.cs` (содержащий прокси-класс для Web-службы) невидим и окне `Solution Explorer`. Он похож на заголовочный файл, который мы создавали при написании клиента C++. В рассматриваемом примере клиентская программа также создаст экземпляр прокси-класса для вызова методов удаленного объекта.

Изменение кода формы

Проделав подготовительную работу, теперь можно добавить код в файлы трех форм для реализации нужной функциональности. Как и раньше, начнем со стартовой страницы.

Файл `StartPage.aspx.cs`

Форма `StartPage.aspx` играет роль классификационной страницы, которая позволяет новым пользователям зарегистрироваться и стать участниками торгов. После регистрации они могут входить в систему брокера для получения биржевых котировок и размещения заказов. Чтобы реализовать эту функциональность, напишем дополнительный код в файле Web-формы. Ниже приведен код для кнопки `Become Member` (Стать участником):

```
private void btnNewMember_Click(object sender, System.EventArgs e)
{
    Response.Redirect("Registration.aspx");
}
```

Когда пользователь нажимает кнопку, открывается новая форма в результате вызова метода `Response.Redirect()`, который принимает в строковом формате ссылку на страницу.

Зарегистрированный участник может получить доступ к Web-службе, введя свой пользовательский идентификатор и пароль. Проверка корректности этих данных выполняется в обработчике события `click` кнопки `DtnAccept`:

```
private void btnAccept_Click(object sender, System.EventArgs e)
{
    localhost.StockQuoteService obj = new localhost.StockQuoteService();
    string result = obj.ValidateUser(txtUserID.Text, txtPassword.Text);
    if (result == "Success")
    {
        Session.Add("UserID", txtUserID.Text);
        Response.Redirect("ServicePage.aspx");
    }
    else
    {
        lblMessage.Text = "Invalid UserID";
    }
}
```

Здесь создается объект Web-службы и вызывается метод `ValidateUser()` этого объекта. Метод оценивает данные, вводимые пользователем, и возвращает положительный или отрицательный результат проверки их корректности. Если данные введены правильно, код передаст их в форму `ServicePage` после сохранения идентификатора пользователя в переменную сеанса `UserID`. Данный идентификатор будет отображен на странице `ServicePage`.

Файл `Registration.aspx.cs`

Регистрационная форма предназначена для предоставления сведений о себе возможными новыми участниками, которые должны ввести свое имя, пользовательский идентификатор и пароль. Код проверяет допустимость этих сведений с помощью

компонентов проверки корректности данных, добавленных нами на Web-форму. Для надежности пароль вводится дважды. Все данные являются обязательными. Когда пользователь щелкает кнопку Submit (Зарегистрироваться), введенные величины записываются в базу данных. Ниже приведен код для кнопки Submit:

```
private void btnSubmit_Click(object sender, System.EventArgs e)
{
    localhost.StockQuoteService obj= new localhost.StockQuoteServiceO;
    string result= obj.AddMemberInfo(txtUserID.Text,
                                     txtPassword.Text,
                                     txtName.Text);
    lblMessage.Text= result;
    HomePage.Visible= true;
}
```

Здесь данные-члены записываются в базу данных с помощью метода Web-службы. Нам пришлось лишь создать объект Web-службы и затем вызвать его метод AddMemberInfo(). После добавления записи в таблицу мы сделали видимой гиперссылку, которая указывает на начальную страницу, откуда пользователь входит в систему торгов.

Файл ServicePage.aspx.cs

В этом файле, который определяет поведение рабочей страницы Web-службы (service page), мы вначале напишем обработчик события для кнопки Get Stock Quote (Получить биржевые котировки):

```
private void btnGetStockQuote_Click(object sender, System.EventArgs e)
{
    if(txtStockSymbol.Text=="")
    {
        lblPrice.Text= "VALUE REQUIRED FOR FIELDS WITH (*)";
    }
    else
    {
        // Вызываем метод Web-службы для получения биржевых котировок
        float price= obj.getStockQuote(txtStockSymbol.Text);
        if(price== 0)
        {
            lblPrice.Text= "INVALID STOCK SYMBOL";
        }
        else
        {
            lblPrice.Text= "Stock price is $" + String.Format("{0:F2}",
                                                                price);
        }
    }
}
```

Для получения цены акций пользователь должен корректно указать их выпуск. Данные, введенные пользователем, проверяются в методе Web-службы getStockQuote(). Этот метод возвращает 0, если задается неверный выпуск акций. Здесь также обрабатывается ситуация, когда пользователь щелкает кнопку, не вводя никаких данных.

Чтобы сделать заказ, пользователь вводит количество акций, которое он хочет купить или продать, а затем нажимает кнопку Place Order (Разместить заказ). Запись транзакции сохраняется в базе данных вместе с идентификатором пользователя. Код, выполняющий эти операции, представлен ниже:

```

private void btnPlaceOrder_Click(object sender, System.EventArgs e)
{
    short transaction;

    // Проверка корректности данных
    bool results = ValidateInput();
    if (results)
    {
        if (rbtnBuy.Checked)
        {
            transaction = 1; // Покупка акций
        }
        else
        {
            transaction = 0; // Продажа акций
        }

        int stkQuantity = Convert.ToInt32(txtQuantity.Text);

        // Вызов метода Web-службы для получения биржевых котировок
        float price = obj.getStockQuote(txtStockSymbol.Text);

        // Вызов метода Web-службы для размещения заказа
        string message = obj.placeOrder(txtUserID.Text,
                                         txtStockSymbol.Text,
                                         price,
                                         transaction,
                                         stkQuantity);

        lblConfirmation.Text = message;
        txtQuantity.Text = "";
    }
    else
    {
        lblConfirmation.Text = "Values required for fields with (*)";
    }
}

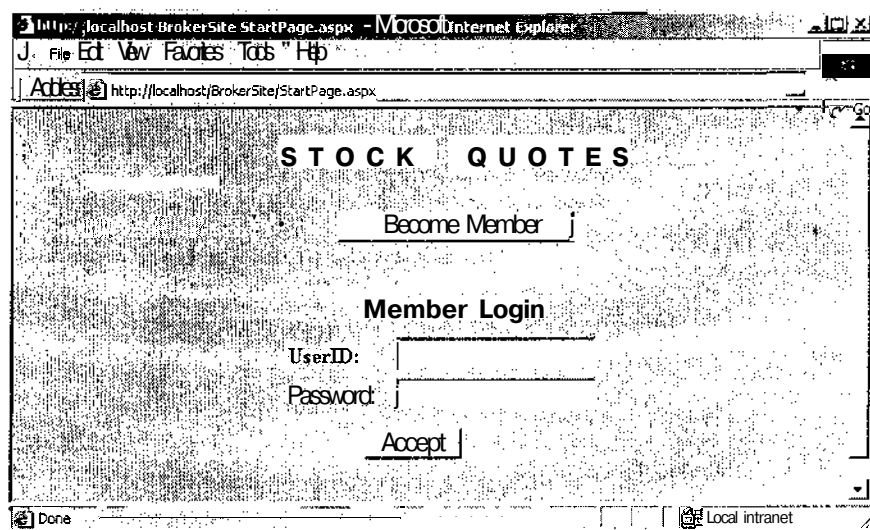
// Метод проверки корректности данных
public bool ValidateInput()
{
    if (txtQuantity.Text != "")
    {
        return (txtQuantity.Text != "");
    }
}

```

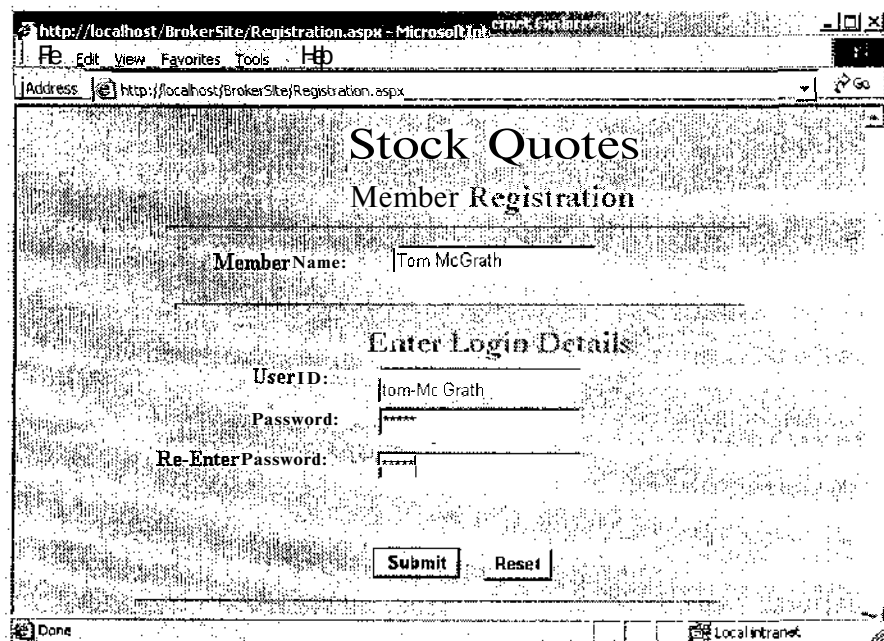
Здесь обработчик событий вызывает Web-метод PlaceOrder(), чтобы записать введенную информацию в базу данных брокера. Полный исходный код этого приложения содержится в загружаемых файлах.

Работа с клиентским приложением

При запуске клиентского приложения браузер должен открыть форму StartPage.aspx. Для этого необходимо ввести указатель URL на файл данной формы: <http://localhost/BrokerSite/StartPage.aspx>. В результате откроется следующее окно:



ДЛЯ ТОГО чтобы стать участником торгов, пользователь должен щелкнуть кнопку Become Member, после чего он получает доступ к регистрационной форме:



Введя всю необходимую информацию, следует нажать кнопку Submit. В результате пользователь получает подтверждение регистрации и в окне появляется ссылка на начальную страницу, откуда участник входит в систему торгов:

The screenshot shows a web browser window titled "http://localhost/BrokerSite/Registration.aspx - Microsoft Internet Explorer". The address bar contains "http://localhost/BrokerSite/Registration.aspx". The page content includes a heading "Stock Quotes" and a sub-heading "Member Registration". Below this is a form with the following fields and labels:

- Member Name:** A text input field containing "Tom McGrath".
- Enter Login Details:** A section header for the login information fields.
- User ID:** A text input field containing "tom-McGrath".
- Password:** A text input field.
- Re-Enter Password:** A text input field.

Below the form, there is a message "THANK YOU! FOR REGISTERING WITH US" and a link "Click Here For LOGIN". At the bottom of the form are two buttons: "Submit" and "Reset". The browser's status bar at the bottom shows "Done" and "Local intranet".

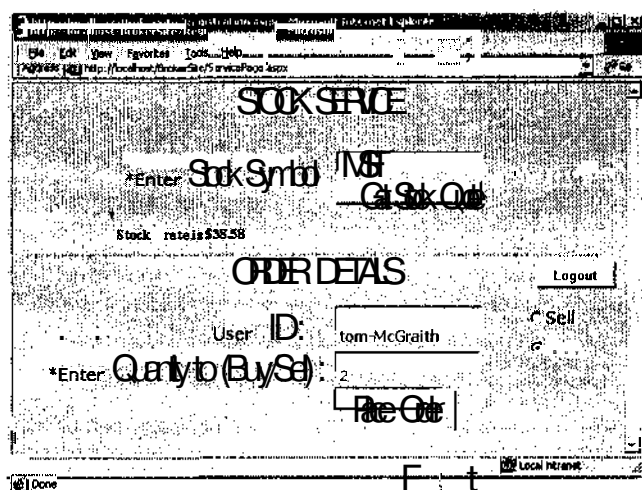
Щелкнув на гиперссылке, пользователь возвращается на начальную страницу:

The screenshot shows a web browser window titled "http://localhost/BrokerSite/StartPage.aspx - Microsoft Internet Explorer". The address bar contains "http://localhost/BrokerSite/StartPage.aspx". The page content includes a heading "STOCK QUOTES" and a link "BecomeMember". Below this is a section titled "Member Login" with the following fields and labels:

- User ID:** A text input field containing "tom-McGrath".
- Password:** A text input field containing "*****".

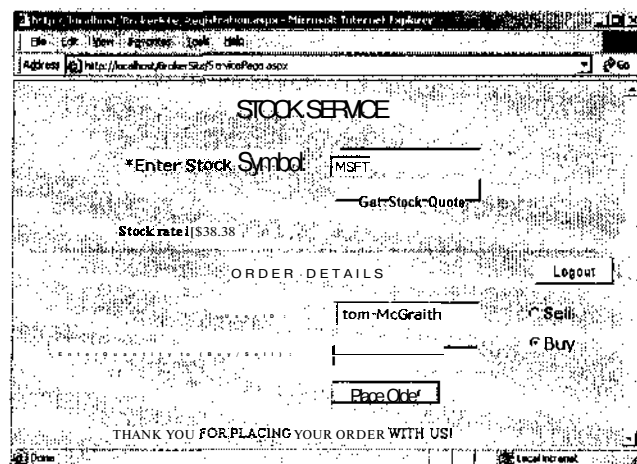
Below the form is a button labeled "Accept". The browser's status bar at the bottom shows "Done" and "Local intranet".

Теперь участник должен указать пользовательский идентификатор и пароль", сконфигурированные им на предыдущем этапе, и нажать кнопку Accept {Ввести данные}. Выполнив аутентификацию пользователя, сервер предоставит доступ к рабочей странице:



Обратимся к этой странице. Введем нужное нам название выпуска акций и щелкнем на кнопке Get Stock Quote (Получить биржевые котировки), чтобы просмотреть последние цены на акции. Для того чтобы разместить у брокера заказ на заключение сделки по этим ценам, введем данные в нижней части экрана и нажмем кнопку Place Order. Идентификатор пользователя здесь берется из переменной сессии, значение которой было введено нами ранее. В результате этих действий заказ с помощью Web-метода будет добавлен в базу данных брокера.

После этого в нижней части окна появится подтверждение о размещении заказа:



В последних примерах мы рассмотрели действительно новую особенность Visual C++. Вначале мы осторожно заглянули в код, генерируемый мастером по умолчанию, а затем, добавив собственный код, выполнили необходимую работу. Здесь напрашивается сравнение с тем, как определяется место, занимаемое Visual C++.NET в Visual Studio .NET. Вначале не совсем ясно, что же все-таки происходит и какую роль играет этот язык в новой технологии. Но по мере продолжения экспериментов открываются все новые возможности языка и мы, наконец, с удивлением спрашиваем себя: неужели можно было в них сомневаться?