

Visual C++ 6

Руководство разработчика

- Введение

Часть 1 Краткий обзор Visual C++

- Глава 1. Компилятор Visual C++, версия 6
- Глава 2. Краткое знакомство со средой Visual C++
- Глава 3. Написание, компиляция и отладка простейшей программы

Часть 2 Процедурное программирование

- Глава 4. Введение в С и C++
- Глава 5. Работа с данными
- Глава 6. Инструкции
- Глава 7. Функции
- Глава 8. Массивы
- Глава 9. Указатели
- Глава 10. Ввод-вывод в языке С
- Глава 11. Основы ввода-вывода в языке C++
- Глава 12. Дополнительные типы данных

Часть 3. Объектно-ориентированное программирование

- Глава 13. Основы ООП
- Глава 14. Классы
- Глава 15. Классы ввода-вывода в языке C++

Часть 4. Основы программирования Windows

- Глава 16. Концепции и средства программирования в Windows
- Глава 17. Процедурные приложения для Windows
- Глава 18. Основы библиотеки MFC
- Глава 19. Создание MFC -приложений

Часть 5. Мастера

- Глава 20. Мастера AppWizard и ClassWizard

- Глава 21. Введение в OLE
- Глава 22. Основы создания элементов управления ActiveX
- Глава 23. COM и ATL

Введение

Самоучитель ставит перед собой три основные задачи: помочь начинающим программистам освоить компилятор Microsoft Visual C++, разобраться в особенностях программирования на C/C++ и познакомить их с основами создания программных продуктов в 32-разрядной среде Windows. Это достаточно объемные задачи даже для издания, содержащего несколько сотен страниц, но мы постарались сделать изложение материала как можно более кратким и ясным.

Общие задачи можно разбить на ряд частных вопросов.

- Речь пойдет в первую очередь о таком мощном средстве программирования, как компилятор Microsoft Visual C++. Данный пакет программ включает в себя собственно компилятор, отладчик и всевозможные вспомогательные утилиты. Наша книга в сочетании с техническим руководством, распространяемым Microsoft, и интерактивной справочной системой, поставляемой на установочных компакт-дисках, поможет вам в освоении базовых компонентов, составляющих пакет Microsoft Visual C++.
- Вы узнаете, как отладить программный код и устранить из него синтаксические и логические ошибки.
- Для успешной работы программист должен четко представлять основные принципы, на которых базируется программирование в той или иной среде. Эта книга раскрывает основные концепции программирования на C/C++, а также в среде Windows, включая использование библиотеки MFC.

Авторы являются сторонниками обучения на практике, поэтому приложили максимум усилий для того, чтобы приведенные в книге примеры были просты для понимания, представляли практический интерес и не содержали ошибок. Вы можете модернизировать предоставленный код и свободно использовать его в своих программах.

Как организован самоучитель

Главы 1—3 познакомят вас с компонентами компилятора Microsoft Visual C++.

В главах 4—12 будут рассмотрены основные концепции программирования на C/C++. В этих главах представлен традиционный, процедурно-ориентированный подход к программированию.

В главах 13—15 рассматривается объектно-ориентированное программирование на C++. В этих главах вы познакомитесь с терминологией, основными определениями и примерами программ, которые помогут вам при создании собственных объектно-ориентированных приложений.

В главах 16 и 17 раскрываются базовые принципы программирования в среде Windows(95, 98 и NT), а также показывается, как с помощью компилятора Microsoft Visual C++ создавать приложения, включающие различные элементы графического интерфейса, в частности указатели мыши, значки, меню и диалоговые окна.

Главы 18 и 19 посвящены программированию с применением библиотеки MFC. Благодаря использованию готовых классов C++ вы сможете не только существенно уменьшить код программы, но и сократить время, затрачиваемое на разработку приложения.

В главах 20 и 21 мы продолжим разговор об MFC и познакомим вас с несколькими служебными мастерами, предназначенными для автоматизации процесса создания программного кода. Вы также узнаете о наиболее важных концепциях технологии OLE, научитесь создавать собственные OLE-приложения.

Изучение MFC и базовых мастеров будет продолжено в главе 22, где описаны основные принципы разработки элементов управления ActiveX.

Последняя, 23 глава содержит объяснение принципов создания COM-объектов с помощью библиотеки ATL и специальных мастеров.

Глава 1. Компилятор Visual C++, версия 6

- Стандартный вариант
- Профессиональный вариант
- Корпоративный вариант
- Инструменты разработчика
 - Интегрированный отладчик
 - Встроенные редакторы ресурсов
 - Дополнительные утилиты
- Возможности компилятора
 - Средства автоматизации и макросы
 - ClassView
 - Настраиваемые панели инструментов и меню
 - Рабочие пространства и файлы проектов
 - Предварительно скомпилированные файлы заголовков
 - MFC
 - Макроподстановка функций
- Опции компиляции
 - General
 - Debug
 - C/C++
 - Link
 - Resources
 - MIDL
 - Browse Info
 - Custom Build

Новая версия VisualC++ позволит вам создавать любые приложения для Windows95, 98 и NT с использованием новейших программных технологий и методик. Пакет программ MicrosoftVisualC++ версии 6 поставляется в трех различных вариантах: стандартном, профессиональном и корпоративном.

Стандартный вариант

Стандартный вариант VisualC++ (ранее он назывался учебным) содержит почти **все** те же средства, что и профессиональный, но в отличие от последнего в нем отсутствуют модуль Profiler, несколько мастеров, возможности по оптимизации кода и статической компоновке библиотеки MFC, некоторые менее важные функции. Этот вариант в наибольшей мере подходит для студентов и иных категорий индивидуальных пользователей, чему, в частности, способствует и сравнительно низкая его **цена**. Лицензионное соглашение стандартного

варианта программы в новой версии, в отличие от старой, разрешает ее использование для создания коммерческих программных продуктов.

Профессиональный вариант

В этом варианте программы можно создавать полнофункциональные графические и консольные приложения для всех платформ Win32, включая Windows95, 98 и NT.

Перечислим новые возможности профессионального варианта:

- поддержка автоматического дополнения выражений (технология IntelHSense);
- шаблоны OLEDB;
- средства визуального проектирования приложений, работающих с базами данных
- (без возможности модификации данных).

Корпоративный вариант

С помощью корпоративного варианта VisualC++ вы можете создавать приложения типа клиент/сервер для работы в Internet или в корпоративной среде (intranet). Приобретая корпоративный вариант программы, вы получаете в свое распоряжение не только все возможности профессионального варианта, но и ряд дополнительных средств, включая:

- Microsoft Transaction Server;
- средства визуального проектирования приложений, работающих с базами данных;
- модуль SQL Editor;
- модуль SQL Debugger;
- классы библиотеки MFC для доступа к базам данных;
- ADO Data-Bound Dialog Wizard;
- Поддержка технологии Remote Automation;
- Visual SourceSafe.

Компилятор Microsoft VisualC++ содержит также средства, позволяющие в среде Windows разрабатывать приложения для других платформ, в том числе для AppleMacintosh. Кроме того, программа снабжена редакторами растровых изображений™, значков, указателей мыши, меню и диалоговых окон, позволяющими работать с перечисленными ресурсами непосредственно в интегрированной среде. Коснувшись темы интеграции, следует упомянуть о мастере ClassWizard, помогающем в кратчайшие сроки создавать приложения OLE с использованием библиотеки MFC.

Инструменты разработчика

Новая версия компилятора Microsoft VisualC++ содержит множество интегрированных средств визуального программирования. Ниже перечислены утилиты, которые вы можете использовать непосредственно из VisualC++.

Интегрированный отладчик

Разработчики компании Microsoft встроили первоначальный отладчик CodeView непосредственно в среду VisualC++. Команды отладки вызываются из меню **Debug**. Встроенный отладчик позволяет пошагово выполнять программу, просматривать и изменять значения переменных и многое другое.

Встроенные редакторы ресурсов

Редакторы ресурсов позволяют создавать и модифицировать ресурсы Windows, такие как растровые изображения, указатели мыши, значки, меню, диалоговые окна и т.д.

Редактор диалоговых окон

Редактор диалоговых окон — это достаточно удобное средство, позволяющее легко и быстро создавать сложные диалоговые окна. С помощью этого редактора в разрабатываемое диалоговое окно можно включить любые элементы управления (например, надписи, кнопки, флажки, списки и т.д.). При этом вы можете изменять как внешний вид элементов управления, так и их свойства (наряду со свойствами самого диалогового окна).

Редактор изображений

Этот редактор позволяет быстро создавать и редактировать собственные растровые изображения, значки и указатели мыши. Пользовательские ресурсы данного типа сохраняются в файле с расширением RC и включаются в файлы сценариев ресурсов. Более подробно об использовании ресурсов в приложениях рассказывается в главах 16-19.

Редактор двоичных кодов

Данный редактор позволяет вносить изменения непосредственно в двоичный код ресурса. Редактор двоичных кодов следует использовать только для просмотра ресурсов или внесения мелких изменений в те из них, тип которых не поддерживается в VisualC++.

Редактор строк

Таблица строк представляет собой ресурс, содержащий список идентификаторов и значений всех строковых надписей, используемых в приложении. К примеру, в этой таблице могут храниться сообщения, отображаемые в строке состояния. Каждое приложение содержит единственную таблицу строк. Наличие единой таблицы позволяет легко менять язык интерфейса программы — для этого достаточно перевести на другой язык строки таблицы, не затрагивая код программы.

Дополнительные утилиты

ActiveX Control Test Container

С помощью этой утилиты, разработанной специалистами **Microsoft**, вы можете быстро протестировать созданные вами элементы управления. При этом можно изменять их свойства и характеристики.

APITextViewer

Данная утилита позволяет просматривать объявления констант, функций и типов данных Win32 API, а также копировать эти объявления в приложения VisualBasic или в буфер обмена.

AVIEditor

Эта утилита позволяет просматривать, редактировать и объединять AVI-файлы.

DataObjectViewer

Утилита DataObjectViewer отображает список форматов данных, предлагаемых объектами ActiveX и OLE, которые находятся в буфере обмена или участвуют в операции перетаскивания (drag-and-drop).

DDESpy

Эта утилита предназначена для отслеживания всех DDE-сообщений.

DocFileViewer

Эта утилита отображает содержимое составных OLE-документов.

ErrorLookup

Эта утилита позволяет просматривать и анализировать всевозможные сообщения об ошибках.

HeapWalkUtility

Эта утилита выводит список блоков памяти, размещенных в указанной динамической области (куче).

HelpWorkshop

Эта утилита позволяет создавать и редактировать файлы справки.

OLE Client/Server, Tools и View

Утилита OLEViewer отображает информацию об объектах ActiveX и OLE, установленных на вашем компьютере. Эта утилита также позволяет редактировать реестр и просматривать библиотеки типов. Утилиты OLE Client и OLE Server предназначены для тестирования OLE-клиентов и серверов.

TheProcessViewer

Эта утилита позволяет следить за состоянием выполняющихся процессов и потоков.

ROTViewer

Эта утилита отображает информацию об объектах ActiveX и OLE, в данный момент загруженных в память.

Spy++

Эта утилита выводит сведения о выполняющихся процессах, потоках, существующих окнах и оконных сообщениях.

StressUtility

Эта утилита позволяет захватывать системные ресурсы и используется для тестирования системы в ситуациях, связанных с недостатком системных ресурсов. В число захватываемых ресурсов входят глобальная и пользовательская динамические области (кучи), динамическая область GDI, свободные области дисков и дескрипторы файлов. Утилита Stress может выделять фиксированное количество ресурсов, а также производить выделение в ответ на получение различных сообщений. Кроме того, утилита способна вести журнал событий, что помогает обнаруживать и воспроизводить аварийные ситуации в работе программы.

MFC Tracer

Эта утилита позволяет устанавливать флаги трассировки в файле AFX.INI. С помощью данных флагов можно выбрать типы сообщений, которые будут посылаться приложением в окно отладки. Таким образом, утилита Tracer является средством отладки.

UUIDGenerator

Эта утилита предназначена для генерации универсального уникального идентификатора (UUID), который позволяет клиентским и серверным приложениям распознавать друг друга.

WinDiff

Эта утилита дает возможность сравнивать содержимое файлов и папок.

Zooming

Эту утилиту можно использовать для захвата и просмотра в увеличенном виде выбранной области на рабочем столе.

Возможности компилятора

Компилятор VisualC++ содержит много новых инструментальных средств и улучшенных возможностей. В следующих параграфах дается их краткий обзор.

Средства автоматизации и макросы

С помощью сценариев VisualBasic вы можете автоматизировать выполнение рутинных и повторяющихся задач. VisualC++ позволяет записывать в макрокомандах самые разные операции со своими компонентами, включая открытие, редактирование и закрытие документов, изменение размеров окон. Можно также создавать надстроечные модули, интегрируя их в среду с использованием объектной модели VisualC++.

ClassView

Вкладка ClassView теперь позволяет работать с классами Java так же, как с классами C++. Вы можете просматривать и редактировать интерфейсы COM-объектов, созданных на базе MFC или ALT, а также разбивать классы по папкам удобным для вас образом.

Настраиваемые панели инструментов и меню

В новой версии VisualC++ стало легче настраивать панели инструментов и меню в соответствии с вашими предпочтениями. В частности, вы можете выполнять следующие действия:

- добавлять меню в панель инструментов;
- добавлять и удалять команды меню и кнопки панели инструментов;
- заменять кнопки панели инструментов соответствующими командами меню;
- создавать копии команд меню или кнопок панелей инструментов на разных панелях, с тем чтобы облегчить доступ к ним в разных ситуациях;
- создавать новые панели инструментов и меню;
- настраивать внешний вид существующих панелей инструментов и меню;
- назначать команды меню новым кнопкам панелей инструментов.

Рабочие пространства и файлы проектов

Файлы рабочего пространства теперь имеют расширение DSW(раньше использовалось расширение MDP). Создаваемые проекты записываются в файлы двух типов: внутренние (DSP) и внешние (MAK). Файлы с расширением DSP создаются при выборе нового проекта или при открытии файла проекта, созданного в ранней версии программы. (Обратите внимание, что DSP-файлы не совместимы с утилитой NMAKE.) Чтобы сохранить проект во внешнем файле с расширением MAK, используйте команду **Export Makefile** из меню **Project**.

Проекты теперь могут содержать активные документы, например электронные таблицы или текстовые документы Word. Вы можете редактировать их, даже не покидая VisualStudio.

Когда создается новое рабочее пространство, VisualC++ создает файл имя_рабочего_пространства.DSW. Эти файлы больше не содержат данных, специфичных для вашего компьютера.

Предварительно скомпилированные файлы заголовков

VisualC++ помещает описания типов данных, прототипы функций, внешние ссылки и объявления функций-членов в специальные файлы, называемые файлами заголовков. Эти файлы содержат важные определения, необходимые во многих местах программы. Части файлов заголовков обычно повторно компилируются при компиляции каждого из включающих их модулей. К сожалению, повторная компиляция значительно замедляет работу компилятора.

VisualC++ позволяет существенно ускорить этот процесс за счет возможности предварительной компиляции файлов заголовков. Хотя идея не нова, для ее реализации специалисты Microsoft использовали принципиально новый подход. Предварительной компиляции может подвергнуться только "стабильная" часть файла; оставшаяся же часть, которая впоследствии может модифицироваться, будет компилироваться вместе с приложением.

Это средство удобно применять, например, в том случае, когда в процессе разработки приложения приходится часто изменять программный код, сохраняя описание классов. Предварительная компиляция файлов заголовков приведет к значительному ускорению работы с программой и в тех случаях, когда заголовки заключают в себе больше программного кода, чем основной модуль.

Компилятор VisualC++ предполагает, что текущее состояние рабочей среды идентично тому, которое было при компиляции заголовков. В случае обнаружения каких-либо конфликтов будет выдано предупреждающее сообщение. Такие ситуации могут возникать при изменении модели использования памяти, значений предопределенных констант или опций отладки/компиляции.

В отличие от многих других компиляторов C++, VisualC++ не ограничивается предварительной компиляцией только файлов заголовков. Благодаря возможности проводить компиляцию до заданной точки программы вы можете предварительно скомпилировать даже какую-нибудь часть основного модуля. В целом, процедура предварительной компиляции используется для тех частей программного кода, которые вы считаете стабильными; таким образом значительно сокращается время компиляции частей программы, изменяемых в процессе работы над ней.

MFC

Приложения Windows просты в использовании, но создавать их довольно сложно. Программистам приходится изучать сотни различных API-функций.

Чтобы облегчить их труд, специалисты Microsoft разработали библиотеку MicrosoftFoundationClasses— MFC . Используя готовые классы C++, можно гораздо быстрее и проще решать многие задачи. Библиотека MFC существенно облегчает программирование в среде Windows. Те, кто обладает достаточным опытом программирования на C++, могут дорабатывать классы или создавать новые, производные от существующих.

Классы библиотеки MFC используются как для управления объектами Windows, так и для решения определенных общесистемных задач. Например, в библиотеке имеются классы для управления файлами, строками, временем, обработкой исключений и другие.

По сути, в MFC представлены практически все функции WindowsAPI. В библиотеке имеются средства обработки сообщений, диагностики ошибок и другие средства, обычные для приложений Windows. MFC обладает следующими преимуществами.

- Представленный набор функций и классов отличается логичностью и полнотой. Библиотека MFC открывает доступ ко всем часто используемым функциям WindowsAPI, включая функции управления окнами приложений, сообщениями, элементами управления, меню, диалоговыми окнами, объектами GDI (GraphicsDeviceInterface— интерфейс графических устройств), такими как шрифты, кисти, перья и растровые изображения, функции работы с документами и многое другое.
- Функции MFC легко изучать. Специалисты Microsoft приложили все усилия для того, чтобы имена функций MFC и связанных с ними параметров были максимально близки к их эквивалентам из WindowsAPI. Благодаря этому программисты легко смогут разобраться в их назначении.
- Программный код библиотеки достаточно эффективен. Скорость выполнения приложений, основанных на MFC , будет примерно такой же, как и скорость выполнения приложений, написанных на C с использованием стандартных функций WindowsAPI, а дополнительные затраты оперативной памяти будут весьма незначительными.
- MFC содержит средства автоматического управления сообщениями. Библиотека MFC устраняет необходимость в организации цикла обработки сообщений — распространенного источника ошибок в Windows-приложениях. В MFC предусмотрен

автоматический контроль за появлением каждого сообщения. Вместо использования стандартного блока switch/case все сообщения Windows связываются с функциями-членами, выполняющими соответствующую обработку.

- MFC позволяет организовать автоматический контроль за выполнением функций. Эта возможность реализуется за счет того, что вы можете записывать в отдельный файл информацию о различных объектах и контролировать значения переменных-членов объекта в удобном для понимания формате.
- MFC имеет четкий механизм обработки исключительных ситуаций. Библиотека MFC была разработана таким образом, чтобы держать под контролем появление таких ситуаций. Это позволяет объектам MFC восстанавливать работу после появления ошибок типа "outofmemory" (нехватка памяти), неправильного выбора команд меню или проблем с загрузкой файлов либо ресурсов.
- MFC обеспечивает динамическое определение типов объектов. Это чрезвычайно мощное программное средство, позволяющее отложить проверку типа динамически созданного объекта до момента выполнения программы. Благодаря этому вы можете свободно манипулировать объектами, не заботясь о предварительном описании типа данных. Поскольку информация о типе объекта возвращается во время выполнения программы, программист освобождается от целого этапа работы, связанного с типизацией объектов.
- MFC может использоваться совместно с подпрограммами, написанными на языке C. Важной особенностью библиотеки MFC является то, что она может "сосуществовать" с приложениями, основанными на WindowsAPI. В одной и той же программе программист может использовать классы MFC и вызывать функции WindowsAPI. Такая прозрачность среды достигается за счет согласованности программных обозначений в обеих архитектурах. Другими словами, файлы заголовков, типы и глобальные константы MFC не конфликтуют с именами из WindowsAPI. Еще одним ключевым моментом, обеспечивающим такое взаимодействие, является согласованность механизмов управления памятью.
- MFC может быть использована для создания программ, работающих в среде MS-DOS. Библиотека MFC была создана специально для разработки приложений в среде Windows. В то же время многие классы предоставляют объекты, часто используемые для ввода/вывода файлов и манипулирования строковыми данными. Такие классы общего назначения могут применяться в приложениях как Windows, так и MS-DOS.

Макроподстановка функций

Компилятор MicrosoftVisualC++ поддерживает возможность макроподстановки функций. Это означает, что вызов любой функции с любым набором инструкций может быть заменен непосредственной подстановкой тела функции. Многие компиляторы C++ разрешают производить макроподстановку только для функций, содержащих определенные операторы и выражения. Например, иногда оказывается невозможной макроподстановка функций, содержащих операторы switch, while и for. VisualC++ не накладывает ограничений на содержимое функций. Чтобы задать параметры макроподстановки, выберите в меню **Project** команду **Settings**, затем активизируйте вкладку C/C++ и, наконец, выберите элемент **Optimizations** из списка **Category**.

Опции компиляции

Компилятор MicrosoftVisualC++ предоставляет огромные возможности в плане оптимизации приложений, в результате чего вы можете получить выигрыш как в отношении размера программы, так и в отношении скорости ее выполнения, независимо от того, что представляет собой ваше приложение. Перечисленные ниже опции компиляции позволяют оптимизировать программный код, сокращая его размер, время выполнения и время компиляции. Чтобы получить к этим опциям доступ, нужно в меню **Project** выбрать команду **Settings**.

General

На вкладке **General** можно включить или отключить возможность использования библиотеки MFC (список **MicrosoftFoundationClasses**). Здесь также можно указать папки, в которые компилятор будет помещать промежуточные (поле **Intermediatefiles**) и выходные (поле **Outputfiles**) файлы.

Debug

На вкладке **Debug** можно указать местонахождение исполняемого файла и рабочей папки, задать аргументы командной строки, а также путь и имя удаленного исполняемого файла на сетевом диске. Кроме того, в списке **Category** можно выбрать элемент **Additional DLLs**, предназначенный для задания дополнительных библиотек динамической компоновки (DLL).

C/C++

Вкладка C/C++ содержит следующие категории опций: **General**, **C++ Language**, **Code Generation**, **Customize**, **Listing Files**, **Optimizations**, **Precompiled Headers** и **Preprocessor**. В поле **Project Options** отображается командная строка проекта.

General

Опции категории **General** позволяют установить уровень контроля за ошибками (список **Warning level**), указать, какую отладочную информацию следует включать (список **Debug info**), выбрать тип оптимизации при компиляции (список **Optimizations**) и задать директивы препроцессора (поле **Preprocessor definitions**).

C++ Language

Опции категории C++ **Language** позволяют выбрать способ представления указателей на члены классов (группа **Pointer-to-member representation**), включить обработку исключительных ситуаций (**Enable exception handling**), разрешить проверку типов объектов на этапе выполнения (**Enable Run-time Type Information**) и запретить замещение конструкторов при вызове виртуальных функций (**Disable construction displacements**).

Code Generation

Опции категории **Code Generation** позволяют задать тип процессора, на который должен ориентироваться компилятор (список **Processor**), выбрать тип соглашения о вызовах функций (список **Calling convention**), указать тип компоновки динамических библиотек (список **User runtime library**) и установить порядок выравнивания полей структурированных переменных (список **Struct member alignment**).

Customize

В категории **Customize** можно задать следующие опции:

- **Disable language extensions** (компиляция производится в соответствии с правилами ANSI C, а не Microsoft C);
- **Enable function-level linking** (функции при компиляции обрабатываются особым образом, что позволяет компоновщику упорядочивать их и исключать неиспользуемые);
- **Eliminate duplicate strings** (в таблицу строк модуля не включаются повторяющиеся строки);
- **Enable minimal rebuild** (позволяет компилятору обнаруживать изменения в объявлениях классов C++ и выявлять необходимость повторной компиляции исходных файлов);
- **Enable incremental compilation** (дает возможность компилировать только те функции, код которых изменился с момента последней компиляции);
- **Suppress startup banner and information messages** (в процессе компиляции запрещается вывод сообщения с информацией об авторских правах и номере версии компилятора).

Listing Files

Опции категории **Listing Files** позволяют задавать сведения, необходимые для создания SBR-файла (группа **Generate browse info**), который используется при построении специальной базы

данных с информацией о всех классах, функциях, переменных и константах программы. Кроме того, в этой категории можно указать, следует ли создавать файл с ассемблерным кодом программы, какого он должен быть типа и где располагаться (список **Listingfiletype** и поле **Listingfilename**).

Optimizations

Опции категории **Optimizations** позволяют устанавливать различные параметры оптимизации программного кода (список **Optimizations**). Также можно указать, каким образом следует выполнять макроподстановку функций (список **Inlinefunctionexpansion**).

PrecompiledHeaders

Опции категории **PrecompiledHeaders** определяют, следует ли использовать файлы предварительно скомпилированных заголовков (файлы с расширением PCH). Наличие таких файлов ускоряет процесс компиляции и компоновки. После компиляции всего приложения эти файлы следует удалить из папки проекта, поскольку они занимают очень много места.

Preprocessor

Опции категории **Preprocessor** позволяют задавать параметры работы препроцессора. Здесь же можно указать дополнительные папки для включаемых файлов заголовков (поле **Additional #include directories**), а также установить опцию **Ignore standard include paths**, которая служит указанием игнорировать папки, перечисленные в переменных среды PATH или INCLUDE.

Link

Вкладка **Link** содержит опции пяти категорий: **General**, **Customize**, **Debug**, **Input** и **Output**.

General

В категории **General** в поле **Outputfilename** можно задать имя и расширение выходного файла. Как правило, для файла проекта используется расширение EXE. В поле **Object/librarymodules** указываются объектные и библиотечные файлы, компонуемые вместе с проектом. Также могут быть установлены следующие опции:

- **Generateddebuginfo**(в исполняемый файл включается отладочная информация);
- **Linkincrementally**(частичная компоновка; эта опция доступна, если в категории **Customize** установлен флажок **Useprogramdatabase**);
- **Enableprofiling**(в исполняемый файл включается информация для профилировщика);
- **Ignorealldefaultlibraries**(удаляются все стандартные библиотеки из списка библиотек, который просматривается компоновщиком при разрешении внешних ссылок);
- **Generatemapfile**(создается MAP-файл проекта).
- **Customize**

В категории **Customize** можно установить такие опции:

- **Linkincrementally**(аналогична одноименной опции из категории **General**);
- **Useprogramdatabase**(в служебную базу данных программы помещается отладочная информация);
- **Outputfilename**(задает имя выходного файла);
- **Printingprogressmessages**(в процессе компиляций выводятся сообщения о ходе компоновки);
- **Suppressstartupbanner**(аналогична подобной опции категории **Customize** вкладки C/C++).

Debug

Опции категории **Debug** позволяют указать, следует ли генерировать MAP-файл проекта, а также задают различные параметры отладки.

Input

Посредством опций категории **Input** приводится различная информация об объектных и библиотечных файлах, компонуемых вместе с проектом.

Output

Опции категории **Output** позволяют задать базовый адрес программы (**Baseaddress**), точку входа (**Entry-pointsymbol**), объем виртуальной и физической памяти, выделяемой для стека (группа **Stackallocations**), и номер версии проекта (группа **Versioninformation**).

Resources

Вкладка **Resources** позволяет указать имя файла ресурсов (обычно это файл с расширением RES) и задать некоторые дополнительные параметры, такие как язык представления ресурсов, папки включаемых файлов и макросы препроцессора.

MIDL

Вкладка **MIDL** предназначена для задания различных параметров генерации библиотеки типов.

BrowseInfo

На вкладке **BrowseInfo** можно указать имя файла базы данных, содержащей описания классов, функций, констант и переменных программы.

CustomBuild

Вкладка **CustomBuild** предназначена для задания дополнительных команд компиляции, которые будут выполняться над выходным файлом.

Глава 2. Краткое знакомство со средой Visual C++

- Запуск Visual C++
- Доступ к контекстной справке
- Вызов команд меню
- Перемещаемые панели инструментов
- Меню File
 - New
 - Open
 - Close
 - Save
 - Save As
 - Save All
 - Page Setup
 - Print
 - Recent Files и Recent Workspaces
 - Exit
- Меню Edit
 - Undo
 - Redo
 - Cut
 - Copy
 - Paste
 - Delete
 - Select All
 - Find
 - Find in Files
 - Replace
 - Go To
 - Bookmarks
 - Breakpoints
 - List Members
 - Type Info
 - Parameter Info
 - Complete Word
- Меню View
 - ClassWizard
 - Resource Symbols и Resource Includes
 - Full Screen
 - Workspace
 - Output
 - Debug Windows
 - Refresh Properties
- Меню Insert
 - New Class
 - Resource
 - Resource Copy
 - File As Text
 - New ATL Object
- Меню Project
 - Set Active Project
 - Add to Project...
 - Dependencies
 - Settings
 - Export Makefile
 - Insert Project Into Workspace
- Меню Build
 - Compile
 - Build
 - Rebuild All
 - Batch Build
 - Clean
 - Start Debug

- Debugger Remote Connection.
- Execute
- Set Active Configuration
- Configurations
- Profile
- Меню Tools
 - Source Browser
 - Close Source Browser File
 - Error Lookup
 - ActiveX Control Test Container
 - OLE/COM Object Viewer
 - Spy++
 - MFC Tracer
 - Visual Component Manager
 - Register Control
 - Customize
 - Options
 - Macro / Record / Play
- Меню Window
 - New Window
 - Split
 - Docking View
 - Close
 - Close All
 - Next
 - Previous
 - Cascade
 - Tile Horizontally
 - Tile Vertically
 - Список открытых окон
- Меню Help
 - Contents / Search / Index
 - Use Extension Help
 - Keyboard Map
 - Tip of the Day
 - Technical Support
 - Microsoft on the Web
 - About Visual C++

Краткое знакомство со средой VisualC++

Microsoft VisualC++ представляет собой интегрированную среду разработки, в которой вы можете легко создавать, открывать, просматривать, редактировать, сохранять, компилировать и отлаживать все свои приложения, написанные на С или C++. Преимуществом этой среды является относительная простота и легкость в изучении.

В настоящей главе мы рассмотрим всевозможные команды и опции меню, имеющиеся в распоряжении пользователя, а в главе 3 перейдем непосредственно к работе в среде VisualC++.

Запуск Visual C++

Запуск оболочки VisualC++ не составит для вас труда. На рис. 2.1 представлено начальное окно с произвольно выбираемым советом дня, которое открывается при этом.

Доступ к контекстной справке

Доступ к справочной системе VisualC++ облегчен благодаря тому, что вся информация предоставляется в интерактивном режиме. Чтобы получить справку, достаточно навести указатель на интересующий вас инструмент и нажать клавишу [F1].



Рис 2.1.

Следует отметить, что использование контекстной справки не ограничивается элементами интерфейса. Если вы наведете указатель на элемент программного кода C/C++ и нажмете [F1], то получите справку о синтаксисе выбранной вами конструкции.

Вызов команд меню

Прежде чем перейти к описанию отдельных команд и опций, остановимся на некоторых моментах, общих для всех меню. Вспомним, например, о том, что существует два способа выбора команд из меню. Более распространенный из них состоит в том, что вы устанавливаете указатель мыши и щелкаете на нужных командах меню левой кнопкой мыши. Второй способ заключается в использовании клавиш быстрого вызова, которые выделяются подчеркиванием в названиях команд. Так, меню **File** можно раскрыть, нажав одновременно [Alt+F].

Существует еще один способ вызова отдельных команд в любой момент времени, а именно с помощью предварительно заданных "горячих" клавиш. Если для команды определено сочетание клавиш, то это сочетание будет указано в меню справа от соответствующего пункта. Например, в меню **File** есть команда New..., которую можно вызвать, просто нажав [Ctrl+N].

Команда меню, показанная серым цветом, является в данный момент недоступной — вероятно, отсутствуют некоторые условия, необходимые для ее выполнения. Например, команда Slave из меню File будет недоступной, если в редактор ничего не загружено. Программа "понимает", что в данный момент сохранять просто нечего, и напоминает вам об этом, отключив команду сохранения.

Если за названием команды меню следует троеточие, значит, после выбора данной команды будет открыто диалоговое окно. Например, после выбора команды Open... в меню File открывается диалоговое окно Open.

Наконец, многие команды меню представлены также кнопками на панелях инструментов и могут быть вызваны простым щелчком мыши. Панели инструментов обычно размещаются в окне программы непосредственно под строкой меню.

Перемещаемые панели инструментов

Любые панели инструментов VisualC++ можно сделать закрепленными или плавающими. Закрепленные панели инструментов фиксируются вдоль одного из четырех краев окна программы. Изменить размер такой панели вы не можете.

Плавающая панель инструментов имеет собственную строку заголовка и может находиться в любом месте. Плавающие панели всегда располагаются поверх других компонентов окна программы. Вы можете свободно изменять как размер, так и местоположение плавающих панелей инструментов.

- Чтобы превратить закрепленную панель инструментов в плавающую, вам необходимо выполнить такие действия:
- щелкнуть левой кнопкой мыши на свободном пространстве панели инструментов;
- удерживая кнопку мыши нажатой, перетащить панель инструментов в требуемое место.

А чтобы закрепить плавающую панель - такие:

- щелкнуть левой кнопкой мыши на строке заголовка или свободном пространстве панели инструментов;
- удерживая кнопку мыши нажатой, перетащить панель инструментов к одному из четырех краев окна программы.

При необходимости разместить плавающую панель инструментов поверх закрепленной вы должны:

- щелкнуть левой кнопкой мыши на строке заголовка или свободном пространстве панели инструментов;
- удерживая кнопку мыши, нажать клавишу [Ctrl] и перетащить панель инструментов в нужное место.

Меню File

В VisualC++ в меню File собран стандартный для многих приложений Windows набор команд, предназначенных для манипулирования файлами (рис. 2.2).

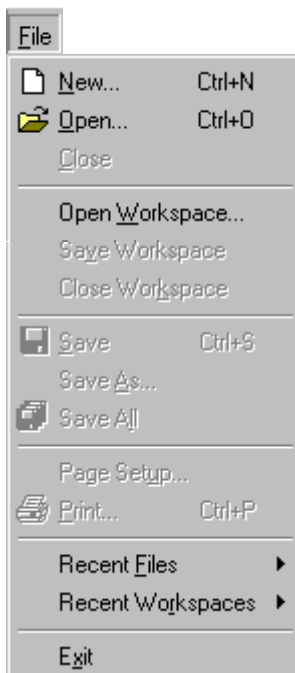


Рис. 2.2.

New...

По команде New... открывается окно для выбора типа создаваемого файла, проекта или рабочего пространства. Именно с этой команды обычно начинается работа над новым приложением. VisualC++ автоматически присваивает название и номер каждому создаваемому файлу (но не проекту), если только вы не сделали этого самостоятельно. Нумерация файлов начинается с 1. Таким образом, первый файл всегда будет иметь имя xxx1, второй — xxx2 и т.д.

Здесь xxx обозначает стандартное имя, меняющееся в зависимости от типа создаваемого файла (программный файл, файл заголовков, значок, указатель мыши и т.п.).

Если вы создадите шесть файлов с именами от xxx1 до xxx6, а затем закроете файл xxx1, то при следующем выборе команды New... программа не восстановит отсутствующее название (в данном случае xxx2), но автоматически присвоит файлу следующий порядковый номер после наибольшего номера файла, открытого на данный момент. То есть в нашем случае новому файлу будет присвоено имя xxx1.

Open...

В отличие от команды New..., предназначенной для создания нового файла, команда Open... открывает диалоговое окно, с помощью которого вы можете выбрать любой ранее сохраненный файл. Окно OpenFile имеет стандартный вид для всех приложений Windows. В случае попытки открыть уже открытый файл будет подан звуковой сигнал и показано предупреждающее сообщение.

Вторая слева кнопка на стандартной панели инструментов является альтернативой команде Open....

Close

Команда **Close** предназначена для закрытия ранее открытого файла. Если у вас в настоящий момент открыто несколько файлов, данная команда закроет активное, т.е. текущее окно. Если вы по ошибке попытаетесь закрыть несохраненный файл, программа предупредит о том, что вы рискуете потерять информацию, и предложит сохранить ее прямо сейчас.

Save

Команда Save сохраняет содержимое текущего окна в соответствующем файле. По строке заголовка окна можно определить, соответствует ли активному окну какой-нибудь файл на жестком диске. Если вы открыли новое окно и еще не сохраняли его, то в строке заголовка будет показано стандартное имя вида xxx1. При попытке сохранить информацию из окна, которому не соответствует ни один файл, автоматически будет открыто диалоговое окно SaveAs.

Для сохранения файла можно также использовать расположенную на панели инструментов кнопку Save (третья слева). Если файл был открыт в режиме только для чтения, то команда Save будет недоступной.

SaveAs...

Команда SaveAs... позволяет сохранить содержимое окна в файле под новым именем. Предположим, вы только что закончили работу над проектом и, имея вполне работоспособную программу, хотите попытаться внести некоторые изменения. В целях безопасности текущую версию программы нужно сохранить. Для этого вы выбираете команду SaveAs и сохраняете проект под новым именем, после чего можете спокойно экспериментировать с дубликатом. Если эксперименты приведут к повреждению программы, вы всегда сможете вернуться к исходной версии.

SaveAll

Если вам никогда ранее не приходилось заниматься программированием на C/C++ в Windows95,98 или NT, то поначалу вы будете ошеломлены обилием файлов, вовлеченных в проект. Неудобство команды Save состоит в том, что она сохраняет содержимое только одного, текущего окна. С помощью команды SaveAll можно сохранить все открытые на данный момент файлы. Если содержимое каких-то окон ранее не сохранялось в файлах, то для них автоматически будет открываться окно SaveAs, где вы сможете вводить имена новых файлов.

PageSetup...

Данную команду обычно используют перед выводом файла на печать. В открывающемся при этом диалоговом окне **PageSetup** вы можете задать верхний и нижний колонтитулы для каждой печатной страницы, а также размеры, верхнего, нижнего, правого и левого полей страницы.

Команды форматирования, которые можно использовать при настройке колонтитулов, перечислены в табл. 2.1.

Таблица 2.1. Команды форматирования используемые в диалоговом окне Page Setup	
Команда форматирования	Назначение
&c	Центрирование текста
&d	Добавление текущей системной даты
&f	Добавление имени файла
&l	Выравнивание текста по левому краю
&p	Нумерация страниц
&r	Выравнивание текста по правому краю
&t	Добавление текущего системного времени

Print...

Чтобы вывести на печать содержимое активного окна, нужно выбрать из меню **File** команду **Print....** Откроется диалоговое окно **Print**, в котором вы сможете установить требуемые параметры печати. Прежде всего необходимо решить, хотите вы вывести на печать все содержимое файла или только предварительно выделенную часть. Если в активном окне был выделен блок, то в окне **Print** станет доступной опция **Selection** группы **PrintRange** (в противном случае переключатель окажется недоступным). Если к компьютеру подключено несколько устройств вывода, вы можете выбрать нужный принтер и произвести настройку параметров печати, щелкнув на кнопке **Setup**.

Recent Files и Recent Workspaces

Под командой **Print...** находятся списки недавно открывавшихся файлов и проектов. Удобная особенность таких списков состоит в том, что они обновляются автоматически. Когда вы в первый раз запускаете VisualC++, оба списка пусты.

Exit

Команда **Exit** закрывает окно VisualC++. Не беспокойтесь, если вы забыли сохранить содержимое какого-нибудь файла. Программа автоматически выдаст предупреждающие сообщения для каждого несохраненного файла.

Меню Edit

Команды меню **Edit** (рис. 2.3) позволяют редактировать текст и проводить поиск по ключевым словам в программном коде, отображаемом в активном окне. Работа этих команд основана на тех же принципах, что и работа аналогичных команд в большинстве текстовых редакторов.

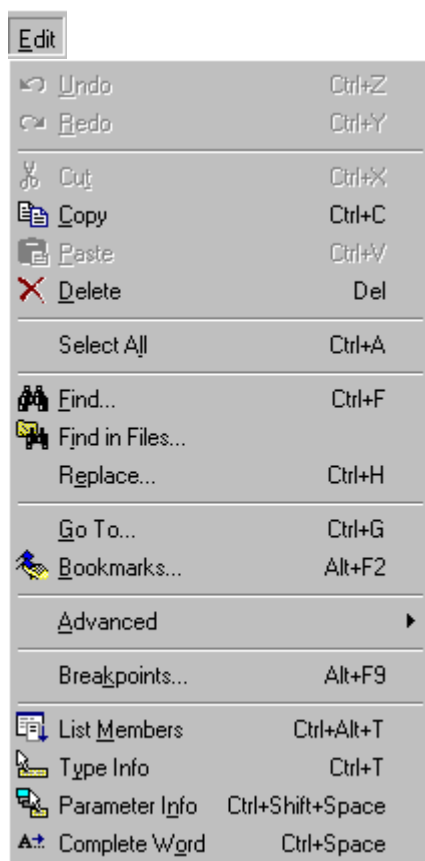


Рис. 2.3.

Undo

Команда Undo позволяет отменять последние выполненные операции редактирования. Данная возможность доступна также и через соответствующую кнопку стандартной панели инструментов (восьмая слева).

Redo

После того как вы отменили последнее действие с помощью команды Undo, вы можете повторить операцию, воспользовавшись командой Redo. Этой команде соответствует девятая слева кнопка стандартной панели инструментов.

Cut

Команда Cut копирует выделенный блок текста из активного окна в буфер обмена, после чего удаляет этот блок из окна. Команду Cut обычно используют в сочетании с командой Paste для перемещения блока текста из одного места в другое. На стандартной панели инструментов ей соответствует пятая кнопка слева.

Copy

Как и команда Cut, команда **Copy** копирует и помещает выделенный блок текста в буфер обмена, но этот блок сохраняется в активном окне. Команду **Copy** обычно используют в сочетании с командой **Paste** при необходимости скопировать блок текста из одного места в другое. Ей соответствует шестая слева кнопка стандартной панели инструментов.

Paste

Команда **Paste** предназначена для вставки информации из буфера обмена в текущий документ (в месторасположение текстового курсора). На стандартной панели инструментов ей соответствует седьмая слева кнопка.

Delete

Чтобы удалить выделенный блок текста, не копируя его в буфер обмена, можно воспользоваться командой **Delete**. Хотя удаленный текст и не будет скопирован в буфер обмена, вы все равно сможете восстановить его, если сразу после удаления выберете в меню **Edit** команду **Undo**.

Select All

Команда **SelectAll** используется для выделения всего содержимого активного окна с целью последующего вырезания, копирования или удаления.

Find...

Модуль поиска, запускаемый командой **Find...**, работает примерно так же, как и аналогичное средство поиска в большинстве текстовых редакторов. Поскольку языки C/C++ чувствительны к регистру символов, опции диалогового окна **Find** позволяют вам организовать поиск как с учетом, так и без учета регистра, а также поиск слова целиком. Можно задать и направление поиска - вверх или вниз от текущего положения курсора.

Одной из удобных особенностей команды **Find...** является возможность применения регулярных выражений. В табл. 2.2 приведены метасимволы, которые можно для этой цели вводить в поле **Findwhat** диалогового окна **Find**.

Таблица 2.2 Метасимволы используемые с командой Find	
Метасимволы	Назначение
*	Заменяет любое количество символов, в том числе нулевое Пример: Data*1 Результат поиска: Data1, DataIn1, DataOut1
.	Заменяет любой отдельный символ Пример: Data. Результат поиска: Data1 и Data2, но не DataIn1
^	Поиск ключевых слов только в начале строк Пример: Ado Результат поиска: все строки, начинающиеся с "do"
+	Заменяет любое число символов, начиная с единицы Пример: +value Результат поиска: i_value, fvalue, lng_value
\$	Поиск ключевых слов только в конце строк Пример: end;\$ Результат поиска: все строки, заканчивающиеся на "end;"
[]	Поиск значений, соответствующих указанному диапазону Пример: Data[A...Z] Результат поиска: DataA, но не DataI Пример: Data[1248] Результат поиска: Data2, но не Data3
\	Отменяет специальное назначение следующего за ним метасимвола Пример: 100\ Результат поиска: "100\$" (в отличие от самого шаблона 100\$, который означает поиск образца "100" в конце строки)
{ }	Поиск ключевых слов, начинающихся с комбинации символов, заключенных в фигурные скобки Пример: {no}* _answer Результат поиска: answer, no_answer, nono_answer, nonono_answer

Find in Files...

При выборе команды **Find in Files...** вы получаете в свое распоряжение все средства команды **Find...** и возможность проводить поиск ключевых слов сразу в нескольких файлах. Вы можете спросить: "С какой стати я стану искать что-нибудь сразу в нескольких файлах?" Чтобы ответить на этот вопрос, вспомним, что проект, написанный на C/C++, состоит из множества взаимосвязанных файлов. Предположим, в процессе программирования вы поймете, что какую-то часто используемую в приложении конструкцию лучше заменить более компактной. В таком случае, выполнив команду **Find in Files...**, вы будете уверены, что произвели замену во

всех файлах проекта. Если над каким-то большим проектом работает группа людей, то с помощью команды **FindinFiles...** вы сможете отобрать файлы, автором которых является определенный сотрудник. Кроме того, помните, что возможности команды **FindinFiles...** не ограничены одной папкой или даже одним диском. С помощью этой команды вы можете вести поиск в локальной сети, в интранет и даже в Internet, отыскивая заданные имена, строки, ключевые слова, методы и многое другое.

Replace...

При выборе команды **Replace...** открывается диалоговое окно, с помощью которого можно менять строки текста. Для этого нужно ввести в соответствующие поля текст для поиска и текст для замены, после чего установить критерии поиска. Вы можете проводить поиск с учетом или без учета регистра символов, искать слова целиком и использовать регулярные выражения, которые мы рассмотрели выше, при знакомстве с командой **Find....**

Хорошенько подумайте, прежде чем щелкнуть на кнопке **ReplaceAll**, поскольку результат выполнения этой команды может оказаться разрушительным для вашей программы. Помните, что вы можете отменить результаты операции замены, если сразу выберете команду **Undo**.

GoTo...

С помощью команды **GoTo...** можно быстро переместить курсор к определенному месту текущего документа. После выбора этой команды откроется диалоговое окно, в котором можно задать номер строки программы, куда следует перейти. Если вы введете значение, превышающее число строк программы, то курсор будет перемещен в конец файла.

Bookmarks...

Команда **Bookmarks...** позволяет помещать закладки в тех местах программы, к которым вы часто обращаетесь. После того как закладка будет установлена, вы сможете быстро перейти к ней с помощью команды меню или определенного сочетания клавиш. Закладку, которая больше не понадобится, можно в любой момент удалить. Вы можете создавать как именованные (они будут сохраняться между сеансами редактирования), так и безымянные закладки. К именованной закладке можно перейти в любое время, даже если файл, к которому она относится, в данный момент не открыт. Именованная закладка хранит как номер строки, так и позицию курсора на строке, которую он занимал во время ее создания. Причем позиция будет автоматически обновляться по мере редактирования файла. Даже удалив все символы вокруг закладки, вы все равно сможете перейти к указанному месту в файле.

Breakpoints...

Данная команда позволяет устанавливать точки прерывания в различных местах программы.

ListMembers

Команда **ListMembers** отображает список доступных переменных-членов или функций выбранного класса либо структуры.

TypeInfo

Данная команда отображает окно подсказки, содержащее описание всех идентификаторов.

ParameterInfo

Эта команда отображает полное описание (включая список параметров) функции, имя которой расположено слева от курсора. Параметр, выделенный полужирным шрифтом, соответствует тому параметру, который вы должны ввести в данный момент.

CompleteWord

При выборе команды **CompleteWord** программа автоматически допишет вместо вас название функции или имя переменной, которое вы только начали вводить. Эта опция способна заметно сохранить ваше время, избавив от необходимости каждый раз вводить с клавиатуры длинные, часто повторяющиеся имена.

Меню View

Меню View(рис. 2.4) содержит команды, позволяющие настроить внешний вид рабочего пространства.

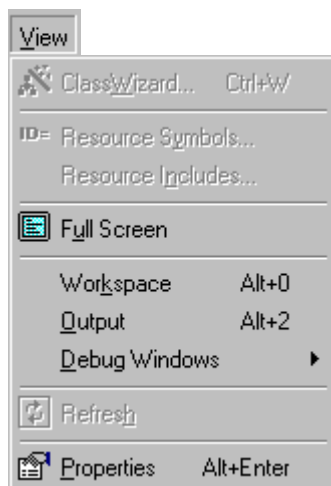


Рис. 2.4

ClassWizard...

Мастер ClassWizard облегчит выполнение таких повторяющихся задач, как создание новых классов и обработчиков сообщений, переопределение виртуальных функций MFC и сбор данных от элементов управления диалоговых окон. Одно очень важное замечание: ClassWizard работает только с приложениями, использующими библиотеку MFC, что отличает его от таких средств, как ClassView и WizardBar, работающих с MFC, ATL и вашими собственными производными классами. К тому же ClassView не распознает классы, если они не зарегистрированы в файле базы данных ClassView (файл с расширением CLW). С помощью мастера ClassWizard можно выполнять следующие действия:

- добавлять к новому классу методы и свойства;
- порождать новые классы от базовых классов MFC;
- создавать новые обработчики сообщений;
- объявлять переменные-члены, которые автоматически инициализируют, собирают и проверяют данные, вводимые в диалоговых окнах или формах;
- удалять существующие обработчики сообщений;
- определять, какие сообщения уже связаны с обработчиками, и просматривать их код;
- работать с существующими классами и библиотеками типов.

Resource Symbols... и Resource Includes...

Вы очень быстро убедитесь в том, что по мере увеличения и усложнения кода вашего приложения стремительно возрастает и число задействованных в проекте ресурсов, все труднее становится отслеживать постоянно растущее число идентификаторов ресурсов, разбросанных по многочисленным файлам проекта. Команды **Resource Symbols...** и **Resource Includes...** существенно облегчат процесс контроля за ресурсами. Перечислим операции, которые можно выполнить с их помощью:

- изменение имен и значений символических идентификаторов, которые в данный момент не используются;
- определение новых идентификаторов;
- удаление ненужных идентификаторов;
- быстрая загрузка соответствующих редактор ресурсов;

- просмотр описаний существующих идентификаторов и определение ресурсов, связанных с каждым идентификатором.

FullScreen

Подобно большинству программистов, вы в процессе программирования наверняка хотите видеть на экране как можно больше кода. В этом вам поможет команда **FullScreen**, при выборе которой окно редактирования будет развернуто на весь экран.

Workspace

При выборе команды **Workspace** на экране появляется панель **Workspace** (обычно в левой части рабочего окна), с помощью которой можно обращаться к текущим классам, файлам, ресурсам. Вы можете переходить от списка к списку, щелкая на вкладках, расположенных в нижней части панели.

Output

По команде **Output** открывается окно **Output**, в котором отображается ход выполнения таких процессов, как компиляция и компоновка программы. В это окно выводятся также все предупреждающие сообщения и сообщения об ошибках, генерируемые компилятором и компоновщиком.

Debug Windows

В подменю **Debug Windows** содержатся команды вызова различных окон отладчика, включая **Watch**, **Call Stack**, **Memory**, **Variables**, **Registers** и **Disassembly**.

Refresh

Команда **Refresh** предназначена для обновления вида текущего окна - аналогично тому, как с помощью клавиши [F5] мы обновляем внешний вид окна программы Explorer (Проводник) в Windows.

Properties

При выборе этой команды выводятся данные о текущем файле, такие как дата создания, размер, тип файла, особенности редактирования и многое другое, что зависит от типа файла.

Меню Insert

Меню **Insert** (рис. 2.5) содержит команды, позволяющие вставлять в проект новые файлы, ресурсы, объекты и т.д.

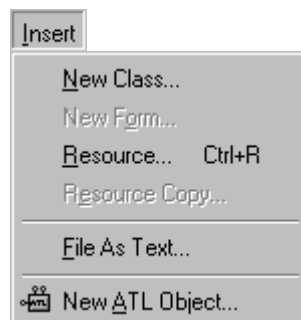


Рис 2.5.

NewClass...

При выборе данной команды открывается диалоговое окно **NewClass**, в котором можно задать имя нового класса (таковым может быть класс библиотек MFC и ATL или класс общего назначения) и указать его базовый класс. В результате создается файл заголовков и файл реализации нового класса.

Resource...

Эта команда позволяет добавить в проект новые ресурсы, включая горячие клавиши, растровые изображения, указатели мыши, диалоговые окна, значки, HTML-файлы, меню, таблицу строк, панели инструментов и идентификатор версии.

ResourceCopy...

VisualC++ дает возможность создать копию ресурса при изменении языка его описания. Язык ресурса выбирается в списке **Language** окна **InsertResourceCopy**. В поле **Condition** можно задать символический идентификатор, наличие которого в данной конфигурации проекта является условием для подключения ресурса. Язык ресурса отображается в окне **Workspace** после имени ресурса.

File As Text...

Данная команда применяется для добавления в проект текста указанного файла. Перед выбором команды необходимо открыть окно редактирования и установить курсор в месте ввода текста.

New ATL Object...

Данная команда позволяет добавить в существующий проект ATL-класс. ATL-объекты представляют собой шаблонные классы C++, которые реализуют основные средства COM, включая интерфейсы IUnknown, IClassFactory, IClassFactory2 и IDispatch, поддержку двусторонних и отключаемых интерфейсов, интерфейсов перечисления (IEnumXXXX), точек подключения, элементов управления ActiveX и многое другое.

Меню Project

Команды меню **Project**(рис. 2.6) позволяют управлять открытыми проектами.

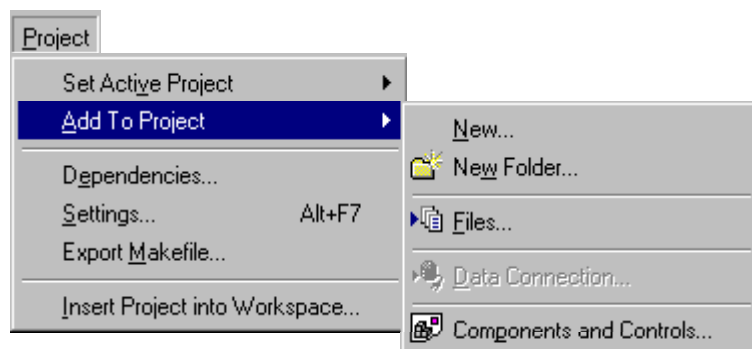


Рис 2.6.

SetActiveProject

В данном подменю отображается список загруженных проектов, из которых можно выбрать активный.

AddtoProject

Это подменю состоит из команд, предназначенных для добавления в проект новых компонентов. Добавляемый файл помещается во все конфигурации проекта.

Dependencies

Если ваш большой проект разбит на несколько подпроектов, то для отображения иерархических связей между ними следует использовать команду **Dependencies**.

Settings...

При выборе команды Settings... открывается довольно сложное диалоговое окно, позволяющее устанавливать практически все параметры конфигурации проекта, включая опции компилятора и компоновщика.

ExportMakefile...

С помощью этой команды можно сохранить в файле всю информацию, необходимую для построения проекта. Файл, созданный с применением команды ExportMakefile..., хранит все установки, которые вы сделали в среде VisualC++.

Insert Project Into Workspace...

Данная команда добавляет существующий проект в ваше рабочее пространство. Сказанное может звучать несколько странно, если вы нечетко представляете, какая разница между проектом и рабочим пространством. Последнее представляет собой область, содержащую совокупность проектов и их конфигураций. Проектом называется группа файлов, позволяющих построить программу или выходной двоичный файл (файлы). Рабочее пространство может содержать несколько проектов, причем эти проекты часто относятся к разным типам.

Меню Build

В меню Build(рис. 2.7) содержатся всевозможные команды, предназначенные для генерации кода приложения, отладки и запуска созданной вами программы.

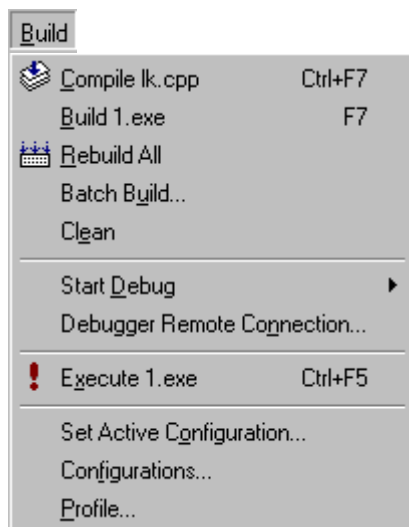


Рис 2.7.

Compile

Выбор этой команды приводит к компиляции содержимого текущего окна.

Build

Обычно проекты, написанные на C/C++, включают в себя много файлов. Поскольку поочередная компиляция всех файлов займет много времени, полезной окажется команда **Build**, которая автоматически проанализирует файлы проекта, компилируя только те из них, которые были созданы позже, чем исполняемый файл проекта.

Прежде чем выбрать команду **Build**, вам следует принять решение, следует ли в конечный файл включать отладочную информацию (конфигурация Debug) или же исключить эти данные из файла (конфигурация Release). Чтобы установить тот или иной режим, необходимо в меню **Build** выбрать команду **Set Active Configuration...**. Если вы закончили работу над программой и убедились в ее работоспособности, отладочную информацию из выходного файла целесообразно исключить — в таком случае он станет значительно компактнее.

Сообщения об обнаруживаемых в процессе компиляции и компоновки ошибках будут появляться в окне **Output**.

RebuildAll

Различие между командами **Build** и **RebuildAll** состоит в том, что последняя не обращает внимания на дату и время создания файлов и компилирует все файлы проекта.

Если при выполнении команды **RebuildAll** будут обнаружены синтаксические ошибки, как фатальные, так и потенциально опасные, то предупреждения и сообщения о них появятся в окне **Output**.

BatchBuild...

Эта команда аналогична команде **Build**, но с ее помощью можно обработать сразу несколько конфигураций одного проекта.

Clean

С помощью команды **Clean** из всех конфигураций текущего проекта удаляются промежуточные файлы. Построить файлы заново можно путем выбора команды **Build**.

StartDebug

Данное подменю содержит команды, предназначенные для выполнения программы в режиме отладки: до курсора или до заданной точки останова.

Debugger Remote Connection...

Благодаря наличию этой команды можно осуществлять отладку проекта, выполняющегося на удаленном компьютере.

Execute

Если компиляция прошла успешно, выберите команду **Execute**, и построенная программа будет запущена.

Set Active Configuration...

Если вы трудитесь над большим проектом, состоящим из нескольких под-проектов, каждый из которых имеет собственный исполняемый файл, то перед выбором команды **Build** или **RebuildAll** вам следует указать, с каким именно исполняемым файлом вы собираетесь работать в данный момент. Для выполнения этой задачи воспользуйтесь командой **SetActiveConfiguration...**, которая позволяет выбрать требуемую конфигурацию проекта.

Configurations...

Команда **Configurations...** позволяет добавлять или удалять конфигурации текущего проекта. Например, если вы начали работу только с конфигурацией Debug(отладочная версия программы), то теперь можете добавить конфигурацию Release(финальная версия программы).

Profile...

Данная команда представлена только в профессиональной и корпоративной версиях VisualC++. Но чтобы ею можно было воспользоваться, необходимо при **создании** проекта установить опцию, задающую подключение профилировщика (опция Enable profiling категории General вкладки Link диалогового окна Project Settings). Профилировщик используется для анализа работы программы во время ее выполнения. В процессе профилирования в окне Output отображается информация, на основании которой вы можете выяснить, какие части вашего программного кода работают эффективно, а какие не выполняются или требуют больших временных затрат.

Меню Tools

Меню **Tools**(рис. 2.8) содержит команды вызова вспомогательных утилит, программирования макросов и настройки среды VisualC++.

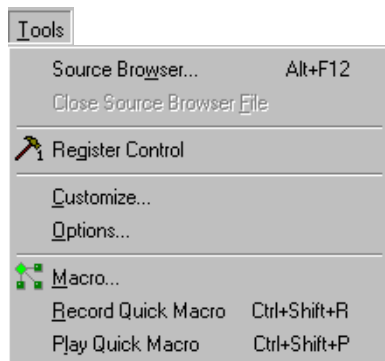


Рис. 2.8.

SourceBrowser...

Этой командой можно воспользоваться при необходимости просмотреть информацию об исходных файлах. Вы можете поручить компилятору создавать по вспомогательному SBR-файлу для каждого объектного (OBJ) файла, который будет встречаться в процессе компиляции. Когда вы создаете или обновляете основной информационный BSC-файл, все SBR-файлы проекта должны быть представлены на диске. Для того чтобы создать SBR-файл, содержащий всю возможную информацию, установите опцию **Generate browse info** в категории **Listing Files** вкладки C/C++ диалогового окна **Project Settings**. Если из файла необходимо исключить информацию о локальных переменных, задайте там же опцию **Exclude local variables from browse info**.

Close Source Browser File

Данная команда закрывает текущий SBR-файл.

ErrorLookup

Утилиту ErrorLookup используют при необходимости получить текст сообщений, связанных с кодами системных ошибок. Введите код ошибки в поле **Value**, и в поле **ErrorMessage** автоматически отобразится связанное с ним сообщение.

ActiveX Control Test Container

Данная утилита предназначена для тестирования элементов управления ActiveX. Она позволяет менять свойства элемента управления, вызывать его методы, моделировать возникновение требуемых событий и многое другое.

OLE/COM Object Viewer

Эта утилита отображает сведения обо всех объектах ActiveX и OLE, установленных на вашем компьютере, а также о поддерживаемых ими интерфейсах. Она также позволяет редактировать реестр и просматривать библиотеки типов.

Spy++

Утилита Spy++ выводит информацию о выполняющихся системных процессах и потоках, существующих окнах и поступающих оконных сообщениях. Указанная утилита также предоставляет набор инструментов, облегчающих поиск нужных процессов, потоков и окон.

MFC Tracer

Дополнительные возможности для отладки оконных приложений, построенных на основе MFC, предоставляет утилита MFC Tracer. Эта утилита отображает в окне отладки сообщения о выполнении операций, связанных с использованием библиотеки MFC, а также предупреждения об ошибках, если при выполнении приложения происходят какие-либо сбои.

VisualComponentManager

Данная утилита предназначена для ведения базы данных готовых программных компонентов.

RegisterControl

Элементы управления OLE, как и другие OLE-серверы, могут использоваться различными приложениями, поддерживающими технологию OLE. Но для этого необходимо зарегистрировать библиотеку типов и класс элемента управления, что как раз и выполняет команда **RegisterControl**.

Customize...

При выборе данной команды открывается диалоговое окно **Customize**, которое позволяет настраивать меню и панели инструментов, а также назначать различным командам сочетания клавиш.

Options...

Данная команда открывает окно **Options**, в котором задаются различные параметры среды VisualC++.

Macro... / Record... / Play...

Эти команды используются для создания и воспроизведения макросов на VBScript. Макросы представляют собой небольшие процедуры, содержащие команды VBScript и не принимающие параметров. Макросы позволяют значительно упростить и ускорить работу в среде VisualC++. Например, вы можете записать в виде макроса некоторую часто выполняемую последовательность команд, в результате чего для осуществления той же самой задачи вам достаточно будет ввести простую комбинацию клавиш или нажать единственную кнопку панели инструментов.

Меню Window

Все команды меню **Window**(рис. 2.9), за исключением команды **DockingView**, в принципе соответствуют стандартному набору команд данного меню во всех приложениях Windows.

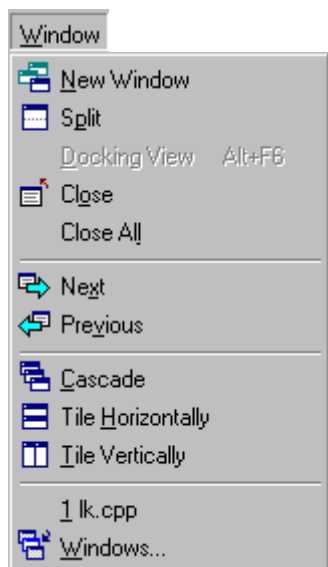


Рис 2.9.

NewWindow

Данная команда создает новое окно редактирования для текущего проекта.

Split

Команда Split позволяет разбить окно редактирования на несколько частей.

DockingView

С помощью этой команды можно закрепить панель инструментов у любого края родительского окна либо сделать ее свободно перемещаемой.

Close

Данная команда закрывает активное окно. Если содержимое окна не было предварительно сохранено, то будет выдано предупреждающее сообщение.

CloseAll

Эта команда закрывает все открытые окна. Если содержимое хотя бы одного из окон не было предварительно сохранено в файле, будет выдано предупреждающее сообщение.

Next

Посредством команды Next, относящейся к меню **Window**, можно переключаться между открытыми окнами.

Previous

Эта команда аналогична команде Next, но в отличие от последней осуществляет переходы между окнами в обратном порядке.

Cascade

Данная команда отображает на экране все открытые окна каскадом, что дает возможность пользователям видеть имена файлов в заголовках всех окон.

TileHorizontally

Эта команда располагает все открытые окна одно над другим. Такой вид отображения удобен для сравнения исходного и модифицированного текста программ.

TileVertically

Команда **Tile Vertically** располагает все открытые окна рядом друг с другом. Такой вид отображения удобен при необходимости произвести сравнение иерархических структур.

Список открытых окон

В нижней части меню показан динамически обновляемый список всех открытых на данный момент окон. Вы можете сделать активным любое окно, щелкнув на его имени в списке.

Меню Help

Меню **Help**(рис. 2.10) содержит стандартные для приложений Windows команды **Contents**, **Search** и **Index**, а также некоторые дополнительные команды.

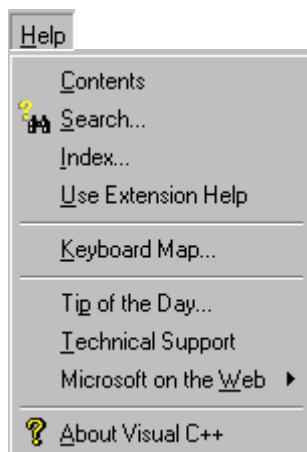


Рис 2.10

Contents / Search... / Index

Эти стандартные команды предоставляют доступ к интерактивной справочной системе программы.

UseExtensionHelp

Когда включена данная опция, в качестве справочной системы вызывается расширенный файл справки, а не MSDN.

KeyboardMap...

Данная команда выводит список и описание всех команд VisualC++ с перечнем связанных с ними сочетаний клавиш.

Tip of the Day...

Команда **TipoftheDay...** выводит окно с различными советами, касающимися работы в среде VisualC++.

TechnicalSupport

Данная команда отображает раздел справочной системы, посвященный вопросам технической поддержки.

Microsoft on the Web

В этом подменю содержатся команды перехода к Web-страницам в Internet, посвященным VisualC++ и различным техническим вопросам.

About Visual C++

About... — стандартная команда всех приложений Windows - отображает информацию о версии программы, авторских правах, зарегистрированном пользователе и установленных дополнительных компонентах.

- Создание вашей первой программы
- Сохранение файла
- Создание исполняемого файла
-

- Добавление файлов в проект
- Построение программы
- Отладка программы
 - Предупреждающие сообщения
 - Первое сообщение об ошибке
 - Использование команд Find и Replace
 - Быстрое обнаружение ошибочных строк
 - Продолжение отладки
- Запуск программы
 - Использование встроенного отладчика
- Дополнительные средства отладки
 - Работа с точками останова
 - Окно QuickWatch

Написание, компиляция и отладка простейшей программы

Для новичка VisualC++ может показаться, на первый взгляд, средой пугающе сложной и непонятной. Впрочем, подобные ощущения возникают при знакомстве с любой современной средой программирования. Начальное окно, открывающееся при запуске программы, выглядит вполне обычно, но по мере того как вы станете выбирать различные пункты меню и подменю, а затем просматривать многочисленные диалоговые окна, обилие настраиваемых параметров и опций будет приводить вас все в большее смятение.

Действительно, создание многофункционального приложения Windows, поддерживающего работу в Internet, является задачей совсем не простой, тем более если начинать приходится с нуля. Но хотим вас успокоить: современные средства программирования позволяют автоматически генерировать программный код для выполнения многих стандартных задач. И мы, авторы, поставили своей целью научить вас как основным принципам программирования на C/C++, так и использованию базовых возможностей одной из мощнейших современных сред программирования, каковой является VisualC++. Надеемся, вы сможете создавать вполне профессиональные приложения, соответствующие требованиям сегодняшнего дня.

Создание вашей первой программы

Первое, что вы должны сделать, приступая к работе над новой программой, — это создать новый проект. Для этого в меню **File** задайте команду New..., а в открывшемся диалоговом окне New выберите вкладку **Projects** (рис. 3.1).

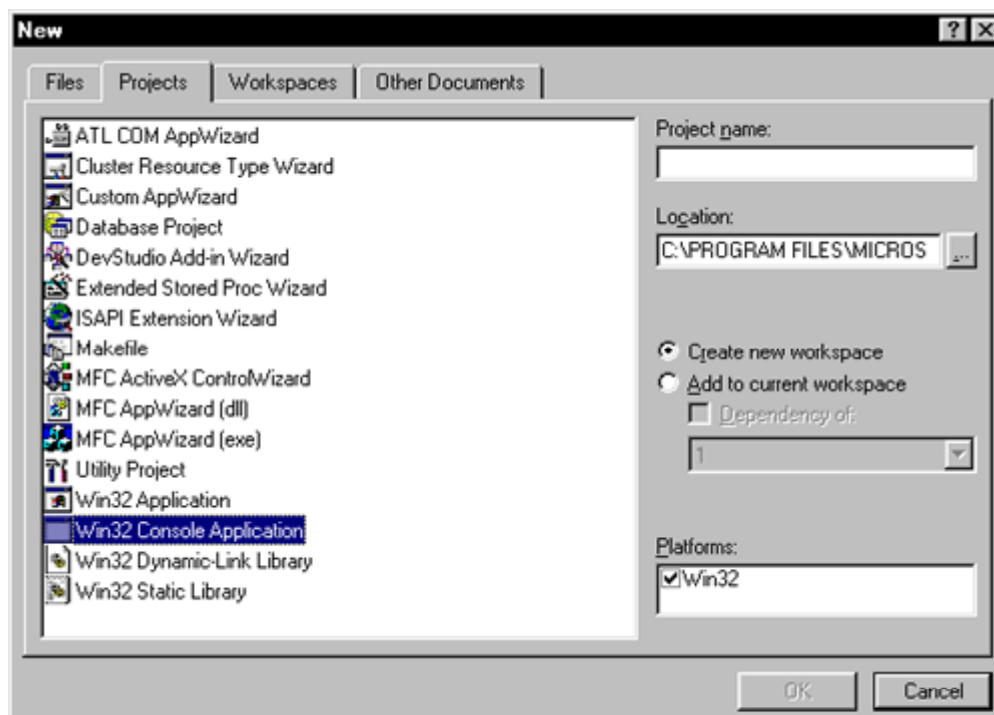


Рис 3.1. Выбор типа проекта

Теперь необходимо ввести имя файла проекта. Отнеситесь к данному моменту серьезно, так как это имя впоследствии будет использовано при построении исполняемого файла приложения.

Следующий шаг состоит в указании типа проекта. В нашем случае это будет простое консольное приложение — **Win32 ConsoleApplication**. В текстовом поле **Location** укажите папку, которую система автоматически создаст для файлов нового проекта.

Далее необходимо указать платформу, для которой создается проект. Для 32-разрядной версии компилятора VisualC++ в поле **Platforms** по умолчанию задана опция **Win32**.

После того как вы нажмете кнопку **OK**, отобразится окно мастера с набором опций (рис. 3.2).

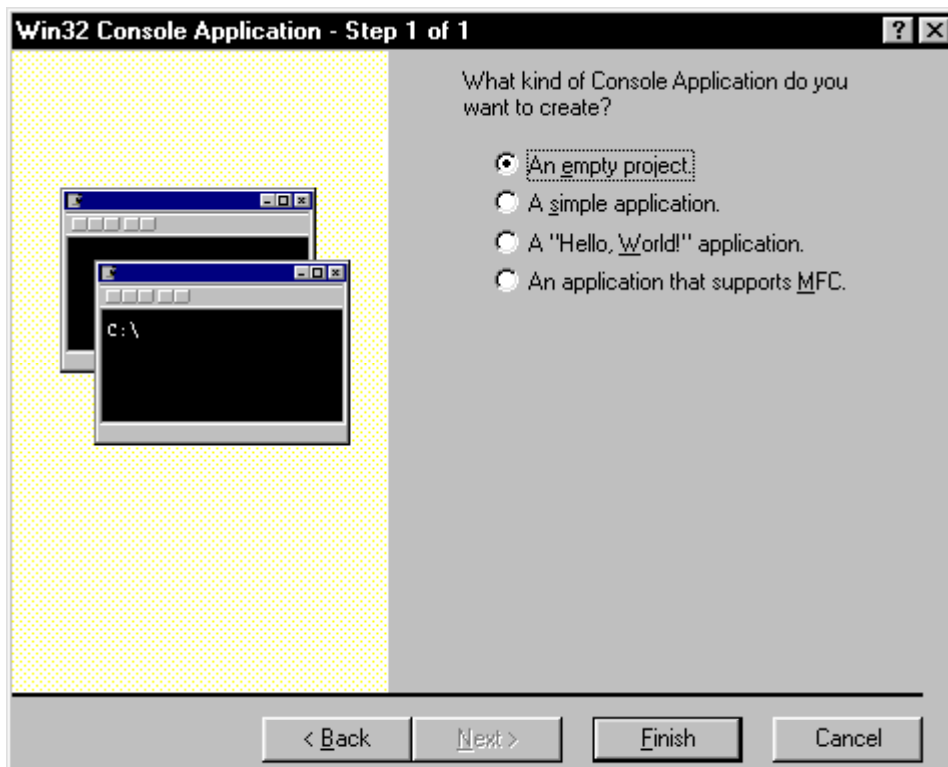


Рис. 3.2. Окно, появляющееся после выбора типа проекта **Win32 Console Application**

Поскольку мы будем создавать проект с нуля, выберите опцию **An empty project** и щелкните на кнопке **Finish**. Далее можете просто щелкнуть на кнопке, расположенной первой слева на стандартной панели инструментов. Когда перед вами откроется новое окно редактирования, введите такой код:

```
/* ПРИМЕЧАНИЕ: данная программа содержит ошибки, */
/* введенные с целью обучить вас использованию */
/* средств отладки */
#include <stdio.h>
/* Следующая константа определяет размер массива */
#define SIZE 5
/* Прототип функции */
void print_them(int offset, char continue, -int iarray[SIZE]);
void main( void ) {
    intoffset; /* индекс элемента массива */
    int iarray[SIZE]; /* массив чисел */
    char continue= 0; /* содержит ответ пользователя */
    /* Первый раз функция выводит значения неинициализированных переменных */
    /* print_them(offset,continue, iarray); */
    /* Приглашение для пользователя */
    Printf("\n\nWelcome to a trace demonstration!");
    printf("\nWould you like to continue (Y/N)");
    scanf("%c",continue);
    /* Пользователь вводит новые значения в массив */
    if (continue== 'Y' )
        for (offset=0; offset < SIZE, offset++) { printf ("\nPlease enter an
            integer: "); scanf("%d",siarray [of f set] ) ;
        }
    /* Второй раз функция отображает значения, введенные пользователем */
    print_them(offset, continue, iarray) ;
    /* Функция, выводящая значения элементов массива */
    void print_them(int offset, char continue, int iarray[SIZE])
```

```

{
printf("\n\n%d",offset);
printf("\n\n%d",continue);
for(offset=0; offset < SIZE, offset++)
printf("\n%d",iarray[offset]); }

```

Если вы хорошо знакомы с языком C, то наверняка заметили в программе ошибки. Не исправляйте их. Они были допущены специально, с целью обучить вас использованию различных средств отладки.

Сохранение файла

Желательно сохранить файл до того, как вы приступите к его компиляции и компоновке, а тем более до того, как попытаетесь запустить программу на выполнение. Опытные программисты могут рассказать много печальных историй о тех, кто пытался запустить на выполнение несохраненные файлы, после чего, вследствие непредвиденных фатальных ошибок, работу над приложением приходилось начинать сначала.

Чтобы сохранить введенный только что код, вы можете либо щелкнуть на третьей кнопке слева на стандартной панели инструментов, либо выбрать в меню **File** команду **Save**, либо нажать [Ctrl+S]. Когда вы в первый раз выбираете команду **Save**, открывается диалоговое окно **Save**. Сохраните файл под именем **ERROR. C**.

На рис. 3.3 окно редактирования показано до сохранения файла. После сохранения в строке заголовка окна отобразится новое имя файла.

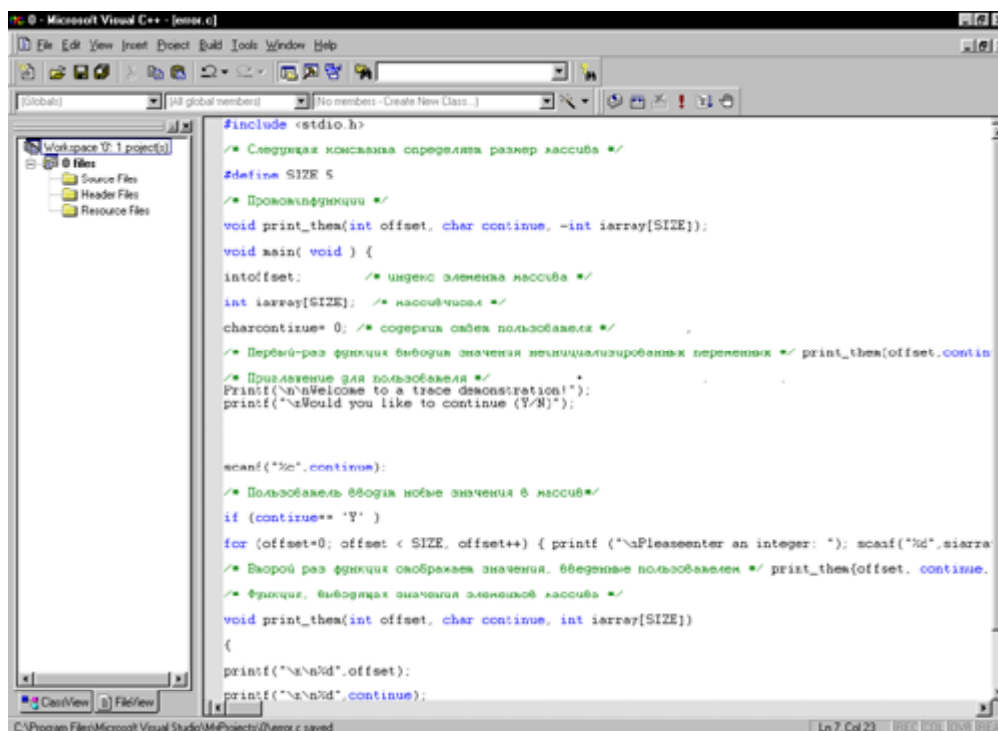


Рис 3.3. Окно редактирования до того, как его содержимое будет сохранено в файле

Создание исполняемого файла

Как правило, проекты для Windows 3.x, Windows95/98 и Windows, NT включают большое число файлов. Но сейчас мы начали работу над простейшей программой, состоящей из единственного файла. Этого вполне достаточно, чтобы разобраться с основными этапами создания полноценного приложения на C/C++.

Добавление файлов в проект

После того как проект создан, в него можно сразу же добавлять новые файлы. Перейдите на вкладку **FileView** и щелкните правой кнопкой мыши на элементе **ERRORfiles**. Контекстное

меню с выделенной командой **AddFilestoProject...**, которое при этом откроется, показано на рис. 3.4.

При выборе данной команды появляется окно **InsertFilesintoProject**, где вы можете отметить файлы, подлежащие включению в проект. Одно замечание по поводу типа файлов: файлы заголовков (с расширением H) не включаются в список файлов проекта, а добавляются в проект непосредственно во время построения программы с помощью директив препроцессора `#include`.

В нашем случае нужно найти созданный ранее файл **ERROR.C** и выполнить на нем двойной щелчок, в результате чего файл автоматически будет добавлен в проект.

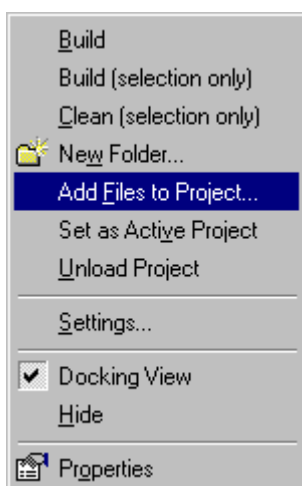


Рис 3.4. Меню, содержащее команду **Add files to Project ...**

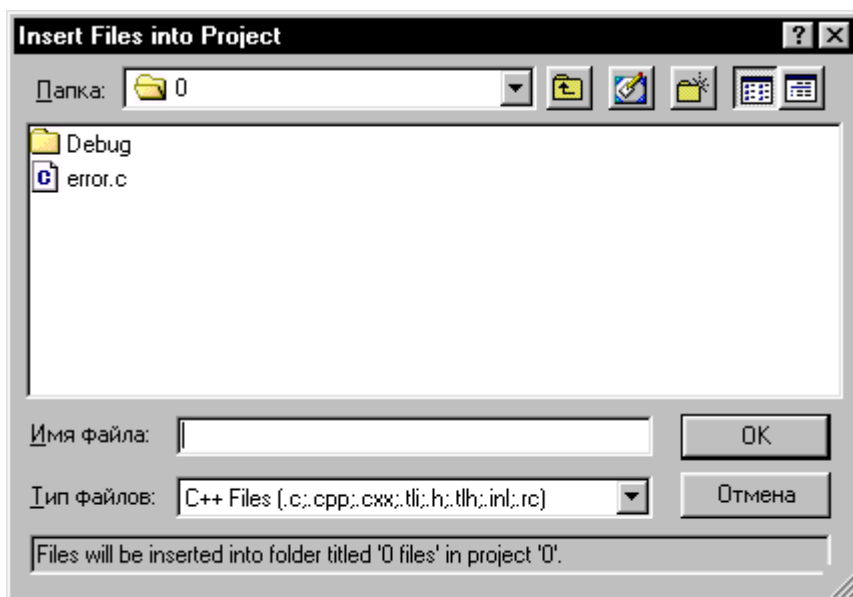


Рис. 3.5. Добавление файла **ERROR.C** в проект

Построение программы

После создания исходного файла можно приступить к созданию файла исполняемого. Согласно терминологии разработчиков VisualC++, этот процесс называется построением программы. Обратимся к показанному на рис. 3.6 меню **Build**с выделенной командой **RebuildAll**.

Единственное различие между командами **Build** и **RebuildAll**, как вы помните, состоит в том, что команда **RebuildAll**не проверяет дату создания используемых в проекте файлов, т.е. компилирует и компоует все файлы проекта, независимо от того, когда они были созданы.

Чтобы избежать недоразумений, связанных с тем, что системное время компьютера может быть легко изменено, при работе с небольшими приложениями рекомендуется использовать команду **RebuildAll**.

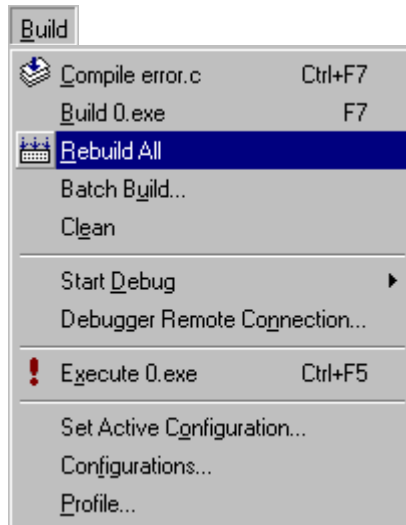


Рис. 3.6. Команда **RebuildAll** выполнит компиляцию и компоновку файлов нового приложения

Отладка программы

Если в программе были допущены синтаксические ошибки, при выполнении команд **Build** и **RebuildAll** сообщения о них будут отображаться на вкладке **Build** окна **Output** (рис. 3.7).

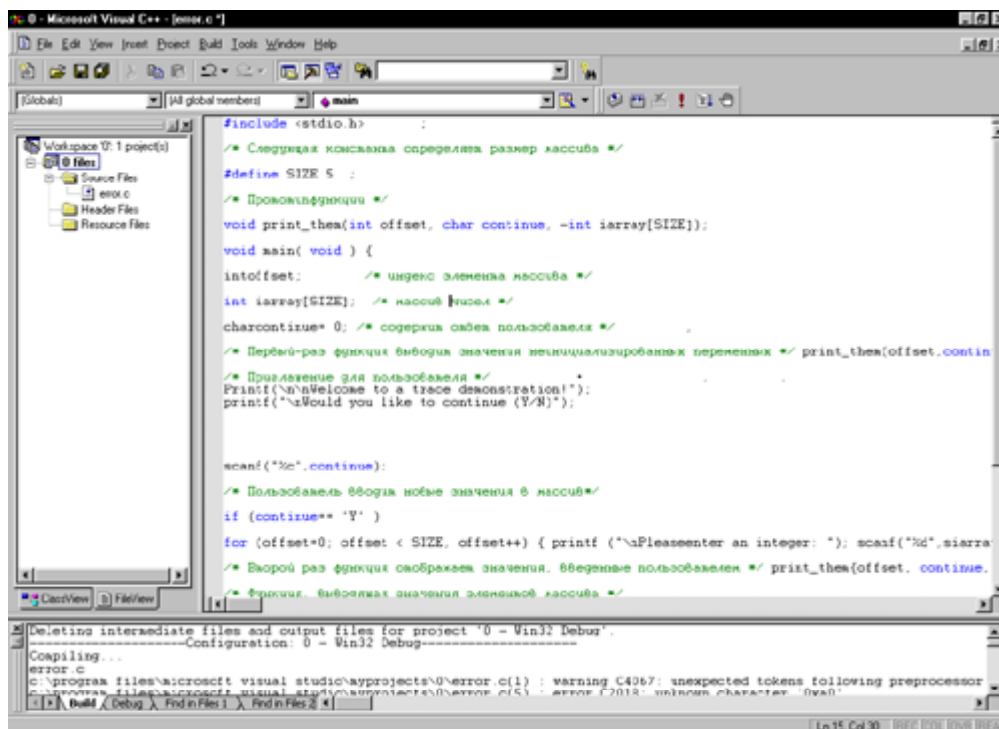


Рис. 3.7. Вывод сообщений об ошибках в окне VisualC++

Каждое сообщение начинается с указания имени файла. Поскольку приложения **Windows** обычно содержат много файлов, это очень ценное свойство.

Сразу за именем файла, в круглых скобках, указан номер строки, в которой была обнаружена ошибка. В нашем случае первая ошибка найдена в восьмой строке. После номера строки, за двоеточием, идут слово **error** или **warning** и номер ошибки.

Разница между сообщениями **error** и **warning** состоит в том, что программы с предупреждениями могут быть выполнены, а приложения с ошибками обязательно требуют исправлений. В конце каждой строки сообщения дается краткое описание обнаруженной ошибки.

Предупреждающие сообщения

Предупреждающие сообщения могут появляться в тех случаях, когда компилятор автоматически выполняет некоторые стандартные преобразования и сообщает об этом программисту. Например, если переменной типа `int`(целое число) будет присвоено дробное значение, то автоматически произойдет округление. Это не означает, что в программе допущена ошибка, но поскольку преобразование типов данных происходит незаметно для программиста, компилятор считает своим долгом сообщить об этом.

Приведем еще один пример. Большинство функций, объявленных в файле `MATH.H`, принимают аргументы и возвращают значения типа `double`(действительное число двойной точности). Если программа передаст одной из таких функций аргумент типа `float`(действительное число одинарной точности), компилятор, прежде чем направить данные в стек аргументов функции, выведет предупреждающее сообщение о том, что тип данных `float` был преобразован в `double`.

Вы можете предотвратить появление предупреждающих сообщений, если будете явно преобразовывать типы данных переменных в соответствии с правилами, принятыми в языке C. Так, в рассматриваемом случае явное приведение аргументов к типу данных `double` перед выполнением функции предотвратит появление предупреждающего сообщения.

Первое сообщение об ошибке

Сообщение об ошибке, представленное на рис. 3.7, является вполне типичным. Наиболее часто с ним приходится сталкиваться тем, кто только осваивает новый язык программирования. В данном случае программист попытался задать переменной имя, зарезервированное для ключевого слова. Это хороший принцип — присваивать переменным имена, созвучные с их назначением, однако выбранное нами имя вступило в конфликт с ключевым словом `continue`, существующим в C/C++.

Использование команд **Find** и **Replace**

Довольно часто в процессе программирования возникают ситуации, когда вам нужно найти и заменить какое-то слово в тексте программы. Вы, конечно же, можете сделать это с помощью диалогового окна, открываемого командой **Replace...** из меню **Edit**, но имеется и более быстрый способ. Рассмотрите внимательно панель инструментов, показанную на рис. 3.8, и найдите в поле списка **Find** слово `continue`.

Чтобы воспользоваться этим средством поиска, щелкните мышью в поле и введите слово, которое хотите найти, после чего нажмите [Enter]. На рис. 3.8 показан результат такого поиска. В тексте программы выделено слово `continue`, обнаруженное первым.

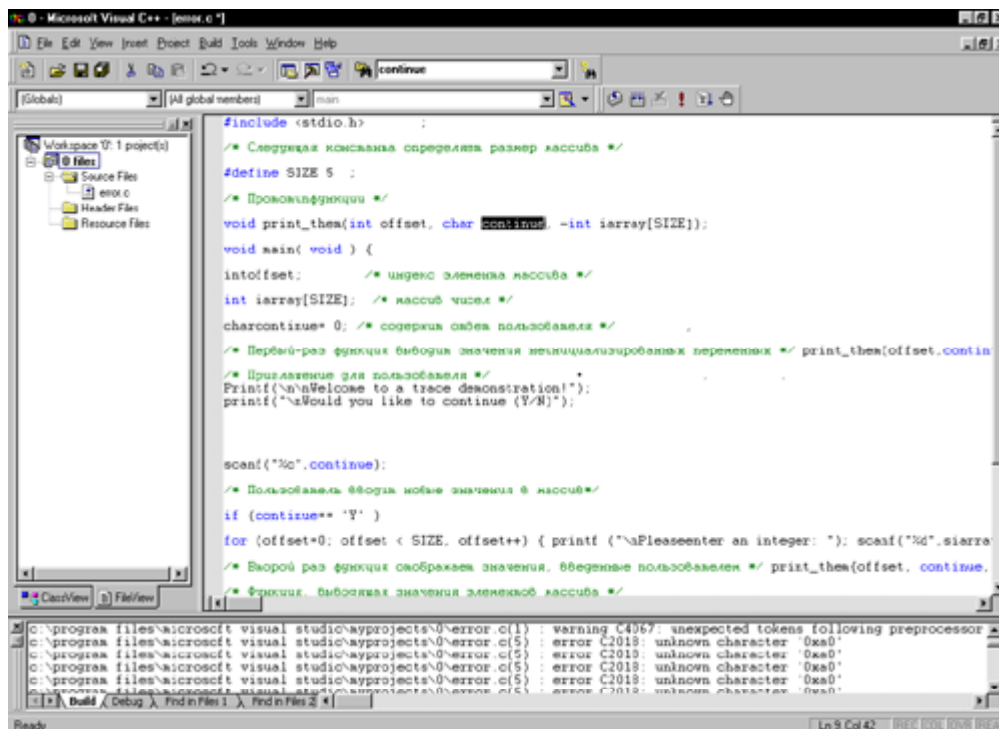


Рис. 3.8. Использование средств быстрого поиска

Данный метод достаточно удобен для поиска нужного слова. Но наша задача этим не ограничивается, поскольку имя переменной `continue` необходимо заменить во всей программе другим именем. В таком случае целесообразнее воспользоваться командой `Replace...` из меню `Edit` (рис. 3.9).

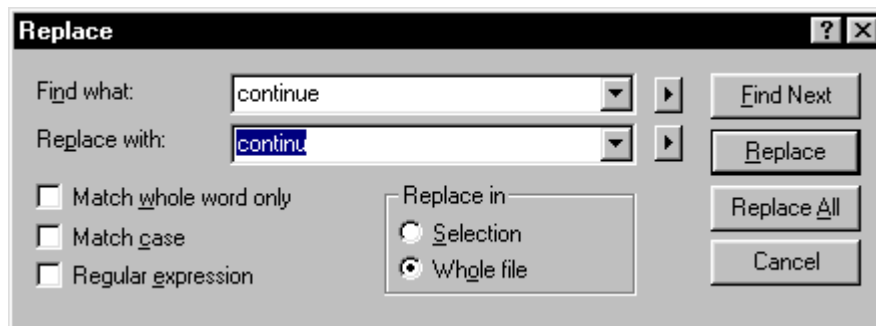


Рис. 3.9. Окно поиска

Наша цель состоит в том, чтобы заменить имя переменной `continue` словом, также указывало бы на назначение этой переменной, но отличалось бы от еривированных имен. С этой целью введем в поле `Replace with` слово `contin`. Но осталась маленькая проблема. В программе имеется строка `"\nWould you like to continue (Y/N)"`. Если вы выполните автоматическую замену во всем файле, щелкнув на кнопке `Replace All`, то сообщение, выдаваемое программой, будет содержать грамматическую ошибку. Поэтому замену следует проводить последовательно, переходя от слова к слову, а в указанном месте щелкнуть на кнопке `Find Next`.

Быстрое обнаружение ошибочных строк

Теперь необходимо выполнить повторную компиляцию программы, после которой окно сообщений будет выглядеть так, как показано на рис. 3.10.

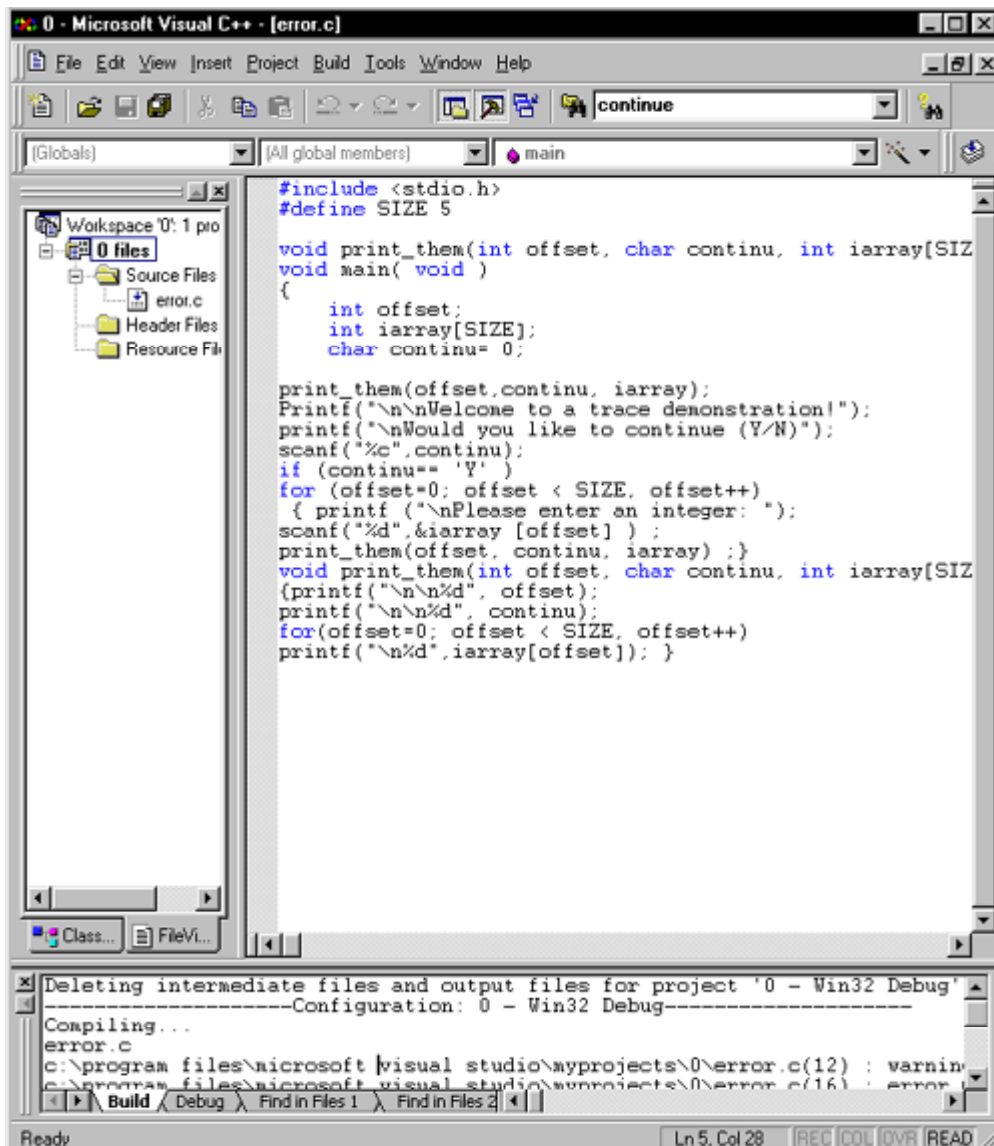


Рис. 3.10 Обновленное окно сообщений

Существует достаточно быстрый способ перехода от окна сообщений к окну редактирования, и мы вам о нем расскажем. Поместите курсор на интересующей вас строке сообщения, например на первом предупреждении:

warning. C4013: 'Printf' undefined;...

А теперь просто нажмите [Enter]. Курсор в окне редактирования будет автоматически помещен в строку программы, вызвавшую появление сообщения об ошибке, а слева от строки появится стрелка (рис. 3.11).

Как вы уже знаете, языки C/C++ чувствительны к регистру символов. Поэтому компилятор совершенно точно установил причину ошибки. Первая буква функции printf() в нашей программе ошибочно была введена в верхнем регистре. Компилятор, конечно же, не смог найти в библиотеке функцию Printf(). Эту ошибку нетрудно исправить — достаточно заменить букву P буквой p. Затем обязательно сохраните файл.

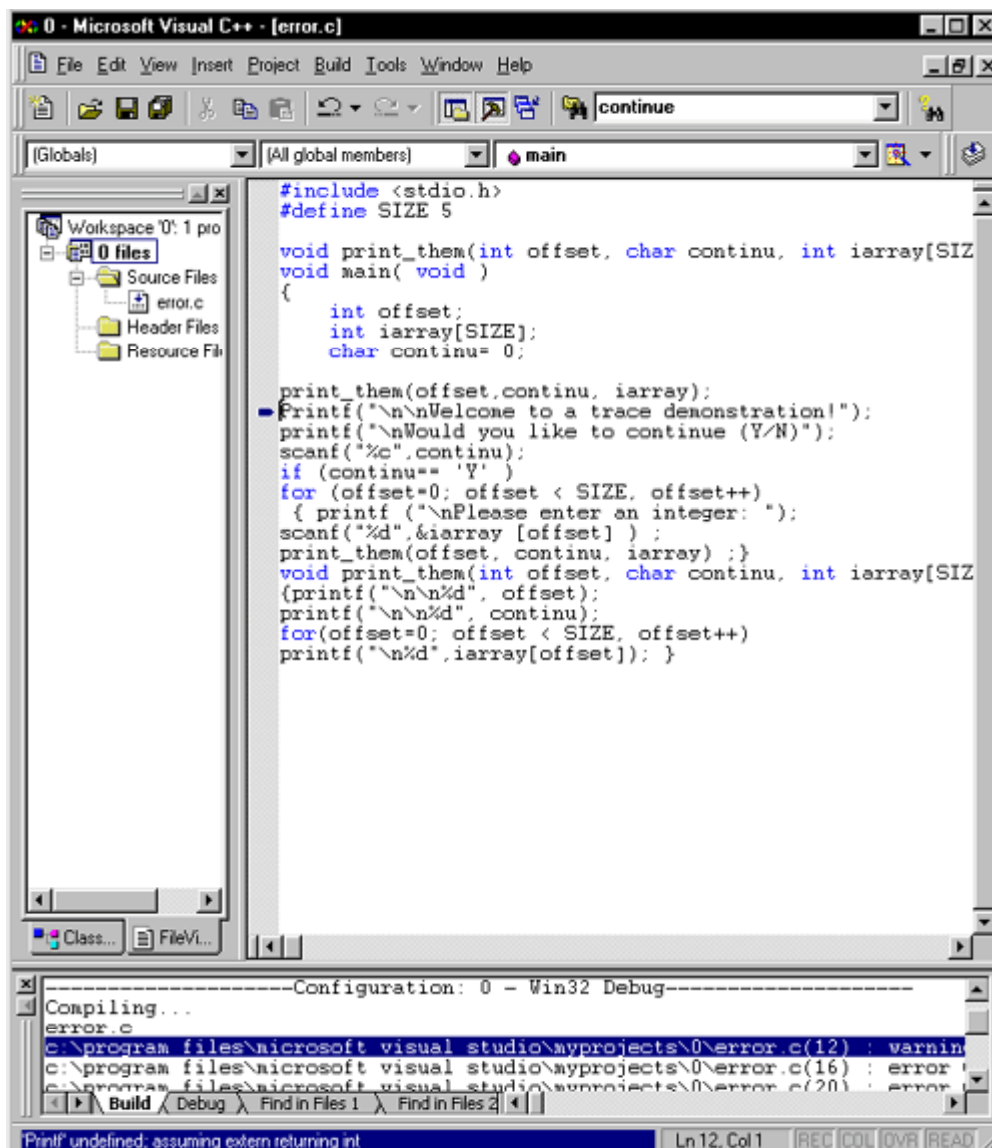


Рис. 3.11. Помечанная стрелкой строка программного кода, содержащая ошибку

Продолжение отладки

После того как вы внесли исправления, программа готова к новой попытке построить исполняемый файл. Перейдите в меню **Project** и вновь выберите команду **RebuildAll**. На рис. 3.12 показано обновленное окно сообщений.

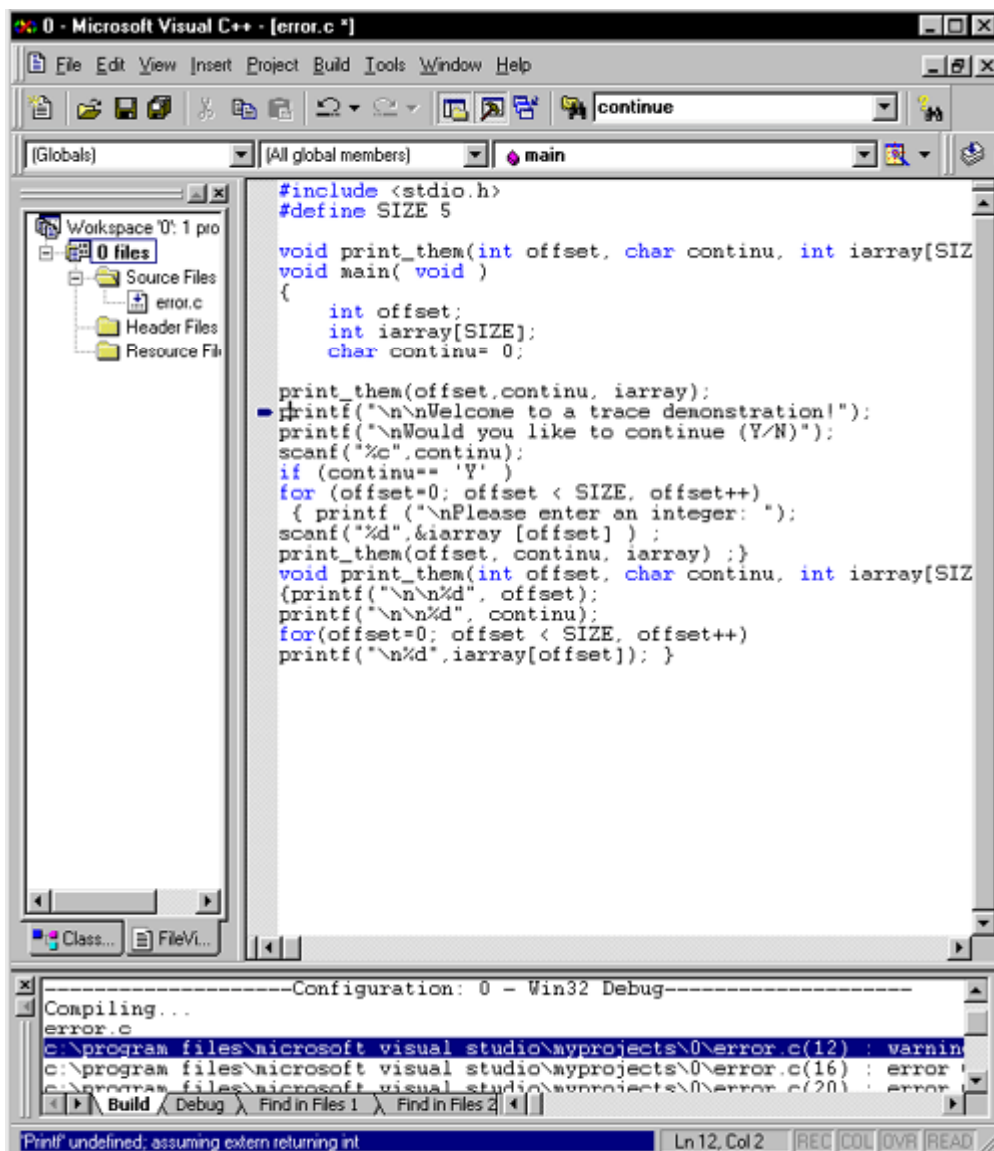


Рис.3.12. Окно сообщений во время третьей попытки построить исполняемый файл

Теперь обнаруживаем, что та же строка, которая содержала некорректное имя функции (Printf()), включает в себе еще одну ошибку. В C/C++ все строковые значения должны браться в двойные кавычки. Значит, в нашем случае открывающие кавычки в функции printf () следует поставить сразу после открывающей круглой скобки, т.е. строка должна начинаться следующим образом: printf(").

Сохраните файл и попытайтесь еще раз построить программу. Как выглядит окно сообщений после очередного обновления, показано на рис. 3.13.

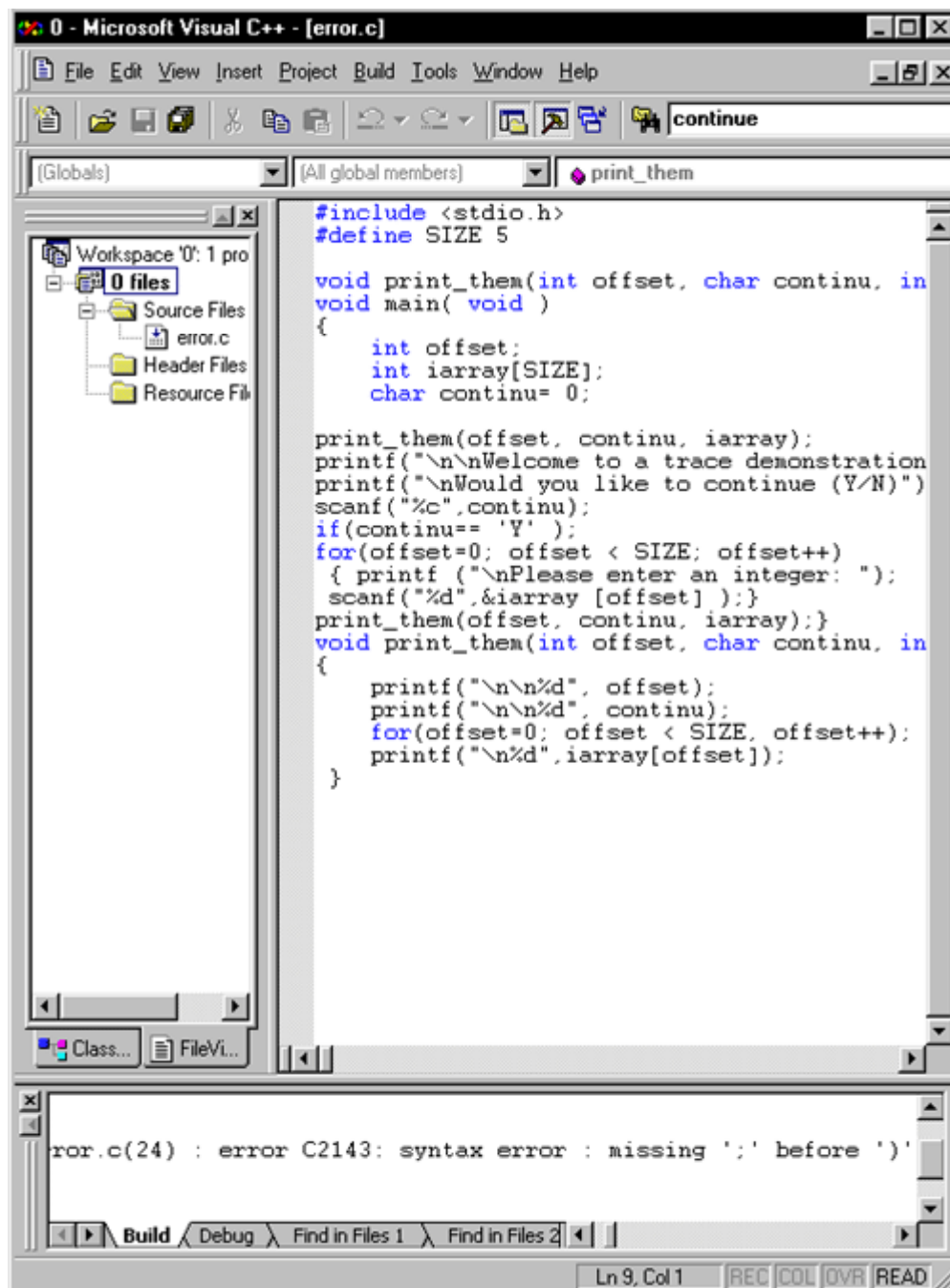


Рис. 3.13. Окно сообщений во время четвертой попытки построить исполняемый файл

На этот раз выдается такое сообщение об ошибке:

syntax error: missing ';'before ')'

В C/C++, в отличие от Pascal, символ точки с запятой используется для обозначения окончания выражения, а не в качестве разделителя. Таким образом, в конце второй проверки в цикле `for` необходимо ввести точку с запятой. После исправления сохраните файл и вновь выполните команду **RebuildAll**.

Все в порядке? Судя по окну сообщений, у компилятора нет больше претензий к вашей программе, и команда **RebuildAll** успешно сгенерировала исполняемый файл ERROR.EXE.

В окне сообщений должны отсутствовать сообщения об ошибках, но присутствовать одно предупреждение, которое можно проигнорировать. Если это не так, значит, вы допустили ошибки при вводе программы. Проверьте программный код и исправьте его.

Запуск программы

Чтобы запустить программу, просто выберите в меню **Project** команду **Execute**. Если в ответ на запрос программы *Would you like to continue(Y/N)* вы нажмете клавишу [Y], а затем [Enter], на экране отобразится следующее:

-858993460

0

-858993460

-858993460

-858993460

-858993460

-858993460

Welcome to a trace demonstration! Would you like to continue (Y/N)y

На рис. 3.14 показано, что произойдет далее.

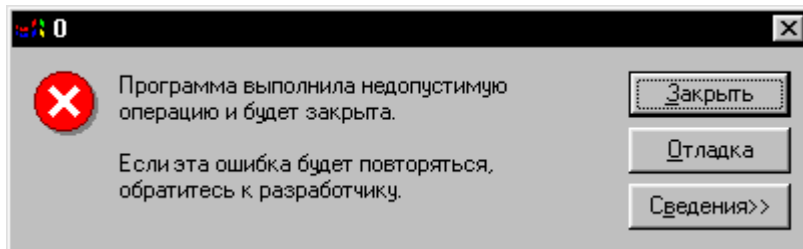


Рис. 3.14. Сообщение об ошибке выполнения программы

Использование встроенного отладчика

Созданная нами программа в начале своей работы отображает на экране исходное содержимое массива данных, после чего спрашивает, хотите ли вы продолжить работу. Ответ Y (yes— да) сигнализирует о том, что вы хотите заполнить массив собственными данными и отобразить их на экране.

Из рис. 3.14 можно сделать вывод о том, что хотя программный код набран совершенно правильно, т.е. в нем нет синтаксических ошибок, программа работает не так, как нам бы хотелось. Ошибки такого рода называются логическими. К счастью, встроенный в VisualC++ отладчик содержит ряд средств, которые послужат для вас спасательным кругом в подобной ситуации. Во-первых, вы можете выполнять программу пошагово, строка за строкой. Во-вторых, вам предоставляется возможность анализировать значения переменных в любой момент выполнения программы.

Разница между командами StepInto и StepOver

Когда вы начинаете процесс отладки, появляется панель инструментов **Debug**. Из множества представленных на ней кнопок наиболее часто задействуются **StepInto** (четвертая справа в верхнем ряду) и **StepOver** (третья справа). В обоих случаях программа будет запущена на выполнение в пошаговом режиме, а в тексте программы выделяется та строка, которая сейчас будет выполнена.

Различия между командами StepInto и StepOver проявляются только тогда, когда в программе встречается вызов функции. Если выбрать команду StepInto, то отладчик войдет в функцию и начнет выполнять шаг за шагом все ее операторы. При выборе команды StepOver отладчик выполнит функцию как единое целое и перейдет к строке, следующей за вызовом функции. Эту команду удобно применять в тех случаях, когда в программе делается обращение к стандартной функции или созданной вами подпрограмме, которая уже была протестирована.

Давайте выполним пошаговую отладку нашей программы.

Как видно из рис. 3.15, в окне редактирования появилась стрелка (ее называют индикатором трассировки), указывающая на строку программы, которая будет выполнена на следующем шаге. В данный момент она указывает на функцию `print_then()`.

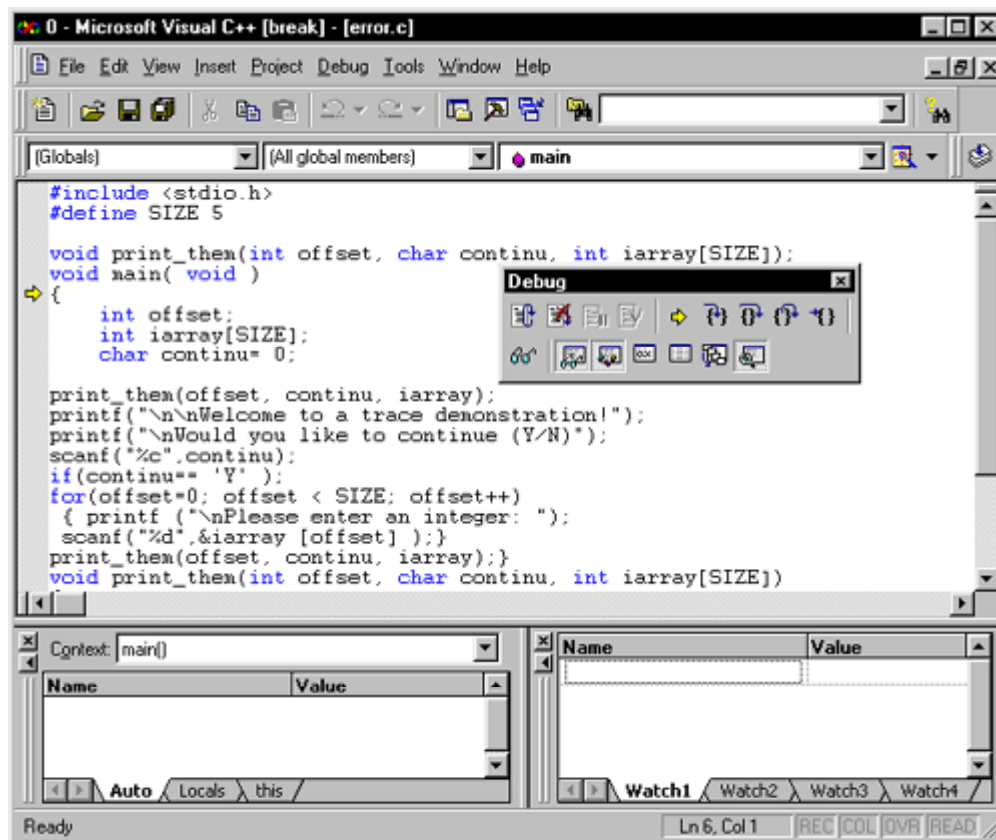


Рис. 3.15. Окно редактирования после того, как трижды была выполнена команда **StepInto** или **StepOver**

Имеет смысл выполнить эту функцию как одно целое. Для этого выберем команду `StepOver`. Функция будет выполнена, и индикатор трассировки укажет на первый вызов функции `printf()`. Теперь три раза нажмите клавишу `[F10]`, пока стрелка не остановится напротив функции `scanf()`.

В этот момент вам нужно перейти в окно программы и в ответ на приглашение `Would you like to continue(Y/N)` ввести `Y` и нажать `[Enter]` (рис. 3.16).

Сразу после этого на экране появится сообщение об ошибке (рис. 3.17).

Это сообщение было сгенерировано программой после попытки выполнить функцию `scanf()`. Давайте попытаемся разобраться, в чем, собственно, состоит проблема.

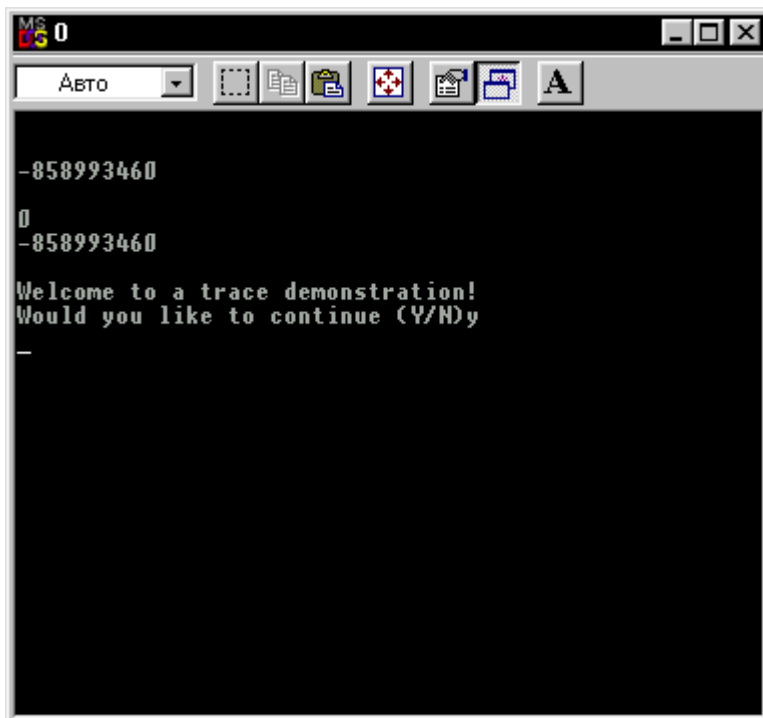


Рис. 3.16. Введите "Y" и нажмите [Enter], чтобы продолжить выполнение программы



Рис. 3.17. Отладчик сообщает об ошибке в программе

Ошибка связана с некорректным использованием функции `scanf()`. Функция `scanf()` ожидает указания адреса ячейки памяти для заполнения. Рассмотрим такое выражение:

```
scanf("%C", continu);
```

Как видите, здесь указывается не адрес переменной, а сама переменная. Чтобы указать адрес, нужно поместить оператор взятия адреса (&) перед `continu`. Внесите исправления в выражение, чтобы оно выглядело следующим образом:

```
scanf("%C", &continu);
```

Сохраните файл и вновь выберите команду **RebuildAll**.

Дополнительные средства отладки

Вы, очевидно, слышали о точках останова, которые применяются в программе при необходимости прервать ее выполнение в определенных местах. Смысл использования точек останова состоит в том, что отладчик не тратит времени на пошаговое выполнение программы вплоть до указанной точки, по достижении которой переходит в пошаговый режим.

Точки останова проще всего расставлять с помощью кнопки **Breakpoint** (первая справа) панели инструментов **Build**. Для этого достаточно установить курсор на нужной строке программы и щелкнуть на указанной кнопке. Если же выделенная строка уже содержит точку останова, то после щелчка на кнопке **Breakpoint** она, точка останова, будет удалена. При выборе команды **Go** программа будет выполняться от текущего местоположения курсора до ближайшей точки останова.

Обратимся к нашей программе. Мы знаем, что все строки программы до вызова функции `scanf()` отлично работают. Чтобы не тратить время на пошаговое выполнение всех строк,

которые уже были проверены ранее, поставим точку останова на 20-й строке, содержащей вызов функции `scanf()`.

Имеется и другой способ задания точек останова — с помощью диалогового окна **Breakpoints**(рис. 3.18), вызываемого командой **Breakpoints...** из меню **Edit**. По умолчанию при щелчке на кнопке со стрелкой открывается контекстное меню, в котором первым пунктом указывается команда создания точки останова на той строке, где в данный момент в окне редактирования находится курсор. В нашем случае это строка 20.

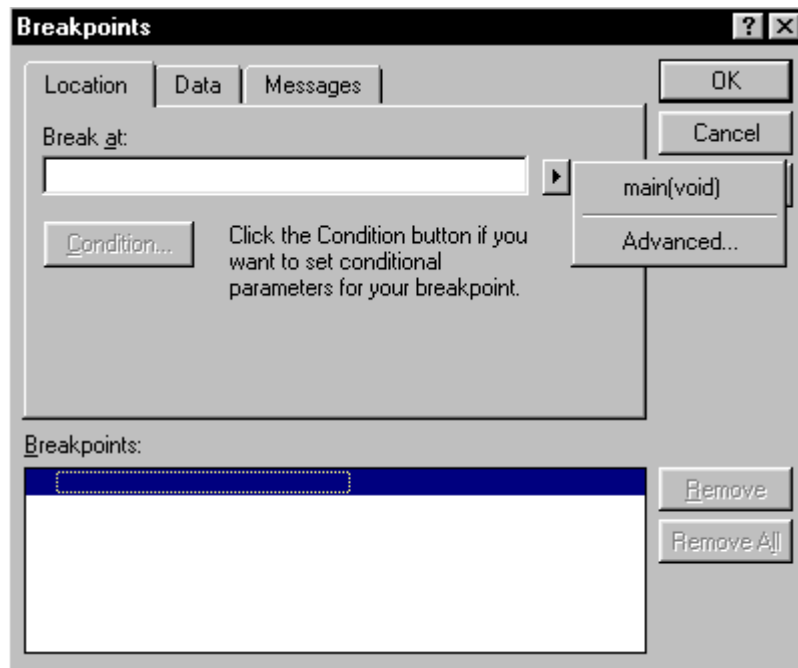


Рис. 3.18. Задание точки останова

Работа с точками останова

Предположим, что вы поставили точку останова в строке программы, содержащей вызов функции `scanf()`. Теперь выберите команду **Go**— либо из меню, либо нажав клавишу [F5]. Обратите внимание, что выполнение программы прерывается не на первой строке программы, а на строке, содержащей точку останова.

Далее можно продолжить выполнение программы в пошаговом режиме либо проанализировать текущие значения переменных. Нас интересует, будет ли функция `scanf()` работать корректно после того, как в программный код были внесены изменения. Выберите команду **StepOver**, перейдите к окну программы, введите букву Y в верхнем регистре и нажмите клавишу [Enter]. (Мы применили команду **StepOver** для того, чтобы избежать пошагового анализа отладчиком всех операторов функции `scanf()`. При выборе команды **StepIn** появляется предложение указать местонахождение файла `SCANF.C`)

Все отлично! Отладчик не выдал окна с сообщением об ошибке. Но означает ли это, что все проблемы разрешены? Чтобы ответить на этот вопрос, достаточно будет проанализировать текущее значение переменной `contin`.

Окно QuickWatch

Команда **QuickWatch...** открывает диалоговое окно **QuickWatch**(рис. 3.19), которое позволяет по ходу выполнения программы анализировать значения переменных. Простейший способ определить значение переменной с помощью данного окна состоит в том, что курсор помещается на имени переменной в окне редактирования, а затем нажимается комбинация клавиш [Shift+F9]. Прodelайте указанную операцию с переменной `contin`.

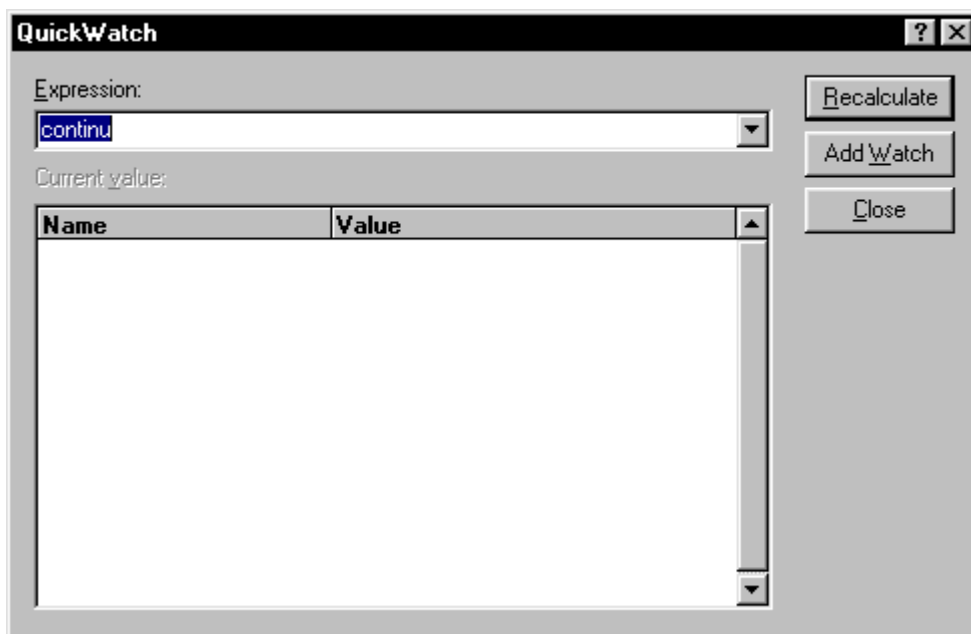


Рис. 3.19. Диалоговое окно **QuickWatch**

Теперь, когда мы определили, что переменная `continu` имеет правильное значение, можно продолжить выполнение программы до конца, выбрав в меню **Debug** команду **Go**(рис. 3.20).

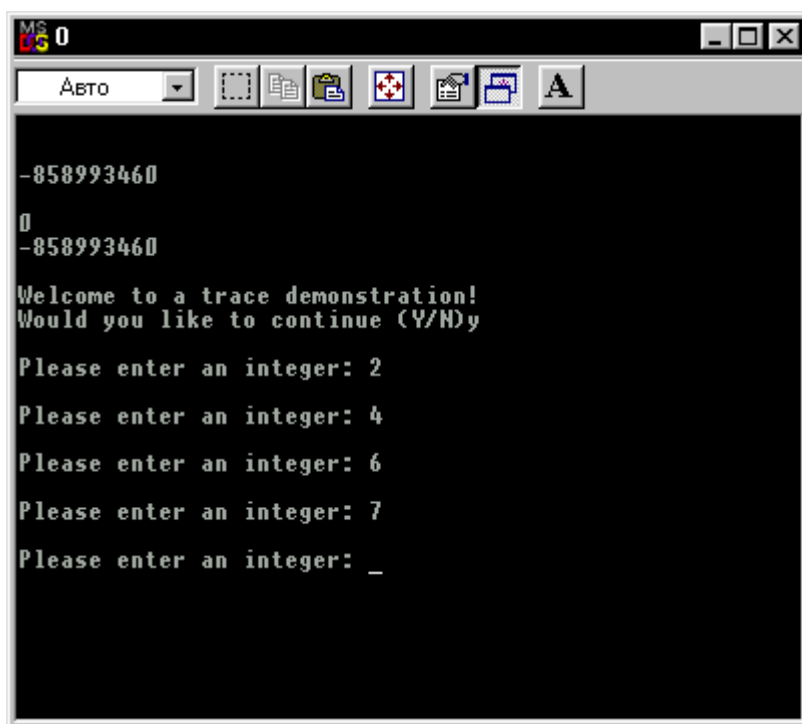


Рис. 3.20. Исправленная версия программы

Глава 4. Введение в С и С++

- Из истории языка С
 - Отличия С от других ранних языков высокого уровня
 - Достоинства языка С
 - Недостатки языка С
 - Язык С — не для любителей!
- Стандарт ANSI C
- Переход от С к С++ и объектно-ориентированному программированию
- Из истории языка С++
 - Эффективность объектно-ориентированного подхода
 - Незаметные различия между С и С++
 - Ключевые различия между С и С++
- Основные компоненты программ на языках С/С++
 - Простейшая программа на языке С
 - Простейшая программа на языке С++
 - Получение данных от пользователя в языке С
 - Получение данных от пользователя в языке С++
 - Файловый ввод-вывод

Знакомство с языками программирования С и С++ мы начнем с истории. Узнать историю появления языков С и С++ будет полезно, поскольку так нам легче будет понять концепции, положенные в основу этих языков, а также ответить на вопрос, почему С на протяжении вот уже ряда десятилетий остается столь популярным среди программистов, а его более молодой "родственник" С++ не уступает ему по популярности.

Приводимые далее примеры можно набирать и запускать в среде Microsoft Visual C++. Понимается, что в предыдущей главе вы научились выполнять элементарные операции по компиляции и отладке программ.

Из истории языка С

Наше исследование происхождения языка С начнется с операционной системы UNIX, поскольку она сама и большинство программ для нее написаны на С. Тем не менее, это не означает, что С предназначен исключительно для UNIX. Благодаря популярности UNIX язык С был признан в среде программистов как язык системного программирования, который можно использовать для написания компиляторов и операционных систем. В то же время он удобен для создания многих прикладных программ.

Операционная система UNIX была разработана в 1969 г. на маломощном, по современным представлениям, компьютере DEC PDP-7 в компании Bell Laboratories, город Мюррей Хилл, штат Нью-Джерси. Система была полностью написана на языке ассемблера для PDP-7 и претендовала на звание "дружественной для программистов", поскольку содержала довольно мощный набор инструментов разработки и являлась достаточно открытой средой. Вскоре после создания UNIX Кен Томпсон (Ken Thompson) написал компилятор нового языка В. С этого момента мы можем начать отслеживать историю языка С, поскольку язык В Кена Томпсона был непосредственным его предшественником. Рассмотрим родословную языка С:

Algol 60	Разработан международным комитетом в 1960 г.
-----------------	--

CPL	Combined Programming Language — комбинированный язык программирования. Разработан в 1963 г. группой программистов из Кембриджского и Лондонского университетов
BCPL	BasicCombinedProgrammingLanguage — базовый комбинированный язык программирования. Разработан в Кембридже Мартином Ричардсом (MartinRichards) в 1967 г.
B	Разработан в 1970 г. Кеном Томпсоном, компания BellLabs
C	Разработан в 1972 г. Деннисом Ритчи (DennisRitchie), компания BellLabs

Позже, в 1983 г., при Американском институте национальных стандартов (American National Standards Institute — ANSI) был создан специальный комитет с целью стандартизации языка C, в результате чего был разработан стандарт ANSI C.

Язык Algol 60 появился всего на пару лет позже языка FORTRAN. Это был значительно более мощный язык, который во многом предопределил пути дальнейшего развития большинства последующих языков программирования. Его создатели уделили много внимания логической целостности синтаксиса команд и модульной структуре программ, с чем, собственно, впоследствии и ассоциировались все языки высокого уровня. К сожалению, Algol 60 не стал популярным в США. Многие считают, что причиной тому была определенная абстрактность этого языка.

Разработчики языка CPL попытались приблизить "возвышенный" Algol 60 к реалиям конкретных компьютеров. Тем не менее, данный язык остался таким же трудным для изучения и практического применения, как и Algol 60, что предопределило его судьбу. В наследство от CPL остался язык BCPL, представляющий собой упрощенную версию CPL, сохранившую лишь основные его функции.

Когда Кен Томпсон взялся за разработку языка B для ранней версии UNIX, он попытался еще больше упростить язык CPL. И действительно, ему удалось создать довольно интересный язык, который эффективно работал на том оборудовании, для которого был спроектирован. Однако языки B и BCPL, очевидно, были упрощены больше, чем нужно. Их использование было ограничено решением весьма специфических задач.

Например, когда Кен Томпсон закончил создание языка B, появился новый компьютер PDP-11. Система UNIX и компилятор языка B были сразу же модернизированы под новую машину. Хотя PDP-11 был, несомненно, мощнее, чем его предшественник PDP-7, возможности этого компьютера оставались все еще слишком скромными по сравнению с современными стандартами. Он имел только 24 Кб оперативной памяти, из которых 16 Кб отводились операционной системе, и 512 Кб на жестком диске. Возникла идея переписать UNIX на языке B. Но B работал слишком медленно, поскольку оставался языком интерпретирующего типа. Была и другая проблема: язык B ориентировался на работу со словами, тогда как компьютер PDP-11 оперировал байтами. Стала очевидной необходимость усовершенствования языка B. Работа над более совершенной версией, которую назвали языком C, началась в 1971 г.

Деннис Ритчи, который известен как создатель языка C, частично восстановил независимость языка от конкретного оборудования, что было во многом утеряно в языках BCPL и B. Так, были успешно введены в практику типы данных и в то же время сохранена возможность прямого доступа к оборудованию — идея, заложенная еще в языке CPL.

Многие языки программирования, разработанные отдельными авторами (C, Pascal, Lisp и APL), отличались большей целостностью, чем те, над которыми трудились группы разработчиков (Ada, PL/I и Algol 60). Кроме того, для языков, разработанных одним автором, характерна большая специализированность в решении тех вопросов, в которых автор разбирался лучше всего. Деннис Ритчи был признанным специалистом в области системного программирования, а именно: языков программирования, операционных систем и генераторов программ. Учитывая профессиональную направленность автора, нетрудно понять, почему C был признан прежде всего разработчиками системных программ. Язык C представлял собой язык программирования относительно низкого уровня, что позволяло контролировать каждую мелочь в работе алгоритма и достигать максимальной эффективности. Но в то же время в C заложены принципы языка высокого уровня, что позволяло избежать зависимости программ от особенностей архитектуры конкретного компьютера. Это повышало эффективность процесса программирования.

Отличия C от других ранних языков высокого уровня

Вам, конечно, интересно узнать, какое место занимает C в ряду других языков программирования. Чтобы ответить на этот вопрос, рассмотрим примерную иерархию языков, показанную на рис. 4.1, где точками представлены промежуточные этапы развития. К примеру,

давние предки компьютеров, такие как станок Джакарда (Jacquard'sloom) (1805 г.) и счетная машина Чарльза Беббиджа (CharlesBabbage) (1834 г.)-, программировались механически для выполнения строго определенных функций. Не исключено, что в скором будущем мы сможем управлять компьютерами, посылая нейронные импульсы непосредственно из коры головного мозга, например подключив воспринимающие контакты к височной области мозга (участвующей в запоминании слов) или к области Брока (управляющей функцией речи).

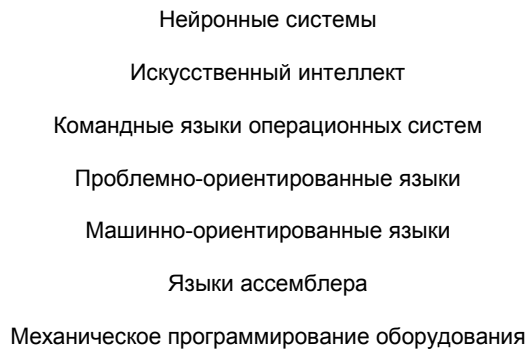


Рис. 4.1. Теоретические этапы развития языков программирования

Первые ассемблерные языки, которые появились вместе с электронными вычислительными машинами, позволяли работать непосредственно со встроенным набором команд компьютера и были достаточно простыми для изучения. Но они заставляли смотреть на проблему с точки зрения работы оборудования. Поэтому программисту приходилось постоянно заниматься перемещением байтов между регистрами, осуществлять их суммирование, сдвиг и, наконец, запись в нужные области памяти для хранения результатов вычислений. Это была очень утомительная работа с высокой вероятностью допущения ошибок.

Первые языки высокого уровня, например FORTRAN, разрабатывались как альтернатива языкам ассемблера. Они обеспечивали определенный уровень абстракции от аппаратной среды и позволяли строить алгоритм с точки зрения решаемой задачи, а не работы компьютера. К сожалению, их создатели не учли динамики развития компьютерной техники и в погоне за упрощением процесса программирования упустили ряд существенных моментов. Языки FORTRAN и Algol оказались слишком абстрактными для системных программистов. Эти языки были проблемно-ориентированными, рассчитанными на решение общих инженерных, научных и экономических задач. Программистам, занимающимся разработкой новых системных продуктов, по-прежнему приходилось полагаться только на старые языки ассемблера.

Чтобы разрешить эту проблему, разработчикам языков пришлось сделать шаг назад и создать класс машинно-ориентированных языков. К таким языкам низкого уровня относились BCPL и B. Но при работе с ними возникла другая проблема: они были приспособлены только для компьютеров определенной архитектуры. Язык C оказался шагом вперед по сравнению с машинно-ориентированными языками, сохранив при этом достаточно "низкий" уровень программирования в сравнении с большинством проблемно-ориентированных языков. Язык C позволяет контролировать процесс выполнения программы компьютером, игнорируя в то же время особенности аппаратной среды. Вот почему C рассматривается одновременно как язык программирования высокого и низкого уровней.

Достоинства языка C

Язык программирования часто можно определить, просто взглянув на исходный текст программы. Так, программа на языке APL напоминает иероглифы, текст на языке ассемблера представляется столбцами мнемоник, язык Pascal выделяется своим читабельным синтаксисом. А что можно сказать о языке C? Многие программисты, впервые столкнувшиеся с ним, находят его слишком замысловатым и пугающим. Конструкции, напоминающие выражения на английском языке, которые характерны для многих языков программирования, в C встречаются довольно редко. Вместо этого программист сталкивается с необычного вида операторами и обилием указателей. Многие возможности языка уходят своими корнями к особенностям программирования на компьютерах, существовавших на момент его появления. Ниже рассматриваются некоторые сильные стороны языка C.

Оптимальный размер программы

В основу C положено значительно меньше синтаксических правил, чем у других языков программирования. В результате для эффективной работы компилятора языка достаточно всего 256 Кб оперативной памяти. Действительно, список операторов и их комбинаций в языке C обширнее, чем список ключевых слов.

Сокращенный набор ключевых слов

Первоначально, в том виде, в каком его создал Деннис Ритчи, язык C содержал всего 27 ключевых слов. В ANSI C было добавлено несколько новых зарезервированных слов. В Microsoft C набор ключевых слов был еще доработан, и общее их число превысило 50.

Многие функции, представленные в большинстве других языков программирования, не включены в язык C. Например, в C нет встроенных функций ввода/вывода, отсутствуют математические функции (за исключением базовых арифметических операций) и функции работы со строками. Но если для большинства языков отсутствие таких функций было бы признаком слабости, то C взамен этого предоставляет доступ к самостоятельным библиотекам, включающим все перечисленные функции и многие другие. Обращения к библиотечным функциям в программах на языке C происходят столь часто, что эти функции можно считать составной частью языка. Но в то же время их легко можно переписать без ущерба для структуры программы — это безусловное преимущество C.

Быстрое выполнение программ

Программы, написанные на C, отличаются высокой эффективностью. Благодаря небольшому размеру исполняемых модулей, а также тому, что C является языком достаточно низкого уровня, скорость выполнения программ на языке C соизмерима со скоростью работы их ассемблерных аналогов.

Упрощенный контроль за типами данных

В отличие от языка Pascal, в котором ведется строгий контроль типов данных, в C понятие типа данных трактуется несколько шире. Это унаследовано от языка B, который так же свободно обращался с данными разных типов. Язык C позволяет в одном месте программы рассматривать переменную как символ, а в другом месте — как ASCII-код этого символа, от которого можно отнять 32, чтобы перевести символ в верхний регистр.

Реализация принципа проектирования "сверху вниз"

Язык C содержит все управляющие конструкции, характерные для современных языков программирования, в том числе инструкции for, if/else, switch/case, while и другие. На момент появления языка это было очень большим достижением. Язык C также позволяет создавать изолированные программные блоки, в пределах которых переменные имеют собственную область видимости. Разрешается создавать локальные переменные и передавать в подпрограммы значения параметров, а не сами параметры, чтобы защитить их от модификации.

Модульная структура

Язык C поддерживает модульное программирование, суть которого состоит в возможности раздельной компиляции и компоновки разных частей программы. Например, вы можете выполнить компиляцию только той части программы, которая была изменена в ходе последнего сеанса редактирования. Это значительно ускоряет процесс разработки больших и даже среднего размера проектов, особенно если приходится работать на медленных машинах. Если бы язык C не поддерживал модульное программирование, то после внесения небольших изменений в программный код пришлось бы компилировать всю программу целиком, что могло бы занять слишком много времени.

Возможность использования кодов ассемблера

Большинство компиляторов C позволяет обращаться к подпрограммам, написанным на ассемблере. В сочетании с возможностью раздельной компиляции и компоновки это позволяет легко создавать приложения, в которых используется код как высокого, так и низкого уровня. Кроме того, в большинстве систем из ассемблерных программ можно вызывать подпрограммы, написанные на C.

Возможность управления отдельными битами данных

Очень часто в системном программировании возникает необходимость управления переменными на уровне отдельных битов. Поскольку своими корнями язык С прочно связан с операционной системой UNIX, в нем представлен довольно обширный набор операторов побитовой арифметики.

Наличие указателей

Язык, используемый для системного программирования, должен предоставлять возможность обращаться к области памяти с заданным адресом, что значительно повышает скорость выполнения программы. В С эта возможность реализуется за счет использования указателей. Хотя указатели применяются и во многих других языках программирования, только С позволяет выполнять над указателями арифметические операции. Например, если переменная `student_record_ptr` указывает на первый элемент массива `student_records`, то выражение `student_record_ptr + 1` будет указывать на второй элемент массива.

Возможность гибкого управления структурами данных

Все массивы данных в языке С одномерны. Но С позволяет создавать конструкции из одномерных массивов, получая таким образом многомерные массивы.

Эффективное использование памяти

Программирование на языке С позволяет достаточно эффективно использовать память компьютера. Отсутствие встроенных функций дает программе возможность загружать только те библиотеки функций, которые ей действительно нужны.

Возможность кросс-платформенной переносимости

Важной характеристикой любой программы является возможность ее запуска на компьютерах разных платформ или с разными операционными системами. В современном компьютерном мире программы, написанные на языке С, отличаются наибольшей независимостью от платформы. Это особенно справедливо в отношении персональных компьютеров.

Наличие мощных библиотек готовых функций

Имеется множество специализированных коммерческих библиотек функций, доступных для любого компилятора С. Это библиотеки работы с графикой, базами данных, окнами, сетевыми ресурсами и многим другим. Использование библиотек функций позволяет значительно уменьшить время разработки приложений.

Недостатки языка С

Не существует абсолютно совершенных языков программирования. Дело в том, что различные задачи требуют различных решений. При выборе языка программирования программист должен исходить из того, какие именно задачи он собирается решать с помощью своей программы. Это первый вопрос, на который вы должны дать ответ еще до того, как начнете работу над программой, поскольку если впоследствии вы поймете, что выбранный язык не подходит для решения поставленных задач, то всю работу придется начинать сначала. От правильного выбора языка программирования в конечном счете зависит успех всего проекта. Ниже будут рассмотрены некоторые слабые стороны языка С.

Упрощенный контроль за типами данных!

То, что язык С не осуществляет строгого контроля за типами данных, можно считать как достоинством, так и недостатком. В некоторых языках программирования присвоение переменной одного типа значения другого типа воспринимается как ошибка, если при этом явно не указана функция преобразования. Благодаря этому исключается появление ошибок, связанных с неконтролируемым округлением значений.

Как было сказано выше, язык С позволяет присваивать целое число символьной переменной и наоборот. Это не проблема для опытных программистов, но для новичков данная особенность может оказаться одним из источников побочных эффектов. Побочными эффектами называются неконтролируемые изменения значений переменных. Поскольку С не осуществляет строгого контроля за типами данных, это дает большую гибкость при манипулировании переменными. Например, в одном выражении оператор присваивания (=) может использоваться несколько раз. Но это также означает, что в программе могут

встречаться выражения, тип результата которых неясен или трудно определить. Если бы C требовал однозначного определения типа данных, то это устранило бы появление побочных эффектов и неожиданных результатов, но в то же время сильно уменьшило бы мощь языка, сведя его к обычному языку высокого уровня.

Ограниченные средства управления ходом выполнения программы

Вследствие того, что выполнение программ на языке C слабо контролируется, многие их недостатки могут остаться незамеченными. Например, во время выполнения программы не поступит никаких предупреждающих сообщений, если осуществлен выход за границы массива. Это та цена, которую пришлось заплатить за упрощение компилятора ради его скорости и эффективности использования памяти.

Язык C — не для любителей!

Огромный набор предоставляемых возможностей — от побитового управления данными до форматированного ввода/вывода, а также относительная независимость и стабильность выполнения программ на разных машинах сделали язык C чрезвычайно популярным в среде программистов. Во многом благодаря тому, что операционная система UNIX была написана на языке C, она получила такое широкое распространение во всем мире и используется на самых разных компьютерах.

Тем не менее, как и всякое другое мощное средство программирования, язык C требует ответственности от программистов, использующих его. Программист должен крайне внимательно относиться к правилам и соглашениям, принятым в этом языке, и тщательно документировать программу, чтобы в ней мог разобраться другой программист, а так же сам автор по прошествии некоторого времени.

Стандарт ANSI C

Специальный комитет Американского института национальных стандартов (ANSI) занимался разработкой стандартов языка C. Давайте рассмотрим, какие изменения были предложены и внесены в язык программирования в результате работы этого комитета. Основные цели внесения изменений заключались в том, чтобы повысить гибкость языка и стандартизировать предлагаемые различными компиляторами возможности.

Ранее единственным стандартом языка C была книга Б. Кернигана и Д. Ритчи "Язык программирования C" (изд-во PrenticeHall, 1978 г.). Но эта книга не опускалась до описания отдельных технических деталей языка, что не гарантировало стандартности компиляторов C, создаваемых разными фирмами. Стандарт ANSI должен был устранить эту неоднозначность. Хотя некоторые из предложенных изменений могли привести к возникновению проблем при выполнении ранее написанных программ, ожидалось, что негативный эффект не будет существенным.

Введение стандарта ANSI должно было в еще большей степени, чем раньше, обеспечить совместимость языка C с различными компьютерными платформами. Хотя, конечно, внесенные изменения не смогли устранить всех противоречий, возникающих в процессе использования C-программ на разных компьютерах. По-прежнему в большинстве случаев, чтобы эффективно использовать имеющуюся программу на компьютере с другой архитектурой, в программный код должны быть внесены некоторые изменения.

Комитет ANSI также подготовил ряд положений в виде меморандума, которые выражали "дух языка C". Вот некоторые из этих положений.

- Доверяйте программистам
- Не ограничивайте возможность программиста делать то, что должно быть сделано.
- Язык должен оставаться компактным и простым.

В дополнение были предприняты усилия для того, чтобы обеспечить соответствие стандарта ANSI (американского) стандарту, предложенному ISO (International Standard Organization — Международная организация по стандартизации). Благодаря этим усилиям язык C оказался, пожалуй, единственным языком программирования, который эффективно применяется в разноязычных средах с различными кодовыми таблицами. В табл. 4.1. перечислены некоторые аспекты языка C, стандартизированные комитетом ANSI.

Таблица 4.1. Рекомендации комитета ANSI разработчикам компиляторов языка C

Аспект	Предложенные стандарты
Типы данных	Четыре: символьный, целочисленный, с плавающей запятой и перечисление
Комментарии	/ * — начало, * / — конец; добавлен — It:любой набор символов в строке справа будет игнорироваться компилятором
Длина идентификатора	31 символ; этого достаточно для обеспечения уникальности идентификатора
Стандартные идентификаторы и файлы заголовков	Разработан минимальный набор идентификаторов и файлов заголовков, необходимый для осуществления базовых операций, например ввода/вывода
Директивы препроцессора	Значку #, с которого начинается директива препроцессора, может предшествовать отступ (любая комбинация пробелов и символов табуляции), помогающий отличить директиву от остального программного кода; в некоторых ранних компиляторах существовало требование помещать директивы препроцессора только начиная с первой позиции строки
Новые директивы препроцессора	Определена конструкция if условие (выражение) #elif (выражение)
Запись выражений в несколько строк	Комитет принял решение, что смежные литералы должны объединяться; таким образом, выражение с оператором #define может быть записано в две строки
Стандартные библиотеки	В предложенном стандарте ANSI определен базовый набор внешних и системных функций, таких как read () и write ()
Управление выводом	Был согласован набор управляющих последовательностей, включающий символы форматирования, такие как разрыв строки, разрыв страницы и символ табуляции
Ключевые слова	Был согласован минимальный набор ключевых слов, необходимых для построения работоспособных выражений на языке C
size of ()	Комитет пришел к выводу, что оператор sizeof () должен возвращать значение типа size_t вместо системно-зависимой целочисленной переменной
Прототипы функций	Комитет постановил, что все компиляторы языка C должны поддерживать программы, как использующие, так и не использующие прототипы функций
Аргументы командной строки	Был согласован и утвержден единый синтаксис использования аргументов командной строки
Тип данных void	Ключевое слово void может использоваться в функциях, не возвращающих значения; для функции, возвращающей значение, результат может быть приведен к типу void: это служит указанием компилятору, что возвращаемое значение умышленно игнорируется
Использование структур	Отменено требование уникальности имен членов структур и объединений; структуры могут передаваться в виде аргументов функций и возвращаться функциями, а также присваиваться другим структурам того же типа
Объявления функций	Объявление функции может включать список типов аргументов, на основании которого компилятор определяет число и тип аргументов
Шестнадцатеричные числа	Шестнадцатеричное число должно начинаться с обозначения \x, за которым следует несколько шестнадцатеричных цифр (0-9, a-f, A-F); например, десятичному числу 16 соответствует Шестнадцатеричное \x10 (допускается также запись 0x10)
Триграммы	Триграммами называются последовательности символов, которые представляют клавиши, отсутствующие в некоторых клавиатурах; например, комбинацию символов ??<можно использовать вместо символа (

Переход от C к C++ и объектно-ориентированному программированию

Язык C++ можно рассматривать как надмножество для языка C. C++ сохранил все возможности, предоставляемые языком C, дополнив их средствами объектно-ориентированного программирования. Он позволяет решать задачи на достаточно высоком уровне абстракции, превосходя в этом отношении даже язык Ada, поддерживает модульное программирование, как в Modula-2, но сохраняет при этом простоту, компактность и эффективность языка C.

В этом языке органично сочетаются стандартные процедурные подходы, хорошо знакомые большинству программистов, и объектно-ориентированные методики, позволяющие находить чисто объектные решения поставленных задач. На практике в одном приложении на языке C++ можно использовать одновременно как процедурные, так и объектноориентированные принципы. Этот дуализм языка C++ представляет собой особую проблему для начинающих программистов, поскольку от них требуется не только изучить новый язык программирования, но и освоить новый стиль мышления и новые подходы к решению проблем.

Из истории языка C++

Вряд ли вас удивит тот факт, что своими корнями C++ восходит к языку C. В то же время C++ впитал в себя многие идеи, реализованные не только в языках BCPL и Algol 60, но и в Simula 67. Возможность перегрузки операторов и объявления переменных непосредственно перед их первым использованием сближает C++ с языком Algol 60. Концепция подклассов (или производных классов) и виртуальных функций была заимствована из Simula 67. Впрочем, все популярные языки программирования представляют собой набор усовершенствованных средств и функций, взятых из других, более ранних языков программирования. Но, безусловно, ближе всего язык C++ стоит к языку C.

Язык C++ был разработан в начале 80-х в Bell Labs Бьярном Страуструпом (Bjarne Stroustrup). Сам доктор Страуструп утверждает, что название C++ было предложено Риком Масситти (Rick Mascitti). C++ изначально был создан для целей разработки некоторых высокоточных событийных моделей, поскольку необходимая эффективность не могла быть достигнута с помощью других языков программирования.

Впервые C++ был использован вне стен лаборатории доктора Страуструпа в 1983 г., но еще до лета 1987 г. шли работы по отладке и совершенствованию этого языка.

При создании языка C++ особое внимание уделялось сохранению совместимости с языком C. Необходимо было сохранить работоспособность миллионов строк программных кодов, написанных и скомпилированных на C, а также сохранить доступ ко множеству разработанных библиотек функций и средств программирования языка C. Надо отметить, что в этом направлении были достигнуты значительные успехи. Во всяком случае многие программисты утверждают, что преобразование программных кодов от C к C++ выполняется значительно проще, чем это происходило ранее, например, при переходе от FORTRAN к C.

С помощью C++ можно создавать широкомасштабные программные проекты. Благодаря тому, что в языке C++ усилен контроль за типами данных, удалось преодолеть многие побочные эффекты, характерные для языка C.

Но наиболее важным приобретением языка C++ все-таки является объектно-ориентированное программирование (ООП). Чтобы воспользоваться всеми преимуществами C++, вам придется изменить привычные подходы к решению проблем. Основной задачей теперь становится определение объектов и связанных с ними операций, а также формирование классов и подклассов.

Эффективность объектно-ориентированного подхода

Остановимся ненадолго на объектно-ориентированном программировании, чтобы на примере показать, как использование абстрактных объектов языка C++ может облегчить решение прикладных задач по сравнению с более старыми процедурными языками. Предположим, например, что нам нужно написать программу на языке FORTRAN для оперирования таблицей успеваемости студентов. Чтобы решить эту задачу, необходимо будет создать ряд массивов, представляющих различные поля таблицы. Все массивы должны быть связаны посредством общего индекса. Для создания таблицы из десяти полей необходимо запрограммировать доступ к десяти массивам с помощью единого индекса, чтобы данные из разных массивов возвращались как единая запись таблицы.

В C++ решение этой задачи сводится к объявлению простого объекта `student_database`, способного принимать сообщения `add_student`, `delete_student`, `access_student` и `display_student` для оперирования данными, содержащимися в объекте. Обращение к объекту `student_database` реализуется очень просто. Допустим, нужно добавить новую запись в таблицу. Для этого достаточно ввести в программу следующую строку:

```
student_database . add_student (new__recruit)
```

В данном примере функция `add__student ()` является методом класса, связанного с объектом `student_database`, а параметр `new_recruit` представляет собой набор данных, добавляемый в таблицу. Обратите внимание на то, что класс объектов `student_database` не является встроенным типом данных языка C++. Наоборот, мы расширили язык программирования собственными средствами, предназначенными для решения конкретной задачи. Возможность создавать новые классы или модифицировать существующие (порождать от них подклассы) позволяет программисту естественным образом устанавливать соответствие между понятиями предметной области и средствами языка программирования.

Незаметные различия между C и C++

Ниже мы рассмотрим некоторые отличия языка C++ от языка C, не связанные с объектным программированием.

Синтаксис комментариев

В C++ комментарии могут вводиться с помощью конструкции `//`, хотя старый метод маркирования блока комментариев посредством символов `/*` и `*/` по-прежнему допустим.

Перечисления

Имя перечисления является названием типа данных. Это упрощает описание перечислений, поскольку устраняет необходимость в указании ключевого слова `enum` перед именем перечисления.

Структуры и классы

Имена структур и классов являются названиями типов данных. В языке C понятие класса отсутствует. В C++ нет необходимости перед именами структур и классов указывать ключевое слово `struct` или `class`.

Область видимости в пределах блока

Язык C++ дает возможность объявлять переменные внутри блоков программного кода, т.е. непосредственно перед их использованием. Переменная цикла может объявляться даже непосредственно внутри инициализатора цикла, как в следующем примере:

```
// Объявление переменной непосредственно в месте ее использования
for(int index = 0; index < MAX_ROWS; index++)
```

Оператор расширения области видимости

Для разрешения конфликтов, связанных с именами переменных, введен оператор `::`. Например, если некоторая функция имеет локальную переменную `vector_location` и существует глобальная переменная с таким же именем, то выражение `::vector_location` позволяет обратиться к глобальной переменной в пределах указанной функции.

Ключевое слово const

С помощью ключевого слова `const` можно запретить изменение значения переменной. С его помощью можно также запретить модификацию данных, адресуемых указателем, и даже модификацию значения адреса, хранящегося в самом указателе.

Безымянные объединения

Безымянные объединения могут быть описаны всюду, где допускается объявление переменной или поля. Эта возможность обеспечивает экономичное использование памяти, поскольку к одной и той же области памяти могут получать доступ несколько полей одной структуры.

Явные преобразования типов

Существует возможность использования имен встроенных или пользовательских типов данных в качестве функций преобразования. В некоторых случаях удобнее использовать явное преобразование, чем обычную операцию приведения типов.

Уникальные особенности функций

Язык C++ понравится программистам, работавшим ранее с языками Pascal, Modula-2 и Ada, поскольку позволяет задавать имена и типы параметров функции прямо внутри круглых скобок, следующих за именем функции. Например:

```
void* vfunc(void *dest, int c, unsigned count)
{
    .
    .
    .
}
```

```
}
```

В языке C после принятия стандарта ANSI также появилась возможность использования таких выражений. Таким образом, стандарт ANSI оказал влияние на создателей языка C++.

Транслятор языка C++ проверит соответствие фактических типов значений, переданных в функцию, формальным типам аргументов функции. Также будет проверено соответствие типа возвращаемого значения типу переменной, которой присваивается это значение. Подобная проверка типов не предусмотрена в большинстве версий языка C.

Перегрузка функций

В C++ можно использовать одинаковые имена для нескольких функций. Обычно разные функции имеют разные имена. Но иногда требуется, чтобы одна и та же функция выполняла сходные действия над объектами различных типов. В этом случае имеет смысл определить несколько функций с одинаковым именем, но разным телом. Такие функции должны иметь отличающиеся наборы аргументов, чтобы компилятор мог различать их. Ниже показан пример объявления перегруженной функции с именем `total()`, принимающей в качестве аргументов массивы чисел типа `int`, `float` и `double`.

```
int total(int isize, int iarray[]);
float total(int isize, float farray[]);
double total(int isize, double darray[]);
.
.
```

Несмотря на то что три разные функции имеют одно имя, по типу аргументов компилятор легко сможет определить, какую версию функции следует вызвать в каждом конкретном случае:

```
total( isize, iarray);
total( isize, farray);
total( isize, darray);
```

Стандартные значения параметров функций

В C++ можно задавать параметрам функций значения по умолчанию. В таком случае при вызове функции могут быть указаны значения только некоторых параметров, тогда как остальным они будут назначены автоматически.

Списки аргументов переменного размера

В C++ с помощью многоточия (...) могут быть описаны функции с неопределенным набором параметров. Контроль за типами параметров таких функций не ведется, что повышает гибкость их использования.

Во вторую редакцию языка C данная возможность также была включена. В этом смысле язык C++ оказал влияние на язык C.

Использование ссылок на аргументы функций

С помощью оператора `&` можно задать передачу аргументов функции по ссылке, а не по значению. Например:

```
int i;
increment(i);
void increment(int &variable_reference)
{
    variable_reference++;
}
```

Поскольку параметр `variable_reference` определен как ссылка, его адрес присваивается адресу переменной `i` при вызове функции `increment()`. Последняя выполняет приращение значения параметра, записывая его в переменную `i`. При этом, в отличие от языка C, нет необходимости в явной передаче адреса переменной `i` функции `increment()`.

Макроподстановка функций

Ключевое слово `inline` говорит о том, что при раскрытии вызова функции компилятор должен не записывать ссылку на нее, а выполнять подстановку ее кода целиком, что в случае небольших функций повышает быстроедействие программы.

Оператор `new` и `delete`

Новые операторы `new` и `delete`, добавленные в язык C++, позволяют выделять и удалять в программе динамические области памяти.

Указатели типа `void`

В C++ тип `void` используется для обозначения того, что функция не возвращает никаких значений. Указатель, имеющий тип `void`, может быть присвоен любому другому указателю базового типа.

Ключевые различия между C и C++

Наиболее существенное отличие C++ от языка C состоит в использовании концепции объектно-ориентированного программирования. Ниже рассмотрены связанные с этим новые средства языка C++.

Классы и инкапсуляция данных

Классы являются фундаментальной концепцией объектно-ориентированного программирования. Определение класса включает в себя объявления всех полей, возможно, с начальными значениями, а также описания функций, предназначенных для манипулирования значениями полей и называемых методами. Объекты являются переменными типа класса. Каждый объект может содержать собственные наборы закрытых и открытых данных.

Структуры в роли классов

Структура представляет собой класс, все члены которого являются открытыми, то есть нет закрытых и защищенных разделов. Такой класс может содержать как поля данных (что предполагается в стандарте ANSI C), так и функции.

Конструкторы и деструкторы

Конструкторы и деструкторы являются специальными методами, предназначенными для создания и удаления объектов класса. При объявлении объекта вызывается конструктор инициализации. Деструктор автоматически удаляет из памяти указанный объект, когда тот выходит за пределы своей области видимости.

Сообщения

Основным средством манипуляции объектами являются сообщения. Сообщения посылаются объектам (переменным типа класса) посредством особого механизма, напоминающего вызов функции. Набор сообщений, которые могут посылаться объектам класса, задается при описании этого класса. Каждый объект отвечает на полученное сообщение, выполняя определенные действия. Например, если есть объект `Palette_Colors`, для которого задан метод `SetNumColors_Method`, требующий указания одного целочисленного параметра, то передача объекту сообщения будет выглядеть следующим образом:

```
Palette_Colors.SetNumColors_Method(16);
```

Дружественные функции

Концепция инкапсуляции данных внутри класса делает невозможным доступ к <<внутренним данным объекта извне. Другими словами, закрытые члены классов недоступны функциям, не являющимся членами этого класса. Но в C++ предусмотрена возможность объявления внешних функций и классов друзьями определенного класса. Дружественные функции, не являясь членами класса, получают доступ к описанным в нем переменным и методам.

Перегрузка операторов

Уникальной особенностью C++ является возможность изменения смысла большинства базовых операторов языка, что позволяет применять их к объектам различных классов, а не только к данным стандартных типов. С помощью ключевого слова `operator` в класс добавляется

функция, имя которой совпадает с именем одного из базовых операторов. Впоследствии эта функция может вызываться без скобок, точно так же, как и обычный оператор. Компилятор отличит "настоящий" оператор от "ненастоящего" на основании типа операндов, так как перегруженные операторы могут вызываться только для объектов классов.

Производные классы

Производный класс можно рассматривать как подкласс некоторого базового класса. Возможность порождения новых классов позволяет формировать сложные иерархические модели. Объекты производного класса наследуют все открытые переменные и методы родительского класса, но в дополнение к ним могут содержать собственные переменные и методы.

Полиморфизм и виртуальные функции

Чтобы понять смысл термина полиморфизм, рассмотрим древовидную иерархию родительского класса и порожденных от него подклассов. Каждый подкласс в этой структуре может получать сообщения с одинаковым именем. Когда объект какого-либо подкласса принимает такое сообщение, он на основании типа и количества переданных параметров определяет, к какому классу относится данное сообщение, и предпринимает соответствующие действия. Если сообщения сразу нескольких классов в иерархии имеют одинаковый формат, считается, что сообщение относится к ближайшему классу в иерархии. Методы родительского класса, которые могут переопределяться в подклассах, называются виртуальными и создаются с указанием ключевого слова `virtual`.

Потоковые классы

Язык C++ содержит дополнительные средства ввода/вывода. Три объекта `cin`, `cout` и `cerr`, подключаемые к программе посредством файла `iostream.h`, служат для выполнения операций консольного ввода и вывода. Все операторы классов этих объектов могут быть перегружены в производных классах, создаваемых пользователем. Благодаря этой возможности операции ввода/вывода можно легко настраивать в соответствии с особенностями работы конкретного приложения.

Основные компоненты программ на языках C/C++

Возможно, вам приходилось слышать, что язык C очень трудно изучать. Действительно, первое знакомство с программой на языке C может поставить вас в тупик, но виной тому не сложность языка, а его несколько необычный синтаксис. К концу этой главы вы получите достаточно информации, чтобы легко разбираться в синтаксисе языка C и даже создавать небольшие, но работоспособные программы.

Простейшая программа на языке C

Ниже показан пример простейшей программы на языке C. Советуем вам вводить программы по мере чтения, чтобы наглядно представлять, как они работают.

```
/*
 *      simple.c.
 *      Ваша первая программа на C.
 */
#include <stdio.h>
int main ()
{
    printf("Здравствуй, мир! ");
    return(0);
}
```

В этом маленьком тексте программы скрыто много интересного. Начнем с блока комментариев:

```
/*
 *      simple.c
 *      Ваша первая программа на C.
 */
```

Любая программа, написанная профессиональным программистом, всегда начинается с комментариев. В языке C блок комментариев начинается символами `/*`, а заканчивается символами `*/`. Все, что находится между ними, игнорируется компилятором.

Следующая строка, называемая директивой препроцессора, характерна для языка C.

```
#include <stdio.h>
```

Директивы препроцессора — это своего рода команды компилятора. В данном случае компилятор получает указание поместить в этом месте программы код, хранящийся в библиотечном файле `STDIO.H`. Файлы с расширением `H` называются файлами заголовков и обычно содержат объявления различных констант и идентификаторов, а также прототипы функций. Хранение такого рода информации в отдельном файле облегчает доступ к ней из разных программ и улучшает структурированность программы.

Вслед за директивой препроцессора расположен блок описания функции:

```
int main ()
return(0); /* или return 0; */
```

Все программы на языке C обязательно содержат функцию `main ()`. С нее начинается выполнение программы, которое завершается вызовом инструкции `return`. Ключевое слово `int` слева от имени функции указывает на тип возвращаемых ею значений. В нашем случае возвращается целое число. Значение 0 в инструкции `return` будет воспринято как признак успешного завершения программы. Допускается использование инструкции `return` без круглых скобок.

Тело функции `main ()` расположено между символами `{` и `}`, называемыми фигурными скобками. Фигурные скобки широко применяются для выделения в программе блоков инструкций. Это может быть тело функции, как в данном примере, тело цикла, например `for` или `while`, либо операторная часть условных конструкций `if/else` или `switch/case`.

В нашем случае тело функции `main ()`, помимо стандартного вызова инструкции `return`, состоит из единственной команды, осуществляющей вывод на экран строки приветствия:

```
printf("Привет, юзер!");
```

Прототип функции `printf ()` описан в файле `STDIO.H`.

Простейшая программа на языке C++

В следующем примере мы реализуем те же самые действия, но на этот раз средствами языка C++.

```
//
//  simple.cpp
//  Ваша первая программа на C++
#include <iostream.h>
int main ()
{
cout << "    Здравствуй,    мир!    ";
return(0);
}
```

Имеется три различия между этой программой и той, что мы рассмотрели ранее. Во-первых, комментарии выделены не парой символов `/* */`, а символами `//`, расположенными в каждой строке блока комментариев. Во-вторых, имя файла в директиве `include` было изменено на `IOSTREAM.H`. И наконец, вывод данных осуществляется посредством объекта `cout`, который появился в программе "благодаря" файлу `IOSTREAM.H`. В следующих примерах книги мы будем обращать ваше внимание на другие отличия языка C++ от C.

Получение данных от пользователя в языке C

Следующий пример немного более сложен. Данная программа не только выводит информацию, но и предлагает пользователю ввести свои данные.

```
/*
*  ui.c
```

```

* Данная программа предлагает пользователю ввести длину в футах,
* после
* чего переводит полученное значение в метры и сантиметры.
*/
#include <stdio.h>
int main ()
{
float feet, meters, centimeters;
printf("Введите количество футов: ") ;
scanf("%f",&feet);
while(feet > 0) {
    centimeters = feet * 12 * 2.54;
    meters = centimeters/100; printf("%8.2f(футы)
равняется\n", feet);
    printf("%8.2f(метры) \n",meters);
    printf("%8.2f(сантиметры) \n",centimeters);
    printf("\nВведите другое значение \n");
    printf("(0- конец программы): ") ;
    scanf("%f", &feet);
}
printf(">>>До свидания! <<<") ;
return(0);
}

```

Объявление переменных

Первое, что бросается в глаза в этом примере, — объявление переменных:

```
float feet, meters, centimeters;
```

В языке C все переменные должны быть объявлены до того, как на них будет осуществлена ссылка где-либо в программе. Ключевое слово `float` перед именами переменных говорит о том, что им назначается стандартный тип данных языка C — действительное число с плавающей запятой одинарной точности.

Ввод данных пользователем

Следующие строки, вид которых может показаться довольно необычным, обеспечивают ввод данных пользователем:

```
printf("Введите количество футов: ");
scanf("%f",&feet);
```

Функция `scanf()` должна содержать строку форматирования, которая определяет, в каком порядке будут вводиться внешние данные и как они будут интерпретироваться программой. Заключенный в кавычки параметр `%f` указывает, что вводимые данные будут приведены к типу `float`. В языках C и C++ для значений этого типа отводится 4 байта. (Более подробно о различных типах данных, существующих в C/C++, см. в следующей главе.)

Оператор взятия адреса

Обратите внимание, что в рассмотренном выше примере переменной `feet` в функции `scanf()` предшествует символ амперсанда (&). Это оператор взятия адреса. Всюду, где имени переменной предшествует этот оператор, компилятор будет использовать вместо значения переменной ее адрес. Особенность функции `scanf()` заключается в том, что она ожидает именно адрес переменной, которой будет присвоено новое значение.

Простейший цикл while

Один из самых простых способов создания цикла в программе на языке C заключается в использовании инструкции `while`:

```
while(feet > 0)
```

```
{
...
}
```

Это так называемый цикл предусловием. Он начинается с ключевого слова `while`, за которым следует логическое выражение, возвращающее `TRUE` или `FALSE`. Фигурные скобки необходимы, когда цикл содержит более одной команды. В противном случае скобки необязательны. Строки, заключенные между фигурными скобками, формируют тело цикла.

Старайтесь придерживаться общепринятого стиля оформления циклов и условных конструкций наподобие `if/else`. Хотя для компилятора не имеет значения наличие символов пробела, табуляции и пустых строк (в процессе компиляции они все равно будут отброшены), следует помнить о некоторых общих правилах форматирования текста программы, чтобы сделать ее удобочитаемой для других. Так, при выделении тела цикла с предусловием открывающая фигурная скобка ставится после закрывающей круглой скобки условного выражения, в той же строке, а закрывающая скобка — после всех инструкций цикла в отдельной строке, причем с тем же отступом, что и первая строка цикла.

Вывод данных на экран

В рассматриваемом примере мы повстречались с более сложным способом вывода информации на экран:

```
printf("%8.2f(футы) равняется\n", feet);
printf("%8.2f(метры) \n", meters);
printf("%8.2f(сантиметры) \n", centimeters);
printf("\nВведите другое значение' \n");
printf("(0— конец программы): ");
```

Принцип вывода заключается в использовании строк форматирования, которые применяются всегда, когда функция `printf()` выводит не только литералы (наборы символов, заключенные в двойные кавычки), но и значения переменных. В строках форматирования определяется тип данных выводимых переменных, а также способ их представления на экране.

Давайте рассмотрим элементы строки форматирования первой из функций `printf()`.

Элемент строки	Назначение форматирования
<code>%8.2f</code>	Задаёт интерпретацию переменной <code>feet</code> как числа типа <code>float</code> в следующем формате: 8 символов до запятой и 2 символа после
<code>(футы) равняется</code>	После вывода значения переменной <code>feet</code> будет сделан пробел и отображена строка (футы) равняется
<code>\n</code>	Символ новой строки
<code>,</code>	Запятая отделяет строку форматирования от списка обозначенных в ней переменных

Расшифровка строк форматирования двух следующих функций `printf()` аналогична приведенной. Каждая функция выводит на экран отформатированное значение соответствующей переменной и строку символов, завершающуюся символом новой строки. В результате работы программы на экран будет выведено следующее:

**Введите количество футов: 10 10.00 (футы) равняется
3.05(метры)
304.80 (сантиметры)
Введите другое значение (0 — конец программы): 0**

При выводе данных можно в строках форматирования указывать так называемые управляющие последовательности наподобие символа новой строки в рассматриваемом примере. Все они перечислены в табл. 4.2. В случае записи ASCII-кода символа в восьмеричном/шестнадцатеричном представлении ведущие нули игнорируются компилятором, а сама последовательность завершается, когда встречается не используемый в данной системе счисления символ или четвертый/третий подряд восьмеричный/шестнадцатеричный символ, не считая ведущих нулей.

Таблица 4.2. Управляющие последовательности

Последовательность	Что обозначает
<code>\a</code>	Предупреждающий звуковой сигнал
<code>\b</code>	Стирание предыдущего символа

\f	Перевод страницы
\n	Новая строка
\r	Возврат каретки
\t	Горизонтальная табуляция
\v	Вертикальная табуляция
\?	Знак вопроса
\'	Одинарная кавычка
\"	Двойная кавычка
\\	Обратная косая черта
\ddd	ASCII-код символа в восьмеричной системе
\xdd	ASCII-код символа в шестнадцатеричной системе

Получение данных от пользователя в языке C++

Ниже показана версия предыдущего примера, переписанная в соответствии с синтаксисом языка C++:

```
//
//ui.cpp
//Данная программа предлагает пользователю ввести длину в футах, после
//чего переводит полученное значение в метры и сантиметры.
//
#include <iostream.h>
#include <iomanip.h>
int main()
float feet, meters, centimeters;
cout << "Введите количество футов: ";
cin >> feet;
while(feet > 0)
{
centimeters = feet * 12 * 2.54;
meters = centimeters/100;
cout << setw(8) << setprecision(2)
<< setiosflags(ios::fixed) << feet
<< " (футы) равняется \n"; cout << setw(8)
<< meters << " (метры) \n"; cout << setw(8)
<< centimeters << " (сантиметры) \n";
cout << "\nВведите другое значение\n";
cout << "(0- конец программы): ";
cin >> feet;
}
cout << ">>> До свидания! <<<";
return(0);
```

Можно обнаружить пять основных различий между показанной программой на языке C++ и ее аналогом на С. Первые два состоят в применении объектов cin и cout вместо функций scanf () и printf (). В выражениях с ними также используются операторы << (для вывода) и >> (для ввода) классов ostream и istream, подключаемых в файле IOSTREAM. H. Оба этих оператора являются перегруженными и поддерживают ввод/вывод данных всех базовых типов. Их можно также перегрузить для работы с пользовательскими типами данных.

Оставшиеся три различия связаны с особенностями форматированного вывода в C++. Чтобы добиться такого же формата вывода, какой был получен с помощью простой строки форматирования %8.2f в программе на языке С, в C++ потребуются три дополнительных выражения. Файл IOMANIP.H, подключаемый в начале программы, содержит объявления трех функций, являющихся членами класса ios (базовый в иерархии классов ввода/вывода): setw(), setprecision () и setios-flags (). Функция setw () задает минимальную ширину (в символах) выводимого поля. Функция setprecision () задает число цифр после десятичной точки при выводе чисел с плавающей запятой. Функция setw () вынуждена повторяться три раза, поскольку определяет формат вывода только следующей за ней переменной, после чего все сделанные установки сбрасываются. В отличие от нее функция setiosflags () вызывается один

раз, устанавливая флаг `fixed`, который задает вывод чисел с плавающей запятой в фиксированном формате, т.е. без экспоненты. Те программисты, которые работают с языком C++, но хотят использовать привычную функцию `printf()`, могут подключить библиотеку `STDIO.H`.

Файловый ввод-вывод

Часто требуется вводить данные не с клавиатуры, а из файла и не выводить полученный результат на экран, а записывать его в файл. Ниже показан простейший пример того, как организовать в программе работу с файлами:

```
/*
 * file1.c
 * Эта программа на языке C демонстрирует использование файлов как
 * для
 * ввода, так и для вывода данных. Программа читает значение
 * переменной
 * forder_price из файла customer.dat, вычисляет значение переменной
 * fbilling_price и записывает его в файл billing.dat.
 */
#include <stdio.h> #define MIN_DISCOUNT .97
#define MAX_DISCOUNT .95
int main ()
{
    float forder_price, fbilling_price;
    FILE *fin,*fout;
    fin = fopen("customer.dat","r"); fout = fopen("billing.dat","w");
    while (fscanf(fin,"%f",&forder_price) != EOF) {
        fprintf(fout,"Для заказа на сумму \t$%8.2f\n",
            forder_price); if (forder_price < 10000)
            fbilling_price = forder_price * MIN_DISCOUNT;
        else fbilling_price = forder_price * MAX_DISCOUNT;
        fprintf(fout,"цена со скидкой равна \t$%8.2f\n\n",
            fbilling_price);
    }
    return(0);
}
```

Каждому файлу в программе соответствует свой указатель типа `file`. Структура `file`, описанная в библиотеке `STDIO.H`, содержит всевозможную информацию о файле, включая путь к нему, имя и атрибуты. В приводимом ниже выражении создаются два указателя файла:

```
FILE *fin, *fout;
```

В следующих двух строках файлы открываются для чтения и записи соответственно:

```
fin = fopen ("customer.dat","r"); fout = fopen("billing.dat","w");
```

Каждая функция `fopen()` возвращает инициализированный указатель файла. В процессе выполнения программы их значения не должны меняться вручную.

Второй параметр функции `fopen()` определяет режим доступа к файлу (табл. 4.3). Файл может быть также открыт в следующих режимах: текстовом (включается путем добавления символа `t` к обозначению режима доступа) и двоичном (включается путем добавления символа `b`). В текстовом режиме компилятор языка C в процессе ввода данных заменяет пары символов возврата каретки и перевода строки одним символом новой строки. При выводе выполняется обратное преобразование. В двоичных файлах эти символы никак не обрабатываются.

Таблица 4.3. Режимы доступа к файлу в языке C	
Режим доступа	Описание
a	Файл открывается для добавления данных; если файл не существует, он создается; все новые данные добавляются в конец файла
a+	Аналогичен предыдущему режиму, но допускает считывание данных

r	Открывает файл только для чтения; если файл не существует, открытия файла не происходит
r+	Открывает файл как для чтения, так и для записи; если файл не существует, открытия файла не происходит
w	Открывает пустой файл для записи; если файл существует, его содержимое стирается
w+	Открывает пустой файл как для записи, так и для чтения; если файл существует, его содержимое стирается

Режимы r+, w+ и a+ позволяют как читать данные из файла, так и осуществлять их запись. При переходе от считывания к записи не забудьте модифицировать текущую позицию указателя файла с помощью функций fsetpos (), fseek() или rewind() .

f В языке C нет необходимости закрывать файлы, так как все открытые файлы закрываются автоматически по завершении программы. Впрочем, иногда требуется самостоятельно управлять этим процессом. Ниже показана версия предыдущего примера с добавлением функций закрытия файлов:

```
/*
 *   file1.c
 *   Эта программа на языке C демонстрирует использование файлов как
для
 *   ввода, так и для вывода данных. Программа читает значение
переменной
 *   forder_price из файла customer.dat, вычисляет значение переменной
 *   fbilling_price и записывает его в файл billing.dat.
 */
#include <stdio.h>
#define MIN_DISCOUNT .97
#define MAX_DISCOUNT .95
int main() {
float forder_price, fbilling_price;
FILE *fin,*fout;
fin = fopen ("customer.dat","r"); fout = fopen("billing.dat","w");
while (fscanf (fin,"%f",&forder_price) != EOF)
{ fprintf(fout,"Для заказа на сумму \t$%8.2f\n",
forder_price); if (forder_price < 10000)
fbilling_price = forder_price * MIN_DISCOUNT;
else fbilling_price = forder_price * MAX_DISCOUNT; fprintf(fout,"цена
со скидкой равна \t$%8.2f\n\n", fbilling_price);}
fclose(fin); fclose(fout) ;
return(0);}
```

Следующая программа, написанная на языке C++, выполняет те же функции, что и рассмотренная нами программа на языке C.

```
//
//   file2.cpp
//   Эта программа на языке C++ демонстрирует использование
файлов как для
//   ввода, так и для вывода данных. Программа читает
значение переменной
//   forder_price из файла customer.dat, вычисляет значение
переменной
//   fbilling_price и записывает его в файл billing.dat.
//
#include <fstream.h>
#include <iomanip.h>
#define MIN_DISCOUNT .97
#define MAX_DISCOUNT .95
```

```

int main( ) {
float forder_price, fbilling_price;
ifstream fin("customer.dat");
ofstream fout("billing.dat");
fin >> forder_price; while (!fin.eof()){
fout << setiosflags(ios::fixed); fout << "Для заказа на сумму\t\t$" <<
setprecision(2)
<< setw(8)<< forder_price << "\n";
if (forder_price < 10000)
fbilling_price = forder_price * MIN_DISCOUNT; else fbilling_price =
forder_price * MAXDISCOUNT;
fout << "цена со скидкой равна\t$"
<< setw(8) << fbilling_price << "\n\n"; fin >> forder_price;
fin. close ( ) ;
fout .close ( ) ;
return(0);
}

```

Операции записи данных в файл и считывания информации из файла отличаются в языках C++ и C. Как видно из сравнения двух примеров, вместо вызова функции `fopen()` происходит создание двух объектов классов `ifstream` (содержит функции файлового ввода) и `ofstream` (содержит функции файлового вывода). Далее работа выполняется с помощью уже знакомых нам операторов `>>` и `<<`, а также функций форматирования `setw()`, `setprecision()` и `setiosflags()`. В целом в C++ для ввода/вывода данных посредством консолей и файлов используются те же операторы с тем же синтаксисом, что существенно упрощает программирование операций ввода/вывода — область программирования, всегда считавшаяся сложной и чреватой ошибками.

Глава 5. Работа с данными

- Идентификаторы
- Ключевые слова
- Стандартные типы данных
 - Символы
 - Три типа целых чисел
 - Модификаторы signed и unsigned
 - Числа с плавающей запятой
 - Перечисления
 - Новый тип данных языка C++ - bool
- Квалификаторы
 - Квалификатор const
 - Директива #define
 - Квалификатор volatile
 - Одновременное применение квалификаторов const и volatile
- Преобразование типов данных
 - Явное преобразование типов
- Классы памяти
 - Объявление переменных на внешнем уровне
 - Объявление переменных на внутреннем уровне
 - Правила определения области видимости переменных
 - Объявление функций
- Операторы
 - Побитовые операторы
 - Операторы сдвига
 - Инкрементирование и декрементирование
 - Арифметические операторы
 - Оператор присваивания
 - Комбинированные операторы присваивания
 - Операторы сравнения и логические операторы
 - Условный оператор
 - Оператор запятая

- Приоритеты выполнения операторов

Есть одно справедливое утверждение: "Вы не сможете называть себя профессиональным программистом на C/C++ до тех пор, пока не прекратите мыслить на любых других языках и переводить свои мысли на C/C++". Вы можете разбираться в языках COBOL, FORTRAN, Pascal или PL/I и, зная хотя бы один из этих языков, заставить себя изучить другие языки из этого списка, чтобы успешно программировать на любом из них. Но такой подход может не сработать в случае изучения языков C и C++. Дело в том, что они достаточно сильно отличаются от других языков как по используемым средствам программирования, так и по уникальной логике поиска решений. Если вы не измените свой стиль мышления при написании программ, то не сможете реализовать все возможности и преимущества C/C++. И, что хуже, вы даже не сможете разобраться в логике программ, написанных настоящими специалистами. Техника написания программ на языках C/C++ изобилует таким количеством нюансов, что беглого взгляда на программу достаточно, чтобы оценить профессиональный уровень программиста!

С этой главы мы начнем детальное изучение структуры языков C/C++. Прежде всего следует познакомиться со стандартными типами данных и операторами, с помощью которых можно манипулировать данными.

Идентификаторы

Идентификаторами называются имена, присваиваемые переменным, константам, типам данных и функциям, используемым в программах. После описания идентификатора можно ссылаться на обозначаемую им сущность в любом месте программы.

Идентификатор представляет собой последовательность символов произвольной длины, содержащую буквы, цифры и символы подчеркивания, но начинающуюся обязательно с буквы или символа подчеркивания. Компилятор распознает только первые 31 символ. (Учтите, что другие программы, принимающие данные от компилятора, например компоновщик, могут распознавать последовательности даже еще меньшей длины.)

Языки C и C++ чувствительны к регистру букв. Другими словами, компилятор распознает прописные и строчные буквы как разные символы. Так, переменные `NAME_LENGTH` и `Name_Length` будут рассматриваться как два разных идентификатора, представляющих различные ячейки памяти. То есть вы можете создавать идентификаторы, одинаково читаемые, но отличающиеся написанием одной или нескольких букв.

Использование как прописных, так и строчных букв в идентификаторах позволяет сделать программный код более читабельным. Например, идентификаторы, которые объявлены в файлах заголовков, включаемых в программу с помощью директив `#include`, часто записывают прописными буквами, чтобы они бросались в глаза. В результате, где бы вы ни встретили идентификаторы, записанные прописными буквами, сразу станет ясно, где они описаны.

Хотя допускается использование символа подчеркивания в начале имени идентификатора, мы не рекомендуем так поступать, поскольку данный способ записи применяется в именах встроенных системных подпрограмм и переменных. Совпадение имени идентификатора с зарезервированным именем вызовет конфликт в работе программы. Два символа подчеркивания (`__`) в начале имени идентификатора применяются в стандартных библиотеках языка C++.

Среди программистов на C принято негласное соглашение начинать любое имя с префикса типа данных этого идентификатора. Например, все целочисленные идентификаторы должны начинаться буквой `i` (integer), идентификаторы с плавающей запятой — буквой `f` (float), строки, завершающиеся нулевым символом, — буквами `sz` (stringzero), указатели — буквой `p` (pointer) и т.д. Зная об этих соглашениях, вы, бросив лишь беглый взгляд на текст программы, сможете сразу определить не только идентификаторы, используемые в этой программе, но и их типы данных. Это существенно упрощает восприятие программных текстов.

Ниже показаны примеры идентификаторов:

```
i
itotal
frangel
szfirst_name
Ifrequency
```

```
imax
iMax
iMAX
NULL
EOF
```

Попробуйте определить, почему следующие имена идентификаторов недопустимы:

```
1st_year
#social_security
Not Done!
```

Первое имя недопустимо, поскольку начинается с цифры. Второе имя начинается с символа #, а третье — содержит недопустимый символ в конце имени. Посмотрите теперь на следующие идентификаторы и попробуйте определить, допустимы ли они:

```
o
oo
ooo
```

Как это ни покажется странным, все четыре имени допустимы. В первых трех используется буква o в верхнем регистре. Поскольку длина идентификаторов разная, никаких конфликтов не происходит. Имя четвертого идентификатора состоит из пяти символов подчеркивания (_). Но достаточно ли содержательны эти имена? Определенно нет, хотя они вполне допустимы. Таким образом, программист должен позаботиться о том, чтобы все имена функций, констант, переменных и другие идентификаторы несли какой-то смысл.

Обратите также внимание, что, поскольку строчные и прописные буквы различимы, следующие три идентификатора уникальны:

```
MAX_RATIO
max_ratio
Max_Ratio
```

Чувствительность компилятора языка C к регистру букв может вызвать головную боль у начинающих программистов. Например, если вместо printf () в программе будет записано Printf(), то компилятор выдаст сообщение об ошибке вида "unknown identifier" (неизвестный идентификатор). На языке Pascal можно использовать любое написание имен, например: writeln, WRITELN или writeLn.

Впрочем, вы быстро научитесь отличать неправильную запись вызовов функций, а вот сможете ли вы разобраться, где ошибка в этой строке:

```
printf("%D", integer_value);
```

Предположив, что переменная integer_value записана правильно, вы можете не заподозрить ошибки. Тем не менее, ошибка есть. Формат вывода может задаваться в языке C только оператором %d, но не %D.

И последнее замечание касательно идентификаторов: имя идентификатора не должно совпадать (с учетом регистра) с именем ключевого слова.

Ключевые слова

Ключевые слова являются встроенными идентификаторами, каждому из которых соответствует определенное действие. Изменить назначение ключевого слова нельзя. (С помощью директивы препроцессора #define можно создать "псевдоним" ключевого слова, который будет дублировать его действия, возможно, с некоторыми изменениями.) Помните, что имена идентификаторов, создаваемых в программе, не могут совпадать с ключевыми словами языков C/C++ (табл. 5.1).

Таблица 5.1. Ключевые слова языков C/C++ (начинающиеся с символов подчеркивания специфичны для компилятора Microsoft)			
asm	else	main	struct

assume	enum	multiple inheritance	switch
auto	except	single inheritance	template
based	explicit	virtual inheritance	this
bool	extern	mutable	thread
break	false	naked	throw
case	_fastcall	namespace	true
catch	_finally	new	try
cdecl	float	noreturn	_try
char	for	operator	typedef
class	friend	private	typeid
const	goto	protected	typename
const cast	if	public	union
continue	inline	register	unsigned
declspec	inline	reinterpret cast	using
default	int	return	uuid
delete	int8	short	_uuidof
dllexport	_int16	signed	virtual
dll import	_int32	sizeof	void
do	_int64	static	volatile
double	leave	static cast	while
dynamic cast	long	_stdcall	wmain

Стандартные типы данных

Все программы обрабатывают какую-то информацию. В языках C/C++ данные представляются одним из восьми базовых типов: char (текстовые данные), int (целые числа), float (числа с плавающей запятой одинарной точности), double (числа с плавающей запятой двойной точности), void (пустые значения), bool (логические значения), перечисления и указатели. Остановимся на каждом из типов данных.

- Текст (тип данных char) представляет собой последовательность символов, таких как a, Z, ? и 3, которые могут быть разделены пробелами. Обычно каждый символ занимает 8 бит, или один байт, с диапазоном значений от 0 до 255.
- Целые числа (тип данных int) находятся в диапазоне от -32768 до 32767 и занимают 16 бит, т.е. два байта, или одно слово. В Windows 98 и Windows NT используются 32-разрядные целые, что позволяет расширить диапазон их значений от -2147483648 до 2147483647.

- Числа с плавающей запятой одинарной точности (тип данных `float`) могут представляться как в фиксированном формате, например число л (3,14159), так и в экспоненциальном (7,56310). Диапазон значений — $\pm 3,4E-38$ — $3,4E+38$, размерность — 32 бита, т.е. 4 байта, или 2 слова.
- Числа с плавающей запятой двойной точности (тип данных `double`) имеют диапазон значений от $\pm 1,7E-308$ до $\pm 1,7E+308$ и размерность 64 бита, т.е. 8 байтов, или 4 слова. Ранее существовал тип `longdouble` с размерностью 80 бит и диапазоном от $\pm 1,18E-4932$ до $\pm 1,18E+4932$. В новых 32-разрядных версиях компиляторов он эквивалентен типу `double` и поддерживается из соображений обратной совместимости с написанными ранее приложениями.
- Перечисления представляются конечным набором именованных констант различных типов.
- Тип данных `void`, как правило, применяется в функциях, не возвращающих никакого значения. Этот тип данных также можно использовать для создания обобщенных указателей, как будет показано в главе "Указатели".
- Указатели, в отличие от переменных других типов, не содержат данных в обычном понимании этого слова. Вместо этого указатели содержат адреса памяти, где хранятся данные.
- Переменные нового типа данных `bool` в C++ могут содержать только одну из двух констант: `true` или `false`.

Символы

В каждом языке (не только компьютерном) используется определенный набор символов для построения значимых выражений. Например, в книгах, написанных на английском языке, используются комбинации из 26 букв латинского алфавита, десяти цифр и нескольких знаков препинания. Аналогично, выражения на языках C и C++ записываются с помощью 26 строчных букв латинского алфавита:

abcdefghijklmnopqrstuvwxyz

26 прописных букв латинского алфавита:

ABCDEFGHIJKLMNOPQRSTUVWXYZ .

десяти цифр:

0123456789

и следующих специальных символов:

+ - * / = , . _ : ; ? \ " ' ~ | ! # % \$ & () [] { } ^ @

К специальным символам относится также пробел. Комбинации некоторых символов, не разделенных пробелами, интерпретируются как один значимый символ:

++ - == && || << >> >= <= += -= *= /= ?: :: /*
*/ //

В следующем примере программы на языке C показана работа с переменными типа `char`:

```
/*
 *      char.c
 *      Эта программа на языке C демонстрирует работу с переменными
типа
 *      char и показывает, как интерпретировать символьное значение
 *      в целочисленном виде.
 */
#include <stdio.h>
#include <ctype.h>
int main() {
char csinglechar, cuppercase, clowercase;
printf("\nВведите одну букву: ");
scanf("%c", &csinglechar);
```



```

cuppercase = toupper(csinglechar);
clowercase = tolower(csinglechar);
printf("ВВЕРХНЕМ регистре буква \'%c\'имеет"
      " десятичный ASCII-код %d\n",cuppercase,
      " cuppercase);
printf("ASCII-код в шестнадцатеричном формате"
      " равен %X\n", cuppercase); printf("Если прибавить
      шестнадцать, то получится
      " \'%c\'\n", (cuppercase + 16));
printf("ASCII-код полученного значения в
      " шестнадцатеричном формате"
      " равен %X\n", (cuppercase + 16)); printf("ВНИЖНЕМ
      регистре буква \'%c\'имеет"
      " десятичный ASCII-код%d\n",clowercase,
      " clowercase);
return(0); }

```

В результате работы программы на экран будет выведена следующая информация:

Введите одну букву: d

В ВЕРХНЕМ регистре буква 'D'имеет десятичный ASCII-код 68

ASCII-код в шестнадцатеричном формате равен 44

Если прибавить шестнадцать, то получится 'T'

ASCII-код полученного значения в шестнадцатеричном формате равен 54

В НИЖНЕМ регистре буква 'd' имеет десятичный ASCII-код 100

Спецификатор формата %X служит указанием отобразить значение переменной в шестнадцатеричном виде в верхнем регистре.

Три типа целых чисел

В C/C++ поддерживаются три типа целых чисел. Наравне со стандартным типом `int` существуют типы `shortint` (короткое целое) и `longint` (длинное целое). Допускается сокращенная запись `short` и `long`. Хотя синтаксис самого языка не зависит от используемой платформы, размерность типов данных `short`, `int` и `long` может варьироваться. Гарантируется лишь, что соотношение размерностей таково: `short <= int <= long`. В Microsoft Visual C/C++ для переменных типа `short` резервируется 2 байта, для типов `int` и `long` — 4 байта.

Модификаторы `signed` и `unsigned`

Компиляторы языков C/C++ позволяют при описании переменных некоторых типов указывать модификатор `unsigned`. В настоящее время он применяется с четырьмя типами данных: `char`, `short`, `int` и `long`. Наличие данного модификатора указывает на то, что значение переменной должно интерпретироваться как беззнаковое число, т.е. самый старший бит является битом данных, а не битом знака.

Предположим, мы создали новый тип данных, `my_octal`, в котором для записи каждого числа выделяется 3 бита. По умолчанию считается, что значения этого типа могут быть как положительными, так и отрицательными. Поскольку из 3-х доступных битов старший будет указывать на наличие знака, то диапазон возможных значений получится от -4 до 3.

Но если при описании переменной типа `my_octal` указать модификатор `unsigned`, то тем самым мы освободим первый бит для хранения полезной информации и получим диапазон возможных значений от 0 до 7. Аналогичные преобразования выполняются и для стандартных типов данных, как видно из табл. 5.2.

Таблица 5.2. Характеристики стандартных типов данных языков C/C++			
Тип данных	Байтов	Эквивалентные названия	Диапазон значений
<code>int</code>	2/4	<code>signed</code> , <code>signed int</code>	зависит от системы

unsigned int	2/4	unsigned	зависит от системы
_int8	1	char, signed char	от -128 до 127
_int16	2	short, short int, signed short int	от -32768 до 32767
_int32	4	signed, signed int	от -2147483648 до 2147483647
int64	8	нет	от -9223372036854775808 до 9223372036854775807
char	1	signed char	от -128 до 127
unsignedchar	1	нет	от 0 до 255
short	2	short int, signed short int	от -32768 до 32767
unsigned short	2	unsigned short int	от 0 до 65535
long	4	long int, signed long int	от -2147483648 до 2147483647
unsigned long	4	unsigned long int	от 0 до 4294967295
lenum	—	нет	то же, что int
float	4	нет	приблизительно +/-3.4E+/-38
double	8	long double	приблизительно +/-1,8E+/-308

Модификаторы signed и unsigned могут использоваться с любыми целочисленными типами данных. Тип char по умолчанию интерпретируется как знаковый. Типы данных int и unsigned int имеют размерность системного слова: 2 байта в MS-DOS и 16-разрядных версиях Windows и 4 байта в 32-разрядных операционных системах. При построении переносимых программ не следует полагаться на конкретную размерность целочисленных типов. В то же время в компиляторе Microsoft C/C++ поддерживается использование целых чисел фиксированной размерности, для чего введены типы _intx. В табл. 5.3 показаны все возможные комбинации типов данных и модификаторов signed и unsigned.

Таблица 5.3. Возможные варианты использования модификаторов типов данных	
Полное название	Эквиваленты
signed char	char
signed int	signed, int
signed short int	short, signed short
signed long int	long, signed long
unsigned char	нет
unsigned int	unsigned
unsigned short int	unsigned short
unsigned long int	unsigned long

Числа с плавающей запятой

В C/C++ используется три типа данных с плавающей запятой: float, double и longdouble. Хотя в стандарте ANSI C не указан точный диапазон их значений, общепринято, что переменные любого из этих типов должны как минимум поддерживать диапазон от 1E-37 до 1E+37. Как видно из табл. 5.2, в Microsoft Visual C/C++ возможности типов данных с плавающей запятой значительно превышают минимальные требования. Тип longdouble отсутствовал в первоначальном варианте языка и был предложен комитетом ANSI. Ранее, в 16-разрядных

системах, он имел размерность 80 бит (10 байтов) и диапазон значений приблизительно +/- 1,2E+/-4932. В Windows 95/98/NT он эквивалентен типу double.

В следующем фрагменте программы на языке C++ показано описание и использование переменных с плавающей запятой:

```
// float.cpp
// Эта программа на языке C++ демонстрирует использование переменных
// типа float.
#include <iostream.h>
#include <iomanip.h>
int main() {
    long loriginal_flags = cin.flags();
    float fvalue;
    cout << "Введите число с плавающей запятой: ";
    cin >> fvalue;
    cout << "Стандартный формат: " << fvalue << "\n";
    cout << setiosflags(ios::scientific);
    cout << "Научный формат: " << fvalue << "\n";
    cout << resetiosflags(ios::scientific);
    cout << setiosflags(ios::fixed) ;
    cout << "Фиксированный формат: " << fvalue << "\n";
    cout.flags (loriginal_flags);
    return(0);
}
```

В результате работы программы на экран будет выведена следующая информация:

Введите число с плавающей запятой: 12.34 Стандартный формат: 12.34 Научный формат: 1.234000e+001 Фиксированный формат: 12.340000

Обратите внимание на возможность отображения чисел с плавающей запятой в различных форматах: стандартном, научном и фиксированном.

Перечисления

Перечислением называется список именованных целочисленных констант, называемых перечислителями. Переменная данного типа может принимать значение одной из перечисленных констант, ссылаясь на эту константу по имени. Например, в следующем фрагменте описывается перечисление `air_supply` с константами `EMPTY`, `useable` и `FULL` и создается переменная `instructor_tank` данного типа.

```
enum air_supply { EMPTY, USEABLE, FULL = 5 } instructor_tank;
```

Все константы имеют тип `int`, и каждой из них автоматически присваивается значение по умолчанию, если не указано какое-нибудь другое значение. В нашем примере константе `EMPTY` по умолчанию присвоено нулевое значение, так как это первый элемент в списке. Константе `useable` присвоено значение 1, так как это второй элемент списка после константы со значением 0. Для константы `full` вместо значения по умолчанию назначено значение 5. Если бы после константы `full` стояла еще одна константа, то ей по умолчанию было бы присвоено значение 6.

Имея перечисление `air_supply`, объявим еще одну переменную — `student_tank`:

```
enum air_supply student_tank
```

Обе переменные теперь можно использовать независимо друг от друга:

```
instructor_tank = FULL; >> student_tank = EMPTY;
```

В этом примере переменной `instructor_tank` присваивается значение 5, а переменной `student_tank` — 0.

и C++ при описании переменной типа перечисления нет необходимости указывать ключевое слово `enum`, хотя его наличие не является ошибкой.

Часто ошибочно полагают, что `air_supply` — это переменная. В действительности это особый тип данных, который можно использовать для создания переменных, таких как `instructor_tank` и `student_tank`.

Переменные типа перечисления являются адресуемыми и могут стоять в выражениях слева от оператора присваивания. Именно это и происходит в показанных выше двух строках программы. `empty`, `useable` и `full` — это имена констант, а не переменных, поэтому их значения не могут быть изменены в ходе выполнения программы.

Переменные типа перечисления можно сравнивать друг с другом и с константами-перечислителями. Эту возможность удобно использовать в программах, как показано в следующем примере:

```
/*
 *      enum.c
 *      Эта программа на языке C демонстрирует использование перечислений.
 */
#include <stdio.h>
#include <stdlib.h>
int main () {
    enum air_supply { EMPTY, USEABLE, FULL = 5 } instructor_tank;
    enum air_supply student_tank;
    instructor_tank = FULL;
    student_tank = EMPTY;
    printf ("Значение переменной instructor__tank
    равно%d\n",instructor_tank) ;
    if (student_tank < USEABLE) {
        printf("Наполните бак горючим.\n");
        printf("Занятие отменено.\n");
        exit(1); }
    if (instructor_tank >= student_tank)
        printf("Продолжайте занятие.\n"); else
        printf("Занятие отменено.\n");
    return(0); }
```

В языке C тип `enum` эквивалентен типу `int`. Это позволяет присваивать целочисленные значения непосредственно переменным типа перечисления. В C++ ведется более строгий контроль за типами данных, и такие присваивания не допускаются.

В результате работы программы на экран будет выведена следующая информация:

Значение переменной `instructor_tank` равно 5 Наполните бак горючим. Занятие отменено.

Новый тип данных языка C++ — `bool`

Тип данных `bool` относится к семейству целых типов. Переменные этого типа могут принимать только значения `true` или `false`. В C++ все условные выражения возвращают логические значения. Например, выражение `myvar != 0`, может вернуть только `true` или `false` в зависимости от значения переменной `myvar`.

Значения `true` и `false` связаны друг с другом следующими отношениями:

```
!false == true
!true == false
```

Рассмотрим следующий фрагмент:

```
if (условие) выражение;
```

Если условие равно `true`, то выражение будет выполнено, если же условие равно `false`, то выражение выполнено не будет. При создании условных выражений следует помнить о том,

что как в С, так и в С++ любое значение, отличное от нуля, воспринимается как истинное, а равное нулю — как ложное.

Когда к переменной типа `bool` применяются операции префиксного и постфиксного инкремента (`++`), переменная принимает значение `true`. Операторы префиксного и постфиксного декремента (`—`) не разрешены с переменными типа `bool`. Кроме того, поскольку тип данных `bool` относится к целочисленным, переменные этого типа могут быть приведены к типу `int`, при этом значение `true` преобразуется в `1`, а значение `false` — в `0`.

Квалификаторы

В С и С++ используются два квалификатора доступа: `const` и `volatile`. Они появились в стандарте ANSI С для обозначения неизменяемых переменных (`const`) и переменных, чьи значения могут измениться в любой момент (`volatile`).

Квалификатор `const`

Иногда требуется, чтобы значение переменной оставалось постоянным в течение всего времени работы программы. Такие переменные называются константными. Например, если в программе вычисляется длина окружности или площадь круга, то часто придется использовать число `pi` — `3.14159`. В бухгалтерских программах постоянной величиной является НДС. Поскольку НДС время от времени все же может меняться, то удобно будет описать его как переменную, значение которой остается постоянным в процессе выполнения программы.

Есть еще одно преимущество константных переменных. Если вместо них использовать числовые или строковые литералы, то при частом их вводе в разные места программы могут быть допущены случайные ошибки, которые достаточно трудно будет обнаружить. Если же ошибка допущена при вводе имени переменной, то компилятор не позволит скомпилировать такую программу и выдаст сообщение о том, что обнаружена незнакомая переменная.

Предположим, вы создаете программу, в которой используется число `pi`. Можно объявить переменную с именем `pi` и присвоить ей начальное значение `3.14159`. Но оно не должно больше меняться, так как это нарушит правильность вычислений с числом `pi`, поэтому должен существовать формальный способ запретить размещение данной переменной в левой части оператора присваивания. Этой цели служит квалификатор `const`. Например:

```
const float pi = 3.14159;
const int iMIN = 1, iSALE_PERCENTAGE = 25;
int irow_index = 1, itotal = 100;
double ddistance => 0;
```

Поскольку значение константной переменной не может меняться в программе, такая переменная может быть только проинициализирована. Это выполняется один раз во время объявления переменной. Так, в показанном выше примере целочисленным константам `iMIN` и `iSALE_PERCENTAGE` присвоены постоянные значения `1` и `25` соответственно. В то же время переменные `irow_index`, `itotal` и `ddistance`, инициализируемые аналогичным образом, могут впоследствии быть модифицированы обычным путем.

Константные и простые переменные используются в программе совершенно одинаково. Единственное отличие состоит в том, что начальные значения, присвоенные константным переменным при их инициализации, не могут быть изменены в ходе выполнения программы. Другими словами, константы не являются l-значениями (левосторонними значениями), т.е. не могут располагаться в выражениях слева от оператора присваивания. (Примером l-значения, ссылающегося на изменяемую ячейку памяти, является переменная, в объявлении которой не указан квалификатор `const`.) При выполнении присваивания оператор записывает значение правого операнда в ячейку памяти, адресуемую именем левого операнда. Таким образом, левый операнд (или единственный операнд унарной операции) должен ссылаться на изменяемую ячейку.

Директива `#define`

В С и С++ существует другой способ описания констант: директива препроцессора `#define`. Предположим, в начале программы введена такая строка:

```
#define SALES_TEAM 10
```

Общий формат данной строки следующий: директива `#define`, литерал `sales_team` (имя константы), литерал `10` (значение константы). Встретив такую команду, препроцессор

осуществит глобальную замену в программном коде имени `sales_team` числом 10. Причем никакие другие значения идентификатору `sales_team` не могут быть присвоены, поскольку переменная с таким именем в программе формально не описана. В результате идентификатор `sales_team` может использоваться в программе в качестве константы. Обратите внимание, что строка с директивой `#define` не заканчивается точкой с запятой. После числа 10 этот символ будет воспринят как часть идентификатора, в результате чего все вхождения `SALES_TEAM` будут заменены литералом 10;.

Итак, мы рассмотрели два способа создания констант: с помощью квалификатора `const` и директивы `#define`. В большинстве случаев они дают одинаковый результат, хотя имеется и существенное различие. Директива `#define` создает макроконстанту, и ее действие распространяется на весь файл. С помощью же ключевого слова `const` создается переменная. Далее в этой главе при рассмотрении классов памяти мы узнаем, что область видимости переменной можно ограничить определенной частью программы. Это же относится и к переменным, описанным с помощью квалификатора `const`. Таким образом, последний дает программисту больше гибкости, чем директива `#define`.

Квалификатор `volatile`

Ключевое слово `volatile` указывает на то, что данная переменная в любой момент может быть изменена в результате выполнения внешних действий, не контролируемых программой. Например, следующая строка описывает переменную `event_time`, значение которой может измениться без ведома программы:

```
volatile int event_time;
```

Подобное описание переменной может понадобиться в том случае, когда переменная `event_time` обновляется системными устройствами, например таймером. При получении сигнала от таймера выполнение программы прерывается и значение переменной изменяется.

Квалификатор `volatile` также применяется при описании объектов данных, совместно используемых разными процессами в многозадачной среде.

Одновременное применение квалификаторов `const` и `volatile`

Допускается одновременное употребление ключевых слов `const` и `volatile` при объявлении переменных. Так, в следующей строке создается переменная, обновляемая извне, но значение которой не может быть изменено самой программой:

```
const volatile constant_event_time;
```

Таким способом реализуются два важных момента. Во-первых, в случае обнаружения компилятором выражения, присваивающего переменной `constant_event_time` какое-нибудь значение, будет выдано сообщение об ошибке. Во-вторых, компилятор не будет выполнять оптимизацию, связанную с подстановкой вместо адреса переменной `constant_event_time` ее действительного значения, поскольку во время выполнения программы это значение в любой момент может быть изменено.

Преобразования типов данных

До сих пор в рассмотренных примерах мы в отдельно взятых выражениях использовали переменные только одного типа данных, например `int` или `float`. Но часто бывают случаи, когда в операции участвуют переменные разных типов. Такие операции называются смешанными. В отличие от многих других языков программирования, языки C и C++ могут автоматически преобразовывать данные из одного типа в другой.

Данные разных типов по-разному сохраняются в памяти. Возьмем число 10. Формат, в котором оно будет храниться, зависит от назначенного этому числу типа данных. Другими словами, сочетание нулей и единиц в представлении одного и того же числа 10 будет разным в зависимости от того, интерпретируется ли оно как целое или как число с плавающей запятой.

Давайте рассмотрим следующее выражение, где для двух переменных, `fresult` и `ivalue`, задан тип данных `float`, а для переменной `ivalue` — тип `int`:

```
Iresult = fvalue * ivalue;
```

Это типичный пример смешанной операции, в процессе которой компилятор автоматически выполняет определенные преобразования. Целочисленное значение переменной `ivalue` будет

прочитано из памяти, приведено к типу с плавающей запятой, умножено на исходное значение переменной `fvalue`, и результат, также в виде значения с плавающей запятой, будет сохранен в переменной `fresult`. Обратите внимание на то, что значение переменной `ivalue` при этом никак не меняется, оставаясь целочисленным.

Автоматические преобразования типов данных при выполнении смешанных операций производятся в соответствии с иерархией преобразований. Суть состоит в том, что с целью повышения производительности в смешанных операциях значения целых типов временно приводятся к тому типу данных, который имеет больший приоритет в иерархии. Ниже перечислены типы данных в порядке уменьшения приоритета:

```
double
float
long
int
short
```

Если значение преобразуется к типу, имеющему большую размерность, потери информации не происходит, вследствие чего не страдает точность вычислений.

Посмотрим, что происходит в случае приведения значения типа `int` к типу `float`. Предположим, переменные `ivaluel` и `ivalue2` имеют тип `int`, а переменные `fvalue` и `fresult` — тип `float`. Выполним следующие операции:

```
ivaluel = 3;
ivalue2 = 4;
fvalue = 7.0;
fresult = fvalue + ivaluel/ivalue2;
```

Выражение `ivaluel/ivalue2` не является смешанной операцией: это деление двух целых чисел, результатом которого будет ноль, поскольку дробная часть (в данном случае 0,75) отбрасывается. Таким образом, переменной `fresult` будет присвоено значение 7,0.

Как изменится результат, если переменная `ivalue2` будет описана как `float`? В таком случае операция `ivaluel/ivalue2` станет смешанной. Компилятор автоматически приведет значение переменной `ivaluel` к типу `float` — 3,0, и результат деления будет 0,75. В сумме с переменной `fvalue` получим 7,75.

Важно помнить, что тип переменной, находящейся слева от оператора присваивания, предопределяет тип результата вычислений. Предположим, что переменные `fx` и `fy` описаны как `float`, а переменная `ireult` — как `int`. Рассмотрим следующий фрагмент:

```
fx = 7.0;
fy = 2.0;
ireult = 4.0 + fx/fy;
```

Результатом выполнения операции `fx/fy` будет 3,5. Можно предположить, что переменной `ireult` будет присвоена сумма $3,5 + 4,0 = 7,5$. Но, поскольку это целочисленная переменная, компилятор преобразует число 7,5 в целое, отбрасывая дробную часть. Полученное значение 7 и присваивается переменной `ireult`.

Явное преобразование типов

Иногда возникают ситуации, когда необходимо изменить тип переменной, не дожидаясь автоматического преобразования. Этой цели служит специальный оператор приведения типа. Если где-либо в программе необходимо временно изменить тип переменной, нужно перед ее именем ввести в круглых скобках название соответствующего типа данных. Например, если переменные `ivaluel` и `ivalue2` имеют тип `int`, а переменные `fvalue` и `fresult` — тип `float`, то благодаря явному преобразованию типов в следующих трех выражениях будут получены одинаковые результаты:

```
fresult = fvalue+(float)ivaluel/ivalue2;
fresult = fvalue+ivaluel/(float)ivalue2;
fresult = fvalue+(float)ivaluel/(float)ivalue2;
```

Во всех трех случаях перед выполнением деления происходит явное приведение значения одной или обеих переменных к типу `float`. Даже если преобразованию подвергается только

одна переменная, как мы уже знаем, в операции деления к типу float будет автоматически приведен и результат.

Классы памяти

В Visual C/C++ имеется четыре спецификатора класса памяти:

```
auto
register
static
extern
```

Спецификатор класса памяти может предшествовать объявлениям переменных и функций, указывая компилятору, как следует хранить переменные в памяти и как получать доступ к переменным или функциям. Переменные, объявленные со спецификаторами auto или register, являются локальными, а со спецификаторами static и extern — глобальными. Память для локальной переменной выделяется заново всякий раз, когда выполнение программы достигает блока, в котором объявлена переменная, и удаляется по завершении выполнения этого блока. Память для глобальной переменной выделяется один раз при запуске программы и удаляется по завершении программы.

Указанные четыре спецификатора определяют также область видимости переменных и функций, т.е. часть программы, в пределах которой к идентификатору можно обратиться по имени. На область видимости переменной и функции влияет место ее объявления в программе. Если объявление расположено вне любой функции, то говорят о внешнем уровне объявления. Если же оно находится в теле функции, то говорят о внутреннем уровне объявления.

Смысл спецификаторов класса памяти несколько различается в зависимости от того, дается ли объявление переменной или функции и на внешнем или внутреннем уровне объявляется данный идентификатор.

Объявление переменных на внешнем уровне

Переменная, объявленная на внешнем уровне, является глобальной и по умолчанию имеет класс памяти extern. Внешнее объявление может включать инициализацию (явную либо неявную) или просто быть ссылкой на переменную, инициализируемую в другом месте программы.

```
static int ivaluel;           // по умолчанию неявно присваивается 0
static int ivaluel = 10;      // явное присваивание
int ivalue2 =20;              // явное присваивание
```

Область видимости глобальной переменной распространяется до конца файла. Обращение к переменной не может находиться выше той строки, где она была объявлена.

Переменная объявляется на внешнем уровне только один раз. Если в одном из файлов создана переменная с классом памяти static, то она может быть объявлена под тем же именем с тем же спецификатором static в любом другом исходном файле. Так как статические переменные доступны только в пределах своего файла, конфликтов имен не возникнет.

С помощью спецификатора extern можно объявить переменную, которая будет доступна из любого места программы. Это может быть ссылка на переменную, описанную в другом файле или ниже в том же файле. Последняя особенность делает возможным размещение ссылок на переменную до того, как она будет инициализирована.

В следующем примере программы на языке C++ демонстрируется использование ключевого слова extern:

```
//
//      Файл А
//
#include <iostream.h>
extern int ivalue ;
```



```

// переменная ivalue становится доступной до того,
// как будет инициализирована
void function_a(void); void function_b(void);
int main()
ivalue++; // ссылка на объявленную выше переменную
cout << ivalue << "\n"; // выводит значение 11
function_a();
return(0);
int ivalue =10; void function_a(void) ivalue++;
cout << ivalue << "\n"; function_b ();
//инициализация переменной ivalue
//ссылка на объявленную выше переменную
//выводит значение 12
//-----
//   Файл B
#include <iostream.h> extern int ivalue ;
void function_b (void) f
ivalue++;
cout << ivalue << "\n";
// ссылка на переменную ivalue ,
// описанную в файле A
// выводит значение 13

```

Объявление переменных на внутреннем уровне

При объявлении переменных на внутреннем уровне можно использовать любой из четырех спецификаторов класса памяти (по умолчанию устанавливается класс auto). Переменные, имеющие класс auto, являются локальными. Их область видимости ограничена блоком, в котором они объявлены.

Спецификатор register указывает компилятору, что данную переменную необходимо сохранить в регистре процессора, если это возможно. В результате сокращается время доступа к данным и упрощается программный код. Область видимости у регистровых переменных такая же, как и у автоматических. В случае отсутствия свободных регистров переменной присваивается класс auto и она сохраняется в памяти.

Стандарт ANSI C не позволяет запрашивать адрес переменной, сохраненной в регистре. Но это ограничение не распространяется на язык C++. Просто при обнаружении оператора взятия адреса (&), примененного к регистровой переменной, компилятор сохранит переменную в ячейке памяти и возвратит ее адрес.

Переменная, объявленная на внутреннем уровне со спецификатором static, будет глобальной, но доступ к ней получить можно только внутри ее блока. В отличие от автоматической переменной, статическая переменная сохранит свое значение после завершения блока. По умолчанию статической переменной присваивается нулевое значение, но вы можете инициализировать ее некоторым константным выражением.

Спецификатор extern на уровне блока используется для создания ссылки на переменную с таким же именем, объявленную на внешнем уровне в любом исходном файле программы. Если в блоке определяется переменная с тем же именем, но без спецификатора extern, то внешняя переменная становится недоступной в данном блоке. В следующем примере показаны все рассмотренные выше ситуации.

```

#include <iostream.h>
int ivaluel = 1;
void function_a (void) ;
void main ( )
{ // ссылка на переменную ivaluel, описанную выше extern int ivaluel;
// создание статической переменной, видимой только внутри функции
main(),
//а также инициализация ее нулевым значением static int ivalue2;

```

```

// создание регистровой переменной с присвоением ей нулевого значения
// и сохранением в регистре процессора (если возможно) register int
rvalue = 0;
// создание автоматической переменной с присвоением ей нулевого
значения intint_value3 = 0;
// вывод значений 1, 0, 0, 0:
cout << ivaluel << "\n" << rvalue << "\n"
<< ivalue2 << "\n" << int_value3 << "\n"; function_a () ;
void function_a (void) {
// сохранение адреса глобальной переменной ivaluel
static int *pivaluel = &ivaluel;
// создание новой, локальной переменной ivaluel;
// тем самым глобальная переменная ivaluel становится недоступной
int ivaluel = 32;
// создание новой статической переменной ivalue2,
// видимой только внутри функции function_a () static int ivalue2 = 2;
ivalue2 += 2;
// вывод значений 32, 4 и 1: cout << ivaluel << "\n" << ivalue2 <<
"\n" << *pivaluel << "\n";
cout << ivaluel << "\n" << ivalue2 << "\n"
<< *pivaluel << "\n";
}

```

Поскольку переменная `ivaluel` переопределяется внутри функции `function_a ()`, доступ к глобальной переменной `ivaluel` блокируется. Тем не менее, с помощью указателя `pivaluel` можно получить доступ к глобальной переменной, сославшись на нее по адресу.

Правила определения области видимости переменных

Чтобы определить, в какой части программы будет доступна та или иная переменная, нужно воспользоваться следующими правилами. Областью видимости переменной может быть программный блок, функция, файл и вся программа. Переменные, объявленные внутри блока или функции, доступны только в их пределах. Переменные, объявленные на уровне файла, т.е. вне любой функции, и имеющие спецификатор `static`, доступны во всем файле, начиная со строки объявления. Переменные, объявленные в одном из файлов программы на внешнем уровне без спецификатора `static` или со спецификатором `extern`, видимы в пределах всей программы.

Объявление функций

При объявлении функций на внешнем или внутреннем уровне можно указывать спецификаторы `static` или `extern`. В отличие от переменных, функции всегда глобальны. Правила видимости функций слегка отличаются от правил видимости переменных.

Функции, объявленные как статические, можно вызывать внутри файла, в котором они реализованы, но они не доступны из других файлов программы. Кроме того, в разных файлах можно создавать статические функции с одинаковыми именами, что не приведет к конфликтам имен.

Функцию, объявленную как внешняя, можно использовать во всех файлах программы (если только в одном из них она не переопределена как статическая). Если при объявлении функции не указан класс памяти, то по умолчанию она считается внешней.

Операторы

В языках C/C++ есть ряд операторов, которых вы не встретите в других языках программирования. В их числе побитовые операторы, операторы инкрементирования и декрементирования, условный оператор, оператор запятая, а также операторы комбинированного присваивания.

Побитовые операторы

Побитовые операторы обращаются с переменными как с наборами битов, а не как с числами. Эти операторы используются в тех случаях, когда необходимо получить доступ к отдельным битам данных, например при выводе графических изображений на экран. Побитовые операторы могут выполнять действия только над целочисленными значениями. В отличие от логических операторов, с их помощью сравниваются не два числа целиком, а отдельные их биты. Существует три основных побитовых оператора: И (&), ИЛИ (|) и исключающее ИЛИ (^). Сюда можно также отнести унарный оператор побитового отрицания (~), который инвертирует значения битов числа.

Побитовое И

Оператор & записывает в бит результата единицу только в том случае, если оба сравниваемых бита равны 1, как показано в следующей таблице:

Бит0	Бит1	Результат
0	0	0
0	1	0
1	0	0
1	1	1

Этот оператор часто используют для маскирования отдельных битов числа.

Побитовое ИЛИ

Оператор | записывает в бит результата единицу в том случае, если хотя бы один из сравниваемых битов равен 1, как показано в следующей таблице:

Бит 0	Бит 1	Результат
0	0	0
0	1	1
1	0	1
1	1	1

Этот оператор часто используют для установки отдельных битов числа.

Побитовое исключающее ИЛИ

Оператор ^ записывает в бит результата единицу в том случае, если сравниваемые биты отличаются друг от друга, как показано в следующей таблице:

Бит 0	Бит 1	Результат
0	0	0
0	1	1
1	0	1
1	1	0

Этот оператор часто применяется при выводе изображений на экран, когда необходимо накладывать друг на друга несколько графических слоев. Ниже показаны примеры использования этих операторов с шестнадцатеричными, восьмеричными и двоичными числами.

0xF1 & 0x35	результат 0x31 (шестнадцатеричное)
0361 & 0065	результат 061 (восьмеричное)
11110011 & 00110101	результат 00110001 (двоичное)
0xF1 0x35	результат 0xF5 (шестнадцатеричное)
0361 0065	результат 0365 (восьмеричное)

11110011 00110101	результат 11110111 (двоичное)
0xF1 ^ 0x35	результат 0xC4 (шестнадцатеричное)
0361 ^ 0065	результат 0304 (восьмеричное)
11110011 ^ 00110101	результат 11000110 (двоичное)
~0xF1	результат 0xFF0E (шестнадцатеричное)
~0361	результат 0177416 (восьмеричное)
~11110011	результат 11111111 00001100 (двоичное)

В последнем примере результат состоит из двух байтов, так как в процессе операции побитового отрицания осуществляется повышение целочисленности операнда — его тип преобразуется к типу с большей размерностью. Если операнд беззнаковый, то результат получается вычитанием его значения из самого большого числа повышенного типа. Если операнд знаковый, то результат вычисляется посредством приведения операнда повышенного типа к беззнаковому типу, выполнения операции ~ и обратного приведения его к знаковому типу.

Операторы сдвига

Языки C/C++ содержат два оператора сдвига: сдвиг влево (<<) и сдвиг вправо (>>). Первый сдвигает битовое представление целочисленной переменной, указанной слева от оператора, влево на количество битов, указанное справа от оператора. При этом освобождающиеся младшие биты заполняются нулями, а соответствующее количество старших битов теряется.

Сдвиг беззнакового числа на одну позицию влево с заполнением младшего разряда нулем эквивалентен умножению числа на 2, как показано в следующем примере:

```
unsigned int value1 = .65; // младший байт: 0100 0001
value1 <<= 1;           // младший байт: 1000 0010
cout << value1;         // будет выведено 130
```

При правом сдвиге происходят аналогичные действия, только битовое представление числа сдвигается на указанное количество битов вправо. Значения младших битов теряются, а освобождающиеся старшие биты заполняются нулями в случае беззнакового операнда и значением знакового бита в противной ситуации. Таким образом, сдвиг беззнакового числа на одну позицию вправо эквивалентен делению числа на два:

```
unsigned int value1 = 10; // младший байт: 0000 1010
value1 >>= 1;             // младший байт: 0000 0101
printf("%d", value1);     // будет выведено 5
```

Инкрементирование и декрементирование

Операции увеличения или уменьшения значения переменной на 1 столь часто встречаются в программах, что разработчики языка C предусмотрели для этих целей специальные операторы инкрементирования (++) и декрементирования (--). Эти операторы можно применять только к переменным, но не константам.

Так, вместо следующей строки

```
value1 + 1;
```

можно ввести строку

```
value1++;
```

ИЛИ

```
++value1;
```

В подобной ситуации, когда оператор ++ является единственным в выражении, не имеет значения место его расположения: до имени переменной или после. Ее значение в любом случае увеличится на единицу.

Операторы ++ и — находят широкое применение в цикле for:

```
sum = 0;
for(1=1; i <= 20; i++) sum = sum + i;
```

Цикл с декрементом будет выглядеть так:

```
sum = 0;
for(i = 20; i >= 1; i--) sum = sum + i;
```

В сложных выражениях необходимо внимательно следить, когда именно происходит модификация переменной. В этом случае следует различать префиксные и постфиксные операторы, которые ставятся, соответственно, перед или после имени переменной.

Например, при постфиксном инкрементировании — `i++` — сначала возвращается значение переменной, после чего оно увеличивается на единицу. С другой стороны, оператор префиксного инкрементирования — `++i` — указывает, что сначала следует увеличить значение переменной, а затем вернуть его в качестве результата. Рассмотрим примеры. Предположим, имеются следующие переменные:

```
int i = 3, j, k = 0;
```

Теперь проанализируем, как изменятся значения переменных в следующих выражениях (предполагается, что они выполняются не последовательно, а по отдельности):

```
k = ++i;           // i = 4,    k = 4
k=i++;             // i = 4,    k = 3
k = --i;           // i = 2,    k = 2
k = i--;           // i = 2,    k = 3
i = j = k--;       // i = 0,    j = 0,    k = -1
```

Арифметические операторы

В языках C/C++ вы найдете все стандартные арифметические операторы, в частности операторы сложения (+), вычитания (-), умножения (*), деления (/) и деления по модулю (%). Первые четыре понятны и не требуют разъяснений. Возможно, имеет смысл остановиться на операции деления по модулю:

```
int a=3,b=8,c=0,d;
d = b % a;    // результат: 2
d = a % b;    // результат: 3
d = b % c;    // результат: сообщение об ошибке
```

При делении по модулю возвращается остаток от операции целочисленного деления. Поскольку в последней строке делается попытка деления на 0, будет выдано сообщение об ошибке.

Оператор присваивания

Присваивание в C/C++ отличается от аналогичных операций в других языках программирования тем, что, как и другие операторы C/C++, оператор присваивания не обязан стоять в отдельной строке и может входить в более крупные выражения. В качестве результата оператор возвращает значение, присвоенное левому операнду. Например, следующее выражение вполне корректно:

```
value1 = 8 * (value2 = 5);
```

В данном случае сначала переменной `value2` будет присвоено значение 5, после чего это значение будет умножено на 8 и результат 40 будет записан в переменную `value1`.

В результате многократного использования оператора присваивания в одной строке может получиться трудночитаемое, но вполне работоспособное выражение. Рассмотрим первый прием, который часто применяется для присваивания нескольким переменным одинакового значения:

```
value1 = value2 = value3 = 0;
```

Второй прием часто можно встретить в условных выражениях цикла `while`, как в следующем примере:

```
while ((c = getchar()) != EOF)
{
    .
    .
}
```

```
}
```

Вначале переменной `c` присваивается значение, возвращаемое функцией `getchar()`, после чего осуществляется проверка значения переменной на равенство константе `eof`. Цикл завершается при обнаружении конца файла. Использование круглых скобок необходимо из-за того, что оператор присваивания имеет меньший приоритет, чем подавляющее большинство других операторов, в частности оператор неравенства. Без круглых скобок данная строка будет воспринята следующим образом:

```
c = (getchar() != EOF)
```

То есть переменной `c` будет присваиваться значение 1 (true) всякий раз, когда функция `getchar()` возвращает значение, отличное от признака конца файла.

Комбинированные операторы присваивания

Набор операторов присваивания в языках C/C++ значительно богаче, чем в других языках программирования. Всевозможные комбинированные операторы присваивания позволяют предельно сжать программный код. В следующих строках показаны некоторые выражения присваивания, стандартные для большинства языков высокого уровня:

```
irow_index = irow_index + irow_increment;  
ddepth = ddepth - dl_fathom;  
fcalculate_tax = fcalculate_tax * 1.07;  
fyards = fyards / ifeet_convert;
```

Теперь посмотрим, как эти же выражения будут выглядеть в C/C++:

```
irow_index += irow_increment;  
ddepth -= dl_fathom; fcalculate_tax *= 1.07;  
fyards /= ifeet_convert;
```

Если вы внимательно рассмотрите эти примеры, то легко поймете синтаксис комбинированных операторов присваивания. Когда в левой и правой частях выражения встречается одна и та же переменная, вы можете удалить ссылку на нее из правой части, а соответствующий оператор объединить с оператором присваивания. Комбинированный оператор указывает, что над переменными из левой и правой частей выражения нужно выполнить операцию, указанную перед знаком равенства, а результат вновь присвоить переменной из левой части выражения. Все комбинированные операторы присваивания перечислены в нижней части табл. 5.4, которая приведена в конце главы.

Операторы сравнения и логические операторы

Операторы сравнения предназначены для проверки равенства или неравенства сравниваемых операндов. Все они возвращают true в случае установления истинности выражения и false в противном случае. Ниже показан список операторов сравнения, используемых в языках C и C++:

Оператор	Выполняемая проверка
<code>==</code>	Равно (не путать с оператором присваивания)
<code>!=</code>	Не равно
<code>></code>	Больше
<code><</code>	Меньше
<code><=</code>	Меньше или равно
<code>>=</code>	Больше или равно

Логические операторы И (`&&`), ИЛИ (`||`) и НЕ (`!`) возвращают значение true или false в зависимости от логического отношения между их операндами. Так, оператор `&&` возвращает true, когда истинны (не равны нулю) оба его аргумента. Оператор `||` возвращает false только в том случае, если ложны (равны нулю) оба его аргумента. Оператор `!` просто инвертирует значение своего операнда с false на true и наоборот.

Следующий пример программы на языке C демонстрирует применение перечисленных выше операторов сравнения и логических операторов.

```
/*
```

```

*   oprs.c
*   Эта программа на языке C демонстрирует применение
*   операторов сравнения и логических операторов.
*/
#include <stdio.h>
int main ()
{
    float foperand1, foperand2;
    printf("\nВведите значения переменных foperand1 и foperand2: ");
    scanf("%f%f", &foperand1, &foperand2);
    printf("\n foperand1 > foperand2 =%d", (foperand1 > foperand2));
    printf("\n foperand1 < foperand2 =%d", (foperand1 < foperand2));
    printf("\n foperand1 >= foperand2 =%d", (foperand1 >= foperand2));
    printf("\n foperand1 <= foperand2 = %d", (foperand1 <= foperand2));
    printf("\n foperand1 == foperand2 =%d", (foperand1 == foperand2));
    printf("\n foperand1 != foperand2 =%d", (foperand1 != foperand2));
    printf("\n foperand1 && foperand2 =%d", (foperand1 && foperand2));
    return(0);
}

```

Иногда полученные результаты могут вас удивить. Следует помнить, особенно при сравнении значений типа float и double, что отличие даже в последней цифре после запятой будет воспринято как неравенство.

Ниже показана аналогичная программа на языке C++:

```

/*
*   oprs.cpp
*   Эта программа на языке C++ демонстрирует применение
*   операторов сравнения и логических операторов..
#include <iostream.h>
int main () {
    float foperand1, foperand2;
    cout << "\nВведите значения переменных foperand1 и foperand2:
    cin >> foperand1 >> foperand2;
    cout << "foperand1 > foperand2 = " << (foperand1 > foperand2) << "\n";
    cout << "foperand1 < foperand2 = " << (foperand1 < foperand2) << "\n";
    cout << "foperand1 >= foperand2 = " << (foperand1 >= foperand2) <<
    "\n";
    cout << "foperand1 <= foperand2 = " << (foperand1 <= foperand2) <<
    "\n";
    cout << "foperand1 == foperand2 = " << (foperand1 == foperand2) <<
    "\n";
    cout << "foperand1 != foperand2 = " << (foperand1 != foperand2) <<
    "\n";
    cout << "foperand1 && foperand2 = " << (foperand1 && foperand2) <<
    "\n";
    cout << "foperand1 | foperand2 = " << (foperand1 | foperand2) <<
    "\n";
    return(0);
}

```

Условный оператор

Оператор ?: часто применяется в различных условных конструкциях. Он имеет следующий синтаксис:

Условие ? выражение1 : выражение2

Если условие равно true выполняется выражение1 в противном случае выражение2.

Оператор запятой

Позволяет последовательно выполнить два выражения записанных в одной строке. Синтаксис оператора следующий:

левое_выражение, правое_выражение

Приоритеты выполнения операторов

Последовательность выполнения различных действий в выражении определяется компилятором. Вследствие этого могут получаться неправильные результаты, если не был учтен порядок разбора выражения компилятором. Побочные эффекты также могут возникать при неправильном использовании простого и комбинированного операторов присваивания. Вот почему важно установление четких приоритетов в выполнении операторов и порядка вычисления операндов. К примеру, операция логического И (&&) всегда выполняется слева направо:

```
while ((c= getchar()) != EOF) && (c != '\n')
```

Оператор && предопределяет, что сначала будет выполнено выражение слева от него, т.е. переменной c будет присвоено значение, возвращаемое функцией getchar (), и только потом это значение проверяется на равенство символу новой строки.

В табл. 5.4 перечислены все операторы языков C и C++ в порядке уменьшения их приоритета (для компилятора Microsoft Visual C++) и указывается направление вычисления операндов (ассоциативность): слева направо или справа налево.

Таблица 5.4. Операторы языков C и C++ (в порядке уменьшения приоритета)		
Оператор	Операция	Ассоциативность
:::	Расширение области видимости	—
::	Доступ к члену класса по имени класса	—
[]	Доступ к элементу массива	Слева направо
()	Вызов функции	Слева направо
()	Приведение объекта к другому типу	—
.	Прямой доступ к члену класса (через объект)	Слева направо
->	Косвенный доступ к члену класса (через указатель на объект)	Слева направо
++	Постфиксный инкремент	—
--	Постфиксный декремент	—
new	Динамическое создание объекта	—
delete	Динамическое удаление объекта	—
delete[]	Динамическое удаление массива	—
++	Префиксный инкремент	—
--	Префиксный декремент	—
*	Раскрытие указателя	—
&	Взятие адреса	—
+	Унарный плюс	—
-	Унарный минус	—
!	Логическое НЕ	—
~	Побитовое НЕ	—
sizeof	Получение размерности выражения в байтах	—
sizeof ()	Получение размерности типа данных в байтах	—
typeid()	Получение информации о типе операнда	—
(тип данных)	Приведение типа	Справа налево
const_cast	Приведение типа	—
dynamic_cast	Приведение типа	—
reinterpret_cast	Приведение типа	—
static_cast	Приведение типа	—

. *	Прямой доступ к указателю на член класса (через объект)	Слева направо
-> *	Косвенный доступ к указателю на член класса (через указатель на объект)	Слева направо
*	Умножение	Слева направо
/	Деление	Слева направо
%	Деление по модулю	Слева направо
+	Сложение	Слева направо
-	Вычитание	Слева направо
<<	Сдвиг влево	Слева направо
>>	Сдвиг вправо	Слева направо
<	Меньше	Слева направо
>	Больше	Слева направо
<=	Меньше или равно	Слева направо
>=	Больше или равно	Слева направо
==	Равно	Слева направо
! =	Не равно	Слева направо
&	Побитовое И	Слева направо
^	Побитовое исключающее ИЛИ	Слева направо
	Побитовое ИЛИ	Слева направо
&&	Логическое И	Слева направо
	Логическое ИЛИ	Слева направо
? :	Условное выражение	—
=	Простое присваивание	Справа налево
*=	Присваивание с умножением	Справа налево
/=	Присваивание с делением	Справа налево
%=	Присваивание с делением по модулю	Справа налево
+=	Присваивание со сложением	Справа налево
-=	Присваивание с вычитанием	Справа налево
<<=	Присваивание со сдвигом влево	Справа налево

>>=	Присваивание со сдвигом вправо	Справа налево
&=	Присваивание с побитовым И	Справа налево
=	Присваивание с побитовым ИЛИ	Справа налево
^=	Присваивание с побитовым исключающим ИЛИ	Справа налево
,	Запятая	Слева направо

Глава 6. Инструкции

- Инструкции выбора
 - Инструкция if
 - Инструкция if/else
 - Условный оператор ?:
 - Конструкция switch/case
- Циклы
 - Цикл for
 - Цикл while
 - Цикл do/while
- Инструкции перехода
 - Инструкция break
 - Инструкция continue
 - Инструкция return

Чтобы приступить к созданию первых программ на языках C/C++, вам следует познакомиться с некоторыми дополнительными средствами программирования. В этой главе будут рассмотрены базовые инструкции, составляющие основу любой программы. Большинство из них вам должно быть хорошо знакомо по другим языкам программирования высокого уровня. Это такие инструкции, как if, if /else и switch/case, а также циклы for, while и do/while. В то же время ряд инструкций уникален для C/C++, например условный оператор ?:, ключевые слова break и continue. Они не имеют аналогов в более ранних языках программирования, таких как FORTRAN, COBOL или Pascal. По этой причине начинающие программисты часто оставляют их без внимания и не используют в своих программах. В результате остаются невопользованными некоторые уникальные возможности, предоставляемые языками C/C++. Кроме того, это выдает в вас новичка, которому вряд ли доверят работу над серьезным проектом.

Инструкции выбора

В языках C/C++ имеются четыре базовые инструкции выбора: if, if /else, switch/case и оператор ?:. Прежде чем приступить к изучению каждой из них, следует упомянуть об общих принципах построения условных выражений. Инструкции выбора используются для выборочного выполнения определенных блоков программы, состоящих либо из одной строки, либо из нескольких. В первом случае строка не выделяется фигурными скобками, во втором — выделяется весь блок.

Инструкция if

Одиночная инструкция if предназначена для выполнения команды или блока команд в зависимости от того, истинно заданное условие или нет. Ниже показан простейший пример инструкции if:

if (условие) выражение;

Обратите внимание, что условие заключается в круглые скобки. Если в результате проверки условия возвращается значение true, то выполняется выражение, после чего управление передается следующей строке программы. Если же результатом условия будет false, то выражение будет пропущено. Перейдем к конкретному примеру. В следующем фрагменте на экран выводится приветствие "Добрый день!" всякий раз, когда значение переменной ioutside_temp оказывается большим или равным 72:

```
if(ioutside_temp >= 72)
printf("Добрыйдень!");
```

В более сложном случае, когда инструкция if охватывает несколько выполняемых команд, синтаксис немного меняется:

```
if(условие) { . выражение1; выражение2;
...
выражение-n; }
```

Весь блок команд выделяется фигурными скобками, а каждая строка заканчивается точкой с запятой. Ниже показан пример программы, в которой применяется инструкция if:

```
/*
 *      if.c
 *  В этой программе на языке C демонстрируется использование
инструкции if.
 */
#include <stdio.h>
int main ()
{
int   inum_As,   inum_Bs,   inum-Cs;
float  fGPA;
printf("\nВведите число предметов, по которым вы получили оценку
ОТЛИЧНО:  ");
scanf("%d", &inum_As);
printf("\nВведите число предметов, по которым вы получили оценку
ХОРОШО:  ");
scanf("%d",&inum_Bs);
printf("\nВведите число предметов, по которым получили оценку
УДОВЛЕТВОРИТЕЛЬНО: ' ') ;
scanf("%d",&inum-Cs);
fGPA = (inum_As*5 + inum_Bs*4 + inum-Cs*3)/(float)(inum_As + inum_Bs +
inum-Cs);
printf("\nВаш средний балл: %5.2f\n",fGPA);
if (fGPA >= 4.5){
printf("\nПОЗДРАВЛЯЕМ!\n");
printf ("\nВы прошли по конкурсу.");
return(0); }
```

Обратите внимание, что инструкция if контролирует вывод только поздравительного сообщения, которое отображается, если значение переменной fGPA больше или равно 4,5.

Инструкция if/else

Инструкция if/else позволяет выборочно выполнять одно из двух действий в зависимости от условия. Ниже показан синтаксис данной инструкции:

```
if (условие)
выражение1; else
выражение2;
```

Если проверка условия дает результат true, то выполняется выражение1, в противном случае — выражение2. Рассмотрим пример:

```
if (ckeypressed — UP)
iy_pixel_coord++; else
iy_pixel_coord--;
```

В этом фрагменте выполняется увеличение или уменьшение текущей горизонтальной координаты указателя мыши в зависимости от значения переменной ckeypressed (предполагается, что она содержит код нажатой клавиши).

Если операторная часть ветви if или else содержит не одно выражение, а несколько, необходимо заключать их в фигурные скобки, как показано в следующих примерах:

```
if(условие) {
    выражение1;
    выражение.?.
    выражение3;
}

else
    выражение 4;

if (условие) (
    выражение1; else {
    выражение2;
    выражение3; выражение4 ;
}

if (условие) {
    выражение1;
    выражение2;
    выражение3;
}

else {
    выражение 4;
    выражение5;
    выражение 6; }
```

Обратите внимание, что после закрывающей фигурной скобки точка с запятой не ставится.

В следующем примере демонстрируется использование инструкции if /else:

```
/*
 *      crapif.c
 * В этой программе на языке C демонстрируется использование
 * инструкции if /else.
 */
#include <stdio.h>
int main ()
{
    char c;
    int ihow_many, i, imore;
    while (imore == 1) {
        printf ("Введите название товара: ");
        if (scanf ("%c",&c) != EOF)
        { while (c != '\n') scanf ("%c",&c) ;
        printf ("Сколько заказано? ");
        scanf ("%d",&ihow_many) ;
        scanf ("%c",&c) ;
        for(i= 1; i <= ihow_many; i++)
            printf ("*");
        printf ("\n"); } else
        imore = 0 ;}
    return(0); }
```

Эта программа предлагает пользователю ввести название товара, и, если не получен признак конца файла [Ctrl+C] (EOF), то название будет прочитано буква за буквой, пока не встретится символ конца строки (генерируется при нажатии клавиши [Enter]). В следующей строке отобразится надпись "Сколько заказано?", после которой необходимо ввести число заказанных единиц товара. Затем в цикле for на экран выводится ряд звездочек, количество которых соответствует введенному числу. При обнаружении признака конца файла программный блок ветви if пропускается и выполняется строка программы в ветви else. В этой строке

устанавливается нулевое значение флага `imore`, что служит признаком завершения программы.

Вложенные инструкции `if/else`

Если используется несколько вложенных инструкций `if`, следует тщательно следить, какой из них соответствует ветвь `else`. Взгляните на следующий пример и попытайтесь определить порядок выполнения команд:

```
if(itemperature < 50)
if(itemperature < 30)
printf("Наденьте меховую куртку.");
else printf("Наденьте шубу!");
```

В этом примере мы специально не делали отступов с помощью табуляции, чтобы не давать вам подсказок. Что произойдет, если значение переменной `itemperature` будет 55? Появится ли сообщение "Наденьте шубу"? Конечно же, нет. В данном примере ветвь `else` связана со второй инструкцией `if`. Дело в том, что компилятор всегда связывает инструкцию `else` с ближайшей к ней несвязанной инструкцией `if`.

Безусловно, профессиональный программист должен позаботиться о том, чтобы отступы в программе позволяли четко выявить связи между инструкциями:

```
if(itemperature < 50)
if (itemperature < 30)
printf ("Наденьте меховую куртку.");
else printf("Наденьте шубу!");
```

Еще нагляднее следующая нотация:

```
if(itemperature < 50) if(itemperature < 30)
printf("Наденьте меховую куртку."); else
printf("Наденьте шубу!");
```

Вы значительно облегчите работу и себе, и другим программистам, использующим ваши программы, если всюду будете четко и правильно выделять отступами программные блоки.

Рассмотрим пример:

```
if(условие1)
if(условие2)
выражение2; else
выражение1 ;
```

Нетрудно заметить, что выделения отступами в этом примере ошибочны, и данный код ничем не отличается от предыдущего примера. Многие начинающие программисты часто пытаются устанавливать связи внутри блоков путем изменения отступов. Но ведь компилятор не учитывает отступы! Как же сделать так, чтобы выражение1 действительно было связано с условием1, а не условием2?

Чтобы исправить ошибку, необходимо использовать фигурные скобки:

```
if(условие1) {
if (условие.2)
выражение2;
}
else
выражение1;
```

С помощью фигурных скобок мы объединили условие2 и выражение2 в единый блок, выполняемый в том случае, если условие1 равно `true`. В результате ветвь `else` связывается с, первой, а не со второй инструкцией `if`.

Конструкция `if/else/if`

Конструкция `if/else/if` используется для того, чтобы последовательно проверить несколько условий. Ее синтаксис выглядит следующим образом:

```

if(условие!)
выражение1; else if(условие2)
выражение2; else if (условие3)
выражение3;

```

Каждое выражение может представлять собой не одну строку, а блок команд. В таком случае требуются фигурные скобки (но помните, что после закрывающей фигурной скобки не ставится точка с запятой). В данном примере программа будет проверять условие за условием до тех пор, пока одно из них не возвратит значение true. Как только это произойдет, все остальные операторы else и связанные с ними действия будут пропущены. Если ни одно из условий не равно true, ни одно из действий не будет выполнено.

Теперь рассмотрим слегка видоизмененную версию предыдущего примера:

```

if(условие!)
выражение1; else if(условие2)
выражение2; else if(условие3)
выражение3; else
действие_по_умолчанию;

```

В данном случае какое-то действие обязательно будет выполнено. Если ни одно из условий не равно true, выполняется действие_по_умолчанию. Чтобы понять, в каких случаях лучше применять эту конструкцию, рассмотрим следующий пример. Здесь переменная fconverted_value содержит значение длины в футах, а переменная econvert_to — признак преобразования. Если значение последней не соответствует ни одному из известных типов преобразования, отображается соответствующее сообщение.

```

if (econvert_to == YARDS)
fconverted_value = length/3; else if (econvert_to == INCHES)
fconverted_value = length * 12; else if (econvert_to == CENTIMETERS)
fconverted_value = length * 30.48; else if (econvert_to == METERS)
fconverted_value = length * 30'. 48/100; else
printf ("Преобразование невозможно");

```

Условный оператор ?:

Оператор ? : позволяет создавать простые однострочные условные выражения, в которых выполняется одно из двух действий в зависимости от значения условия. Данный оператор можно использовать вместо инструкции if/else, а синтаксис его таков:

```
условие ? выражение1 : выражение2;
```

Оператор ? : называют тернарным, поскольку он требует наличия трех операндов. Рассмотрим пример, в котором определяется модуль числа:

```

if (f value >= 0.0)
fvalue = fvalue; else
fvalue = -fvalue;

```

С помощью условного оператора его можно записать в одну строку:

```
fvalue = (fvalue >= 0.0)? fvalue : -fvalue;
```

В следующей программе на языке C++ условный оператор применяется для того, чтобы выбрать правильный вид выводимых данных:

```

//
//  condit.cpp
//  В этой программе на языке C++ демонстрируется использование
//  условного оператора.
#include <math.h>
#include <iostream.h>
int main() <
float fbalance, fpayment;
cout << "Введите сумму займа: ";

```

```

cin >> fbalance;
cout << "\nВведите сумму погашения: ";
cin >> fpayment;
cout << "\n\nВаш баланс: ";
cout << ((fpayment > fbalance) ? "переплачено на $" : "уплачено $");
cout << ((fpayment > fbalance) ? (fpayment - fbalance) : fpayment);
cout << " по заему на сумму $" << fbalance << ".";
return(0);

```

Первый раз условный оператор используется для того, чтобы выяснить, какой текст выводить на экран: "переплачено на \$" или "уплачено \$". В следующем выражении с условным оператором определяется, какую денежную сумму следует отобразить:

```

cout << ((fpayment > fbalance) ? (fpayment - fbalance) : fpayment);

```

Конструкция switch/case

Часто возникает необходимость проверить переменную на равенство целому ряду значений. Это можно выполнить либо с помощью конструкции if/else/if, либо с помощью похожей конструкции switch/case. Обратим ваше внимание на то, что инструкция switch в языке C имеет ряд особенностей. Ее синтаксис таков:

```

switch(целочисленное_выражение) { ,
case константа1:
    выражение1;
    break;
case константа2:
    выражение2;
    break;
.
.
.
case константа-п:
    выражение-п;
    break;
default:
    действие_по_умолчанию; }

```

Заметьте: инструкция break повторяется во всех ветвях, кроме самой последней. Если, скажем, из первой ветви удалить эту инструкцию, то после выражения1 будет выполняться выражение2, что не всегда желательно. Таким образом, инструкция break отвечает за то, что после выполнения одной из ветвей case все остальные ветви будут пропущены. Рассмотрим несколько примеров.

Предположим, имеется такой фрагмент:

```

if(emove == SMALL_CHANGE_DP)
fycoord = 5; else iftemove == SMALL_CHANGE_DOWN)
fycoord = -5; else if(emove == LARGE_CHANGE_DP)
fycoord = ,10;
else
fycoord = -10;

```

Его можно переписать с использованием инструкции switch:

```

switch(emove) {
case SMALL_CHANGE_UP:
    fycoord = 5;
    break; case SMALL_CHANGE_DOWN:
    fycoord = -5;
    break; case LARGE_CHANGE_UP:
    fycoord = 10;

```



```
break; default:
fycoord = -10;
}
```

Здесь значение переменной `fycoord` последовательно сравнивается с рядом констант в ветвях `case`. Если обнаруживается совпадение, переменной `fycoord` присваивается нужное приращение. Затем выполняется инструкция `break`, которая передает управление строке, следующей после закрывающей фигурной скобки. Если же ни одно из сравнений не дало результата, то переменной `fycoord` присваивается значение по умолчанию (-10). Поскольку это последняя строка всего блока, нет необходимости ставить после нее инструкцию `break`. Следует также упомянуть о том, что ветвь `default` является необязательной.

Умелое манипулирование инструкциями `break` позволяет рациональнее строить блоки `switch/case`, как показано в следующем примере:

```
/*
 *   switch.c
 *   В этой программе на языке C демонстрируется, как рационально
 *   строить блоки switch/case.
 */
int main () {
char c = 'a' ;
int ivowelct = 0, iconstantct = 0;
switch(c)
{
case 'a'
case 'A'
case 'e'
case 'E'
case 'i'
case 'I'
case 'o'
case 'O'
case 'u'
case 'U'
ivowelct++;
break; default : iconstantct++;
return(0);
}
```

Если переменная `c` содержит ASCII-код любой гласной буквы, увеличивается счетчик гласных `ivowelct`, в противном случае осуществляется приращение счетчика согласных `iconstantct`.

В других языках высокого уровня допускается объединение в одной проверке всех констант, приводящих к выполнению одного и того же действия. Но в C/C++ требуется, чтобы каждая константа была представлена отдельной ветвью `case`. Эффект объединения достигается за счет того, что связанные ветви не разделяются инструкциями `break`.

В следующей программе на языке C инструкция `switch` используется для организации вызова правильной функции:

```
/*
 *   fnswitch.c
 *   В этой программе на языке C демонстрируются другие возможности
 *   инструкции switch.
 */
#include <stdio.h>
#define QUIT 0
#define BLANK ' '
double      fadd(float fx,float fy) ;
```

```

double      fsub(float fx,float fy) ;
double      fmul(float fx,float fy) ;
double      fdiv(float fx,float fy) ;
int main() {
float fx, fy;
char cblank, coperator = BLANK;
while(coperator != QUIT) {
printf("\nВведите выражение вида (a (оператор) b): ");
scanf("%f%c%f", &fx, &cblank, &coperator, &fy) ;
switch (coperator) {
case '+':printf("ответ= %8.2f\n",fadd(fx, fy)); break;
case '-':printf("ответ= %8.2f\n",fsub(fx, fy)); break;
case '*':printf("ответ= %8.2f\n",fmul(fx, fy)); break;
case '/':printf("ответ= %8.2f\n",fdiv(fx, fy)); break;
case 'x': coperator = QUIT;
           break;
default : printf("\nОператор не распознан.");}
        }
    return (0);
}
double fadd (float fx, float fy)
{ return (fx+ fy) ; }
double fsub (float fx, float fy)
{ return (fx - fy) ; }
double fmul (float fx, float fy)
{ return (fx* fy) ; }
double fdiv (float fx, float fy)
{ return (fx/ fy) ; }

```

Хотя синтаксис описания и вызова функций может быть еще непонятен вам (с функциями нам предстоит познакомиться в следующей главе), в целом использование инструкции switch в данном примере позволяет сделать текст программы максимально компактным и наглядным. Если пользователь введет какое-нибудь математическое выражение, скажем 10+10 или 23*15, соответствующий оператор (+ или *) будет сохранен в переменной coperator, значение которой затем сравнивается в блоке switch с набором стандартных арифметических операторов. Если же вместо арифметического оператора введен символ x, то программа присвоит переменной coperator значение quit, что является признаком завершения цикла, а с ним и всей программы. В случае, когда пользователь случайно ввел какой-нибудь не распознаваемый программой символ, например %, управление передается ветви default, в которой на экран выводится предупреждающее сообщение.

В языке C++ инструкция switch используется аналогичным образом, в чем можно убедиться на следующем примере:

```

//
//  calendar. Сpp
//  В этой программе на языке C++ демонстрируется использование
//  инструкции switch для создания ежегодного календаря.
#include <fstream.h>
int main() {
int jan_1_start_day, num_days_per_month, month, date, year;
bool leap_year_flag;
ofstream fout("output.dat");
cout << "Укажите, на какой день недели приходится 1-е января\n";
cout << "\n(0- понедельник, ";
cout << "\n 1 - вторник и т.д.): ";
cin >> jan_1_start_day;
cout << "\nВведите год, для которого вы хотите построить календарь:";

```

```

cin >> year;
fout << "\n Календарь на " << year << " год";
if(!(year % 4) && (year % 100) || !(year % 400))
    leap_year_flag = true; else
    leap_year_flag = false;
for(month = 1; month <= 12;month++) {
    switch(month) {
    case 1:
        cout << "\n\n\n Январь\n";
        num_days_per_month = 31;
        break;
    case 2:
        cout << "\n\n\n Февраль\n";
        num_days_per_month = leap_year_flag ? 29 : 28;
        break;

    case 3:
        cout << "\n\n\n Март\n";
        num_days_per_month = 31;.
        break;
    case 4:
        cout << "\n\n\n Апрель\n";
        num_days_per_month =30;
        break;
    case 5:
        cout << "\n\n\n Май\n";
        num_days_per_month =31;
        break;
    case 6:
        cout << "\n\n\n Июнь\n";
        num_days_per_month = 30;
        break;
    case 7:
        cout << "\n\n\n Июль\n";
        num_days_per_month = 31;
        break;
    case 8:
        cout << "\n\n\n Август\n";
        num_days__per_month = 31;
        break;
    case 9:
        cout << "\n\n\n Сентябрь\n";
        num_days_per_month =30;
        break;
    case 10:
        cout << "\n\n\n Октябрь\n";
        num_days_per_month = 31;
        break;
    case 11:
        cout << "\n\n\n Ноябрь\n";
        num_days_per_month = 30;
        break;
    case 12:
        cout << "\n\n\n Декабрь\n";

```

```

num_days_per_month =31;
break;
}
fout << "\nПон Вто Сре Чет Пят Суб Вос\n";
fout << " --- --- --- --- --- --- --- \n";
for (date = 1; date < jan_1_start_day*4; date++)
fout << " ";
for (date = 1; date <= num_days_per_month; date++)
{ fout.width(3) ;
fout << date;
if ((date+ jan_1_start_day) % 7 > 0)
fout << " ";
else
fout << "\n"; }
jan_1_start_day = (jan_1_start_day + num_days_per_month) % 7;
}
fout.close () ;
return (0);
}

```

Программа начинает свою работу с того, что предлагает пользователю указать день недели, на который приходится 1-е января (0 соответствует понедельнику, 1 — вторнику и т.д.) Далее программа просит указать год, для которого вы. хотите построить календарь. Введенное значение отображается в виде заголовка календаря и сохраняется в переменной year. Выполнив ряд проверок, программа определяет, является ли введенный год високосным или нет, и сохраняет результат в переменной leap_year_flag (високосным является год, номер которого делится на 4, но не делится на 100, а также год, номер которого делится на 400).

Затем запускается цикл for, выполняющийся 12 раз — по одному разу для каждого месяца. Сначала выводится название месяца, а в переменную num_days_per_month записывается количество дней в данном месяце. Это осуществляется в блоке switch/case, состоящем из 12-ти ветвей. Вслед за этим отображаются заголовки дней недели и в отдельной строке — ряд пробелов, чтобы начать выводить числа с того дня, на который приходится 1-е число данного месяца. В последнем цикле for выводятся собственно номера дней месяца. В последней строке внешнего цикла for вычисляется порядковый номер дня недели первого числа следующего месяца.

Совместное использование конструкций if/else/if и switch/case

В следующем примере выполняется преобразование имеющегося значения длины в футах в значение другой системы измерений:

```

/*
*          ifelsw.c
*   В этой программе на языке C демонстрируется использование
*   конструкции if /else/if в сочетании с конструкцией switch/ case.
*/
#include <stdio.h>
enum conversion_type {YARDS, INCHES, CENTIMETERS, METERS}
C_Tconversion;
int main()
{
int iuser_response;
float fmeasurement, ffoot;
printf("\nВведите значение длины в футах: "); scanf("%f",&ffoot) ;
printf("\nВозможные единицы измерений: \
\n\t\t0 - ЯРДЫ          \
\n\t\t1 - ДЮЙМЫ         \
\n\t\t2 - САНТИМЕТРЫ    \

```

```

\n\t\t3 - МЕТРЫ          \
\n\n\t\tВаш выбор - >> ");
scanf("%d",&iuser_response) ;
switch (iuser_response) (
case 0 : C_Tconversion = YARDS;
break; case 1 : C_Tconversion = INCHES;
break; case 2 : C_Tconversion = CENTIMETERS;
break; default : C_Tconversion = METERS;
if (C_Tconversion == YARDS)
fmeasurement = ffoot/3; else if (C_Tconversion == INCHES)
fmeasurement = ffoot * 12; else if (C_Tconversion == CENTIMETERS)
fmeasurement = ffoot * 30.48; else
fmeasurement = ffoot * 30.48/100;
switch (C_Tconversion) {
case YARDS      : printf("\n\t\t%4.2f ярдов", fmeasurement); break;
case INCHES     : printf("\n\t\t%4.2f дюймов", fmeasurement);
break;
case CENTIMETERS :
printf("\n\t\t%4.2f сантиметров", fmeasurement) ;
break;
default :
printf("\n\t\t%4.2f метров", fmeasurement); }
return (0);
}

```

В данном примере константы единиц измерений представлены перечислением `conversion_type`. (Подробнее о перечислениях см. в главе "Дополнительные типы данных".) Первый блок `switch/case` предназначен для того, чтобы на основании введенного пользователем значения проинициализировать переменную `C_Tconversion` типа `conversion_type`. Затем в блоке вложенных инструкций `if/else/if` выполняется соответствующее преобразование. Наконец, в последнем блоке `switch/case` полученное значение выводится на экран.

Циклы

В языках C и C++ используются стандартные циклические инструкции: `for`, `while` и `do/while` (в некоторых языках программирования высокого уровня последняя называется `repeat/until`). Особенностью этих языков является то, что они располагают средствами прерывания циклов. Обычно цикл продолжается до тех пор, пока не выполнится некоторое условие, заданное при инициализации цикла. Но в C/C++ цикл можно прервать после обнаружения ожидаемой ошибки или по другой причине с помощью инструкции `break`. Кроме того, допускается принудительный переход на следующую итерацию цикла с помощью инструкции `continue`.

Отличие цикла `for` от циклов `while` и `do/while` состоит в том, что в нем, как правило, число повторений заранее известно. Таким образом, цикл `for` обычно используется в тех случаях, когда можно точно определить необходимое количество повторов. Циклы `while` и `do/while` применяются, когда число повторений неизвестно, но имеется некоторое условие, которое необходимо выполнить.

Цикл for

Цикл `for` имеет следующий синтаксис:

```
for(инициализирующее_выражение; условное_выражение; модифицирующее_выражение)
выражение;
```

При обнаружении в программе цикла `for` первым выполняется `инициализирующее_выражение`, в котором обычно устанавливается счетчик цикла. Это происходит только один раз перед запуском цикла. Затем анализируется `условное_выражение`, которое также называется условием прекращения цикла. Пока оно равно `true`, цикл не прекращается. Каждый раз после всех строк тела цикла выполняется `модифицирующее_выражение`, в котором происходит

изменение счетчика цикла. Как только проверка условного_выражения даст результат false, все строки тела цикла и модифицирующее_выражение будут пропущены и управление будет передано первому выражению, следующему за телом цикла. Если тело цикла содержит более одной команды, следует использовать фигурные скобки и руководствоваться определенными правилами оформления, чтобы сделать текст программы более понятным: for (инициализирующее_выражение; условное_выражение; модифицирующее_выражение) { выражение1;

```
выражение2;  
выражение3;  
выражение-n; }
```

Давайте рассмотрим некоторые примеры использования цикла for. В следующем фрагменте вычисляется сумма ряда целых чисел от 1 до 5. Предполагается, что переменные isum и ivalue имеют тип int:

```
isum = 0;  
for (ivalue = 1; ivalue <= 5; ivalue++)  
isum += ivalue;
```

Сначала переменной isum присваивается нулевое значение, после чего запускается цикл for, который начинается с присваивания переменной ivalue значения 1. Эта операция выполняется только один раз. Затем проверяется условие ivalue <= 5. Поскольку на первом шаге цикла это выражение равно true, к переменной isum прибавляется текущее значение переменной ivalue — единица. По завершении выполнения последней строки цикла (в данном случае единственной) переменная ivalue увеличивается на единицу. Этот процесс будет повторяться до тех пор, пока значение переменной ivalue не достигнет 6, что приведет к завершению цикла.

В программе на языке C++ приведенный фрагмент будет записан следующим образом (найдите одно отличие):

```
isum = 0;  
for (int ivalue = 1; ivalue <= 5; ivalue++) isum += ivalue;
```

В C++ допускается объявление переменных прямо в строке инициализации цикла for. Тут затрагивается достаточно важный вопрос: где вообще следует размещать все объявления переменных? В C++ переменные можно создавать непосредственно перед той строкой, где они впервые используются. Поскольку в нашем случае переменная ivalue используется только внутри цикла, ее объявление в цикле не нарушает стройности программы. Но рассмотрим такой пример:

```
int isum = 0;  
for(int ivalue = 1; ivalue <= 5; ivalue++) isum += ivalue;
```

Подобное объявление переменной isum можно назвать плохим стилем программирования, если ее применение не ограничивается данным циклом. Желательно все объявления локальных переменных собирать сразу после заголовка функции, к которой они относятся, так как это повышает удобочитаемость программы и облегчает ее отладку. Значение счетчика цикла for вовсе не обязательно должно меняться только на единицу. В следующем примере вычисляется сумма ряда нечетных чисел от 1 до 9:

```
iodd_sum = 0;  
for(iodd_value = 1; iodd_value <= 9;  
    iodd_value += 2); iodd_sum += iodd_value;
```

Здесь счетчиком цикла является переменная iodd_value, и ее значение на каждом шаге увеличивается на 2.

Кроме того, счетчик цикла может не только увеличиваться, но и уменьшаться. В следующем примере с помощью цикла for организуется считывание строки символов в массив sa[100], а затем с помощью другого цикла for эта строка выводится на экран в обратном порядке:

```
/*  
 *      forloop.cpp  
 *      В этой программе на языке C++ демонстрируется использование  
 *      цикла for для работы с массивом символов.  
 */  
#include <stdio.h>
```

```

#define CARRAY_SIZE 10
int main()
{
    int ioffset;
    char carray[CARRAY_SIZE];
    for(ioffset = 0; ioffset < CARRAY_SIZE; ioffset++)
        carray[ioffset] = getchar();
    for(ioffset = CARRAY_SIZE - 1; ioffset >= 0; ioffset--)
        putchar(carray[ioffset]);
    return(0); }

```

В первом цикле for переменной ioffset присваивается начальное значение 0, поскольку адресация элементов массива начинается с нуля. Затем происходит считывание символов по одному до тех пор, пока значение переменной ioffset не станет равным размеру массива. Во втором цикле for, в котором элементы массива выводятся на экран в обратном порядке, переменная ioffset инициализируется номером последнего элемента массива и по ходу цикла уменьшается на единицу. Цикл будет выполняться до тех пор, пока переменная ioffset не примет значение 0.

При работе с вложенными циклами for обращайтесь внимание на правильную расстановку фигурных скобок, чтобы четко определить границы каждого цикла:

```

/*
 *      nsloop1.c
 *      В этой программе на языке C демонстрируется важность
 *      правильной расстановки фигурных скобок во вложенных циклах.
 */
#include <stdio.h>
int main()
{
    int iouter_val, iinner_val;
    for (iouter_val = 1; iouter_val <= 4; iouter_val++) ( printf ("\n%3d--",
iouter_val) );
    for (iinner_val = 1; iinner_val <= 5; iinner_val++) printf
("%3d", iouter_val * iinner_val) ;
    return(0); }

```

В результате выполнения программы будут выведены следующие данные:

```

1 — 12345..
2 — 2  4  6  8 10
3 — 3  6  9 12 15
4 — 4  8 12 16 20

```

А теперь представим, что внешний цикл записан без фигурных скобок:

```

/*
 *      nsloop2.c
 *      В этой программе на языке C демонстрируется, что произойдет, если
 *      при задании внешнего цикла не поставит фигурные скобки.
 */
#include <stdio.h>
int main ()
{
    int iouter_val, iinner_val;
    for(iouter_val = 1; iouter_val <= 4; iouter_val++)
        printf("\n%3d-", iouter_val);
    for(iinner_val = >>1; iinner_val <= 5; iinner_val++)
        printf("%3d", iouter_val * iinner_val);
    return(0); }

```

Выводимые данные будут выглядеть совершенно иначе:

```
1 --
2 --
3 --
4 -- 5 10 15 20 25
```

Если тело внешнего цикла не будет выделено фигурными скобками, то к первому циклу будет отнесена только следующая за ним строка с функцией printf(). Только после того как функция printf() будет вызвана четыре раза подряд, начнется выполнение следующего цикла for. Во внутреннем цикле будет использовано самое последнее значение переменной iouter_val, т.е. 5, на основе которого будет сгенерирован и выведен очередной функцией printf() соответствующий ряд чисел.

Очень часто решение использовать или не использовать фигурные скобки в конструкциях со вложенными циклами принимается на основе того, насколько проще для понимания станет текст программы. Посмотрите на следующие два примера и попытайтесь определить, будет ли результат их выполнения одинаковым.

Вот первый пример:

```
/*
 *      nsloop3.c
 *      Еще одна программа на языке C, демонстрирующая использование
 *      фигурных скобок
 *      в конструкциях со вложенными циклами.
 */
#include <stdio.h>
int main () {
    int iouter_val, iinner_val;
    for(iouter_val = 1; iouter_val <= 4; iouter_val++)
        { for(iinner_val = 1; iinner_val <= 5; iinner_val++)
            printf("%3d", iouter_val * iinner_val);
        }
    return(0);
}
```

Запишем эту же программу несколько иначе:

```
/*
 *      nsloop4.c
 *      Видоизмененный вариант предыдущей программы.
 */
#include <stdio.h>
int main() {
    int iouter_val, iinner_val;
    for(iouter_val = 1; iouter_val <= 4; iouter_val++)
        for(iinner_val = 1; iinner_val <= 5; iinner_val++)
            printf("%3d", iouter_val * iinner_val);
    return(0); }
```

В обоих случаях на экран будет выводиться одна и та же последовательность:

```
1 2 3 4 5 2 4 6 8 10 3 6 9 12 15 4 8 12 16 20
```

В рассмотренных программах после строки, задающей внешний цикл, сразу следует строка внутреннего цикла. При этом весь внутренний цикл, независимо от того, состоит ли он из одной или нескольких строк, будет восприниматься как одно выражение. Поэтому без фигурных скобок, ограничивающих внешний цикл, вполне можно было обойтись. Но их можно установить с целью повышения удобочитаемости программы.

Цикл while

В языках C/C++ цикл `while` обычно используется в тех случаях, когда число повторений цикла заранее не известно. Он является циклом с предусловием, как и цикл `for`. Другими словами, программа проверяет истинность условия цикла до того, как начать следующую итерацию. Поэтому, в зависимости от начального условия, цикл может выполняться несколько раз или не выполняться вообще. Цикл `while` имеет следующий синтаксис:

```
while(условие) выражение;
```

Если тело цикла состоит из нескольких строк, необходимо использовать фигурные скобки:

```
while(условие) { выражение1; выражение2; выражение3;
    выражение-п; }
```

В следующей программе на языке C создается цикл, в котором на экран выводится двоичный эквивалент знакового целого числа. Цикл выполняется до тех пор, пока в результате побитового сдвига переменной `ivalue` не будет протестирован каждый ее бит.

```
/*
 *   while.c
 *   В этой программе на языке C демонстрируется использование цикла
   while.
 */
#include <stdio.h>
#define WORD 16
#define ONE_BYTE 8
int main()
{
    int ivalue = 256, ibit_position = 1;
    unsigned int umask = 1;
    printf("Число%d\n", ivalue);
    printf("имеет следующий двоичный эквивалент: ");
    while(ibit_position <= WORD) {
        if((ivalue>> (WORD - ibit_position))
            & umask) /* сдвигаем каждый */
            printf("1"); /* бит на нулевую */
        else /* позицию и сравниваем */
            printf("0"); /* число с константой */
        if(ibit_position == ONE_BYTE) /* umask */
            printf (" "); ibit_position++;
        return(0);
    }
}
```

Данная программа начинается с описания двух констант, `WORD` и `ONE_BYTE`, значения которых при необходимости можно модифицировать. Первая из них определяет длину в битах анализируемого числа, а вторая — позицию, после которой следует поставить пробел, чтобы отделить на экране одну группу разрядов от другой. В цикле `while` осуществляется поочередный (от старшего к младшему) сдвиг битов переменной `ivalue` в первую позицию, сравнение полученного числа со значением переменной `umask` (маска младшего разряда) и вывод нуля или единицы в зависимости от результата.

В следующей программе пользователю предлагается задать имена файлов для ввода и вывода данных. Цикл `while` используется для чтения данных из одного файла с одновременной записью их в другой файл, причем размер файлов изначально не известен.

```
/*
 *   fwhile.c
 *   В этой программе на языке C цикл while используется
 *   при работе с файлами. В программе демонстрируются
 *   дополнительные возможности файлового ввода/вывода.
 */
```

```

#include <stdio.h>
#define MAX_CHARS 30
int main() {
    int c;
    FILE *ifile, *ofile;
    char szi_file_name [MAX_CHARS] ,
          szo_file_name[MAX_CHARS] ;
    printf("Введите имя исходного файла: ");
    scanf("%s",szi_file_name);
    if((ifile= fopen(szi_file_name, "r")>= NULL)
    {
        printf("\nФайл%s не может быть открыт", szi_file_name);
        return(0);
    }
    printf ("Введите имя выходного файла:  ");
    scanf("%s",    szo_f ile_name) ;
    if ((oflie  =  fopen(szp_file_name,    "w"))    == NULL)
        {
            printf ("\nФайл %s не может быть открыт",    szo_f ile_name) ;
            return (0);
        }
    while ((c= fgetc(ifile)) != EOF)
        fputc(c,of ile) ;
    return(0); }

```

В программе также показано использование функций файлового ввода-вывода данных, таких как `fopen ()`, `fgetc ()` и `fputc ()` (более подробно об этих функциях см. в главе "Ввод-вывод в языке C").

Цикл do/while

В цикле `do/while` истинность условия проверяется после выполнения очередной итерации, а не перед этим. Другими словами, тело цикла гарантированно будет выполнено хотя бы один раз. Как вы помните, циклы `for` и `while` с предусловием могут вообще остаться невыполненными, если условное выражение сразу возвратит значение `false`. Таким образом, цикл `do/while` следует использовать тогда, когда некоторое действие в программе необходимо выполнить в любом случае, по крайней мере один раз.

Цикл `do/while` имеет следующий синтаксис:

```

do
    .
выражение; while(условие) ;

```

Если тело цикла состоит более чем из одной строки, необходимо ставить фигурные скобки:

```

do{
    выражение1; выражение2; выражение3;
    выражение-n ;}
while(условие);

```

В следующем примере цикл `do/while` используется для получения от пользователя строки текста:

```

//
//  dowhile.cpp
//  В этой программе на языке C++ демонстрируется
//  использование цикла do/while.
//
#include <iostream.h>
#define LENGTH 80
int main()
{
    char cSentence [LENGTH] ;

```

```

int iNumChars = 0, iNumWords = 1;
do{
cout << "Введите предложение : ";
cin.getline (cSentence, LENGTH); } while (cSentence [0]== '\0');
while (cSentence [iNumChars] != '\0')
{ if (cSentence [iNumChars] !=' '&&
    cSentence [iNumChars] != '\t'&&
    (cSentence [iNumChars+1] == ' ' ||
    cSentence [iNumChars+1] == '\t' ||
    cSentence [iNumChars+1] == '\0'))
    iNumWords++;
    iNumChars++;
cout<< "Вы ввели " << iNumChars<< " символов\n";
cout<< "Вы ввели " << iNumWords<< " слов";
return(0); }

```

В цикле do/while запрос будет повторяться до тех пор, пока пользователь не введет хотя бы один символ. При простом нажатии клавиши [Enter] функция getline() возвращает символ null, в таком случае цикл повторяется. Как только в массив будет записана строка, цикл завершится и программа произведет подсчет количества введенных символов и слов.

Инструкции перехода

В языках C/C++ имеется четыре инструкции перехода: goto, break, continue и return. Инструкция goto(или ее аналог) существует во многих языках высокого уровня и предназначена для принудительного перехода по указанному адресу, определяемому меткой. Ее синтаксис таков:

goto метка;

В отличие от языка типа Basic, где данная инструкция находит широкое применение, в C/C++ наличие в программе команды goto рассматривается как плохой стиль программирования, и считается, что в четко структурированной и грамотно написанной программе этой инструкции быть не должно.

Инструкция break

Инструкция break позволяет выходить из цикла еще до того, как условие цикла станет ложным. По своему действию она напоминает команду goto, только в данном случае не указывается точный адрес перехода: управление передается первой строке, следующей за телом цикла. Рассмотрим пример:

```

/*
 *          break. c
 *   В этой программе на языке C демонстрируется использование
 *   инструкции break.
 */
int main ()
{ int itimes = 1, isum = 0;
while (itimes < 10) {
isum += itimes;
if (isum > 20)
break;
itimes++;
return (0); }

```

Проанализируйте работу этой программы с помощью отладчика, контролируя значения переменных isum и itimes. Обратите внимание на то, что происходит, когда переменная isum достигает значения 21. Вы увидите, что в этом случае выполняется инструкция break, в результате чего цикл прекращается и выполняется первая строка, следующая за телом цикла. В нашем примере это инструкция return, которая завершает программу.

Инструкция continue

Инструкция `continue` заставляет программу пропустить все оставшиеся строки цикла, но сам цикл не завершается. Действие этой инструкции аналогично выполнению команды `goto`, указывающей на строку условия цикла. Если условное выражение возвратит `true`, цикл будет продолжен.

Ниже показан текст программы, реализующей игру в угадывание чисел и использующей возможности инструкции `continue`:

```
/*
 *      continue.c
 *  В этой программе на языке C демонстрируется использование
 *  инструкции continue.
 */
#include <stdio.h>
#define TRUE 1
#define FALSE 0
int main ()
{
    int ilucky_number = 77,
        iinput_val,
        inumber_of_tries = 0,
        iam_lucky = FALSE;
    while(!iam_lucky) {
        printf("Введите число: ");
        scanf("%d",&iinput_val);
        inumber_of_tries++;
        if(iinput_val == ilucky_number)
            iam_lucky = TRUE; else {
            if(iinput_val > ilucky_number)
                printf("Ваше число больше!\n"); else
                printf("Ваше число меньше!\n"); continue; }
        printf("Вам потребовалось всего %d попыток, чтобы отгадать счастливое число!",
            inumber_of_tries); }
    return(0); }
```

В цикле `while` пользователю предлагается ввести свой вариант числа, после него значение переменной `inumber_of_tries`, хранящей число попыток, увеличивается на единицу. Если попытка оказалась неудачной, программа переходит в ветвь `else`, в которой пользователю дается подсказка и выполняется инструкция `continue`, подавляющая вывод поздравительного сообщения функцией `printf()`. Тем не менее, выполнение цикла продолжается. Как только будет получено соответствие введенного и счастливого чисел, переменной `iam_lucky` будет присвоено значение `true`, в результате чего будет выполнена функция `printf()`.

Совместное использование инструкций break и continue

Для решения некоторых задач удобно комбинировать инструкции `break` и `continue`. Рассмотрим следующую программу на языке C:

```
/*
 *      breakcon.c
 *  В этой программе на языке C используются инструкции break и
 *  continue.
 */
#include <stdio.h>
#include <ctype.h>
#define NEWLINE '\n'
int main()
{
```

```

int c;
while((c=getchar()) != EOF) {
    if(isascii(c) == 0) {
        printf("Введенный символ не является ASCII-символом;  ") ;
        printf("продолжение невозможно\n");
        break;
    }
    if(ispunct(c) || isspace(c)) {
        putchar(NEWLINE);
        continue;
    }
    if(isprint(c) == 0)
        continue;
    putchar(c); }
return(0);
}

```

На вход программы поступают такие данные:

```

word control < exclamation! apostrophe' period.
^Z

```

Вот что будет получено в результате:

```

word control
exclamation
apostrophe
period

```

Данная программа считывает поступающие символы до тех пор, пока не будет введен признак конца файла ^Z [Ctrl+Z]. Затем производится анализ введенных символов, удаляются непечатаемые знаки, а все слова разделяются разрывами строк. Все эти действия выполняются с помощью функций, описанных в файле CTYPE.H: isascii(), ispunct(), isspace() и isprint(). Каждая функция получает в качестве аргумента символ и возвращает ноль или единицу в зависимости от принадлежности этого символа соответствующей категории.

Функция isascii() проверяет, является ли символ обычным ASCII-символом (т.е. его код находится в диапазоне 0—127), функция ispunct() — является ли символ знаком препинания, функция isspace() — является ли символ пробельным, а функция isprint() — является ли символ печатаемым. С помощью этих функций программа определяет, следует ли продолжать выполнение, а если продолжать, то как поступать с каждым из введенных символов.

Первая проверка в цикле while позволяет выяснить, поступают ли входные данные в читаемом формате. Ведь на вход программы могут подаваться данные из бинарного файла, в результате чего программа окажется бесполезной. В этом случае отображается предупреждающее сообщение, а выполнение цикла прерывается инструкцией break.

Если формат входных данных подходящий, то проверяется, является ли введенный символ пробелом или знаком препинания. Если это так, выводится пустая строка и выполняется инструкция continue, инициирующая следующую итерацию цикла.

Далее проверяется, является ли введенный символ печатаемым. Обратите внимание, что во входных данных содержится символ < ([Ctrl+Q]). Поскольку это непечатаемый символ, он не отображается вовсе, и выполняется инструкция continue, завершающая текущую итерацию цикла.

В случае, если анализируемый символ является текстовым, выполняется функция putchar(), выводящая его на экран.

Инструкция return

Иногда необходимо прервать выполнение программы задолго до того, как будут выполнены все ее строки. Для этой цели в языках C/C++ существует уже знакомая вам инструкция return, которая завершает выполнение той функции, в которой она была вызвана. Если вызов произошел в функции main(), то завершается сама программа. В этом случае инструкция return принимает единственный целочисленный аргумент, называемый кодом завершения. В

операционных системах UNIX и MS-DOS нулевой код воспринимается как нормальное завершение программы, тогда как любое другое значение является признаком ошибки.

Код завершения программы может проверяться вызывающими ее процессами. Например, если программа была запущена из командной строки и код ее завершения показал наличие какой-то ошибки, операционная система может выдать предупреждающее сообщение. Помимо завершения работы программы инструкция `return` вызывает принудительный вывод всех содержащихся в буфере данных и закрытие всех открытых программой файлов.

Следующая программа вычисляет среднее арифметическое ряда чисел, содержащего не более 30-ти значений, а также находит минимальное и максимальное значение ряда. Если пользователь хочет задать ряд, размер которого превышает максимально допустимый, выдается предупреждающее сообщение и выполнение программы прерывается с помощью инструкции `return`.

```
//
//  return.cpp
//  В этой программе на языке C++ демонстрируется использование
//  инструкции return.
//
#include <iostream.h>
#define LIMIT 30
int main() {
    int irow, irequested_qty, iscores[LIMIT], imin_score, imax_score;
    float fsum = 0, faverage;
    cout<< "\nВведите число значений ряда: ";
    cin >> irequested_qty;
    if(irequested_qty > LIMIT) {
        cout<< "\nВы можете ввести не более " << LIMIT << " значений.\n"
        cout<< "\n>>> Программа завершена. <<<\n";
        return(0);
    }
    for(irow = 0; irow < irequested_qty; irow++) {
        cout << "\nВведите '"<<irow+1 << "-элемент ряда: cin >> iscores[irow];
    }
    for(irow = 0; irow < irequested_qty; irow++) fsum = fsum +
    iscores[irow];
    faverage = fsum/(float)irequested_qty; imax_score = imin_score =
    iscores[0];
    for(irow =1;irow < irequested_qty; irow++) {
        if(iscores[irow] > imax_score)
            imax_score = iscores[irow];
        if(iscores[irow] < imin_score)
            imin_score = iscores[irow];
    }
    cout<<      "\nМаксимальное значение      = "      <<      imax_score;
    cout<<      "\nМинимальное значение      = "      <<      imin_score;
    cout<<      "\nСреднее значение      = "      <<      faverage;
    return(0); }
```

Функция `exit()`

Существует библиотечная функция `exit()`, которая аналогична инструкции `return` и объявлена в файле `STDLIB.H`. В этом файле также описаны две дополнительные константы, которые могут быть переданы в качестве аргументов этой функции: `exit_success` (сигнализирует об успешном завершении программы) и `exit_failure` (сигнализирует об ошибке). Использование этих констант позволяет сделать текст программы чуть более понятным, хотя компилятор выдает предупреждение, если не встречает в функции `main()` ни одной инструкции `return`.

Рассмотрим слегка видоизмененный вариант предыдущей программы.

```
//
// exit2.cpp
// Вариант предыдущей программы, в котором вместо
// инструкции return применяется функция exit() .
//
#include <iostream.h>
#include <stdlib.h>
#define LIMIT 30
int main () {
int irow, irequested_qty, iscores [LIMIT];
float fsum =0,imin_score, imax_score, faverage;
cout << "\nВведите число значений ряда: ";
cin >> irequested_qty;
if(irequested_qty > LIMIT) {
cout<< "\nВы можете ввести не более " << LIMIT<< " значений.\n" cout<<
"\n >>> Программа завершена. <<<\n"; exit(EXIT_FAILURE);
}
for(irow = 0; irow < irequested_qty; irow++) {
cout << "\nВведите " << irow+1 << "-и элемент ряда: cin >>
iscores[irow];
}
for(irow = 0; irow < irequested_qty; irow++) fsum = fsum +
iscores[irow];
faverage = fsum/(float)irequested_qty; imin_score = imax_score =
iscores[0];
for(irow = 0; irow < irequested_qty; irow++) {
if(iscores[irow] > imax_score) imax_score = iscores[irow];
if(iscores[irow] < imin_score) imin_score = iscores[irow]; }
cout<< "\nМаксимальное значение = " << imax_score;
cout<< "\nМинимальное значение = " << imin_score;
cout<< "\nСреднее значение = " << faverage;
exit(EXIT_SUCCESS); }
```

Функция atexit()

При завершении программы, как в нормальном режиме, так и с помощью функции exit() , иногда необходимо выполнить некоторые "финальные" действия. Для этого существует функция atexit(), которая в качестве аргумента принимает имя функции, регистрируя ее как "финальную". В следующей программе на языке С реализован этот принцип:

```
/*
* atexit.c
* В этой программе на языке С демонстрируется способ задания процедур,
* выполняемых при завершении программы, а также показывается, как
* влияет
* порядок их регистрации на очередность выполнения.
*/
#include <stdio.h>
#include <stdlib.h>
void atexit f nl (void) ;
void atexit_fn2(void); void atexit_fn3(void) ;
int main()
{
atexit(atexit_fn1);
atexit(atexit_fn2);
atexit(atexit_fn3);
printf("Программа atexit запущена.\n");
```


Глава 7. Функции

- Прототипы функций
 - Синтаксис объявления функции
 - Способы передачи аргументов
 - Правила видимости переменных
 - Рекурсия
- Аргументы функций
 - Формальные и фактические аргументы
 - Аргументы типа void
 - Аргументы типа char
 - Аргументы типа int
 - Аргументы типа float
 - Аргументы типа double
 - Массивы в качестве аргументов
- Типы значений, возвращаемых функциями
 - Тип результата: void
 - Тип результата: char
 - Тип результата: bool
 - Тип результата: int
 - Тип результата: long
 - Тип результата: float
 - Тип результата: double
- Аргументы командной строки
 - Текстовые аргументы
 - Целочисленные аргументы
 - Аргументы с плавающей запятой
- Дополнительные особенности функций
 - Макроподстановка функций
 - Перегрузка функций
 - Функции с переменным числом аргументов
- Область видимости переменных
 - Попытка получить доступ к переменной вне ее области видимости

- Оператор расширения области видимости

Краеугольным камнем, лежащим в основе языков C и C++, являются функции. В данной главе рассматриваются основные концепции создания и использования функций. Вы познакомитесь с понятием прототипа функции, которое было введено в стандарте ANSI C. На многочисленных примерах будет показано, как работать с аргументами различных типов и как создавать функции, возвращающие значения различных типов. Вы также узнаете, как с помощью стандартных параметров `argc` и `argv` получать аргументы командной строки в функции `main()`. Кроме того, будет рассказано об уникальных возможностях функций в языке C++.

Основную часть программного кода в C/C++ составляют функции. Они позволяют разбивать программу на отдельные блоки или модули. Таким образом, модульное программирование обязано своим появлением именно функциям. Под модульным программированием подразумевается создание программ из отдельных, самостоятельно отлаживаемых частей, совокупность которых образует единое приложение. Например, один модуль может выполнять функцию ввода данных, другой — реализовывать вывод данных на печать, а третий — отвечать за сохранение данных на диске. По сути, программирование на языках C и C++ как раз и заключается в написании функций. Любая программа на этих языках содержит, по крайней мере, одну функцию — `main()`. От того, насколько успешно будут проработаны функции, зависит эффективность и надежность работы программы.

В данной главе рассматривается много примеров программ, которые помогут вам лучше уяснить принципы создания функций. Многие из этих программ, помимо пользовательских, используют также встроенные библиотечные функции C/C++, которые значительно расширяют возможности программирования на этих языках.

Прототипы функций

После принятия стандарта ANSI C принципы программирования функций в языке C претерпели изменения. В основу было положено использование прототипов функций, которые широко применяются в C++. С этого момента в язык C была внесена некоторая сумятица, обусловленная одновременным существованием старого и нового стиля описания функций. Компиляторы поддерживают и тот, и другой, но, естественно, желательно придерживаться нового стиля.

Синтаксис объявления функции

В соответствии со стандартом ANSI C любая функция должна иметь прототип, т.е. заранее объявленный заголовок функции с указанием ее имени, типов аргументов и типа возвращаемого значения. Прототип может размещаться как в теле программы (до первого обращения к функции), так и в отдельном файле заголовков. (В этой книге мы для наглядности размещаем прототипы в теле программы.) Прототип функции снабжает компилятор информацией о том, аргументы какого типа ожидает функция и значение какого типа она возвращает. Это позволяет компилятору выполнять строгую проверку типов. В ранней версии языка C в объявлении функции указывались только имена ее аргументов.

В этой книге мы придерживаемся нового стиля, синтаксис которого таков:

```
тип_результата имя_функции(тип_аргумента необязательное_имя_аргумента[, ...]);
```

Функция может возвращать значения типа `void`, `int`, `float` и т.д. Имя функции может быть произвольным, но желательно, чтобы оно указывало на ее назначение. Если для выполнения функции нужно предоставить ей некоторую информацию в виде аргумента, то в круглых скобках должен быть указан тип аргумента и, при необходимости, его имя. Тип аргумента может быть любым. Если аргументов несколько, их описания (тип плюс имя) разделяются запятыми. Не будет ошибкой указание в прототипе только типа аргумента, без имени. Такая форма записи прототипов применяется в библиотечных функциях.

Описание функции представляет собой часть программного кода, которая, как правило, следует за телом функции `main()`. Синтаксис описания следующий:

```
тип_результата имя_функции(тип_аргумента имя_аргумента[, ...])
{
    тело функции )
```

Обратите внимание на то, что строка заголовка функции идентична строке описания ее прототипа, за одним маленьким исключением: она не завершается точкой с запятой. Ниже показана программа, в которой имеется пример описания функции:

```
/*
 *  prototyp.c
 *  Эта программа на языке C содержит описание функции.
 *  Функция находит сумму двух целочисленных аргументов и возвращает
 *  результат в виде целого числа.
 */
#include <stdio.h>
int iadder(int ix, int iy) ; /* прототип функции */
int main () {
    int ia = 23;
    int ib = 13;
    int ic;
    ic = iadder(ia, ib) ;
    printf("Сумма равна %d\n",ic) ;
    return(0);
}

int ladder(int ix, int iy) /* описание функции */
{
    int iz;
    iz = ix + iy;
    return(iz); /* возвращаемое значение */
}
```

Функция в нашем примере называется iadder(). Она принимает два целочисленных аргумента и возвращает целочисленное значение. В соответствии со стандартом ANSI C рекомендуется, чтобы прототипы всех функций размещались в отдельных файлах заголовков. Вот почему библиотеки .функций распространяются вместе с файлами заголовков. Но в простых программах вроде той, которую мы только что рассмотрели, допускается описание прототипа функции прямо в тексте программы.

Описание функции iadder() в программе на языке C++ будет выглядеть идентично:

```
//
//      prototyp.cpp
//      Эта программа на языке C++ содержит описание функции.
//      Функция находит сумму двух целочисленных аргументов и
//      возвращает
//      результат в виде целого числа.
//
#include <iostream.h>
int ladder(int ix, int iy); // прототип функции
int main() {
    int ia = 23;
    int ib = 13;
    int ic;
    ic = iadder(ia,ib) ;
    cout<< "Сумма равна " << ic<< "\n";
    return (0); }
int ladder(int ix, int iy) // описание функции
{
    int iz; iz = ix + iy;
    return(iz); /* возвращаемое значение */
}
```

Способы передачи аргументов

В предыдущих двух примерах в функцию передавались значения аргументов. Когда происходит передача переменной-аргумента по значению, в функции создается локальная переменная с именем аргумента, в которую записывается его значение. Внутри функции может измениться значение этой локальной переменной, но не самого аргумента.

В случае передачи переменной-аргумента по ссылке функция получает адрес аргумента, а не его значение. Создаваемая при этом локальная переменная является указателем. Такой подход, кроме всего прочего, позволяет немного сэкономить память. И, естественно, во время выполнения функции значение переменной-аргумента может измениться. Особенность этого метода состоит в том, что функция может возвращать сразу несколько значений тому процессу, из которого она была вызвана: как через аргументы-ссылки, так и непосредственно через инструкцию `return`.

В следующем примере рассмотренная выше функция `ladder()` принимает адреса аргументов, передаваемых по ссылкам. Такой способ может быть реализован как в С, так и в С++.

```
/*
 *      ref.c
 *  Эта программа на языке С демонстрирует использование указателей
 *  в качестве аргументов функции.
 */
#include <stdio.h>
int ladder(int *pix, int *piy);
int main ()
int ia = 23;
int ib = 13;
int ic;
ic = iadder(&ia,&ib) ;
printf("Сумма равна %d\n", ic);
return(0); }
int ladder (int*pix, int *piy)
{
int iz;
iz = *pix + *piy; return(iz); }
```

В языке С++ при передаче аргументов можно вместо указателей использовать непосредственно ссылки. Это гораздо удобнее и упрощает текст программы, так как ссылки тоже указывают на аргумент, но не требуют использования оператора раскрытия указателя. Вот пример той же программы на языке С++:

```
//
//      ref.cpp
//  Эта программа на языке С++ демонстрирует использование ссылок
//  в качестве аргументов функции.
//
#include <iostream.h>
int iadder (int Srix, int Sriy) ;
int main ()
{
int ia = 23;
int ib = 13;
int ic;
ic = iadder (ia,ib) ;
cout<< "Сумма равна " << ic<< "\n";
return (0);
}
int ladder(int Srix, int sriy) I
int iz;
iz = rix + riy; return(iz); }
```

Обратите внимание на отсутствие операторов взятия адреса при вызове функции и операторов раскрытия указателей в теле функции. В качестве аргументов функции используются ссылки `gi` и `giy`.

В C++ не допускается использование ссылок на ссылки, ссылок на битовые поля, массивов ссылок и указателей на ссылки.

Правила видимости переменных

Область видимости переменной может быть ограничена функцией, файлом или классом. Локальные переменные объявляются внутри функции, а значит, их использование ограничено телом функции. О таких переменных говорят, что они доступны, или видны, только внутри функции и имеют локальную область видимости.

Переменные, область видимости которых охватывает весь файл, объявляются вне любых функций или классов. Такие переменные называются глобальными, и доступ к ним можно получить из любой точки того же файла.

Одно и то же имя переменной может быть сначала описано на глобальном, а затем на локальном уровне. В таком случае локальная переменная "закрывает" собой глобальную переменную в теле функции. Для подобных ситуаций в языке C++ существует оператор расширения области видимости (`::`). Будучи примененным к переменной в теле функции, он позволяет сослаться на глобальную переменную с указанным именем, даже если в самой функции объявлена одноименная локальная переменная. Синтаксис оператора таков:

`::переменная`

Неправильное понимание правил видимости переменных может привести к возникновению различного рода ошибок, которые будут более подробно проанализированы в конце главы.

Рекурсия

Рекурсия возникает в том случае, когда функция вызывает саму себя. Рекурсивные алгоритмы можно использовать для элегантного решения некоторых задач, из которых одна из наиболее распространенных — нахождение факториала числа. Факториалом называется произведение всех целых чисел от единицы до заданного. Например:

$$8! = 8 * 7 * 6 * 5 * 4 * 3 * 2 * 1 = 40320$$

Обратите внимание на используемый тип данных — `double`, который позволяет работать с числами порядка $1E+308$. Особенностью факториала является то, что даже для сравнительно небольшого числа результат может быть огромным. Например, факториал числа 15 равен 1307674368000.

```
/*
 *      factor.c
 *  Эта программа на языке C демонстрирует применение
 *  рекурсии при вычислении факториала.
 */
#include <stdio.h>
double dfactorial(int inumber);
int main ()
{
    int Inumber =15;
    double dresult;
    dresult = dfactorial(inumber);
    printf ("Факториал%d равен%15.0f\n"; inumber, dresult) ;
    return(0);
}
double dfactorial(int inumber)
{
    if (inumber <= 1)
        return(1.0);
```

```

else
return(inumber * dfactorial(inumber-1));
}

```

Аргументы функций

В настоящем параграфе подробно рассматриваются принципы передачи функциям аргументов различных типов. Некоторые программисты используют вместо термина аргумент термин параметр, хотя правильнее говорить о фактическом аргументе, т.е. значении, передаваемом в функцию в момент вызова, и формальном аргументе (параметре), т.е. идентификаторе, заданном в описании функции.

В принципе, функция может вообще не иметь аргументов. Если же они есть, то их может быть сколь угодно много и все они могут иметь различные типы данных. Ниже будут приведены примеры, иллюстрирующие передачу функциям аргументов стандартных типов.

Формальные и фактические аргументы

Любая функция содержит список формальных аргументов, который может быть и пустым, если функция не принимает аргументов. Когда в программе осуществляется вызов функции, ей передается список значений аргументов, который называется списком фактических аргументов. В соответствии со стандартом ANSI C типы идентификаторов в обоих списках должны совпадать. Но в действительности совпадение может быть нестрогим, и над фактическими аргументами могут выполняться неявные операции преобразования типов данных.

Рассмотрим следующий фрагмент программы на языке C:

```

printf("Пример шестнадцатеричного %x и восьмеричного %o значения",
ians);

```

Хотя в строке форматирования указано два аргумента, в наличии имеется только один. Когда функции предоставляется меньше аргументов, чем содержится в списке параметров, недостающим присваиваются неопределенные значения. Во избежание ошибок в C++ предусмотрено задание аргументам стандартных значений. Тогда, если при вызове функции соответствующий аргумент не указан, ему автоматически будет присвоено значение по умолчанию. Вот пример прототипа функции на языке C++:

```

int имя_функции(int it, float fu = 4.2, int iv = 10)

```

Если в вызове функции не будут указаны аргументы fu и/или iv, то им по умолчанию будут присвоены значения 4,2 и 10 соответственно. Стандарт языка C++ требует, чтобы аргументы, для которых заданы значения по умолчанию, находились в конце списка формальных аргументов. В нашем примере допустимы следующие варианты вызова функции:

```

имя_функции(10)
имя_функции(10,15.2)
имя_функции(10,15.2,8)

```

Если аргумент fu не будет задан, установка аргумента iv также станет невозможной.

Аргументы типа void

В соответствии со стандартом ANSI C ключевое слово void применяется для явного указания на отсутствие аргументов функции. В C++ указывать слово void не обязательно, хотя данное соглашение довольно широко используется. В следующем примере создается функция voutput(), которая не получает аргументов и не возвращает никаких значений. Это, пожалуй, один из простейших видов функций.

```

/*
 *   fvoid.c
 *   Эта программа на языке C содержит пример функции, не принимающей
 *   никаких аргументов .
 */
#include <stdio.h>
#include <math.h>
void voutput (void) ;

```

```
int raain() {
printf("Эта программа вычисляет квадратный корень числа. \n\n");
voutput ();
return (0);
}
```

```
void voutput(void) {
int it = 12345;
double du;
du = sqrt(it);
printf("Квадратный корень числа %d равен %f \n", it, du); }
```

В функции voutput() вызывается стандартная библиотечная функция sqrt(), объявленная в файле MATH.H. Данная функция возвращает квадратный корень своего аргумента в виде значения типа double.

Аргументы типа char

В следующем примере в функции main() считывается символ, введенный с клавиатуры, и передается в функцию voutput(), которая выводит его на экран. Считывание символа осуществляется функцией _getch(). В языке C есть ряд похожих на нее библиотечных функций: getch(), getchar () и _getche (). Данные функции можно использовать и в языке C++, хотя всех их инкапсулирует в себе объект cin, управляющий вводом данных. Функция _getch() запрашивает символ, поступающий со стандартного устройства ввода (как правило, это клавиатура), и записывает его в переменную типа char без эха на экране.

```
/*
*      fchar.c
*  Эта программа на языке C считывает символ, введенный с клавиатуры,
*  и передает его в функцию voutput(), осуществляющую вывод нескольких
*  копий символа на экран.
*/
#include <stdio.h>
#include <conio.h>
void voutput(char c) ;
int main() {
char cyourchar;
printf("Введите символ: ");
cyourchar = _getch();
voutput(cyourchar);
return (0); }
```

```
void voutput(char c) <
int j ;
for (j = 0; j < 16; j++)
printf("\nБыл введен символ %c ", c); )
```

Спецификатор %c в функции printf() указывает, что выводится единственный символ.

Аргументы типа int

В следующем примере введенная пользователем длина ребра куба передается в функцию vside(), которая на основании полученного значения вычисляет площадь грани куба, его объем и площадь поверхности.

```
/*
*      fint.c
*  Эта программа на языке C предназначена для вычисления площади грани
*  и поверхности куба, а также его объема. Функция vside() принимает
*  целочисленное значение, содержащее длину ребра куба.
*/
```

```

#include <stdio.h>
void, vside(int is);
int main()
int iyourlength;
printf ("Введите длину ребра куба: ") ;
scanf("%d",&iyourlength) ;
vside (iyourlength) ;
return (0);
}
void vside(int is) {
int iarea, ivolume, isarea;
iarea = is * is; ivolume = is * is * is; isarea = 6 * iarea;
printf("\nДлина ребра куба: %d\n", is);
printf("Площадь грани куба: %d\n", iarea);
printf("Объем куба: %d\n", ivolume);
printf("Площадь поверхности куба: %d\n", isarea); }

```

Аргументы типа float

В следующем примере в функцию `vhypotenuse()` передаются два аргумента типа `float`, определяющие длину катетов прямоугольного треугольника. В функции вычисляется длина гипотенузы `vhypotenuse()`.

```

/*
 *      ffloat.c
 *      Эта программа на языке C вычисляет длину гипотенузы
 *      прямоугольного треугольника.
 */
#include <stdio.h>
#include <math.h>
void vhypotenuse(float ft,float fu);
int main()
{
float fxlen, fylen;
printf("Введите длину первого катета: ");
scanf("%f",&fxlen);
printf("Введите длину второго катета: ");
scanf("%f",&fylen);
vhypotenuse(fxlen,fylen);
return (0);
}
void vhypotenuse(float ft,float fu)
double dresult;
dresult = hypot((double) ft,(double) fu) ;
printf("\nДлина гипотенузы равна%g \n", dresult); }

```

Функция `hypot()`, возвращающая длину гипотенузы, объявлена в файле `MATH.H` и принимает аргументы типа `double`, поэтому параметры `fx` и `fy` функции `vhypotenuse()` должны быть приведены к этому типу.

Аргументы типа double

Следующая программа считывает два числа типа `double` и возводит первое в степень второго.

```

/*
 *      fdouble.c
 *      Эта программа на языке C возводит число в указанную степень.
 */

```



```

#include <stdio.h>
#include <math.h>
void vpower(double dt, double du);
int main () {
double dtnum, dunum;
printf("Введите основание степени: ");
scanf("%lf",&dtnum);
printf("\nВведите показатель степени: ");
scanf("%lf",&dunum);
vpower(dtnum, dunum);
return(0) ;
}
void vpower(double dt, double du)
double danswer;
danswer = pow(dt, du) ;
printf("\n%fв степени%f равно%f\n",dt, du, danswer);}

```

Массивы в качестве аргументов

В следующих примерах показано, как передать в функцию массив данных. В первой программе на языке С функция получает адрес первого элемента массива в виде указателя.

```

/*
 *      fpointer.c
 *      Эта программа на языке С демонстрирует, как передать в функцию
массив
 *      данных. Функция получает указатель на первый элемент массива.
 */
#include <stdio.h>
void voutput(int *pinums);
int main ()
{
int iyourarray[7] = {2,7, 15,32,45,3, 1} ;
printf("Передаем массив данных ,в функцию.\n");
voutput(iyourarray);
return(0); }
void voutput (int*pinurns) , {
int t ;
for(t= 0; t < 7; t++)
printf("Массив данных: %d \n",pinurns[t]); }
printf("Введите основание степени: ");
scanf("%lf",&dtnum);
printf("\nВведите показатель степени: ");
scanf("%lf",&dunum);
vpower(dtnum, dunum);
return(0);
}
void vpower(double dt, double du) {
double danswer;
danswer = pow(dt,du) ;
printf("\n%fв степени%f равно%f\n",dt,du, danswer), }

```

По сути, функция voutput () получает имя массива iyourarray []. Но данное имя одновременно является указателем на первый элемент массива. Это справедливо для любых массивов.

В объявлении функции можно также указать, что ее аргументом является массив неопределенного размера. В приводимом ниже примере показано, как реализовать это в языке C++. (Аналогичный подход допустим и в языке C.)

```
//
// farray.cpp
// Эта программа на языке C++ вычисляет среднее арифметическое ряда
чисел.
//
#include <iostream.h>
void avg (float fnums [ ] ) ;
int main () {
float iyourarray[8] = (12.3,25.7,82.1,6.0,7.01,0.25,4.2,6.28);
cout<< "Передаем массив в функцию для вычисления среднего значения.
\n"; avg (iyourarray) ;
return (0); }
void avg (float fnums [ ] ) {
int iv;
float fsum = 0.0;
float faverage;
for(iv= 0; iv < 8; iv++) {
fsum += fnums[iv];
cout<< iv+1 << "-и элемент равен " << fnums[iv] << "\n"; }
faverage = fsum/iv; cout << "\nСреднее равно " <<
faverage << "\n";
}
```

Типы значений, возвращаемых функциями

В данном параграфе вы найдете примеры функций, возвращающих значения всех стандартных типов. В предыдущих параграфах мы рассматривали функции, которые не возвращали никаких данных. В таких случаях говорят, что функция возвращает значение типа void. По сути, функция voutput() получает имя массива iyourarray[]. Но данное имя одновременно является указателем на первый элемент массива. Это справедливо для любых массивов.

В объявлении функции можно также указать, что ее аргументом является массив неопределенного размера. В приводимом ниже примере показано, как реализовать это в языке C++. (Аналогичный подход допустим и в языке C.)

```
//
// farray.cpp
// Эта программа на языке C++ вычисляет среднее арифметическое ряда
чисел.
//
#include <iostream.h>
void avg (float fnums [ ] ) ;
int main() {
float iyourarray[8] = (12.3,25.7,82.1,6.0,7.01,0.25,4.2,6.28);
cout<< "Передаем массив в функцию для вычисления среднего значения.
\n"; avg (iyourarray) ;
return (0); }
void avg (float fnums[]) {
int iv;
float fsum = 0.0;
float faverage;
for(iv= 0; iv < 8; iv++) {
fsum += fnums[iv];
cout<< iv+1 << "-и элемент равен " << fnums[iv] << "\n"; }
```

```
faverage = fsum/iv; cout << "\nСреднее павно " << faverage
<< "\n";
}
```

Тип результата: void

Поскольку с функциями, возвращающими значения типа void, мы уже познакомились, рассмотрим чуть более сложный пример. Как вы знаете, в C/C++ можно выводить числовую информацию в шестнадцатеричном, десятичном и восьмеричном формате, но не в двоичном. В следующей программе функция vbinary() преобразовывает полученное десятичное значение в двоичную форму и выводит его на экран. Цифры, составляющие двоичное число, не объединены в единое значение, а представляют собой массив данных.

```
/*
 *      voidf.c
 *      Эта программа на языке C находит двоичный эквивалент заданного
 *      десятичного числа.
 */
#include <stdio.h>
void vbinary(int idata) ;
int main ()
int ivalue;
printf ("Введите десятичное число: "); scanf("%d",&ivalue) ; vbinary
(ivalue);
return (0); }
void vbinary (int idata)
int t = 0;
int iyourarray [50];
while (idata != 0) {
iyourarray [t]= (idata % 2);
idata /= 2;
t++; }
printf("\n");
for( ; t >= 0; t--)
printf("%d",iyourarray[t]); }
```

Преобразование числа в систему счисления более низкого порядка реализуется довольно простым математическим алгоритмом. Например, чтобы перевести десятичное число в двоичную систему, нужно разделить его на основание новой системы — 2 — максимально возможное количество раз. На каждом шаге результатом деления будет дробное число, состоящее из целой части и остатка. Целая часть используется для следующего деления на 2, а остаток определяет цифру двоичного числа в позиции, соответствующей номеру шага. В двоичной системе используются цифры 0 и 1.

В функции vbinary() этот алгоритм реализуется в цикле while. В результате операции деления по модулю (%) остаток от деления (0 или 1) заносится в массив iyourarray[]. Целая часть, полученная при делении, присваивается переменной idata. Этот процесс продолжается до тех пор, пока целая часть результата деления (переменная idata) не станет равной нулю.

Тип результата: char

В следующей программе на языке C функция lowercase() принимает аргумент типа char и возвращает значение такого же типа. Предполагается, что вводится буква в верхнем регистре. В функции lowercase() вызывается библиотечная функция tolower(), прототип которой находится в файле ctype.h, она переводит символ в нижний регистр.

```
/*
 *      charf.c
 *      Эта программа на языке C преобразует символ из верхнего регистра '
в нижний.
 */
#include <stdio.h>
```

```

#include <ctype.h>
char clowcase(char c) ;
int main () {
char clowchar, chichar;
printf("Введите букву в верхнем регистре. \n");
chichar = getchar();
clowchar = clowcase(chichar);
printf("%c\n",clowchar);
return(0);
char clowcase(char c) {
return(tolower(c));
}

```

Тип результата: bool

Ниже дан пример использования двух функций, `is_upper()` и `is_lower()`, которые возвращают значения типа `bool`, проверяя, представлен ли переданный им символ в верхнем или нижнем регистре соответственно. Этот тип данных специфичен для языка C++.

```

//
//  boolf.cpp
//  Эта программа на языке C++ демонстрирует возможность
//  возвращения функциями значений типа bool.
//
#include <iostream.h>
bool is_upper(void) ; bool is_lower(void) ;
int main () {
char cTestChar = 'T';
bool bIsUppercase, bIsLowercase;
bIsUppercase = is_upper (cTestChar) ; bIsLowercase =
is_lower(cTestChar);
cout <<      "Буква " << (bIsUppercase ? "является " : "не является
")
<<      "прописной.\n";
cout<<      "Буква " << {bIsLowercase? "является " : "не является "}
<<      "строчной.\n";
return(0); }
bool is_upper(int ch) {
return(ch>= 'A'&& ch <= 'Z'); }
bool is_lower(int ch) {
return(ch >= 'a'&& ch <= 'z'); }

```

В этой программе для выбора выводимого сообщения применяется оператор `?:`, что позволяет каждую операцию вывода записать в одну строку и не использовать многострочную инструкцию `if/else`.

Тип результата: int

В следующем примере функция `icube()` принимает целое число, возводит его в куб и возвращает результат тоже в виде целого числа.

```

/*
*          intf.c
*  Эта программа на языке C возводит целое число в куб.
*/
#include <stdio.h>
int icube(int ivalue); int n(ain())
int k, inumbercube;
for(k=0;k < 20;k += 2) {

```

```

inumbercube = icube(k);
printf("Куб числа%d равен%d\n",k, inumbercube) }
return(0);
int icube(int ivalue)
return(ivalue * ivalue * ivalue);
}

```

Тип результата: long

В следующей программе, написанной на языке C++, функция Ipower() получает целочисленный аргумент, возводит число 2 в степень, заданную аргументом, и возвращает значение типа long.

```

//
// longf .cpp
// Эта программа на языке C++ демонстрирует возможность возвращения
// функциями значений типа long. Функция Ipower() получает целое
// число.и
// возводит 2 в степень, определяемую этим числом.
//
#include <iostream.h> long Ipower(int ivalue); int main()
int k;
long lanswer;
for(k =0;k < 20; k++) {
lanswer = Ipower(k);
cout<< "2 в степени " << k<< " равно " << lanswer<< "\n"
return(0); }
long Ipower(int ivalue) .
int t;
long Iseed = 1;
for(t =0;t < ivalue; t++) Iseed *= 2;
return(Iseed);
}

```

В функции Ipower() используется цикл, повторяющийся заданное число раз, в котором число 2 последовательно умножается само на себя. На практике для этого лучше применять стандартную функцию pow(), объявленную в файле MATH.H.

Тип результата: float

В следующем примере в функцию fproduct () передается массив чисел типа float и возвращается значение того же типа. Данная программа, написанная на языке C++, вычисляет произведение всех элементов массива.

```

//
// floatf.cpp
// Эта программа на языке C++ демонстрирует возможность возвращения
// функциями значений типа float. Функция fproduct0 получает массив
// данных типа floatи возвращает произведение элементов массива.
//
#include <iostream.h>
float fproduct(float farray[]);
int main()
float fmyarray[7] = {4.3,1.8,6.12,3.19,0.01234,0.1,9876.2}, float
fmultiplied;
fmultiplied = fproduct(fmyarray) ;
cout<< "Произведение всех элементов массива равно: " << fmultiplied <<
"\n";
return(0); }
float fproduct(float farray[])

```

```
int i;
float fpartial;
fpartial = farray[0]; for (i = 1; i < 7; i++)
fpartial *= farray[i];
return (fpartial); }
```

Значение первого элемента массива присваивается переменной `fpartial` еще до начала цикла `for`. Вот почему цикл начинается с индекса 1, а не 0.

Тип результата: `double`

В следующем примере функция `dtrigcosine()` находит значение косинуса угла, заданного в градусах.

```
/*
 *      doublef.c
 *  Эта программа на языке C последовательно находит косинусы углов
 *  от 0 до 90 градусов с интервалом в пять градусов.
 */
#include <stdio.h>
#include <math.h>
const double dPi = 3.14159265359; double dtrigcosine(double dangle);
int main() {
int j;
double dcosine;
for(j= 0; j < 91;j+=5) {>
dcosine = dtrigcosine((double) j);
printf("Косинус угла%d градусов равен%.3f\n",j, dcosine); }
return(0);}
double dtrigcosine(double dangle) {
double dpartial;
dpartial = cos((dPi/180.0) * dangle);
return(dpartial);}
```

Для вычисления косинуса в функции `dtrigcosine()` вызывается стандартная функция `cos()`, объявленная в файле `MATH.H`. Но предварительно необходимо преобразовать величину угла из градусов в радианы. Для этого параметр `dangle` делится на 180 градусов и умножается на число π , заданное в программе как константа `dpi`.

Аргументы командной строки

В языках C/C++ имеются средства ввода данных посредством командной строки. Аргументами командной строки называются числовые или строковые значения, которые указываются пользователем вслед за именем приложения при запуске его из командной строки. Подобный механизм позволяет передавать программе информацию, избегая выдачи всевозможных подсказок и приглашений. Вот синтаксис командной строки:

имя_программы аргумент1, аргумент2 ...

Аргументы командной строки обрабатываются в функции `main()`, которая принимает два параметра: `argc` и `argv[]`. Целочисленный параметр `a` где содержит число аргументов командной строки плюс 1 — с учетом имени программы, которое, в принципе, тоже является аргументом. Вторым параметром является указатель на массив строковых указателей. Все аргументы представляют собой строки, поэтому массив `argv[]` имеет тип `char*`. Имена параметров — `argc` и `argv` — не являются стандартными элементами языка, но называть их именно так — общепринятое соглашение, широко применяемое большинством программистов на C/C++.

Ниже на примерах будет показано, как обрабатывать в программах аргументы командой строки различных типов.

Текстовые аргументы

Аргументы, задаваемые в командной строке, всегда передаются в программу в виде наборов символов, что облегчает работу с ними. В следующей программе на языке С от пользователя ожидается ввод в командной строке ряда аргументов. При запуске программы проверяется значение параметра argc. Если оно меньше двух, то пользователю будет предложено перезапустить программу.

```
/*
 *      sargv.c
 *      Эта программа на языке С демонстрирует процесс
 *      считывания строковых аргументов командной строки.
 */
#include <stdio.h>
#include <process.h>
int main(int argc, char *argv[]) {
    int t;
    if(argc < 2) {
        printf("Необходимо ввести несколько аргументов командной строки.\n");
        printf("Повторите запуск программы.\n");
        exit (1);
        forft =1;t < argc; t++)
        .   printf("Аргумент№%d -  %s\n",    t,    argv[t] );
        exit(0);
    }
}
```

Все аргументы, введенные в командной строке, выводятся на экран в той же последовательности. Если были введены числовые значения, они все равно будут рассматриваться как ASCII-символы.

Целочисленные аргументы

Часто в командной строке нужно указывать числовые значения. В таком случае необходимо выполнить преобразование строк в числа. Следующая программа на языке C++ читает из командной строки символьный аргумент и преобразовывает его в целое число с помощью стандартной функции atoi(). Полученное число сохраняется в переменной ivalueи передается в функцию vbinary(), которая выводит его двоичный эквивалент. В функции main() отображаются также восьмеричный и шестнадцатеричный эквиваленты числа.

```
//
//      iargv.cpp
//      Эта программа на языке C++ преобразует аргумент
//      командной строки в целое число.
//
#include <iostream.h>
#include <stdlib.h>
void vbinary (int idigits);
int main (int argc, char *argv[]) {
    int ivalue;
    if (argc != 2) {
        cout<< "Введите в командной строке целое число. \n";
        cout<< "Это число будет преобразовано в двоичный, \n"
        cout<< "восьмеричный и шестнадцатеричный форматы . \n";
        return (0);
    }
    ivalue = atoi(argv[1]);
    vbinary(ivalue) ;
    cout << "В восьмеричном формате:  " << oct << ivalue << "\n";
    cout << "В шестнадцатеричном формате:  " << hex<< ivalue <<  " \n"
    return(0) ; }
void vbinary (int idigits)
```

```

int t = 0;
int iyourarray[50];
while(idigits != 0) {
    iyourarray[t]= (idigits % 2);
    idigits /= 2;
    t++; }
t-- ;
cout<< "В двоичном формате: ";
for(; t >= 0; t--)
    cout << dec << iyourarray[t] ; cout << "\n"; }

```

Функция `vbinary()` и алгоритм нахождения двоичной формы числа уже были рассмотрены нами, только на примере языка С. Здесь же интересно проанализировать, как происходит преобразование числа в восьмеричную и шестнадцатеричную формы.

Для вывода числа в восьмеричном формате используется следующая запись:

```
cout<< "В восьмеричном формате: " << oct << ivalue << "\n";
```

Чтобы вывести то же число в шестнадцатеричном формате, достаточно просто поменять флаг окна флаг `hex`:

```
cout<< "В шестнадцатеричном формате: " << hex << ivalue << "\n";
```

При отсутствии дополнительного форматирования шестнадцатеричные цифры `a`, `b`, `c`, `d`, `e` и `f` будут отображаться в нижнем регистре. Более подробно о средствах форматирования, используемых в языке С++, рассказано в главе "Основы ввода-вывода в языке С++". Среди прочего в ней объясняется, как управлять выводом шестнадцатеричных цифр в верхнем и нижнем регистре.

Аргументы с плавающей запятой

Следующая программа на языке С принимает в командной строке несколько значений углов в градусах. Полученные данные используются для вычисления косинусов.

```

/*
 *      fargv.c
 *      Эта программа на языке С демонстрирует процесс считывания
 *      аргументов
 *      командной строки с плавающей запятой.
 */
#include <stdio.h>
#include <math.h>
#include <process.h>
const double dPi = 3.14159265359;
int main(int argc, char *argv[]) {
    int t;
    double ddegree;
    if(argc< 2) {
        printf("Введите в командной строке значения углов в градусах.\n") ,
        printf("Программа вычислит косинусы углов.\n"), •
        exit(1);
    }
    for(t= 1;  t <  argc;  t++)    {
        ddegree = atof(argv[t]);
        printf("Косинус угла  %f  равен  %.3f\n",  ddegree,
        cos((dPi/180.0)  *  ddegree)); )
        exit(0);
    }
}

```

Функция `atof()` преобразовывает строковые значения в значения типа `double`. Для вычисления косинуса используется библиотечная функция `cos()`.

Дополнительные особенности функций

Макроподстановка функций

В C++ при использовании функций доступны некоторые дополнительные возможности. Например, можно встраивать весь код функции непосредственно по месту ее вызова. Такие функции, называемые inline-функциями, встраиваются в программу автоматически в процессе компиляции, а в результате может значительно сокращаться время выполнения программы, особенно если в ней часто вызываются короткие функции.

Ключевое слово `inline` является особым видом спецификатора. Его можно представить как рекомендацию компилятору C++ подставить в программный код тело функции по месту ее вызова. Компилятор может проигнорировать эту рекомендацию, если, например, функция слишком велика. Макроподстановка, как правило, применяется, когда какая-нибудь небольшая функция вызывается в программе много раз.

```
//
//  inline. cpp
//  Эта программа на языке C++ содержит пример макроподстановки
//  функции.
//
#include <iostream.h>
inline long squareit(int iValue) (return iValue * iValue; }
int main ()
{
    int iValue;
    for(iValue      = 1; iValue <= 10; iValue++)
        cout<<      "Квадрат числа " << iValue<< " равен "
        << squareit(iValue) << "\n";
    return(0); }
```

Функция `squareit()`, возвращающая квадрат целочисленного аргумента `iValue`, описана со спецификатором `inline`. Когда в программе будет обнаружен вызов данной функции, компилятор подставит в этом месте строку `iValue*iValue`. Другими словами, компилятор заменяет вызов функции ее телом, присваивая при этом аргументам функции их фактические значения.

Преимущество использования inline-функций вместо макросов состоит в том, что появляется возможность контроля за ошибками. Вызов макроса с аргументами неправильного типа останется незамеченным компилятором. А inline-функций имеют прототипы, как и все другие функции, поэтому компилятор проверит соответствие типов формальных аргументов в описании функции и фактических аргументов при ее вызове.

Перегрузка функций

В C++ также допускается использование перегруженных функций. Под перегрузкой понимается создание нескольких прототипов функции, имеющих одинаковое имя. Компилятор различает их по набору аргументов. Перегруженные функции оказываются весьма полезными, когда одну и ту же операцию необходимо выполнить над аргументами разных типов.

В следующей программе создаются два прототипа функции с одним именем и общей областью видимости, но аргументами разных типов. При вызове функции `adder()` будут обрабатываться данные либо типа `int`, либо типа `float`.

```
//
//  overload.cpp
//  Эта программа на языке C++ содержит пример перегрузки функции.
//
#include <iostream.h>
int adder (int iarray[]);
float adder (float f array []);
int main() {
```

```

int iarray[7] = {5,1, 6, 20,15,0, 12};
float farray[7] = {3.3,5.2,0.05,1.49,3.12345,31.0,2.007};
int isum;
float fsum;
isura      = adder (iarray) ;
fsum= adder (f array) ;
cout<< "Сумма массива целых чисел равна " << isura << "\n";
cout<< "Сумма массива дробных чисел равна " << fsum<< "\n";
return ( 0 ).;
}
int adder(int iarrayt) {
int i;
int ipartial;
ipartial = iarray[0]; for(i= 1; i < 7; i++) ipartial += iarray[i];
return(ipartial);
}
float adder (float farrayf) {
int i;
float f partial,
fpartial = f array [0];
for(i= 1; i < 7; i++)
fpartial += farray[i];
return (fpartial) ; }

```

При использовании перегруженных функций часто допускается ряд ошибок. Например, если функции отличаются только типом возвращаемого значения, но не типами аргументов, такие функции не могут иметь одинаковое имя. Также недопустим следующий вариант перегрузки:

```

int имя_функции(int имя_аргумента) ;
int имя_функции(int имя_аргумента) ; // недопустимая
перегрузка имени

```

Это неправильно, поскольку аргументы имеют одинаковый тип.

Функции с переменным числом аргументов

Если точное количество формальных аргументов функции изначально не известно, допускается указывать многоточие в списке аргументов:

```

void имя_функции(int first, float second, . . . ) ;

```

Данный прототип говорит компилятору о том, что за аргументами first и second могут следовать и другие аргументы, если возникнет такая необходимость. При этом тип последних не контролируется компилятором.

В следующей программе, написанной на языке C, создается функция `vsmalltest()` с переменным числом аргументов. Текст этой программы может вам показаться трудным для понимания, поскольку мы еще не рассматривали подробно работу с указателями. В таком случае рекомендуем вернуться к этой программе после прочтения главы "Указатели".

```

/*
 *      ellipsis. c
 *      Эта программа на языке C содержит пример функции с переменным
числом
 *      аргументов и демонстрирует использование макросов va_arg,
va_start и va_end.
 */
#include <stdio.h>
#include <stdarg.h>
#include <string.h>
void vsmallest (char *szmessage, ...);

```

```

int main() {
    vsmallest("Выводим %d целых чисел, %d %d %d", 10, 4, 1);
    return (0); }
void vsmallest (char *szmessage, ...)
{
    int inumber_of_percent_ds = 0;
    va_list type_for_ellipsis;
    int ipercent_d_format = 'd';
    char *pchar;
    pchar = strchr (szmessage, ipercent_d_format) ;
    while (*++pchar != '\0'){ pchar++;
    pchar = strchr (pchar, ipercent_d_format) ; inumber_of_percent_ds++;
    }
    printf("Выводим %d целых чисел, ", inumber_of_percent_ds) ;
    va_start(type_for_ellipsis, szmessage);
    while(inumber_of_percent_ds-->0)
    printf(" %d", va_arg(type_for_ellipsis, int));
    va_end(type_for_ellipsis); }

```

Функция `vsmallest()` ожидает двух формальных аргументов: указателя на строку и списка неопределенной длины. Естественно, функция должна иметь возможность каким-то образом определить, сколько же аргументов она получила на самом деле. В данной программе эта информация передается в строковом аргументе.

Созданная нами функция `vsmallest()` частично имитирует работу стандартной функции `printf()`. Аргумент `szmessage` рассматривается как строка форматирования, в которой подсчитывается число спецификаций `%d`. Полученная информация позволяет вычислить количество дополнительных аргументов.

Функция `strchr()` возвращает адрес позиции спецификатора `d` в строке форматирования. Первый элемент `%d` игнорируется, поскольку на его месте будет выведено общее число аргументов. В первом цикле `while` определяется количество спецификаторов `d` в строке `szmessage`, и полученное значение заносится в переменную `inumber_of_percent_ds`. По завершении цикла на экран выводится первая часть сообщения.

Макрос `va_start()` устанавливает указатель `type_for_ellipsis` (обратите внимание на его тип — `va_list`) в начало списка аргументов функции. Второй параметр макроса является именем обязательного аргумента анализируемой функции, который стоит непосредственно перед многоточием. Макрос `va_arg()` возвращает очередной аргумент из списка. Второй параметр макроса указывает на тип возвращаемого аргумента (в нашей программе это тип `int`). Макрос `va_end()` очищает указатель `type_for_ellipsis`, делая его равным нулю.

Область видимости переменных

При работе с переменными, имеющими разную область видимости, часто возникают непредвиденные ошибки, называемые побочными эффектами. Например, в программе могут быть объявлены две переменные с одинаковым именем, но одна локально, внутри функции, а другая на уровне всего файла. Правила определения области видимости говорят о том, что в пределах функции локальная переменная имеет преимущество перед глобальной, делая последнюю недоступной. Все это звучит достаточно просто, но давайте рассмотрим ряд проблем, часто возникающих в программах и, на первый взгляд, не совсем очевидных.

Попытка получить доступ к переменной вне ее области видимости

В следующем примере в функции `main()` объявляются четыре локальные переменные. В программе как будто бы нет никаких ошибок. Тем не менее, когда функция `iproduct()` попытается обратиться к переменной `ip`, она не сможет ее обнаружить. Почему? Потому что область видимости этой переменной ограничена только функцией `main()`.

```

/*
 *      scope.c
 *
 * Эта программа на языке C иллюстрирует проблему неправильного

```

```

*   определения области видимости переменной. Во время компиляции
*   программы появится сообщение об ошибке.
*/

```

```

#include <stdio.h>
int iproduct(int iw, int ix);
int main () {
int il = 3;
int im = 7;
int in = 10;
int io;
io = iproduct(il, im) ;
printf("Произведение чисел равно %d\n",io) ;
return(0); }
int iproduct (int iw, int ix)
{
int iy;
iy = iw * ix * in;
return (iy); }

```

Компилятор выдаст сообщение об ошибке, в котором говорится о том, что в функции iproduct() обнаружен нераспознанный идентификатор in. Чтобы решить эту проблему, нужно сделать переменную in глобальной.

В следующем примере переменная in описана как глобальная. В результате обе функции, как main(), так и iproduct(), могут использовать ее. Обратите также внимание на то, что и та, и другая функция может изменить значение переменной in.

```

/*
*   fscope.c
*   Это исправленная версия предыдущей программы. Проблема решена
*   путем
*   объявления переменной in как глобальной.
*/
#include <stdio.h>
int iproduct (int iw, int ix) ; int in = 10;
int main() {
int il = 3;
int im = 7;
int io;
io = iproduct (il,im) ;
printf("Произведение чисел равно %d\n",io) ;
return (0);
}
int iproduct(int iw, int ix)
int iy;
iy = iw * ix * in;
return(iy);

```

Эта программа будет корректно скомпилирована, и на экране появится результат - 210.

Как уже было сказано, локальная переменная перекрывает одноименную глобальную переменную. Ниже показан вариант предыдущей программы, иллюстрирующий данное правило.

```

/*
*   Iscope.c
*   Эта программа на языке C иллюстрирует взаимоотношение
*   между одноименными локальной и глобальной переменными.
*   Функция iproduct() находит произведение трех переменных,

```

```

*   из которых две передаются как аргументы функции, а еще
*   одна, in, объявлена и как глобальная, и как локальная.
*/

```

```

#include <stdio.h>
int iproduct(int iw, int ix);
int in = 10;
int main()
int il = 3; int im = 7;
int io;
io = iproduct(il,im) ;
printf("Произведение чисел равно %d\n",io) ;
return (0);
int iproduct(int iw, int ix) {
int iy;
int in = 2;
iy = iw * ix * in;
return(iy); }

```

В этом примере переменная `in` описывается дважды: на уровне файла и на уровне функции. В функции `iproduct()`, где эта переменная объявлена как локальная, будет использовано "локальное" значение. Поэтому результатом умножения будет $3*7*2 = 42$.

Оператор расширения области видимости

В следующей программе, написанной на языке C++, все работает нормально до момента вывода информации на экран. Объект `cout` правильно отобразит значения переменных `il` и `im`, но для переменной `in` будет выбрано "глобальное" значение. Вследствие этого пользователю будет представлен неправильный результат: $3*7*10 = 42$. Как вы уже поняли, ошибка возникает из-за того, что в функции `iproduct()` используется локальная переменная `in`.

```

//
//   scopel.cpp
//   Эта программа на языке C++ содержит логическую ошибку.
//   Функция iproduct() находит произведение трех переменных,
//   используя при этом значение локальной переменной in.
//   В то же время в выводимых данных программа сообщает
//   о том, что значение переменной in равно 10.
//
#include <iostream.h>
int iproduct (int iw, int ix);
int in = 10;
int main()
{
int il = 3;
int im = 7 ;
int io;
io = iproduct (il,im) ;
cout << il << " * " << im << " * " << in << " = " << io << "\n";
return (0);
}
int iproduct(int iw, int ix) {
int iy; int in = 2;
iy = iw * ix * in;
return (iy);}

```

Что же нужно сделать, чтобы получить правильный результат? Для этого достаточно воспользоваться оператором расширения области видимости (`::`), о котором мы уже упоминали:

```

iy = iw * ix * ::in;

```

Следующая программа иллюстрирует сказанное.

```
//
//  scope2.cpp
//  Это исправленная версия предыдущего примера. Проблема решена путем
//  использования оператора расширения области видимости (::).
//
#include <iostream.h>
int iproduct(int iw, int ix) ; int in = 10;
int main ()
int il = 3; int im = 7; int io;
io = iproduct(il,im) ;
cout << il << " * " << im << " * " << in << " = " << io << "\n";
return (0);
}
int iproduct(int iw, int ix)
int iy;
int in = 2;
iy = iw * ix * (::in); return(iy);}
```

Переменная `in` вместе с оператором `::` для наглядности взяты в скобки, хотя в этом нет необходимости, так как данный оператор имеет самый высокий приоритет среди остальных. Таким образом, на экран будет выведен правильный результат: `"3*7*10 = 210"`.

Глава 8. Массивы

- Что такое массивы
- Свойства массивов
- Объявления массивов
- Инициализация массивов
 - Инициализация по умолчанию
 - Явная инициализация
 - Инициализация безразмерных массивов
- Доступ к элементам массива
- Вычисление размера массива в байтах
- Выход за пределы массива
- Массивы символов
- Многомерные массивы
- Массивы как аргументы функций
 - Передача массивов функциям в языке C
 - Передача массивов функциям в языке C++
- Функции работы со строками и массивы символов
 - Функции `gets()`, `puts()`, `fgets()`, `fputs()` и `sprintf()`
 - Функции `strcpy()`, `strcat()`, `strncmp()` и `strlen()`

В этой главе вы узнаете, как создавать массивы данных и работать с ними. В C/C++ темы массивов, указателей и строк взаимосвязаны, во многих книгах они даже рассматриваются в одной главе. С нашей точки зрения, это не лучший подход, поскольку часто для работы с массивами не требуется глубокого знания указателей. Кроме того, с массивами в целом связан достаточно большой объем материала, и параллельное изучение указателей может совершенно запутать дело. В то же время, без уяснения принципов использования указателей вы не сможете полноценно работать с массивами. Поэтому главу "Указатели" можно рассматривать как завершение дискуссии о массивах.

Что такое массивы

Массив можно представить как переменную, содержащую упорядоченный набор данных одного типа. К каждому элементу массива можно получить доступ по его адресу. В языках C/C++ массив не является стандартным типом данных. Напротив, он сам имеет тип: `char`, `int`, `float`, `double` и т.д. Допускается создавать массивы массивов, указателей, структур и др. Принципы построения массивов и работы с ними в основе своей одинаковы в C и C++.

Свойства массивов

Ниже перечислены четыре основных принципа, определяющих свойства массивов: в массиве хранятся отдельные значения, которые называются элементами;

- все элементы массива должны быть одного типа;

- все элементы массива сохраняются в памяти последовательно, и первый элемент имеет нулевое смещение адреса, т.е. нулевой индекс;
- имя массива является константой и содержит адрес первого элемента массива.

Поскольку все элементы массива имеют одинаковый, заранее установленный размер, а имя массива содержит адрес первого его элемента, то нетрудно вычислить адрес любого другого элемента. Но при этом должен соблюдаться еще один принцип — строгой последовательности хранения в памяти всех элементов массива, от нулевого до последнего, причем первый элемент имеет наименьший адрес, а последний — наибольший.

Имя массива представляет собой константное значение, которое не изменяется в ходе выполнения программы, поэтому оно не может размещаться слева от оператора присваивания, т.е. не является левосторонним значением. Если бы этого ограничения не существовало, программа могла бы изменять содержимое имени, а смысл подобного изменения состоял бы в замене адреса нулевого элемента массива. На первый взгляд кажется, что данное ограничение малозначительно, но на самом деле определенные выражения, выглядящие вполне корректными, оказываются недопустимыми.

Объявления массивов

Ниже даны примеры объявления массивов:

```
int iarray[12]; /* массив из двенадцати целых чисел */
char carray[20]; /* массив из двадцати символов */
```

Как и в случае обычных переменных, объявление массива начинается с указания типа данных, после чего следует имя массива и пара квадратных скобок, заключающих константное выражение, которое определяет размер массива. Внутри квадратных скобок может стоять только константа, но не имя переменной, чтобы компилятор точно знал, какой объем памяти резервировать. Таким образом, размер массива должен быть известен заранее и не может быть изменен в ходе выполнения программы.

Вот как устанавливаются размеры массивов с помощью констант:

```
#define iARRAY_MAX 20
#define fARRAY_MAX 15
int iarray[iARRAY_MAX]; char farray[fARRAY_MAX];
```

Подобное использование макроконстант позволяет избежать ошибок, связанных с обращением к несуществующим элементам массива. Например, последовательный доступ к массиву часто организуется с помощью цикла for:

```
#include <stdio.h>
#define iARRAY_MAX 20
int iarray[iARRAY_MAX];
main () {
    int i;
    for(i= 0; i < iARRAY_MAX; i++) {
    }
    return(-0); }
```

Инициализация массивов

Массив можно инициализировать одним из трех способов:

- при создании массива — используя инициализацию по умолчанию (этот метод применяется только для глобальных и статических массивов);
- при создании массива — явно указывая начальные константные значения;
- в процессе выполнения программы — путем записи данных в массив.

При создании в массив могут быть занесены только константные значения. Впоследствии в массив можно записывать и значения переменных.

Инициализация по умолчанию

В соответствии со стандартом ANSI глобальные массивы (расположенные вне любой функции), а также массивы, объявленные статическими внутри функции, по умолчанию заполняются нулями, если не заданы начальные значения элементов массива. Массивы указателей заполняются значениями null. Проверить вышесказанное можно на следующем примере:

```
/*
.*      initar.c
*   Эта программа на языке C демонстрирует инициализацию массивов,
*   выполняемую по умолчанию.
*/
#include <stdio.h>
#define iGLOBAL_ARRAY_SIZE 10
#define iSTATIC_ARRAY_SIZE 20
int iglobal_array[iGLOBAL_ARRAY_SIZE]; /* глобальный массив */
main () {
    static int istatic_array[iSTATIC_ARRAY_SIZE]; /* статический массив */
    int i;
    for (i = 0; i < iGLOBAL_ARRAY_SIZE; i++)
        printf ("iglobal_array [%d]:%d\n",i, iglobal_array [i] )
        ;
    for(i= 0; i < iSTATIC_ARRAY_SIZE; i++)
        printf("istatic_array[%d]: %d\n", i, istatic_array[i]);
    return(0); }
```

После запуска программы на экран будут выведены нулевые значения, присвоенные элементам массива по умолчанию. Данная программа выявляет еще один существенный момент работы с массивами: первый элемент массива всегда имеет нулевой индекс. Это связано с тем, что создатели языка C стремились максимально приблизить его к ассемблерным языкам, где первый элемент таблицы всегда имеет нулевое смещение.

Явная инициализация

Согласно стандарту ANSI элементам как глобальных, так и локальных массивов можно явно присваивать начальные значения. В следующем фрагменте программы содержится объявление четырех массивов, инициализируемых явно:

```
int iarray[3]      =      {-1,0, 1};
static float fpercent[4] =          {1.141579,0.75,55E0,-.33E1};
static int  idecimal[3] =          {0,1, 2, 3, 4, 5, 6, 7, 8, 9};
char cvowels[]    =          {'A','a','E','e','I','i','O','o','U','u'};
```

В первой строке создается массив `iarray`, содержащий три целочисленных элемента, значения которых, разделенные запятыми, указаны в фигурных скобках. В результате еще при запуске программы резервирование ячеек памяти для массива `iarray` будет сопровождаться одновременной записью значений в эти ячейки. Обратите внимание не только на удобство этого метода, но и на то, что инициализация массива выполняется еще до начала выполнения программы. Некоторые компиляторы позволяют проводить инициализацию только глобальных и статических массивов, как, например, во второй строке программы.

В третьей строке показан пример задания большего числа элементов массива, чем в нем на самом деле содержится. Многие компиляторы рассматривают подобную ситуацию как ошибку, тогда как другие автоматически увеличивают размер массива, чтобы вместить дополнительные элементы. Компилятор `Microsoft Visual C++` выдаст ошибку вида `"too many initializers"` (слишком много инициализаторов). В противоположной ситуации, когда при инициализации указано меньше значений, чем элементов массива, оставшиеся элементы по умолчанию примут нулевые значения. С учетом этого можно вообще не задавать размер массива, как в четвертой строке программы. Количество значений, указанных в фигурных скобках, автоматически определит размер массива.

Инициализация безразмерных массивов

Размер массива можно задать либо в квадратных скобках, либо указывая список начальных значений при инициализации массива. В большинстве компиляторов разрешены оба метода. Для примера рассмотрим создание часто используемых во многих программах сообщений об ошибках. Задать соответствующий массив символов можно двумя способами. Вот первый из них:

```
char    szInput_Errdr[41]    = "Введите значения от 0 до 9.\n";
char    szDevice_Error[18]   = "Диск недоступен.\n";
char    szMonitor_Error[49]  = "Для работы программы необходим цветной
монитор. \n";
char    szWarning[36]= "Эта операция приведет к удалению файла!\n";
```

Здесь необходимо предварительно подсчитать число символов в строке, не забыв к полученному числу прибавить 1 для символа \0, обозначающего конец строки. Это нудная работа, к тому же чреватая ошибками. Можно позволить компилятору автоматически вычислить размер массива, как показано ниже:

```
char    szInput_Error[]      = "Введите значения от 0 до 9.\n";
char    szDevice_Error[]     = "Диск недоступен.\n";
char    szMonitor_Error[]    = "Для работы программы необходим
цветной монитор.\n";
char    szWarning[]         = "Эта операция приведет к удалению файла!\n";
```

В процессе инициализации массива, для которого не задан размер, компилятор автоматически создаст массив такого размера, чтобы вместить все указанные элементы.

Доступ к элементам массива

При объявлении переменной происходит резервирование одной или нескольких ячеек памяти, а в специальную таблицу лексем программы заносится имя переменной и адрес связанных с ней ячеек. Например, в следующей строке программы резервируется ячейка памяти для хранения целочисленного значения переменной с именем `iweekend`.

```
int iweekend;
```

В случае показанного ниже массива `iweek` происходит резервирование не одной, а семи ячеек памяти, каждая из которых хранит одно целочисленное значение.

```
int iweek[7] ;
```

Рассмотрим, как организуется доступ к одиночной ячейке памяти, связанной с переменной `iweekend`, и к семи ячейкам памяти, связанным с массивом `iweek`. Чтобы получить значение, хранящееся в переменной `iweekend`, достаточно обратиться к этой переменной по имени. При доступе к массиву следует дополнительно указать индекс, представляющий собой порядковый номер элемента массива, к которому вы хотите обратиться. В следующем примере последовательно выполняется обращение ко всем элементам созданного массива:

```
iweek[0];
iweek[1];
iweek[2];
iweek[3];
.
.
.
iweek[6];
```

При обращении к элементу массива в квадратных скобках указывается целочисленный индекс, представляющий собой смещение адреса по отношению к базовому адресу первого элемента.

Начинающие программисты часто допускают ошибку, считая, что первый элемент имеет индекс 1. Для обращения к первому элементу массива следует указывать индекс 0, так как смещение первого элемента относительно самого себя, естественно, является нулевым. А например, к третьему элементу следует обращаться по индексу 2, так как он на две ячейки смещен по отношению к первому элементу.

При работе с массивами квадратные скобки используются в двух разных случаях. Когда происходит объявление массива, число в квадратных скобках указывает количество резервируемых ячеек памяти:

```
int iweek[7];
```

Если же необходимо получить доступ к определенному элементу массива, вслед за именем массива в квадратных скобках указывается индекс элемента:

```
iweek[3];
```

Для описанного выше массива `iweek` следующее выражение присваивания является неправильным:

```
iweek[7] = 53219;
```

В массиве `iweek` нет элемента со смещением 7 по отношению к первому элементу, другими словами — восьмого элемента. Поскольку массив содержит только семь элементов, данная строка вызовет ошибку. Вы как программист ответственны за то, чтобы индексы в обращениях к массиву имели допустимое значение.

Рассмотрим следующие объявления массива, переменных и константы:

```
#define iDAYS_OF_WEEK 7
int iweek[iDAYS_OF_WEEK];
int iweekend = 1;
int iweekday = 2;
```

и выражения:

```
iweek[2];
iweek[iweekday] ;
iweek[iweekend + iweekday];
iweek[iweekday - iweekend];
iweek[iweekend - iweekday];
```

Первые две строки содержат обращение к третьему элементу массива. В первом случае индекс задается числовой константой, а во втором случае для этих целей используется имя переменной. Последующие три строки демонстрируют применение выражений для установки индексов. Важно только, чтобы результатом выражения было логически корректное число. Например, в третьей строке получаем индекс 3, который указывает на четвертый элемент массива. В четвертой строке индекс равен 1 и указывает на второй элемент массива, а вот последняя строка ошибочна, так как значение индекса -1 недопустимо.

Для того чтобы получить доступ к элементу массива, нет необходимости учитывать размер занимаемой им памяти, так как подобная работа выполняется компилятором автоматически. Предположим, например, что вам нужно получить значение третьего элемента массива `iweek`, содержащего целые числа. Как было сказано в главе "Работа с данными", в различных системах для представления данных типа `int` могут использоваться ячейки памяти разных размеров: иногда 2 байта, иногда 4. Но независимо от системы вы в любом случае сможете получить доступ к третьему элементу массива, используя выражение `iweek[2]`. Индекс указывает на порядковый номер элемента в массиве вне зависимости от того, сколько байтов занимает каждый элемент.

Вычисление размера массива в байтах

Вам уже известен оператор `sizeof`, возвращающий размер указанного операнда в байтах. Этот оператор можно использовать с переменными любых типов, за исключением битовых полей. Часто оператор `sizeof` применяют, чтобы определить размер переменной, тип которой в разных системах может иметь разную размерность. Как говорилось выше, в одном случае для представления целочисленного значения отводится 2 байта, в другом — 4 байта. Поэтому, например, при работе с массивами, размещаемыми в памяти динамически, необходимо точно знать, сколько памяти запрашивать у операционной системы. Следующая программа автоматически вычислит и отобразит на экране число байтов, занимаемых массивом из семи целочисленных элементов:

```
/*
 *      sizeof.c
 *      Эта программа на языке C демонстрирует использование
 *      оператора sizeof для определения физического размера массива.
 */
#include <stdio.h>
```

```

#define iDAY_OF_WEEK 7
main () {
int iweek[iDAY_OF_WEEK] = {1,2, 3, 4, 5, 6, 7} ;
printf("Массив iweek занимает%d байт.\n", (int) sizeof(iweek));
return(0);
}

```

Вы можете спросить, почему значение, возвращаемое оператором sizeof, приводится к типу int. Дело в том, что в соответствии со стандартом ANSI данный оператор возвращает значение типа size_t, поскольку в некоторых системах одного лишь типа int оказывается недостаточно для представления размерностей данных некоторых типов. В нашем примере операция приведения необходима также для того, чтобы выводимое число соответствовало спецификации %d функции printf().

С помощью следующей программы можно проследить, как размещаются в памяти элементы массива iarray.

```

/*
 *      array.c
 *      Эта программа на языке C позволяет убедиться
 *      в смежном размещении в памяти элементов массива.
 */
#include <stdio.h>
#define iDAYS 7
main ()
{
int index, iarray[iDAYS];
printf("sizeof(int)= %d\n\n", (int)sizeof(int));
for(index = 0; index < iDAYS; index++)
printf("siarray[%d]= %X\n", index, &iarray[index]);
return(0); }

```

Запустив программу, вы получите примерно следующий результат:

```

sizeof(int)= 4
&iarray[0]=          64FDDC
Siarrayfl] =          64FDE0
&iarray[2]=          64FDE4
&iarray[3]=          64FDE8
Siarray[4]=          64FDEC
Siarray[5]=          64FDFO
&iarray[6]=          64FDF4

```

Обратите внимание на то, что оператор взятия адреса & можно применять к любым переменным, в том числе к элементам массива. Над элементом массива можно выполнять те же операции, что и над любой другой переменной, использовать его в выражениях, присваивать значения и передавать в качестве аргумента функциям. В рассматриваемом примере по отображаемым адресам можно убедиться, что на каждый элемент действительно отводится 4 байта памяти.

Ниже показан аналог этой же программы на языке C++:

```

//
//      array.cpp
//      Это версия предыдущей программы на языке C++.
//
#include <iostream.h>
#define iDAYS 7
main () {
int index, iarray[iDAYS];
cout << "sizeof (int) = " << (int)sizeof(int) << "\n\n";
}

```

```
for(index = 0; index < iDAYS; index++)
cout << "siarray["<< index << "] = " << &iarray[index] << "\n";
return(0); }
```

Выход за пределы массива

При работе с массивами не следует забывать о том, что индекс в обращении к элементу не должен превышать размер массива. Помните, что поскольку языки С и С++ создавались как альтернатива ассемблерным языкам, функции контроля подобных ошибок не включены в компилятор, чтобы уменьшить его код. Например, при компиляции следующей программы ошибки не будут обнаружены, тем не менее, при ее выполнении могут быть произвольно изменены значения других переменных, и выполнение программы может завершиться крахом из-за того, что в ней есть обращения за пределы массива:

```
/*
 *      norun.c
 *      НЕ запускайте эту программу.
 */
#include <stdio.h>
#define iMAX 10
#define iOUT_OF_RANGE 50
main()
{
int inot_enough_room[iMAX] , index;
for (index = 0; index < iOUT_OF_RANGE; index++)
inot_enough_room[index] = index;
return(0);
}
```

Массивы символов

Хотя языки С и С++ поддерживают тип данных `char`, в переменных этого типа могут храниться только одиночные символы, но не строки текста. Если в программе необходимо работать со строками, их можно создавать как массивы символов. В таком массиве для каждого символа строки отводится своя ячейка, а последний элемент содержит символ конца строки — `\0`.

В следующем примере создаются три символьных массива. Массив `szvehicle1` заполняется символом за символом с помощью оператора присваивания. Данные в массив `szvehicle2` заносятся с помощью функции `scanf()`, а массив `szvehicle3` инициализируется константной строкой.

```
/*
 *      string.c
 *      Эта программа на языке С демонстрирует работу с массивами
 *      символов.
 */
#include <stdio.h>
main() {
char      szvehicle1[7],          /* машина */
szvehicle2[8];                  /* самолет */
static char szvehicleS[8] = "корабль"; /* корабль */
szvehicle1[0]      =      'м'
szvehicle1[1]      =      'а'
szvehicle1 [2]      =      'ш'
szvehicle1[3]      =      'и'
szvehicle1[4]      =      'н'
szvehicle1[5]      =      'а'
szvehicle1[6]      =      '\0';
printf ("\n\n\tВведите слово --> самолет ") ;
scanf("%s",szvehicle2);
```

```
printf("%s\n", szvehicle1);
printf("%s\n", szvehicle2);
printf("%s\n", szvehicle3);
return(0);
}
```

Хотя слово "машина", сохраненное в массиве `szvehicle1`, состоит из шести букв, массив содержит семь элементов: по одному для каждой буквы плюс еще один для символа конца строки. То же самое справедливо и для других двух массивов. Вспомните также, что массив `szvehicle3` можно задать путем указания последовательности символов в фигурных скобках:

```
static char szvehicleS[8] = {'к', 'о', 'п', 'а', 'б', 'л', 'ь', '\\0'};
```

Разница заключается в том, что здесь необходимо явно указывать символ конца строки, тогда как в применяемом в программе способе добавление данного символа происходит автоматически. Объявление массива можно записать также следующим образом:

```
static char szvehicleS[] = "корабль";
```

При этом размер массива определяется компилятором автоматически.

Часто содержимое массива запрашивается у пользователя с помощью функции `scanf()`, как в случае с массивом `szvehicle2`. В нашем примере в функции `scanf()` применяется спецификация `%s`, означающая ввод строки. В результате функция пропустит ведущие пробельные литеры (символы пробела, табуляции и конца абзаца), после чего запишет в массив последовательность символов вплоть до следующей пробельной литеры. В завершение к набору символов будет автоматически добавлен символ конца строки. Помните, что при объявлении массива следует указать такой размер, чтобы в массиве поместился весь вводимый текст вместе с символом `\\0`.

Рассмотрим еще раз строку программы, в которой в массив `szvehicle2` записывается информация:

```
scanf ("%s", szvehicle2);
```

Не удивил ли вас тот факт, что имени `szvehicle2` не предшествует оператор взятия адреса `&`? Это объясняется тем, что имя массива, в отличие от имен других переменных, уже является адресом первого его элемента.

Когда в функции `printf()` задана спецификация `%s`, то предполагается наличие аргумента также в виде адреса строки. Строка текста будет выведена на экран полностью, за исключением завершающего нулевого символа.

Ниже показана программа на языке C++, аналогичная рассмотренной выше:

```
//
//      string. cpp
//      Это версия предыдущей программы на языке C++.
//
#include <iostream.h>
main () {
char szvehicle1 [7],      // машина
szvehicle2 [8];          // самолет
static char szvehicleS[8] = "корабль"; // корабль
szvehicle1[0]             =      'м'
szvehicle1[1]             =      'а'
szvehicle1[2]             =      'ш'
szvehicle1[3]             =      'и'
szvehicle1[4]             =      'н'
szvehicle1[5]             =      'а'
szvehicle1[6]             =      '\\0';
cout<< "\\n\\n\\tВведите слово --> самолет ";
cin >> szvehicle2;
cout << szvehicle1 << "\\n";
cout << szvehicle2 << "\\n";
```

```
cout << szvehicleS << "\n";
return(0);
}
```

При выполнении любой из приведенных программ на экран будет выведено следующее:

```
машина
самолет
корабль
```

Многомерные массивы

Под размерностью массива понимают число индексов, которые необходимо указать для получения доступа к отдельному элементу массива. Все массивы, рассмотренные до сих пор, были одномерными и требовали указания только одного индекса. Чтобы определить размерность массива, достаточно посмотреть на его объявление. Если в нем указана только одна пара квадратных скобок ([]), то перед нами одномерный массив, если две ([][]), то двухмерный, и т.д. Массивы с более чем одной размерностью называются многомерными. В реальных программах размерность массива не превышает трех.

В следующей программе создается двухмерный массив:

```
/*
 *      2darray.c
 *      Эта программа на языке C демонстрирует работу с двухмерным
 *      массивом.
 */
#include <stdio.h>
#define iROWS 4
#define iCOLUMNS 5
main() {
    int irow;
    int icolumn;
    int istatus [iROWS] [iCOLUMNS] ;
    int iadd;
    int imultiple;
    for (irow = 0; irow < iROWS; irow++)
        for (icolumn = 0; icolumn < iCOLUMNS; icolumn++)
            { iadd = iCOLUMNS - icolumn; imultiple = irow; istatus [irow][icolumn]
            =
                (irow+1) *icolumn +
                iadd*imultiple;
            }
    for(irow = 0; irow < iROWS; irow++)
    { printf("ТЕКУЩАЯ СТРОКА: %d\n", irow);
      printf("СМЕЩЕНИЕ ОТ НАЧАЛА МАССИВА:\n");
      for(icolumn = 0; icolumn < iCOLUMNS; icolumn++)
          printf(" %d ", istatus[irow][icolumn]); printf("\n\n");
      }
    return(0);}
```

В программе используются два цикла for, в которых каждый элемент массива инициализируется значением смещения этого элемента относительно первой ячейки массива. Созданный массив имеет 4 строки (константа irows) и 5 столбцов (константа icolumns), что в сумме дает 20 элементов. Многомерные массивы представляются в памяти компьютера в виде линейной последовательности элементов, при этом группировка по индексам осуществляется справа налево, т.е. от самого правого индекса к самому левому.

Хотя процедура вычисления смещения может показаться несколько запутанной, получить доступ к любому элементу многомерного массива очень просто:

```
istatus[irow][icolumn] = ...
```

В результате выполнения программы на экран будет выведена следующая информация:

ТЕКУЩАЯ СТРОКА: 0 СМЕЩЕНИЕ ОТ НАЧАЛА МАССИВА: 01234

ТЕКУЩАЯ СТРОКА: 1 СМЕЩЕНИЕ ОТ НАЧАЛА МАССИВА: 56789

ТЕКУЩАЯ СТРОКА: 2 СМЕЩЕНИЕ ОТ НАЧАЛА МАССИВА: 10 11 12 13 14

ТЕКУЩАЯ СТРОКА: 3 СМЕЩЕНИЕ ОТ НАЧАЛА МАССИВА: 15 16 17 18 19

Многомерные массивы можно инициализировать так же, как и одномерные, что иллюстрируется следующей программой:

```
/*
 *      2dinit.c
 *      Эта программа на языке C демонстрирует инициализацию двухмерного
 массива.
 */
#include <stdio.h>
#include <raath.h>
#define iBASES 6
#define iEXPONENTS 3
#define iBASE 0
#define iRAISED_TO 1
#define iRESULT 2
main () {
double dpowers[iBASES][iEXPONENTS] = {
1,1,1,0,
2,2,2,0,
3,3,3,0,
4,4,4,0,
5,5,5,0,
6,6,6,0,
};
int irow_index;
for(irow_index = 0; irow_index < iBASES; irow_index++)
dpowers[irow_index][iRESULT] = pow(dpowers[irow_index][iBASE],
dpowers[irow_index][iRAISED_TO]);
for(irow_index = 0; irow_index < iBASES; irow_index++) {
printf(" %d\n", (int) dpowers[irow_index][iRAISED_TO]);
printf("%2.1f= %.2f\n\n", dpowers[irow_index][iBASE],
dpowers[irow_index][iRESULT]) ,
}
return(0); }
```

Библиотечная функция `pow(x,y)` возводит `x` в степень `y`. Массив `dpowers` имеет тип `double`, поскольку данная функция работает с числами этого типа. В первом столбце массива записаны числа, возводимые в степень, во втором столбце — показатели степени, в третий столбец помещается результат.

При выполнении программы на экран будет выведена следующая информация:

```
1
1.1= 1.10
2 2.2= 4.84
3 3.3= 35.94
4 4.4= 374.81
5 5.5= 5032.84
6 6.6= 82653.95
```

Массивы как аргументы функций

Как и другие переменные, массивы могут передаваться функциям в качестве аргументов. Поскольку подробно изучить все аспекты подобного процесса можно будет только после знакомства с указателями, данная тема будет продолжена в главе "Указатели".

Передача массивов функциям в языке C

Предположим, имеется функция `isum()`, которая подсчитывает сумму элементов массива `inumeric_values`, содержащего целые числа. Для работы функции необходимы два аргумента: указатель `iarray_address_received`, принимающий копию адреса массива, и переменная `imax_size`, содержащая индекс последнего элемента, участвующего в операции суммирования. Тогда заголовок функции `isum()` будет записан следующим образом:

```
int isum(int iarray_address_received[], int imax_size)
```

Квадратные скобки в имени первого параметра указывают на то, что он является массивом. Обратите также внимание на то, что размер массива не указан. Вызов функции в программе будет выглядеть так:

```
isum(inumeric_values, iactual_index);
```

Это простейший способ передачи массивов функциям. Поскольку функция `isum()` в действительности получает адрес первого элемента массива, она может быть вызвана и так:

```
itotal = isum(Sinumeric_values[0], iactual_index);
```

В отличие от переменных, которые могут передаваться в функцию по значению (т.е. внутри функции создается локальная копия переменной), массивы всегда передаются по ссылке, и в вычислениях участвуют реальные элементы массива, а не их копии. Это предупреждает появление сообщений вида "stack overruns heap" (стек перегружен), которые постоянно беспокоят программистов на языке Pascal, если они забывают при описании формальных аргументов указывать перед именами массивов модификатор `var`. В этом случае массивы передаются по значению, что заставляет компилятор дублировать их содержимое. При работе с большими массивами этот процесс может потребовать много времени и свободной памяти.

В следующем примере создается массив из пяти элементов, а функция `iminimum()` определяет наименьший из них. С этой целью в функцию `iminimum` передаются имя массива и число сравниваемых элементов — в данном случае это весь массив (указана константа `IMAX`).

```
/*
 *      arrayarg.c
 *  Эта программа на языке C демонстрирует передачу массива в качестве
 *  аргумента функции.
 */
#include <stdio.h>
#define IMAX 10
int iminimum(int iarrayU, int imax);
main()
{
    int iarrayt[IMAX] = {3, 7, 2, 1, 5, 6, 8, 9, 0, 4};
    int i, ismallest;
    printf("Вот элементы массива: ");
    for(i = 0; i < IMAX; i++)
        printf("%d", iarray[i]);
    ismallest = iminimum(iarray, IMAX);
    printf("\nМинимальное значение: %d\n", ismallest);
    return(0);
}

int iminimum(int iarrayt[], int imax)
{
    int i, icurrent_minimum;
    icurrent_minimum = iarray[0];
    for(i = 1; i < imax; i++)
        if(iarray[i] < icurrent_minimum) icurrent_minimum = iarray[i];
    return(icurrent_minimum);
}
```

Передача массивов функциям в языке C++

Следующая программа на языке C++ также содержит пример передачи массива в функцию.

```
//
// arrayarg.cpp
// Эта программа на языке C++ демонстрирует передачу массива в
// качестве
// аргумента функции.
//
#include <iostream.h>
#define iSIZE 5
void vadd_l (int iarray[]);
main() {
int iarray[iSIZE] = {0,1, 2, 3, 4 } ;
int i;
cout << "Массив iarray перед вызовом функции vadd_l:\n\n";
for(i = 0; i < iSIZE; i++)
cout << " " << iarray[i];
vadd_l(iarray);
cout << "\n\nМассив iarray после вызова функции vadd_l:\n\n";
for(i =0; i < iSIZE; i++)
cout << " " << iarray[i];
return(0); }
void vadd_l(int iarray[]) {
int i;
for(i =0;i < iSIZE; i++)
iarray [i]++; }
```

В процессе выполнения программы на экран будет выведена следующая информация:

Массив iarray перед вызовом функции vadd_l:

0 1 2 3 4

Массив iarray после вызова функции vadd_l:

1 2 3 4 5

Результаты работы программы дают четкий ответ на вопрос о том, каким способом массив передается в функцию: по значению или по ссылке. Функция vadd_l() добавляет единицу к каждому элементу массива. Так как это действие отражается на массиве iarray в функции main(), можно сделать вывод, что аргумент передается по ссылке.

В следующей программе на языке C++ иллюстрируются многие свойства массивов, которые мы обсуждали выше, включая инициализацию многомерного массива и использование массивов в качестве аргументов функций.

```
//
// 3darray.cpp
// Эта программа на языке C++ демонстрирует, как создавать
// многомерный
// массив, передавать его в функцию и выбирать отдельный элемент
// такого массива.
//
#include <iostream.h>
void vdisplay_results(char carray[][3][4]);
char cglobal_cube[5][4][5]= { {
{'T','A','B','L','E'},
{'Z','E','R','O',' '},
{' ',' ',' ',' ',' '},
{'R','O','W',' ','3'}},
},
```

```

{
    {'T','A','B','L','E'},
    {'O','N','E',' ',' '},
    {'R','O','W',' ','2'},
},
{
    {'T','A','B','L','E'},
    {'T','W','O',' ',' '},
},
{
    {'T','A','B','L','E'},
    {'T','H','R','E','E'},
    {'R','O','W',' ','2'},
    {'R','O','W',' ','3'},
},
{
    {'T','A','B','L','E'},
    {'F','O','U','R',' '},
    {'r','o','w',' ','2'},
    {'a','b','c','d','e'},
} };
int imatrix[4][3]={ {1},{2},{3},{4} };
main () {
    int irow_index, icolumn_index;
    char clocal_cube[2][3][4];
    cout<< "Размер массива clocal_cube = "
    << sizeof(clocal_cube) << "\n";
    cout << "Размер таблицы clocal_cube[0] = "
    << sizeof (clocal_cube[0]) << "\n";
    cout<< "Размер строки clocal_cube[0][0] = "
    << sizeof(clocal_cube[0][0]) << "\n";
    cout << "Размер элемента clocal_cube[0][0][0] = "
    << sizeof(clocal_cube[0][0][0]) << "\n";
    vdisplay_results(clocal_cube);
    cout << "Элемент cglobal_cube[0][1][2] = " << cglobal_cube[0][1][2] <<
    "\n";
    cout << "Элемент cglobal__cube [1][0][2]= " << cglobal_cube[1][0][2]
    << "\n";
    cout << "\nВывод фрагмента массива cglobal_cube (таблица 0)\n";
    for (irow_index = .0; . irow_index < 4; irow_index++) {
        for(icolumn_index = 0; icolumn_index < 5; icolumn_index++) cout
        << cglobal_cube[0][irow_index][icolumn_index];
        cout << "\n"; }
    cout << "\nВывод фрагмента массива cglobal_cube (таблица 4)\n";
    for(irow_index = 0; irow_index < 4; irow_index++) {
        for(icolumn_index = 0; icolumn_index < 5; icolumn_index++) cout
        << cglobal_cube[4][irow_index][icolumn_index];
        cout << "\n"; }
    cout << "\nВывод всего массива imatrix\n";
    for (irow_index = 0; irow_index .< 4; irow_index++) {
        for(icolumn_index =0;icolumn_index < 3; icolumn_index++) cout <<
        imatrix[irow_index][icoluran_index];
        cout << "\n"; }
    return(0); }
void vdisplay_results(char carray[][3][4])

```

```

{
cout << "Размер массива carray          = " <<      sizeof(carray) <<
"\n";
cout << "Размер таблицы carray[0]      = " <<      sizeof(carray[0]) <<
"\n";
cout << "Размер массива cglobal_cube   = " <<      sizeof(cglobal_cube)
<< " \n";
cout << "Размер таблицы cglobal_cube[0]= " <<
sizeof(cglobal_cube[0]) << " \n";
}

```

Прежде всего обратите внимание на то, как объявляется и инициализируется массив `cglobal_cube`. Фигурные скобки используются для выделения групп символов, относящихся к одной размерности массива. Такой подход облегчает работу по заполнению массива данными, так как позволяет четко визуализировать его форму. А в принципе, в фигурных скобках нет необходимости: все элементы можно записать и в один ряд. Разбиение списка элементов на блоки особенно полезно в тех случаях, когда отдельные элементы следует оставить незаполненными. В нашем случае для большей наглядности трехмерный массив лучше всего сгруппировать в пять блоков, каждый из которых представляет собой таблицу из четырех строк и пяти столбцов.

В процессе выполнения программы на экран будут сначала выведены четыре строки с описанием размера всего массива `clocal_cube`, одной его таблицы, строки и отдельного элемента. Эти значения помогут вам лучше представить, из чего складывается размер массива. К примеру, размер трехмерного массива можно вычислить как произведение размеров трех его составляющих, умноженное на размер одного элемента. В нашем случае размер массива `clocal_cube` будет равен $2*3*4*sizeof(char) = 24$.

Обратите внимание на то, что часть массива `clocal_cube[0]` сама является двумерным массивом 3×4 , то есть ее размер составляет 12. Размер строки `clocal_cube[0][0]` равен 4, что соответствует числу элементов в ней, так как размер одного элемента равен 1 (`sizeof(clocal_cube[0][0][0])`).

Чтобы полностью разобраться с многомерными массивами, следует четко уяснить, что выражение `clocal_cube[0]` является константным указателем. В программе не объявлен массив `clocal_cube[0]` — на самом деле это последнее измерение многомерного массива `clocal_cube`. Выражение `clocal_cube[0]` не ссылается ни на один конкретный элемент массива, а лишь указывает на обособленный его фрагмент, поэтому тип данного выражения не `char`, а константный указатель на `char`, который не может выступать адресным операндом, то есть стоять слева от оператора присваивания.

Интересные события происходят, когда имя массива `clocal_cube` передается в качестве аргумента функции `vdisplay_results()`. В теле функции оператор `sizeof` не сможет правильно определить размер параметра `carray`, так как функция получает только копию адреса первого элемента массива, поэтому оператор вернет размер этого адреса (4 байта), а не самого массива и даже не его первого элемента. В то же время, размер фрагмента `carray[0]` будет вычислен правильно — $3 \times 4 = 12$, поскольку в заголовке функции указано, что первый параметр является массивом, две последние размерности которого равны 3 и 4.

Функция `vdisplay_results()` выводит также размер глобального массива `cglobal_cube`, причем вычисляет его правильно. На этом примере мы видим, что из тела функции можно получать непосредственный доступ к глобальному массиву, но если массив передается в качестве аргумента, то функции доступен только его адрес.

Две строки в теле программы, следующие за вызовом функции `vdisplay_results()`, демонстрируют возможность обращения к отдельным элементам многомерного массива `cglobal_cube`. Выражение `cglobal_cube[0][1][2]` возвращает значение элемента нулевой (первой по порядку) таблицы, второй строки и третьего столбца — 'R'. Выражение `cglobal_cube[1][0][2]` ссылается на элемент второй по порядку таблицы, первой ее строки и третьего столбца — 'B'.

Следующий фрагмент программы представлен тремя парами из двух вложенных циклов `for`, которые демонстрируют последовательный доступ к элементам трехмерного массива. Первая пара циклов `for` выводит на экран строки нулевой (первой по порядку) таблицы массива `cglobal_cube`, причем во внешнем цикле выбирается строка, а во внутреннем цикле последовательно выводятся все элементы этой строки. С помощью второй пары циклов `for`

выводится содержимое пятой таблицы массива `cglobal_cube`. В конце отображается содержимое массива `imatrix` в виде таблицы; именно так большинство из нас и представляет себе двумерный массив.

Информация, выводимая программой, будет выглядеть следующим образом:

```
Размер массива clocal_cube          = 24
Размер таблицы clocal_cube[0]       = 12
Размер строки clocal_cube[0][0]     = 4
Размер элемента clocal_cube[0][0][0] = 1
Размер массива carray               = 4
Размер таблицы carray[0]            = 12
Размер массива cglobal_cube         = 100
Размер таблицы cglobal_cube[0]      = 20
Элемент cglobal_cube[0][1][2]       = R
Элемент cglobal_cube[1][0][2]       = B
```

Вывод фрагмента массива `cglobal_cube` (таблица 0)

```
TABLE
ZERO
ROW 3
```

Вывод фрагмента массива `cglobal_cube`(таблица 4)

```
TABLE
FOUR
row 2
abed
```

Вывод всего массива `imatrix`

```
100
200
300
400
```

Вас удивила последняя часть? Тогда рассмотрим инициализацию массива `imatrix`. Каждая внутренняя пара фигурных скобок соответствует новой строке массива, но, поскольку внутри скобок указано меньше значений, чем размер строки, отсутствующие элементы массива остаются нулевыми. Вспомните, что в C/C++ все неинициализированные элементы статического массива по умолчанию автоматически становятся равными нулю.

Функции работы со строками и массивы символов

Многие функции работы со строками принимают в качестве аргумента имя массива символов: `gets()`, `puts()`, `fgets()`, `fputs()`, `sprintf()`, `strcpy()`, `strcat()`, `strncpy()` и `strlen()`. Настало время познакомиться с ними. Сейчас вам будет значительно проще понять принципы их работы, поскольку вы уже изучили основные концепции создания массивов.

Функции `gets()`, `puts()`, `fgets()`, `fputs()` и `sprintf()`

В следующей программе показано, как с помощью функций `gets()`, `puts()`, `fgets()`, `fputs()` и `sprintf()` можно управлять процессом ввода/вывода строк:

```
/*
 *      stringio.c
 *  Эта программа на языке C демонстрирует применение функций работы со
 *  строками.
 */
#include <stdio.h>
#define iSIZE 20
main()
{
```

```

char sztest_array[iSIZE];
fputs("Введите первую строку : ", stdout); gets(sztest_array);
fputs("Выввели      : ", stdout);
puts(sztest_array) ;
fputs("Введите вторую строку      : ", stdout);
fgets(sztest_array, iSIZE, stdin);
fputs("Выввели      : ", stdout);
fputs(sztest_array, stdout);
sprintf(sztest_array, "Это была %s проверка", "только");
fputs("Функция sprintf() создала : ", stdout);
fputs(sztest_array, stdout);
return(0); }

```

Вот что будет выведено на экран в результате работы программы:

```

Введите первую строку:          строка один
Вы ввели                      :          строка один
Введите вторую строку:          строка два
Вы ввели                      :          строка два
Функция sprintf() создала      :  Это была только проверка

```

Если введенные строки содержат меньше символов, чем зарезервировано ячеек в массиве `sztest_array`, программа работает правильно. Тем не менее, если ввести строки большей длины, чем позволяет размер массива `sztest_array`, на экране может появиться абракадабра.

Функция `gets()` принимает символы от стандартного устройства ввода (в большинстве случаев это клавиатура) и помещает их в массив, указанный в качестве аргумента. Когда вы нажимаете клавишу [Enter] для завершения ввода, генерируется символ новой строки (`\n`). Функция `gets()` преобразует его в нулевой символ (`\0`), который служит признаком конца строки. Учтите, что при использовании функции `gets()` нет возможности напрямую определить, превышает ли количество введенных символов размер массива.

Функция `puts()` выводит на экран то, что было получено с помощью функции `gets()`. Она выполняет обратную замену — нулевого символа на символ новой строки.

Функция `fgets()` аналогична функции `gets()`, но позволяет контролировать соответствие числа введенных символов установленным размерам массива. Символы читаются из указанного потока, которым может быть файл или, как в данном случае, стандартное устройство ввода (`stdin`). Введенных символов должно быть на единицу меньше, чем размер массива: в последнюю позицию автоматически добавляется нулевой символ. Если был введен символ новой строки, то он сохранится в массиве непосредственно перед символом `\0`. По аналогии с работающими в паре функциями `gets()` и `puts()`, совместно с функцией `fgets()` используют функцию `fputs()`.

Поскольку первая не удаляет символы новой строки, то и вторая не добавляет их. Функция `fputs()` направляет символы в указанный поток: файл или устройство стандартного вывода `stdout` (как правило, это консоль).

Имя функции `sprintf()` является сокращением от слов "string printf()", т.е. "строковая функция printf()". В этой функции используются те же спецификаторы форматирования, что и в `printf()`. Основное отличие состоит в том, что функция `sprintf()` помещает результат не на экран, а в указанный массив символов. Это может быть полезно, если результат работы данной функции необходимо вывести несколько раз, скажем, на экран и на принтер.

Функции `strcpy()`, `strcat()`, `strncmp()` и `strlen()`

Все функции, рассматриваемые в этом параграфе, объявлены в файле `STRING.H`. Для их работы требуется, чтобы передаваемые им наборы символов обязательно заканчивались признаком конца строки — символом `\0`. В следующей программе демонстрируется использование функции `strcpy()` :

```

/*
*   strcpy.c

```

```

*   Эта программа на языке C демонстрирует использование функции
strcpy().
*/
#include <stdio.h>
#include <string.h>
#define iSIZE 20
main() {
    char szsource_string[iSIZE]= "Исходная строка",
        szdestination_string[iSIZE];
    strcpy(szdestination_string, "Постоянная строка");
    printf("%s\n",szdestination_string);
    strcpy(szdestination_string, szsource_string);
    printf("%s\n",szdestination_string);
    return (0);
}

```

При первом вызове функций `strcpy()` происходит копирование константной строки "Постоянная строка" в массив `szdestination_string`, а вторая функция `strcpy()` копирует туда же массив `szsource_string`, содержащий строку "Исходная строка". В результате получаем следующие сообщения:

Постоянная строка Исходная строка

Ниже показана аналогичная программа на языке C++:

```

//
//   strcpy.cpp
//   Это версия предыдущей программы, написанная на языке C++.
#include <iostream.h>
#include <string.h>
#define iSIZE 20
main () {
    char szsource_string [iSIZE] = "Исходная строка", szdestination_string
        [iSIZE] ;
    strcpy (szdestination_string, "Постоянная строка");
    cout << "\n"<< szdestination_string;
    strcpy(szdestination_string, szsource_string) ;
    cout << "\n"<< szdestination_string;
    return(0); }

```

Функция `strcatf()` конкатенирует (объединяет) две самостоятельные строки в один массив. Обе строки должны заканчиваться нулевыми символами. Результирующий массив также завершается символом `\0`. Применение функции `strcat()` иллюстрируется следующей программой:

```

/*
*       strcat.c
*   Эта программа на языке C демонстрирует использование функции
strcat() .
*/
#include <stdio.h> #include <string.h>
#define lSTRING_SIZE 35
main()
{
    char szgreeting[]= "Доброе утро,",
        szname[] = " Каролина! ",
        szmessage[iSTRING_SIZE] ;
    strcpy (szmessage, szgreeting) ;
    strcat (szmessage, szname) ;
}

```

```

strcat (szmessage, "Какдела?") ;
printf ("%s\n", szmessage);
return(0); }

```

В этом примере два массива, `szgreeting` и `szname`, инициализируются в момент объявления, тогда как массив `szmessage` создается пустым. Первое, что делает программа, это копирует с помощью функции `strcpy()` содержимое массива `szgreeting` в массив `szmessage`. Затем функция `strcat()` добавляет в конец массива `szmessage` ("Доброе утро.") содержимое массива `szname` ("Каролина! "). И наконец, последняя функция `strcat()` добавляет к полученной строке "Доброе утро, Каролина! " текст "Как дела?". В результате программа выведет на экран

Доброе утро, Каролина! Как дела?

Следующая программа демонстрирует возможность сравнения двух строк с помощью функции `strncmp()`:

```

/*
 *          strncmp.c
 *   Эта программа на языке C сравнивает две строки с помощью функции
   strncmp() .
 */
#include <stdio.h>
#include <string.h>
main () {
char szstringA[] = "Вита", szstringB[] = "Вика";
int istringA_length, iresult = 0;
istringA_length = strlen(szstringA);
if(strlen(szstringB) >= strlen(szstringA))
iresult = strncmp(szstringA, szstringB, istringA_length);
printf ("Строка %s обнаружена", iresult == 0 ? "была" : "не была");
return(0); }

```

Используемая в программе функция `strlen()` возвращает число символов в строке, указанной в качестве аргумента, не считая последнего, нулевого символа. В программе эта функция вызывается дважды с разными целями, что дает возможность получить о ней более полное представление. В первом случае функция записывает в переменную `istringA_length` длину массива `szstringA`. Во втором случае значения, возвращаемые двумя функциями `strlen()`, сравниваются с помощью оператора `>=`. Если длина массива `szstringB` больше или равна длине массива `szstringA`, то происходит вызов функции `strncmp()`.

Функция `strncmp()` ищет первую строку во второй, начиная с первого символа. Длина первой строки задается в третьем аргументе. Если строки одинаковы, функция возвращает нулевое значение. Когда строки не идентичны, функция возвращает отрицательное значение, если строка `szstringA` меньше строки `szstringB`, и положительное значение, если строка `szstringA` больше строки `szstringB`.

Результат сравнения заносится в переменную `iresult`, на основе которой с помощью условного оператора (`?:`) формируется строка отчета. В нашем примере программа выведет на экран следующее сообщение:

Строка не была обнаружена

В завершение главы подчеркнем, что двумя наиболее частыми причинами сбоев в работе программ являются выход за пределы массива и отсутствие нулевого символа (`\0`) в конце символьного массива, используемого в качестве строки. Обе эти ошибки могут никак не проявляться в течение многих месяцев, до тех пор пока пользователь не введет, к примеру, слишком длинную строку текста.

Глава 9. Указатели

- Указатели как особый тип переменных
 - Объявление указателей
 - Использование указателей
 - Инициализация указателей
 - Ограничения на использование оператора &
 - Указатели на массивы
 -
 - Указатели на указатели
 -
 - Указатели на строки
 - Арифметические операции над указателями
 - Применение арифметики указателей при работе с массивами
 - Распространенная ошибка при использовании операторов ++ и --
 - Применение квалификатора const совместно с указателями
 -
 - Другие операции над указателями
 - Физическая реализация указателей
- Указатели на функции
- Динамическая память
 - Указатели типа void
- Подробнее об указателях и массивах
- - Строки (массивы типа char)
 - Массивы указателей
 - Дополнительные сведения об указателях на указатели
 - Массивы строковых указателей
- Ссылки в языке C++
 - Функции, возвращающие адреса

Если вы еще не имеете четкого представления о работе указателей, то самое время подробнее познакомиться с ними. Суть концепции указателей состоит в том, что вы работаете с адресом ячейки памяти, получая лишь косвенный доступ к ее содержимому, благодаря чему появляется возможность динамически создавать и уничтожать переменные. Хотя с помощью указателей можно создавать чрезвычайно эффективные алгоритмы, сложность программы, в которой используются указатели, значительно возрастает.

В языках C/C++ вопросы, связанные с указателями, массивами и строками, тесно связаны друг с другом. Поэтому данную главу можно рассматривать как непосредственное продолжение предыдущей главы. Для начинающего программиста глава может показаться чересчур сложной, но лишь в полной мере изучив тему указателей, можно начать создавать действительно профессиональные приложения.

Указатели как особый тип переменных

Начинающие программисты, как правило, работают только с обычными переменными. Для таких переменных память выделяется автоматически при запуске программы или функции, в которой они объявлены, и удаляется также автоматически при завершении программы или функции. Доступ к ним организуется достаточно просто — по имени:

```
imemorycell_contents+= 10;
```

Другой способ получения доступа к значению переменной заключается в использовании другой переменной, которая содержит ее адрес. Предположим, имеется переменная типа `int` именем `imemorycell_contents` и переменная `pimemory_cell_address`, являющаяся указателем на нее. Как вы уже знаете, в C/C++ есть оператор взятия адреса `&`, который возвращает адрес своего операнда. Поэтому вам будет нетрудно разобраться в синтаксисе присвоения одной переменной адреса другой переменной:

```
pimemorycell_address = &imemorycell_contents;
```

Переменные, которые хранят адреса других переменных, называются указателями. На рис. 9.1 схематично показана взаимосвязь между переменной и указателем на нее. Переменная `imemorycell_contents` представлена в памяти компьютера ячейкой с адресом 7751. После выполнения показанной выше строки программы адрес этой переменной будет присвоен указателю `pimemorycell_address`.

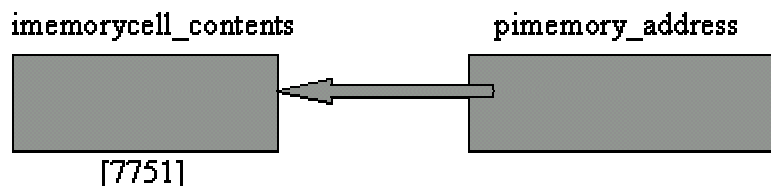


Рис. 9.1. Взаимосвязь между переменной и её указателями

Обращение к переменной, чей адрес хранится в другой переменной, осуществляется путем помещения перед указателем оператора `*`: `*pimemorycell_address`. Такая запись означает, что будет произведен косвенный доступ к ячейке памяти через имя указателя, содержащего адрес ячейки. Например, если выполнить две показанные ниже строки, то переменная `imemorycell_contents` примет значение 20:

```
pimemorycell_address = &imemorycell_contents;  
*pimemorycell_address = 20;
```

С учетом того, что указатель `pimemorycell_address` хранит адрес переменной `imemorycell_contents`, обе следующие строки приведут к одному и тому же результату: присвоению переменной `imemorycell_contents` значения 20.

```
imemorycell_contents = 20;  
*pimemorycell_address = 20;
```

Объявление указателей

В языках C/C++ все переменные должны быть предварительно объявлены. Объявление указателя `pimemorycell_address` выглядит следующим образом:

```
int *pimemorycell_address;
```

Символ `*` говорит о том, что создается указатель. Этот указатель будет адресовать переменную типа `int`. Следует подчеркнуть, что в C/C++ указатели могут хранить адреса только переменных конкретного типа. Если делается попытка присвоить указателю одного типа адрес переменной другого типа, возникнет ошибка либо во время компиляции, либо во время выполнения программы.

Рассмотрим пример:

```
int *pi
float real_value = 98.26;
pi = &real_value;
```

В данном случае переменная `pi` объявлена как указатель типа `int`. Но в третьей строке делается попытка присвоить этому указателю адрес переменной `real_value`, имеющей тип `float`. В результате компилятор выдаст предупреждение вида "несовместимые операнды в операции присваивания", а программа, использующая указатель `pi`, будет работать неправильно.

Использование указателей

В следующем фрагменте программы посредством указателей производится обмен значений между переменными `iresult_a` и `iresult_b`:

```
int iresult_a = 15, iresult_b = 37, itemporary;
int *piresult;
piresult = &iresult_a;
itemporary = *piresult;
*piresult = iresult_b;
iresult_b = itemporary;
```

Первая строка содержит традиционные объявления переменных. При этом в памяти компьютера резервируются три ячейки для хранения целочисленных значений, каждой ячейке присваивается имя, и две из них инициализируются начальными значениями. Предположим, что переменная `iresult_a` хранит свои значения в ячейке с адресом 5328, переменная `iresult_b` связана с ячейкой 7916, а `itemporary` — с ячейкой 2385 (рис. 9.2).

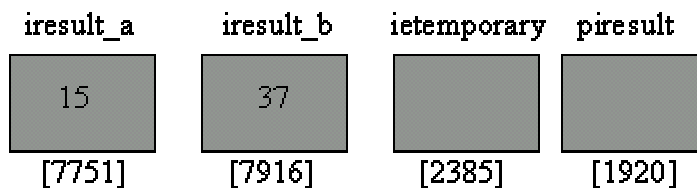


Рис. 9.2. Резервирование и инициализация ячеек памяти

Во второй строке программы создается указатель `piresult`. При этом также происходит резервирование именованной ячейки памяти (скажем, с адресом 1920). Поскольку инициализация не производится, то в данный момент указатель содержит пустое значение. Если попытаться применить к нему оператор `*`, то компилятор не сообщит об ошибке, но и не возвратит никакого адреса.

В третьей строке происходит присваивание указателю `piresult` адреса переменной `iresult_a` (рис. 9.3).

В следующей строке в переменную `itemporary` записывается содержимое переменной `iresult_a`, извлекаемое с помощью выражения `*piresult`:

```
itemporary = *piresult;
```

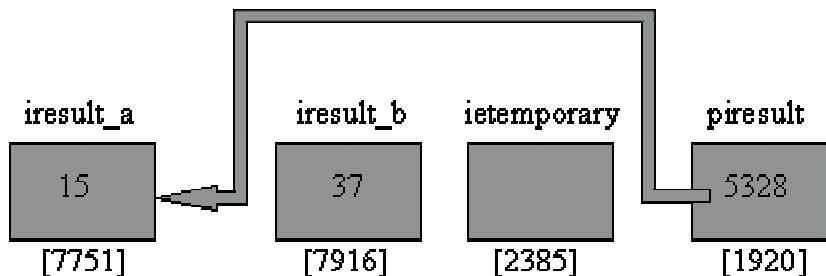


Рис. 9.3. Присваивание указателю `piresult` адреса переменной `iresult_a`

Таким образом, переменной `itemporary` присваивается значение 15 (рис. 9.4). Если перед именем указателя `piresult` не поставить символ `*`, то в результате в переменную `itemporary` будет ошибочно записано содержимое самого указателя, т.е. 5328. Это очень коварная ошибка, поскольку многие компиляторы не выдают в подобных ситуациях предупреждений или сообщений об ошибке. Компилятор VisualC++ отобразит предупреждение вида "разные уровни косвенной адресации в операции присваивания".

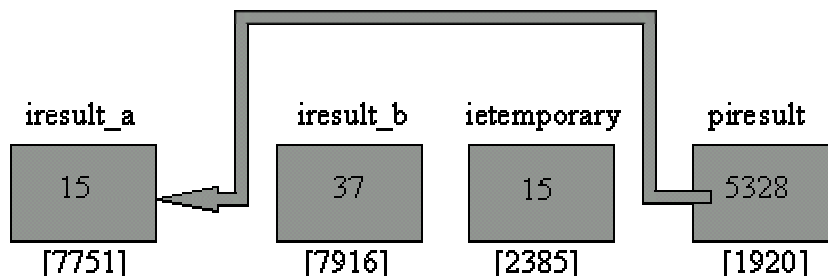


Рис. 9.4. Запись в переменную `itemporary` значения переменной `iresult_a`

В пятой строке содержимое переменной `iresult_b` копируется в ячейку памяти, адресуемую указателем `piresult` (рис. 9.5):

```
*piresult = iresult_b;
```

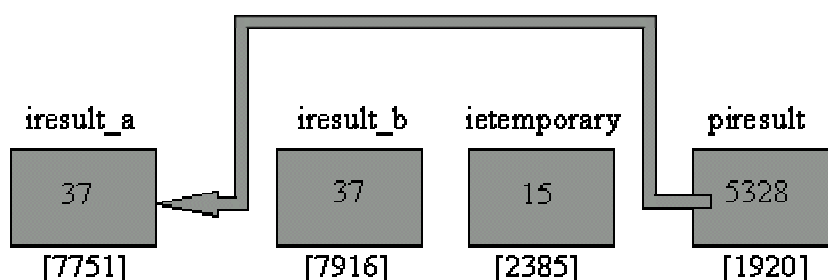


Рис. 9.5. В переменную `iresult_a` записывается значение переменной `iresult_b`

В последней строке число, хранящееся в переменной `itemporary`, просто копируется в переменную `iresult_b` (рис. 9.6).

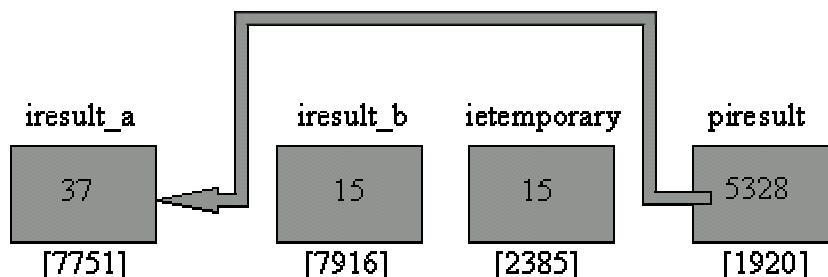


Рис. 9.6. Конечный результат

В следующем фрагменте программы демонстрируется возможность манипулирования адресами, хранящимися в указателях. В отличие от предыдущего примера, где переменные обменивались значениями, здесь осуществляется обмен адресами переменных.

```
char cswitch1 = 'S', cswitch2 = "I" ;
char *pcswitch1, *pcswitch2, *pctemporary;
pcswitch1 = &cswitch1;
pcswitch2 = &cswitch2;
pctemporary = pcswitch1;
```

```
pcswitch1 = pswitch2;
pcswitch2 = ptemporary;
printf("%c%c", *pcswitch1, *pcswitch2);
```

На рис. 9.7 показана схема отношений между зарезервированными ячейками памяти после выполнения первых четырех строк программы. В пятой строке содержимое указателя pswitch1 копируется в переменную ptemporary, в результате чего оба указателя адресуют одну переменную: cswitch1 (рис. 9.8).

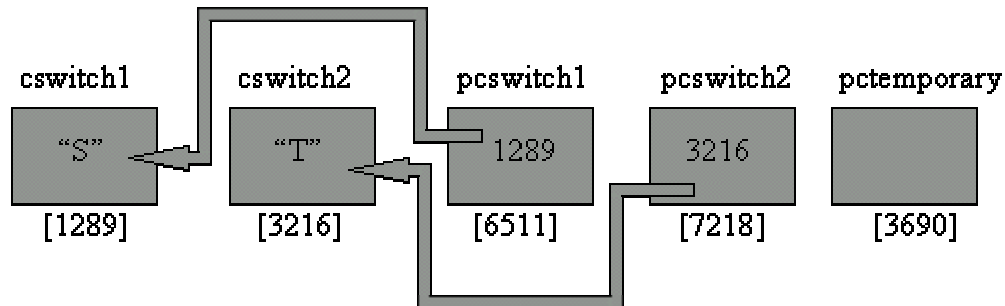


Рис. 9.7. Исходные отношения между переменными

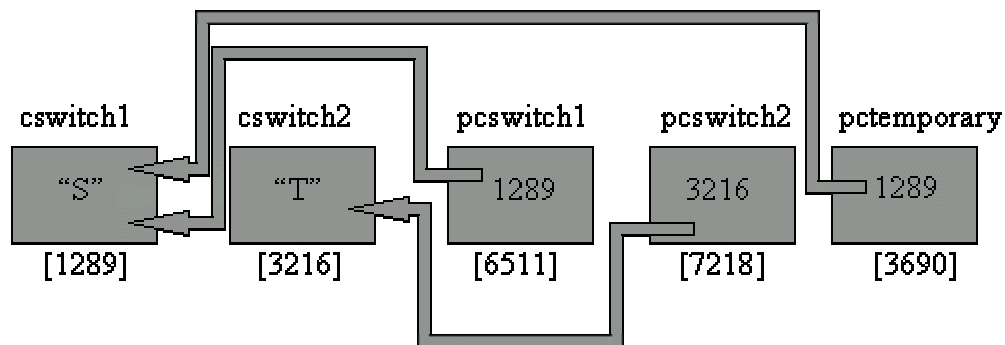


Рис. 9.8. Указателю ptemporary присвоен адрес, хранящийся в указателе pswitch1

В следующей строке содержимое указателя pswitch2 копируется в указатель pswitch1, после чего оба будут содержать адрес переменной cswitch2 (рис. 9.9):

```
pcswitch1 = pswitch2;
```

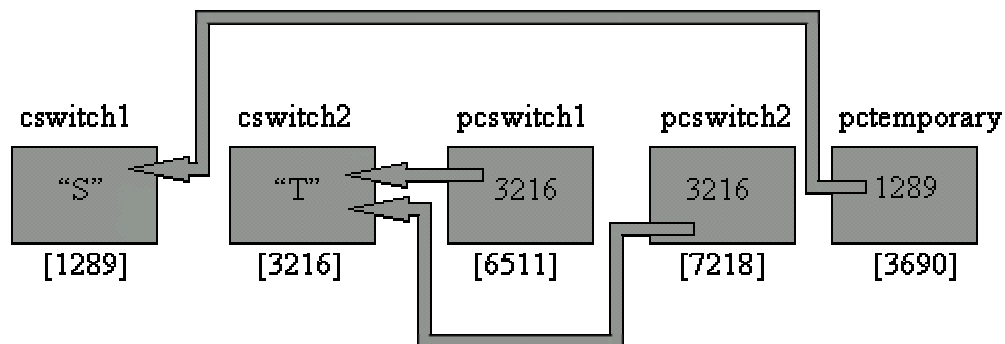


Рис. 9.9. Присвоение указателю pswitch1 адреса, хранящегося в указателе pswitch2

Обратите внимание, что если бы содержимое указателя `pcswitch1` не было продублировано во временной переменной `pctemporary`, то в результате выполнения предыдущего выражения ссылка на адрес переменной `cswitch1` была бы утеряна.

В предпоследней строке происходит копирование адреса из указателя `pctemporary` в указатель `pcswitch2` (рис. 9.10). В результате работы функции `printf()` получаем:

TS

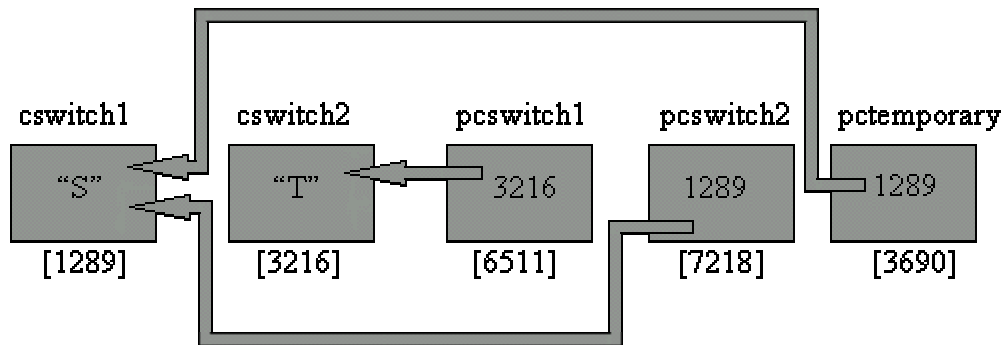


Рис. 9.10. Передача адреса от указателя `pctemporary` к указателю `pcswitch2`

Заметьте: в ходе выполнения программы исходные значения переменных `cswitch1` и `cswitch2` не изменялись. Описанный метод может пригодиться вам в дальнейшем, так как, в зависимости от размеров объектов, часто бывает проще копировать их адреса, чем перемещать содержимое.

Инициализация указателей

Указатели можно инициализировать при их объявлении, как и любые другие переменные. Например, в следующем фрагменте создаются две именованные ячейки памяти: `iresult` и `piresult`.

```
int iresult;
int *piresult = &iresult;
```

Идентификатор `iresult` представляет собой обычную целочисленную переменную, а `piresult`— указатель на переменную типа `int`. Одновременно с объявлением указателя `piresult` ему присваивается адрес переменной `iresult`. Будьте внимательны: здесь инициализируется содержимое самого указателя, т.е. хранящийся в нем адрес, но не содержимое ячейки памяти, на которую он указывает. Переменная `iresult` остается неинициализированной.

В приведенной ниже программе объявляется и инициализируется указатель на строку-палиндром, одинаково читаемую как слева направо, так и справа налево:

```
/*
 *   psz.c
 *   Эта программа на языке C содержит пример инициализации указателя.
 */
#include <stdio.h>
#include <string.h>
void main 0    {
char *pszpalindrome = "Доммод";
int i;
for (i = strlen(pszpalindrome) - 1; i >= 0; i--)
printf("%c",pszpalindrome[i]);
printf("%s",pszpalindrome); }
```

В указателе `pszpalindrome` сохраняется адрес только первого символа строки. Но это не значит, что оставшаяся часть строки пропадает: компилятор заносит все строковые константы, обнаруженные им в программе, в специальную скрытую таблицу, встраиваемую в программу. Таким образом, в указатель записывается адрес ячейки таблицы, связанной с данной строкой.

Функция `strlen()`, объявленная в файле `STRING.H`, в качестве аргумента принимает указатель на строку, заканчивающуюся нулевым символом, и возвращает число символов в строке, не считая последнего. В нашем примере строка состоит из семи символов, но счетчик цикла `for` инициализируется значением 6, поскольку строка в нем интерпретируется как массив, содержащий элементы от нулевого до шестого. На первый взгляд, может показаться странной взаимосвязь между строковыми указателями и массивами символов, но если мы вспомним, что имя массива по сути своей является указателем на первый элемент массива, то станет понятным, почему переход от имени указателя к имени массива в программе не вызвал возражений компилятора.

Ограничения на использование оператора `&`

Оператор взятия адреса (`&`) можно применять далеко не с каждым выражением. Ниже иллюстрируются ситуации, когда оператор `&` используется неправильно:

```
/*с константами*/
pivariable = &48;
/*в выражениях с арифметическими операторами*/
int ireresult = 5;
pivariable = &(ireresult + 15);
/* с переменными класса памяти register*/
register int register1; pivariable = &register1;
```

В первом случае делается недопустимая попытка получить адрес константного значения. Поскольку с константой 48 не связана ни одна ячейка памяти, операция не может быть выполнена.

Во втором случае программа пытается найти адрес выражения `ireresult+15`. Поскольку результатом этого выражения является число, находящееся в программном стеке, его адрес не может быть получен.

В последнем примере объявляется регистровая переменная, смысл которой состоит в том, что предполагается частое ее использование, поэтому она должна располагаться не в памяти, а непосредственно в регистрах процессора. Компилятор может проигнорировать подобный запрос и разместить переменную в памяти, но в любом случае операция взятия адреса считается неприменимой по отношению к регистровым переменным.

Указатели на массивы

Как уже говорилось, указатели и массивы логически связаны друг с другом. Вспомните из предыдущей главы, что имя массива является константой, содержащей адрес первого элемента массива. В связи с этим значение имени массива не может быть изменено оператором присваивания или каким-нибудь другим оператором. Например, ниже создается массив типа `float` именем `ftemperatures`:

```
#define IMAXREADINGS 20
float ftemperatures[IMAXREADINGS]; float *pftemp;
```

В следующей строке объявленному выше указателю `pftemp` присваивается адрес первого элемента массива:

```
pftemp = ftemperatures;
```

Это же выражение можно записать следующим образом:

```
pftemp = &ftemperatures[0];
```

Тем не менее, даже если указатель описан для хранения адреса переменных типа `float`, все равно следующие выражения недопустимы:

```
ftemperatures = pftemp;
&ftemperatures[0] = pftemp;
```

Эти выражения невыполнимы, поскольку в них делается попытка изменить константу `ftemperatures` и эквивалентное ей выражение `&ftemperatures[0]`, что так же бессмысленно, как и строка

```
10 = pftemp;
```

Указатели на указатели

В C/C++ можно создавать указатели на другие указатели, которые, в свою очередь, содержат адреса реальных переменных. Смысл этого процесса проиллюстрирован на рис. 9.11, где ррi является указателем на указатель.

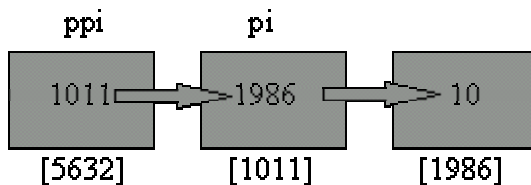


Рис. 9.11. Пример указателя на указатель

Чтобы объявить в программе указатель, который будет хранить адрес другого указателя, нужно просто удвоить число звездочек в объявлении:

```
int **ppi;
```

Каждый символ * читается как указатель на. Таким образом, количество указателей в цепочке, задающее уровень косвенной адресации, соответствует числу звездочек перед именем идентификатора. Уровень косвенной адресации определяет, сколько раз следует выполнить операцию раскрытия указателя, чтобы получить значение конечной переменной.

В следующем фрагменте создается ряд указателей с различными уровнями косвенной адресации (рис. 9.12).

```
int ivalue = 10;
int *pi;
int **ppi;
int ***pppi;
pi = &ivalue;
ppi = &pi;
pppi = &ppi;
```

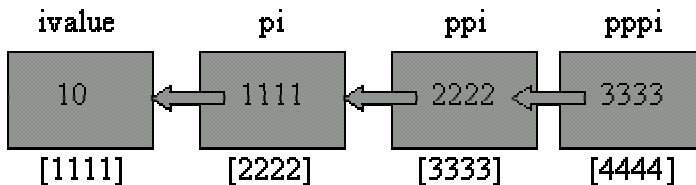


Рис. 9.12. Несколько уровней косвенной адресации

В первых четырех строках объявляются четыре переменные: ivalue типа int, указатель pi на переменную типа int (первый уровень косвенной адресации), указатель ppi (второй уровень косвенной адресации) и указатель pppi (третий уровень косвенной адресации). Этот пример показывает, что при желании можно создать указатель любого уровня.

В пятой строке указателю первого уровня pi присваивается адрес переменной ivalue. Теперь значение переменной ivalue(10) может быть получено с помощью выражения *pi. В шестой строке в указатель второго уровня ppi записывается адрес (но не содержимое) указателя pi, который, в свою очередь, указывает на переменную ivalue. Обратиться к значению переменной можно будет посредством выражения **ppi. В последней строке аналогичным образом заполняется указатель третьего уровня.

В C/C++ указатели можно инициализировать сразу при объявлении, как и любые другие переменные. Например, указатель pppi можно было бы инициализировать следующей строкой:

```
int ***pppi = &ppi;
```

Указатели на строки

Строковые константы в действительности представляют собой символьные массивы с конечным нулевым символом (рис. 9.13). Указатель на строку можно объявить и инициализировать следующим образом:


```
char *psz= "Файл не найден";
```



Рис. 9.13. Схема представления в памяти строки символов

Данное выражение создает указатель psz типа char и присваивает ему адрес первого символа строки — 'Ф'

(рис. 9.14). Сама строка заносится компилятором в специальную служебную таблицу.



Рис. 9.14. Инициализация указателя на строку

Приведенную выше строку можно записать по-другому:

```
char *psz;  
psz= "Файл не найден";
```

Следующий пример иллюстрирует одно часто встречающееся заблуждение, касающееся использования строковых указателей и символьных массивов:

```
char *psz= "Файл не найден";  
char pszarray[] = "Файл не найден";
```

Основное различие между этими двумя выражениями состоит в том, что значение указателя psz может быть изменено (поскольку указатель является разновидностью переменной), а значение имени массива pszarray не может быть изменено, так как это константа. По этой причине показанные ниже выражения ошибочны:

```
/* Ошибочный код */  
char pszarray[15];  
pszarray= "Файл не найден";
```

Хотя, на первый взгляд, все кажется логичным, в действительности в данном выражении оператор присваивания пытается скопировать адрес первой ячейки строки "Файл не найден" в объект pszarray. Но поскольку имя массива pszarray является константой, а не переменной-указателем, то будет выдано сообщение об ошибке.

Следующий фрагмент на языке C++ ошибочен по той причине, что указатель был объявлен, но не инициализирован:

```
/* Ошибочный код */  
char *psz; cin >> psz;
```

Чтобы решить проблему, нужно просто зарезервировать память для указателя:

```
char string [10];  
char *psz = string;  
cin.get (psz,10) ;
```

Поскольку имя string представляет собой адрес первой ячейки массива, то второе выражение не только резервирует память для указателя, но и инициализирует его, присваивая ему адрес первого элемента массива string. В данном случае метод cin.get () (его назначение состоит в чтении из входного потока заданного количества символов) будет выполнен успешно, поскольку в функцию передается действительный адрес массива.

Арифметические операции над указателями

Языки C/C++ дают возможность выполнять различные операции над указателями. В предыдущих примерах мы наблюдали, как адрес, хранящийся в указателе, или содержимое переменной, адресуемой указателем, присваивались другим указателям аналогичного типа. Кроме того, в C/C++ над указателями можно производить два математических действия: сложение и вычитание. Проиллюстрируем сказанное примером.

```
//
// ptarith.cpp
// Эта программа на языке C++ содержит примеры
// арифметических выражений с участием указателей.
//
#include <iostream.h>
void main () {
int *pi;
float *pf;
int an_integer;
float a_real;
pi = &an_integer; pf = &a_real; pi++;
pf++; }
```

Предположим, что в данной системе для целых чисел отводится 2 байта, а для чисел с плавающей запятой — 4 байта. Допустим также, что переменная `an_integer` хранится в ячейке с адресом 2000, а переменная `a_real` — в ячейке с адресом 4000. После выполнения двух последних строк программы значение указателя `pi` становится 2002, а указателя `pf` — 4004. Но, подождите, разве мы не знаем, что оператор `++` увеличивает значение переменной на единицу? Это справедливо для обычных переменных, но не для указателей.

В главе 4 мы познакомились с понятием перегрузки операторов. Операторы инкремента (`++`) и декремента (`--`) как раз являются примерами перегруженных операторов. Они модифицируют значение операнда-указателя на число, соответствующее размеру занимаемой им ячейки памяти. Для указателей типа `int` это 2, для указателей типа `float` — 4 байта. Этот же принцип справедлив для указателей любых других типов. Если взять указатель, связанный со структурой размером 20 байтов, то единичное приращение такого указателя также будет соответствовать 20-ти байтам.

Адрес, хранимый в указателе, можно также изменять путем прибавления или вычитания любых целых чисел, а не только единицы, как в случае операторов `++` и `--`. Например, чтобы сместить адрес на 4 ячейки, можно использовать такое выражение:

```
pf = pf + 4;
```

Рассмотрим следующую программу и попытаемся представить, каким будет результат ее выполнения.

```
//
// sizept.cpp
// Эта программа на языке C++ демонстрирует приращение
// указателя на значение, большее единицы.
//
#include <iostream.h>
void main() {
float fvalues[] = {15.38, 12.34, 91.88, 11.11, 22.22};
float *pf;
size_t fwidth;
pf = &fvalues [0];
fwidth = sizeof (float);
pf = pf + fwidth;
cout << *pf; }
```

Предположим, адрес переменной `fvalue` равен FFCA. Переменная `fwidth` принимает свое значение от оператора `sizeof(float)` (в нашем случае — 4). Что произойдет после выполнения предпоследней строки программы? Адрес в указателе `pf` поменяется на FFDA, а не на FFCE,

как можно было предположить. Почему? Не забывайте, что при выполнении арифметических действий с указателями нужно учитывать размер объекта, адресуемого данным указателем. В нашем случае величина приращения будет: 4x4 (размерность типа данных float) = 16. В результате указатель рfсместится на 4 ячейки массива, т.е. на значение 22,22.

Применение арифметики указателей при работе с массивами

Если вы знакомы с программированием на языке ассемблера, то наверняка сталкивались с задачей вычисления физических адресов элементов, хранящихся в массивах. При использовании индексов массивов в C/C++ выполняются те же самые операции. Разница состоит только в том, что в последнем случае функцию непосредственного обращения к адресам памяти берет на себя компилятор.

В следующих двух программах создаются массивы из 10-ти символов. В обоих случаях программа получает от пользователя элементы массива и выводит их на экран в обратной последовательности. В первой программе применяется уже знакомый нам метод обращения к элементам по индексам. Вторая программа идентична первой, но значения элементов массива считываются по их адресам посредством указателей. Вот первая программа:

```
/*
 *      arrayind.c
 *      Эта программа на языке C демонстрирует
 *      обращение к элементам массива по индексам.
 */
#include <stdio.h>
#define ISIZE 10
void main () (
char string10[ISIZE];
int i;
for(i = 0; i < ISIZE; i++)
string10[i]= getchar();
for(i= ISIZE - 1; i >= 0; i--)
putchar(string10[i]); )
}
```

Теперь приведем текст второй программы:

```
/*
 *      arrayptr.c
 *      Эта программа на языке C демонстрирует обращение к
 *      элементам массива посредством указателей.
 */
#include <stdio.h>
#define ISIZE 10
void main ()
{
char string10[ISIZE];
char *pc;
int icount;
pc = string10;
for(icount = 0; icount < ISIZE; icount++) {
*pc = getchar();
pc++; }
pc = string10 + (ISIZE - 1);
for(icount = 0; icount < ISIZE; icount++)
{ putchar(*pc); pc--; }
}
```

Первая программа достаточно проста и понятна, поэтому сосредоточим наше внимание на второй программе, в которой используются арифметические выражения с указателями.

Переменная `pc` имеет тип `char*`, означающий, что перед нами указатель на символ. Поскольку массив `string10` содержит символы, указатель `pc` может хранить адрес любого из них. В следующей строке программы в указатель `pc` записывается адрес первой ячейки массива:

```
pc = string10;
```

В цикле `for` программа считывает ряд символов, число которых определяется константой `isize`, и сохраняет их в массиве `string10`. Для занесения очередного символа по адресу, содержащемуся в указателе `pc`, применяется оператор `*`:

```
*pc = getchar() ;
```

В следующей строке значение указателя увеличивается на четыре с помощью операции инкрементирования (`++`). Таким способом осуществляется переход к следующему элементу массива.

Чтобы начать вывод символов массива в обратном порядке, следует в первую очередь присвоить указателю `pc` адрес последнего элемента массива:

```
pc = string10 + (ISIZE - 1);
```

Чтобы переместить указатель на 10-й элемент, мы прибавляем к базовому адресу массива значение 9, поскольку первый элемент имеет нулевое смещение.

Во втором цикле `for` указатель `pc` последовательно смещается от последнего к первому элементу массива с помощью операции декрементирования (`--`). Функция `putchar()` выводит на экран содержимое ячеек, адресуемых указателем.

Распространенная ошибка при использовании операторов `++` и `--`

Напомним, что следующие два выражения будут иметь разный результат:

```
*pc++ = getchar ();
*++pc = getchar ();
```

Первое выражение (с постинкрементом) присваивает символ, возвращаемый функцией `getchar()`, текущей ячейке, адресуемой указателем `pc`, после чего выполняется приращение указателя `pc`. Во втором выражении (с преинкрементом) сначала происходит приращение указателя `pc`, после чего в ячейку по обновленному адресу будет записан результат функции `getchar()`. Сказанное справедливо также для префиксной и постфиксной форм оператора — (декремент).

Применение квалификатора `const` совместно с указателями

В рассматриваемой нами теме указателей еще достаточно "подводных камней". Посмотрите на следующие два объявления указателей и попробуйте разобраться в различиях между ними:

```
const MYTYPE *pmytype_1;
MYTYPE * const pmytype_2 = &mytype;
```

В первом случае переменная `pmytype_1` объявляется как указатель на константу типа `mytype`. Во втором случае `pmytype_2` — это константный указатель на переменную `mytype`. Непонятно?

Хорошо, давайте разбираться дальше. Идентификатор `pmytype_1` является указателем. Указателю, как и любой переменной, можно присваивать значения соответствующего типа данных. В данном случае это должен быть адрес, по которому локализован объект типа `mytype`. А ключевое слово `const` в объявлении означает следующее: хотя указателю `pmytype_1` может быть присвоен адрес любой ячейки памяти, содержащей данные типа `mytype`, впоследствии содержимое этой ячейки не может быть изменено путем обращения к указателю. Попробуем прояснить ситуацию на примере:

```
pmytype_1 = &mytype1;          // допустимо
pmytype_1 = &mytype2;          // допустимо
*pmytype_1 = (MYTYPE)float_value; // недопустимая попытка изменить
// значение защищенной ячейки памяти
```

Теперь рассмотрим переменную `pmytype_2`, которая объявлена как константный указатель. Она может содержать адрес ячейки памяти с данными типа `mytype`, но этот адрес недоступен для изменения. По этой причине инициализация константного указателя обязательно должна происходить одновременно с его объявлением (в приведенном выше примере указатель

инициализируется адресом переменной mytype). С другой стороны, содержимое ячейки, на которую ссылается такой указатель, может быть свободно изменено:

```
pmytype_2 = &mytype1; // недопустимая попытка изменить
// защищенный адрес
*pmytype_2 = (MYTYPE)float_value; // допустимое изменение содержимого
ячейки
// памяти
```

А теперь подумайте, какой из двух вариантов применения ключевого слова `const` соответствует объявлению массива? Ответ: второй. Напомним, что имя массива представляет собой неизменяемый адрес первого его элемента. С точки зрения компилятора, ничего не изменилось бы, если бы массив объявлялся следующим образом:

```
тип данных* const имя массива = адрес_первого_элемента;
```

Другие операции над указателями

Мы уже рассмотрели примеры, иллюстрирующие применение операторов `++` и `--` к указателям, а также добавление целого значения к указателю. Ниже перечислены другие операции, которые могут быть выполнены над указателями:

- вычитание целого значения из указателя;
- вычитание одного указателя из другого (оба должны ссылаться на один и тот же массив);
- сравнение указателей с помощью операторов `<=`, `=` и `>=`.

В результате вычитания целого числа указатель будет ссылаться на элемент, смещенный на указанную величину влево по отношению к текущей ячейке.

Результатом вычитания указателя из указателя будет целое число, соответствующее числу элементов между ячейками, на которые они ссылались. Предполагается, что оба указателя одного типа и связаны с одним и тем же массивом. Если в выражении используются два разнотипных указателя или если они ссылаются на разные массивы, то результат операции невозможно предсказать.

Независимо от того, в какой операции участвует указатель, компилятор не может проследить за тем, чтобы в результате операции адрес не выходил за пределы массива.

Указатели одного типа можно сравнивать друг с другом. Возвращаемые значения `true(! 0)` или `false(0)` можно использовать в условных выражениях и присваивать переменным типа `int` так же, как результаты любых других логических операций. Один указатель будет меньше другого, если он указывает на ячейку памяти с меньшим индексом. При этом предполагается, что оба указателя связаны с одним и тем же массивом.

Наконец, указатели можно сравнивать с нулем. В данном случае можно выяснить только равенство/неравенство нулю, поскольку указатели не могут содержать отрицательные значения. Нулевое значение указателя означает, что он не связан ни с каким объектом. Ноль является единственным числовым значением, которое можно непосредственно присвоить указателю независимо от его типа. Кроме того, указатель можно сравнить с константным выражением, результатом которого является ноль, а также с другим указателем типа `void` (в последнем случае указатель следует предварительно привести к типу `void`).

Все остальные операции с указателями запрещены. Например, нельзя складывать два указателя, а также умножать и делить их.

Физическая реализация указателей

В примерах данной главы адреса возвращались как целые числа. Исходя из этого, вы могли сделать заключение, что указатели являются переменными типа `int`. В действительности это не так. Указатель содержит адрес переменной определенного типа, но при этом свойства указателя не сводятся к свойствам рядовых переменных типа `int` или `float`. В некоторых системах значение указателя может быть скопировано в переменную типа `int` и наоборот, хотя

стандарты языков C/C++ не гарантируют подобную возможность. Чтобы обеспечить правильную работу программы в разных системах, такую практику следует исключить.

Указатели на функции

Во всех рассмотренных до сих пор примерах указатели использовались для обращения к определенным значениям, содержащимся в памяти компьютера. Теперь мы познакомимся с указателями, связанными не с данными, а с программными кодами — функциями. Указатели на функции обеспечивают возможность косвенного вызова функций, точно так же как указатели на данные предоставляют косвенный доступ к ячейкам памяти.

С помощью указателей на функции можно решать многие важные задачи. Рассмотрим, к примеру, функцию `qsort()`, осуществляющую сортировку массива. В числе ее параметров должен быть задан указатель на функцию, выполняющую сравнение двух элементов массива. Необходимость в функции-аргументе возникает в связи с тем, что алгоритм сравнения может быть достаточно сложным и многофакторным, работающим по-разному в зависимости от типа элементов. Код одной функции не может быть передан в другую функцию как значение аргумента, но в C/C++ допускается косвенное обращение к функции с помощью указателя.

Концепция использования указателей на функции часто иллюстрируется на примере стандартной функции `qsort()`, прототип которой находится в файле `STDLIB.H`. В приводимом ниже примере программы на языке C мы создаем собственную функцию сравнения `icompare_func()` и передаем указатель на нее в функцию `qsort()`.

```
/*
 *   qsort.c
 *   Эта программа на языке C демонстрирует передачу указателя на
 *   функцию
 *   в качестве аргумента функции qsort().
 */
#include <stdio.h>
#include <stdlib.h>
#define IMAXVALUES 10
int icompare_func(const void *iresult_a, const void *iresult_b);
int (*ifunc_ptr) (const void *, const void *);
void main ()
{
    int i ;
    int iarray[IMAXVALUES] = {0, 5, 3, 2, 8, 7, 9, 1, 4, 6};
    ifunc_ptr = icompare_func;
    qsort(iarray, IMAXVALUES, sizeof(int), ifunc_ptr);
    for(i=0; i < IMAXVALUES; i++)
        printf("%d", iarray[i]); }
int icompare_func(const void *iresult_a, const void *iresult_b) {
    return ((* (int*) iresult_a) - (* (int*) iresult_b)); }
```

Функция `icompare_func()` (которую будем называть адресуемой) соответствует требованиям, накладываемым на нее функцией `qsort()` (которую будем называть вызывающей): она принимает два аргумента типа `void*` и возвращает целочисленное значение. Напомним, что ключевое слово `const` в списке аргументов накладывает запрет на изменение данных, на которые указывают аргументы. Благодаря этому вызывающая функция в худшем случае неправильно отсортирует данные, но она не сможет их изменить! Теперь, когда синтаксис использования функции `icompare_func()` стал понятен, уделите внимание ее телу.

Если адресуемая функция возвращает отрицательное число, значит, первый ее аргумент меньше второго. Ноль означает равенство аргументов, а положительное значение — что первый аргумент больше второго. Все эти вычисления реализуются единственной строкой, составляющей тело функции `icompare_func()` :

```
return ((* (int *) iresult_a) - (* (int *) iresult_b));
```

Поскольку оба указателя переданы в функцию как `void*`, они приводятся к соответствующему им типу `int` и раскрываются (*). Результат вычитания содержимого второго указателя из содержимого первого возвращается в функцию `qsort()` в качестве критерия сортировки.

Важная часть программы — объявление указателя на адресуемую функцию, расположенное сразу вслед за ее прототипом:

```
int (*ifunct_ptr)(const void *, const void *);
```

Это выражение определяет указатель `ifunct_ptr` на некую функцию, принимающую два константных аргумента типа `void*` и возвращающую значение типа `int`. Обратите особое внимание на скобки вокруг имени указателя. Выражение вида

```
int *ifunct_ptr(const void *, const void *);
```

воспринимается не как объявление указателя, а как прототип функции, поскольку оператор вызова функции, `()`, имеет более высокий приоритет, чем оператор раскрытия указателя `*`. В результате последний будет отнесен к спецификатору типа, а не к идентификатору `ifunct_ptr`.

Функция `qsort()` принимает следующие параметры: адрес массива, который нужно отсортировать (массив `array`), число элементов массива (константа `imaxvalues`), размер в байтах элемента таблицы (`sizeof(int)`) и указатель на функцию сравнения (`ifunct_ptr()`)

Рассмотрим еще несколько интересных примеров:

```
int * (* (*ifunct_ptr) (int) ) [5];
float (* (*ffunct_ptr) (int,int)) (float);
typedef double (* (* (*dfunct_ptr) () ) [5]) () ;
dfunct_ptr A_dfunct_ptr;
(* (*function_array_ptrs () ) [5]) () ;
```

Первая строка описывает указатель `ifunct_ptr` на функцию, принимающую один целочисленный аргумент и возвращающую указатель на массив из пяти указателей типа `int`.

Вторая строка описывает указатель `ffunct_ptr` на функцию, принимающую два целочисленных аргумента и возвращающую указатель на другую функцию с одним аргументом типа `float` и таким же типом результата.

Создавая с помощью ключевого слова `typedef` новый тип данных (более подробно эта тема раскрывается в главе "Дополнительные типы данных"), можно избежать повторения сложных деклараций. Третья строка читается следующим образом: тип `dfunct_ptr` определен как указатель на функцию без аргументов, возвращающую указатель на массив из пяти указателей, которые, в свою очередь, ссылаются на функции без аргументов, возвращающие значения типа `double`. В четвертой строке создается экземпляр такого указателя.

Последняя строка содержит объявление функции, а не переменной. Создаваемая функция `function_array_ptrs()` не имеет параметров и возвращает указатель на массив из пяти указателей, которые ссылаются на функции без аргументов, возвращающие значения типа `int` (последнее подразумевается по умолчанию, если не указано иное).

Динамическая память

Во время компиляции программ на языках C/C++ память компьютера разделяется на четыре области: программного кода, глобальных данных, стек и динамическую область ("куча"). Последняя отводится для хранения временных данных и управляется функциями распределения памяти, такими как `malloc()` и `free()`.

Функция `malloc()` резервирует непрерывный блок ячеек для хранения указанного объекта и возвращает указатель на первую ячейку этого блока. Функция `free()` освобождает ранее зарезервированный блок и возвращает эти ячейки в динамическую область для последующего резервирования.

В качестве аргумента в функцию `malloc()` передается целочисленное значение, указывающее количество байтов, которое необходимо зарезервировать. Если резервирование прошло успешно, функция `malloc()` возвращает переменную типа `void*`, которую можно привести к любому необходимому типу указателя. Концепция использования указателей типа `void` описана в стандарте ANSI C. Этот спецификатор предназначен для создания обобщенных указателей неопределенного типа, которые впоследствии можно преобразовывать к

требуемому типу. Обобщенный указатель сам по себе не может быть использован для обращения к каким-нибудь данным, так как он не связан ни с одним из базовых типов данных. Но любой указатель может быть приведен к типу void и обратно без потери информации.

В следующем фрагменте программы резервируется память для 300 чисел типа float:

```
float *pf;
int inura_floats = 300;
pf = (float *) malloc(inum_floats * sizeof(float));
```

В данном примере функция malloc() резервирует блок памяти, достаточный для хранения 300 значений типа float. Выделенный блок надежно отделен от других блоков и не перекрывается ими. Предугадать, где именно он будет размещен, невозможно. Все блоки особым образом помечаются, чтобы система могла легко их обнаруживать и определять их размер.

Если блок ячеек больше не нужен, его можно освободить следующей строкой:

```
free ((void*) pf);
```

В C/C++ резервирование памяти осуществляется двумя способами. Так, при объявлении переменной указатель стека программы продвигается ниже по стеку, "захватывая" новую ячейку. Когда переменная выходит за пределы своей области видимости, область памяти, отведенная для переменной, автоматически освобождается путем перемещения указателя назад, выше по стеку. Размер необходимой стековой памяти всегда должен быть известен еще до начала компиляции.

Но иногда в программе необходимо создавать переменные, размер которых неизвестен заранее. В таких случаях ответственность за резервирование памяти возлагается на программиста, и для этих целей используется динамическая область. В программах на языках C/C++ резервирование и освобождение блоков памяти в динамической области может происходить в любой момент времени. Также важно запомнить, что объекты, хранимые в динамической области, в отличие от обычных переменных, не подчиняются правилам видимости. Они никогда не могут оказаться вне области видимости, поэтому вы же ответственны за то, чтобы освободить зарезервированную память, как только отпадет необходимость в соответствующем объекте. Если вы не позаботитесь об освобождении неиспользуемой памяти, а будете создавать все новые и новые динамические объекты, то рано или поздно программа столкнется с недостатком свободной памяти.

В компиляторах языка C для управления динамической памятью используются библиотечные функции malloc() и free(), которые мы только что рассмотрели. Создатели языка C++ посчитали оперирование свободной памятью столь важной задачей для работы программы, что добавили дополнительные операторы new и delete, аналогичные вышеуказанным функциям. Аргументом для оператора new служит выражение, возвращающее число байтов, которое необходимо зарезервировать. Этот оператор возвращает указатель на начало выделенного блока памяти. Аргументом оператора delete выступает адрес первой ячейки блока, который необходимо освободить. В следующих двух примерах демонстрируются различия в синтаксисе программ на языках C и C++, работающих с динамической областью памяти. Вот пример программы на языке C:

```
/*
 *      malloc.c
 *      Эта программа на языке C демонстрирует применение
 *      функций malloc() и free()
 */
#include <stdio.h>
#include <stdlib.h>
#define ISIZE 512
void main()
{
    int *pimemory_buffer;
    pimemory_buffer = malloc(iSIZE * sizeof (int)) ;
    if(pimemory_buffer == NULL)
        printf("Недостаточно памяти\n"); else
        printf("Память зарезервирована\n");
    free(pimemory_buffer) ;
}
```



```
}
```

Первое, на что следует обратить внимание еще в начале программы, — это подключение к программе файла `STDLIB.H`, содержащего прототипы функций `malloc()` и `free()`. Функция `malloc()` передает указателю `pimemory_buffer` адрес зарезервированного блока размером `ISIZE*sizeof(int)`. Надежный алгоритм всегда должен включать проверку успешности резервирования памяти, поэтому указатель проверяется на равенство значению `null`. Функция `malloc()` всегда возвращает `null`, если выделить память не удалось. Программа завершается вызовом функции `free()`, которая освобождает только что зарезервированную область памяти. Программа на языке C++ мало отличается от предыдущего примера:

```
//
//  newdel.cpp
//  Эта программа на языке C++ демонстрирует применение
//  операторов new и delete.
//
#include <iostream.h>
#define NULL 0
#define ISIZE 512
void main()
(
int *pimemory_buffer;
pimemory_buffer = new int[ISIZE];
if(pimemory_buffer == NULL)
cout<< "Недостаточно памяти\n";
else
cout<< "Память зарезервирована\n";
delete(pimemory_buffer); }
```

Необходимости в подключении файла `STDLIB.H` здесь нет, так как операторы `new` и `delete` являются встроенными компонентами языка. В отличие от функции `malloc()`, где для определения точного размера блока применяется оператор `sizeof`, оператор `new` автоматически выполняет подобного рода вычисления, основываясь на заданном типе объекта. В обеих программах резервируется блок из 512 смежных ячеек типа `int`.

Указатели типа void

Если бы все указатели имели стандартный размер, не было бы необходимости на этапе компиляции определять тип адресуемой переменной. Кроме того, с помощью "стандартизированного" указателя можно было бы передавать в функции адреса переменных любого типа, а затем, в теле функции, привести указатель к требуемому типу в соответствии с полученной дополнительной информацией. Используя такой подход, можно создавать универсальные функции, подходящие для обработки данных разного типа. Вот почему в язык C++ был добавлен указатель типа `void`. Использование ключевого слова `void` при объявлении указателя имеет другой смысл, чем в списке аргументов или в качестве возвращаемого значения (в этих случаях `void` означает "ничего"). Указатель на `void` - это универсальный указатель на данные любого типа. В следующей программе на языке C++ демонстрируется применение таких указателей:

```
//
//  voidpt.cpp
//  Эта программа на языке C++ демонстрирует
//  применение указателей типа void.
//
#include <iostream.h>
#define ISTRING_MAX 50
void voutput(void *pobject, char cflag)
void main() {
int *pi;
char *psz;
float *pf;
char cresponse, cnewline;
```

```

cout << "Задайте тип данных, которые\n";
cout << "вы хотите ввести.\n\n";
cout << "(s)tring, (i)nt, (f)loat: ";
cin >> cresponse;
switch(cresponse) {
case 's':
psz = new char [ISTRING_MAX];
cout << "\nВведите строку: ";
cin.get (psz, cresponse);
voutput (pi, cresponse);
break;
case 'i':
pi = new int;
cout << "\nВведите целое число :";
cin >> *pi;
voutput (pi, cresponse);
break;
case 'f':
pf = new float;
cout << "\nВведите число с плавающей запятой: ";
cin >> *pf;
voutput (pi, cresponse);
break;
default:
cout << "\n\nТакой тип данных не поддерживается!";
}
}
void voutput(void *pobject, char cflag)
{
switch(cflag) {
case 's':
cout << "\nВы ввели строку " << (char *) pobject;
delete pobject;
break;
case 'i':
cout<< "\nВы ввели целое число "
<< *((int*) pobject);
delete pobject;
break;
case 'f':
cout<< "\nВы ввели число с плавающей запятой "
<< *((float *) pobject);
delete pobject;
break;
}
}
}

```

Прежде всего обратите внимание на прототип функции `voutput()`, в частности на то, что первый параметр функции — `pobject` — представляет собой обобщенный указатель `void*`. В функции `main()` создаются указатели трех конкретных типов: `int*`, `char*` и `float*`. Они будут использоваться в зависимости от того, какого типа данные введет пользователь.

Выполнение программы начинается с выдачи пользователю приглашения указать тип данных, которые он собирается вводить. Вас может удивить, почему для считывания ответа используются две различные команды. Первый объект `cin` считывает символ, введенный

пользователем с клавиатуры, но не воспринимает символ новой строки `\n`. Исправляет ситуацию функция `cin.get(cnewline)`.

Обработка ответа пользователя осуществляется с помощью инструкции `switch`, где происходит выбор сообщения, выводимого на экран. В программе используется одна из трех строк инициализации указателя:

```
psz = new char[ISTRING_MAX]; pi = new int; pf = new float;
```

Следующее выражение предназначено для считывания введенной строки текста, длина которой не может превышать значение, заданное константой `istring_max`, — 50 символов:

```
cin.get(psz, ISTRING_MAX) ;
```

Поскольку функция `cin.get()` ожидает в качестве первого аргумента указатель на строку, при вызове функции `voutput()` нет необходимости выполнять операцию раскрытия указателя:

```
voutput(psz, cresponse) ;
```

Две следующие ветви `case` выглядят почти одинаково, за исключением отображаемых сообщений и типа адресуемых переменных. Обратите внимание, что в трех различных обращениях к функции `voutput()` в качестве аргументов используются указатели разных типов:

```
voutput(psz, cresponse) ;
```

```
voutput(pi, cresponse) ;
```

```
voutput(pf, cresponse) ;
```

Функция `voutput()` воспринимает все эти аргументы независимо от их типа только благодаря тому, что в объявлении функции указан параметр типа `void*`. Напомним, что для извлечения содержимого указателей, заданных как `void*`, их сначала нужно привести к конкретному типу данных, например `char*`.

Начинающие программисты склонны к тому, чтобы быстро забывать о созданных ими динамических переменных. В нашей программе каждая ветвь `case` завершается явным удалением созданной перед этим динамической переменной. Где именно в программе происходит создание и удаление динамических переменных, зависит только от привычек программиста и особенностей конкретного приложения.

Подробнее об указателях и массивах

В следующих параграфах на примерах программ мы более подробно рассмотрим взаимосвязь между массивами и указателями.

Строки (массивы типа `char`)

Многие операции со строками в языках C/C++ выполняются с применением указателей и арифметических операций над ними, поскольку доступ к отдельным символам строки осуществляется, как правило, последовательно. Следующая программа на языке C++ является модификацией рассмотренной ранее в этой главе программы для вывода строки-палиндрома:

```
//
// chrarray.c
// Эта программа на C++ выводит на экран массив символов в обратной
// последовательности с применением арифметики указателей.
//
#include <iostream.h>
#include <string.h>
void main () {
char pszpalindrome[] = "Дом мод";
char *pc;
pc = pszpalindrome + (strlen(pszpalindrome) - 1);
do {
cout << *pc;
pc--;
} while (pc >= pszpalindrome); }
```

После объявления и инициализации массива `pszpalindrome` создается указатель `pc` типа `char*`. Вспомните, что имя массива само по себе является адресом. Сначала указателю `pc` присваивается адрес последнего символа массива. Это осуществляется с помощью функции `strlen()`, которая возвращает длину массива символов.

Функция `strlen()` не учитывает символ окончания Строки `\0`.

Почему от полученной длины массива отнимается единица? Не забывайте, что первый элемент массива имеет нулевое смещение, т.е. его индекс равен 0, а не 1. Поэтому вывод следует выполнять начиная с индекса, на единицу меньшего, чем количество символов в массиве.

После того как указатель переместится на последний значащий символ массива, запускается цикл `do/while`. В этом цикле на экран выводится символ, адресуемый текущим значением указателя, а в конце каждой итерации адрес указателя уменьшается на единицу, после чего полученный адрес сравнивается с адресом первого элемента массива. Цикл продолжается до тех пор, пока все элементы массива не будут выведены на экран.

Массивы указателей

В языках C/C++ имеется еще одна интересная возможность — создавать массивы указателей. Такой массив представляет собой совокупность элементов, каждый из которых является указателем на другие объекты. Эти объекты, в свою очередь, также могут быть указателями.

Концепция построения массивов указателей на указатели широко используется при работе с параметрами `argc` и `argv` функции `main()`, с которыми вы познакомились в главе "Функции". В следующей программе определяется максимальное или минимальное из значений, введенных в командной строке (ожидается ввод неотрицательных чисел). Аргументами могут быть либо только числа, либо, в дополнение к ним, специальная опция, задающая режим работы: поиск минимального (`-s`, `-S`) или максимального (`-l`, `-L`) значений.

```
//
//      argcargv.cpp
//      Эта программа на языке C++ демонстрирует работу с массивами
//      указателей
//      на указатели на примере параметров argc и argv функции main().
//
#include <iostream.h>
#include <process.h>      // exit()
#include <stdlib.h>       // atoi()
#define IFIND_LARGEST 1
#define IFIND_SMALLEST 0
#define IMAX32767
int main(int argc, char *argv[]) {
    char *psz;
    int ihow_many;
    int iwhich_extreme = IFIND_SMALLEST; int irange_boundary = IMAX;
    if(argc--< 2) {
        cput<< "\nВведите одну из опций -S, -s, -L, -l"
        << " и, как минимум, одно целое число."; exit(1); }
    if ((*++argv) [0] == '-') {
        psz = argv[0] + 1; switch (*psz) {
            case 's': case 'S':
                iwhich_extreme = IFIND_SMALLEST;
                irange_boundary = IMAX;
                break;
            case 'l':
            case 'L':
                iwhich_extreme = IFIND_LARGEST;
                irange_boundary = 0;
        }
    }
}
```

```

break;
default:
cout << "Нераспознанный аргумент " << *psz << ", \n";
exit(1) ;
}
if(*++psz != '\0'){
cout<< "Нераспознанный аргумент " << *psz << "\n"; exit(1.); ' )
if (--argc == 0) {
cout<< "Введите как минимум одно число\n";
exit(1); }
argv++; }
ihow_many = argc;
while (argc--) {
int present_value;
present_value = atoi(*argv++);
if(iwhich_extreme == IFIND_LARGEST && present_value >
irange_boundary) irange_boundary = present_value;
if(iwhich_extreme == IFIND_SMALLEST && present_value <
irange_boundary) irange_boundary = present_value; }
cout << ( (iwhich_extreme) ? "Максимальное " : "Минимальное ");
cout<< "значение в ряду из " << ihow_many
<< " введенных чисел = "
<< irange_boundary << "\n";
return(0); }

```

Прежде чем приступить к анализу текста программы, давайте разберемся, какие значения могут быть введены в командной строке в качестве аргументов. Ниже показан список возможных комбинаций аргументов:

```

argcargv
argcargv98
argcargv98 21
argcargv -s 98
argcargv -S 98 21
argcargv -l 14
argcargv -L 14 67

```

Напомним, что параметр `a` где функции `main()` представляет собой целое число, равное количеству аргументов командной строки, включая имя программы. Параметр `argv` представляет собой указатель на массив строковых указателей.

Примечание

Параметр `argv` не является константой. Напротив, это переменная, чье значение может быть изменено. Об этом важно помнить при изучении работы программы. Первый элемент массива — `argv[0]` — является указателем на строку символов, содержащую имя программы.

В первую очередь в программе проверяется, не является ли значение параметра `argc` меньшим 2. Если это так, значит, пользователь ввел в командной строке только имя программы без аргументов. По-видимому, пользователь не знает назначения программы или того, какие аргументы должны быть заданы в командной строке. Поэтому программа отобразит на экране информацию о своих аргументах и завершится выполнением функции `exit()`. Обратите внимание, что после выполнения проверки счетчик аргументов уменьшается на единицу (`argc--`). К моменту начала цикла `while` счетчик должен стать равным количеству числовых аргументов, заданных в командной строке, здесь же мы уменьшили его на единицу, чтобы учесть аргумент, содержащий имя программы.

Затем происходит обращение к первому символу второго аргумента, и если полученный символ является дефисом, то программа определяет, какая именно опция, `-s` или `-l`, задана в командной строке. Заметьте, что к указателю массива `argv` сначала добавляется единица (`++argv`), в результате чего аргумент с именем программы (первый элемент массива)

пропускается и активным становится следующий аргумент. Выражение с инкрементом взято в круглые скобки, так как идущий следом оператор индексации (квадратные скобки) имеет более высокий приоритет, и без скобок выражение возвращало бы второй символ имени программы.

Поскольку значение указателя `argv` было изменено и он теперь ссылается на второй аргумент командной строки, выражение `argv[0]` возвращает адрес первого символа этого аргумента. Путем прибавления единицы мы перемещаем указатель ко второму символу, записывая его адрес в переменную `psz`:

```
psz = argv[0] + 1;
```

Анализируя значение этого символа, программа определяет, какую задачу ставит перед ней пользователь: поиск минимального или максимального чисел из списка. При этом в переменной `iwhich_extreme` устанавливается флаг режима

```
(IFIND_SMALLEST — ПОИСК МИНИМАЛЬНОГО ЧИСЛА, IFIND_LARGEST —  
МАКСИМАЛЬНОГО),
```

а в переменную `irange_boundary` записывается начальное значение, с которым будут сравниваться числовые аргументы. Из соображений совместимости мы предполагаем, что неотрицательные числа типа `int` попадут в диапазон 0—32767, что соответствует 16-разрядным системам. В случае, если пользователь задаст неверную опцию, скажем, `-d` или `-su`, в ветви `default` будет выведено соответствующее сообщение. Проверка

```
if(--argc == 0)
```

позволяет убедиться, что после опции указан хотя бы один числовой аргумент. В противном случае программа завершит свою работу. Последняя строка данного блока

```
argv++;
```

необходима для того, чтобы переместить указатель `argv` на аргумент, идущий после опции.

Функция `atoi()` в цикле `while` преобразовывает каждый из аргументов в целое число (не забывайте, что на данный момент все они представлены в виде массивов символов) и сохраняет результат в переменной `present_value`. Опять-таки после выполнения этой функции указатель `argv` будет перемещен на следующий аргумент командной строки. В оставшихся двух инструкциях `if` в переменную `irange_boundary` записывается текущее минимальное или максимальное значение из списка в зависимости от содержимого флага `iwhichextreme`.

Дополнительные сведения об указателях на указатели

В следующей программе также демонстрируется применение указателей на указатели. Мы не рассмотрели ее раньше, когда обсуждали соответствующую тему, по той причине, что в этой программе используется динамическая память.

```
/*  
 *      dblptr.c  
 *      Эта программа на языке C демонстрирует работу с указателями на  
 *      указатели.  
 */  
#include <stdio.h>  
#include <stdlib.h>  
#define IMAXELEMENTS 3  
void voutputdnt (**ppiresult_a, int **ppiresult_b, int **ppiresult_c)  
;  
void vassignfint (*pivirtual_array[], int *pinewblock);  
void main ()  
{  
    int **ppiresult_a, **ppiresult_b, **ppiresult_c;  
    int *pivirtual_array[IMAXELEMENTS];  
    int *pinewblock, *pioldblock;  
    ppiresult_a = &pivirtual_array[0]; ppiresult_b = &pivirtual_array[1];  
    , ppiresult_c = &pivirtual_array[2];  
    pinewblock = (int *) malloc(IMAXELEMENTS * sizeof(int)); pioldblock =  
    pinewblock;  
    vassign(pivirtual_array, pinewblock);
```

```

**ppiresult_a = 7;
**ppiresult_b = 10;
**ppiresult_c =15;
voutput(ppiresult_a, ppiresult_b, ppiresult_c);
pinewblock = (int *) malloc(IMAXELEMENTS * sizeof(int));
*pinewblock = **ppiresult_a;
*(pinewblock +1)= **ppiresult_b;
*(pinewblock +2)= **ppiresult_c;
free (pioldblock) ;
vassign(pivirtual_array, pinewblock);
voutput(ppiresult_a, ppiresult_b, ppiresult_c);
void yassign(int *pivirtual_array [], int *pinewblock)
pivirtual_array [0]      = pinewblock;
pivirtual_array [1]      = pinewblock + 1;
pivirtual_array [2]      = pinewblock + 2;
void voutput (int **ppiresult_a, int **ppiresult_b, int
**ppiresult_c){
printf ("%d\n",**ppiresult_a) ;
printf ("%d\n",**ppiresult_b) ;
printf ("%d\n",**ppiresult_c) ;
}

```

В этой программе указатели ppiresult_a, ppiresult_b и ppiresult_c ссылаются на указатели с постоянными адресами (&pivirtual_array[0], &pivirtual_array[1], &pivirtual_array[2]), но содержимое последних может меняться динамически.

Рассмотрим блок объявлений в функции main(). Переменные ppiresult_a, ppiresult_b и ppiresult_c объявлены как указатели на указатели, содержащие адреса значений типа int. Переменная pivirtual_array является массивом указателей типа int с числом элементов, заданным константой imaxelements. Последние две переменные, pinewblock и pioldblock, являются одиночными указателями такого же типа, что и массив pivirtual_array. Взаимоотношения между всеми переменными после того, как указателям ppiresult_a, ppiresult_b и ppiresult_c были присвоены адреса соответствующих элементов массива pivirtual_array, показаны на рис. 9.15.

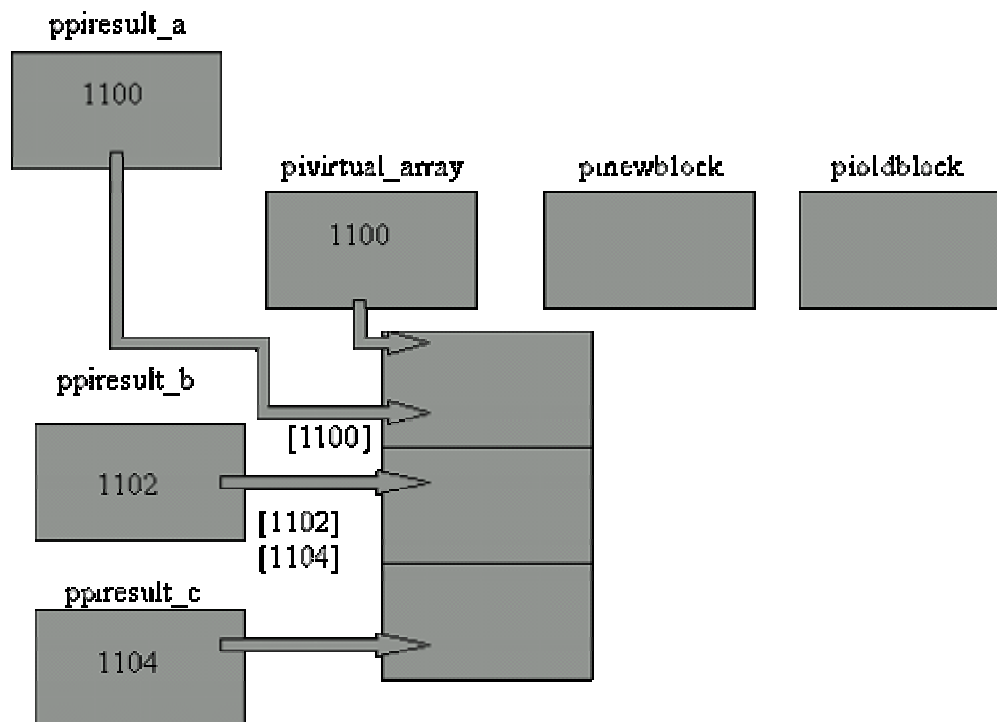


Рис. 9.15. Взаимоотношения между переменными программы после инициализации указателей `ppirestult_a`, `ppirestult_b` и `ppirestult_c`

Идея программы заключается в том, чтобы в упрощенной форме проиллюстрировать процессы, на самом деле происходящие в многозадачной среде. Программа считает, что содержит действительный адрес переменной, хотя на самом деле она имеет в своем распоряжении лишь фиксированный адрес ячейки массива, в которой хранится текущий адрес переменной. Когда операционная система в соответствии со своими потребностями перемещает данные, хранимые в оперативной памяти, она автоматически обновляет ссылки на эти данные в массиве указателей. Используемые в программе указатели при этом сохраняют прежние адреса.

На рис. 9.16 показано, что произойдет после того, как в динамическую память будет помещен массив `pinewblock`, переменной `pioldblock` будет присвоен его адрес и выполнена функция `vassign()`. Обратите внимание на соответствие адресов, хранящихся в "физическом" массиве `pinewblock` "виртуальном" массиве `pivirtual_array`.

В функцию `vassign()` в качестве аргументов передаются массив `pivirtual_array` и адрес только что созданного динамического блока, представленного переменной `pinewblock`. В теле функции каждому элементу массива присваивается соответствующий ему адрес динамической ячейки памяти. Поскольку обращение к массиву из функции осуществляется по адресу, все изменения вступают в силу и в теле функции `main()`.

На рис. 9.17 показано состояние переменных после присвоения ячейкам динамической памяти трех целочисленных значений.

Далее ситуация становится интереснее. С помощью функции `malloc()` в динамической памяти создается еще один блок ячеек, адрес которого записывается в указатель `pinewblock`. А указатель `pioldblock` по-прежнему связан с созданным ранее блоком. Все это сделано для того, чтобы примерно смоделировать процесс перемещения блока ячеек в новую позицию.

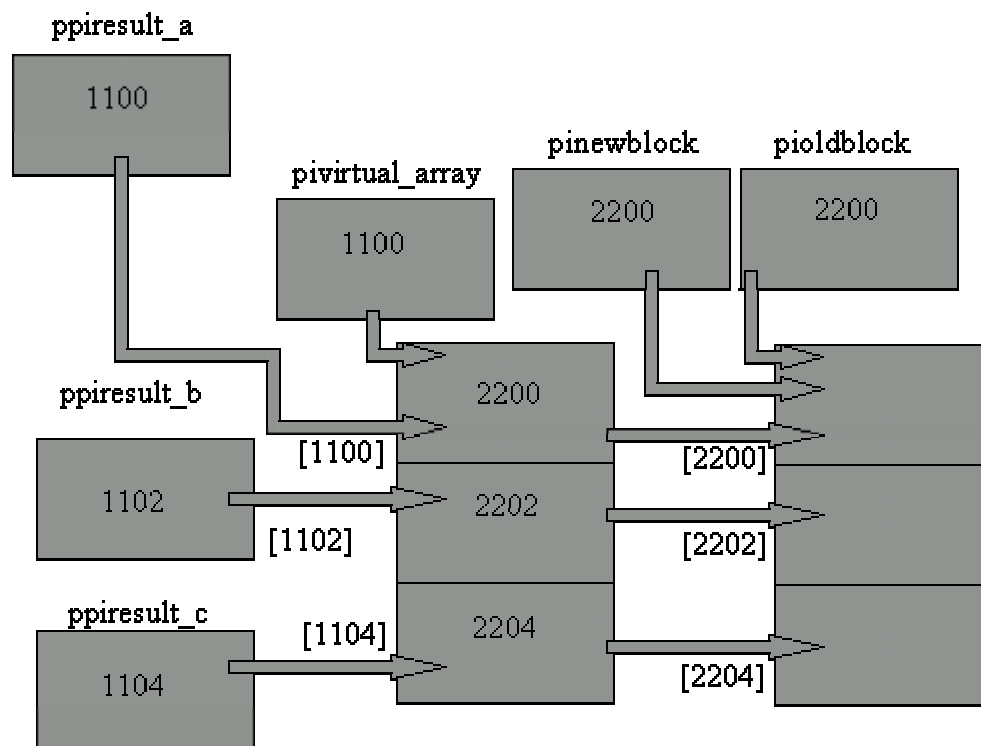


Рис.9.16. Динамическое создание блока ячеек и инициализация массива `pivirtual_array`

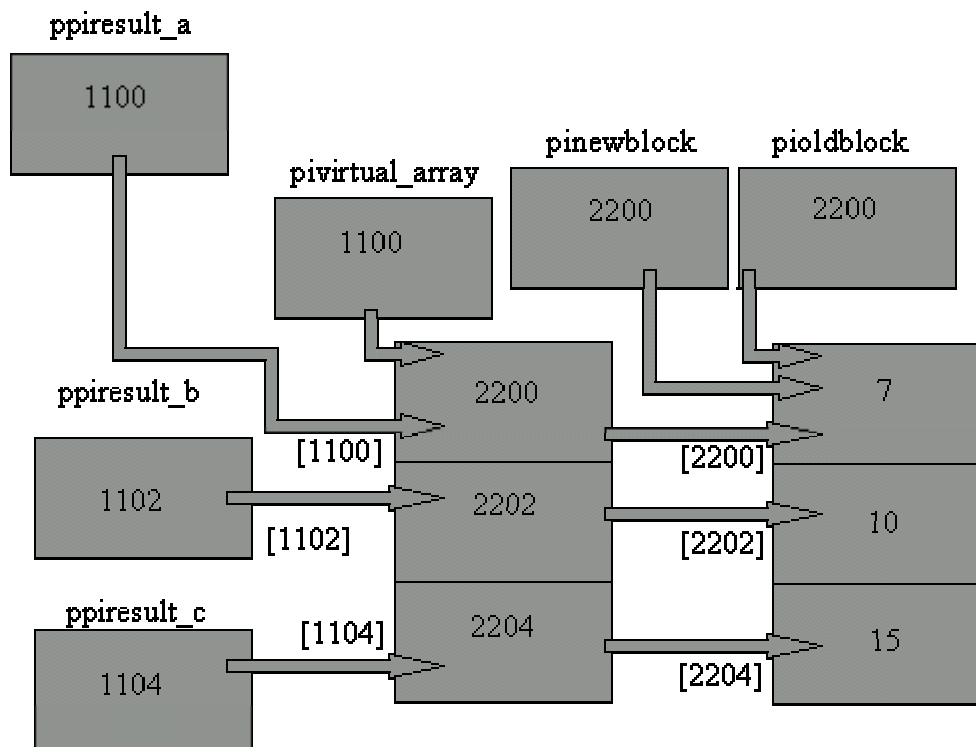


Рис. 9.17. Заполнение ячек данными

Как показано на рис. 9.18, перемещение блока сопровождается перемещением всех связанных с ним данных, что реализуется следующими строками программы:

```
*pinewblock = **ppiresult_a;
*(pinewblock + 1) = **ppiresult_b;
*(pinewblock + 2) = **ppiresult_c;
```

Поскольку указатель `pinewblock` содержит адрес первой ячейки блока, то для того чтобы обратиться к следующим ячейкам, необходимо воспользоваться уже знакомыми нам арифметическими операциями над указателями. При этом арифметическое выражение следует взять в скобки, чтобы первой была выполнена операция сложения, а затем применен оператор раскрытия указателя (*).

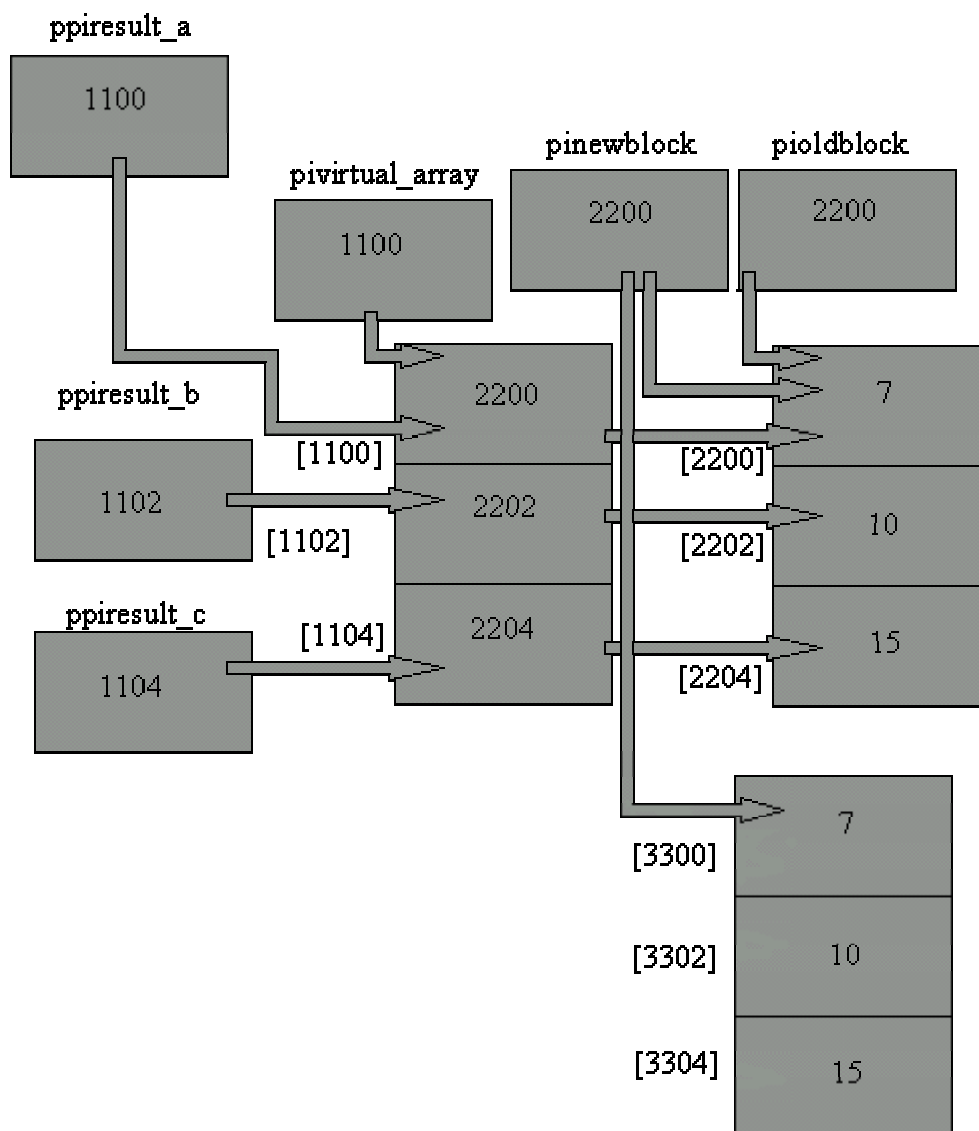


Рис. 9.18. Создание и заполнение нового динамического блока ячеек

На рис. 9.19 показано, что произойдет после вызова функций `free()` и `vassign()` для переадресации указателей массива `pivirtual_array` на ячейки нового блока.

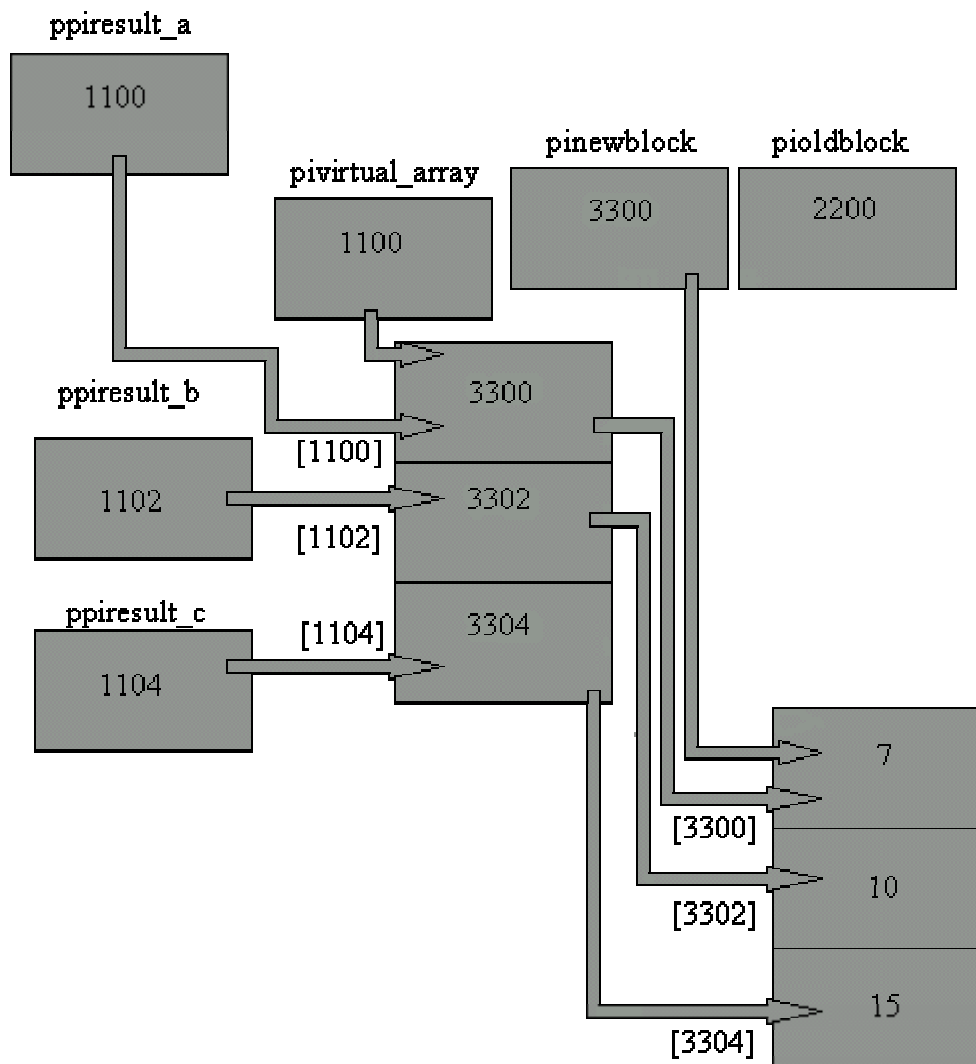


Рис. 9.19. Переадресация указателей массива `pivirtual_array`

Самое важное, что наглядно демонстрирует рис. 9.19, заключается в том, что указатели `ppiresult_a`, `ppiresult_b` и `ppiresult_c` не изменились в ходе выполнения программы. Поэтому когда программа выводит на экран их значения, мы видим все ту же последовательность чисел 7, 10 и 15, хотя размещение соответствующих ячеек в памяти изменилось.

Массивы строковых указателей

Простейший способ управления массивом строк состоит в создании массива указателей на эти строки. Это гораздо проще, чем объявлять двухмерный массив символов. В следующей программе для отображения на экране трех разных сообщений об ошибках применяется массив указателей на строки.

```
/*
 *      arraystr.c
 *  Эта программа на языке C демонстрирует работу
 *  с массивом указателей на строки.
 */
#include <ctype.h>
#include <stdio.h>
#define INUMBER_OF_ERRORS 3
char *pszarray[INUMBER_OF_ERRORS] =
{
```

```

"\nфайл недоступен.\n",
"\nВведенный символ не является алфавитно-цифровым.\n",
"\nЧисло не попадает в диапазон от 1 до 10.\n"
};
FILE *fopen_a_file (char *psz) char cget_a_char (void) ; int
,iget_an_integer (void) ; FILE *pfa_file;
void main () {
char cvalue;
int ivalue;
fopen_a_file("input.dat"); cvalue = cget_a_char() ; ivalue =
iget_an_integer() ,
}
FILE *fopen_a_file(char *psz)
(
const ifopen_a_file_error = 0;
pfa_file = fopen(psz, "r");
if (!pfa_file)
printf("%s",pszarray [ifopen_a_file_error] );
return (pfa_file );
}
char cget_a_char(void) {
char cvalue;
const .icget_a_char_error = 1;
printf("\nВведитебукву: ");
scanf("%c",&cvalue);
if(!isalpha (cvalue))
printf("%s",pszarray[icget_a_char_error]);
return(cvalue); }
int iget_an_integer (void) {
int ivalue;
const iiget_an_integer =2;
printf("\nВведите целое число в диапазоне от 1 до 10: ");
scanf("%d", &ivalue) ;
if ((ivalue< 1) || (ivalue > 10) )
printf("%s", pszarray [iiget_an_integer] );
return (ivalue) ; }

```

Массив pszarray объявлен вне функций, поэтому является глобальным. Обратите внимание, что в каждой из трех функций — `fopen_a_file()`, `cget_a_char()` и `iget_an_integer()` — используется свой индекс для обращения к массиву. Подобный подход позволяет организовать весьма гибкую и надежную систему сообщений об ошибках, так как все такие сообщения локализованы в одном месте и ими легче управлять.

Ссылки в языке C++

Язык C++ предоставляет возможность прямого обращения к переменным по адресам, что даже проще, чем использовать указатели. Аналогично языку C, в C++ допускается создание как обычных переменных, так и указателей. В первом случае резервируется область памяти для непосредственного сохранения данных, тогда как во втором случае в памяти компьютера выделяется место для хранения адреса некоторого объекта, который может быть создан в любое время. В дополнение к этому язык C++ предлагает третий вид объявления переменных — ссылки. Как и указатели, ссылки содержат данные о размещении других переменных, но для получения этих данных не нужно использовать специальный оператор косвенного обращения. Синтаксис использования ссылок в программе довольно прост:

```

int iresult_a = 5;
int& rresult_a = iresult_a; // допустимое создание ссылки

```

```
int$ rresult_b; // недопустимо, поскольку нет
инициализации
```

В данном примере создается ссылка (на это указывает символ & после имени типа данных) `rresult_a`, которой присваивается адрес существующей переменной `ireult_a`. С этого момента на одну и ту же ячейку памяти ссылаются две переменные: `ireult_a` и `rresult_a`. По сути, это два имени одной и той же переменной. Любое присвоение значения переменной `rresult_a` немедленно отразится на содержимом переменной `ireult_a`. Справедливо и обратное утверждение: любое изменение переменной `ireult_a` вызовет изменение значения переменной `rresult_a`. Таким образом, можно сказать, что использование ссылок в языке C++ позволяет реализовать систему псевдонимов переменных.

В отличие от указателей, на ссылки накладывается следующее ограничение: их инициализация должна производиться непосредственно в момент объявления, и однажды присвоенное значение не может быть изменено в ходе выполнения программы. То есть если при объявлении ссылки вы присвоили ей адрес какой-либо переменной, то эта ссылка на протяжении всего времени работы программы будет связана только с этой ячейкой памяти. Вы можете свободно изменять значение переменной, но адрес ячейки, указанный в ссылке, изменить невозможно. Таким образом, ссылке можно представить как указатель с константным значением адреса.

Рассмотрим некоторые примеры. В следующей строке значение, присвоенное выше переменной `ireult_a` (в нашем примере — 5), будет удвоено:

```
rresult_a *= 2;
```

Теперь присвоим новой переменной `icopy_value` (предположим, что она имеет тип `int`) значение переменной `ireult_a`, используя для этого ссылку:

```
icopy_value = rresult_a;
```

Следующее выражение также является допустимым:

```
int *piresult_a = &rresult_a;
```

Здесь адрес ссылки `rresult_a` (т.е., по сути, адрес переменной `ireult_a`) присваивается указателю `piresult_a` типа `int`.

Функции, возвращающие адреса

Когда функция возвращает указатель или ссылку, результатом выполнения функции становится адрес в памяти компьютера. Пользователь может прочитать значение, сохраненное по этому адресу, и даже, если указатель не был объявлен со спецификатором `const`, записать туда свои данные. Это может привести к возникновению трудно обнаруживаемых ошибок в программах. Посмотрим, сможете ли вы разобраться в том, что происходит в следующей программе.

```
//
// refvar.cpp
// Эта программа на языке C++ демонстрирует,
// чего НЕ нужно делать с адресами переменных.
//
#include <iostream.h>
int *ifirst_function(void) ;
int *isecond_function(void);
void main(){
    int *pi = ifirst_function ();
    isecond_function();
    cout<< "Это значение правильное? " << *pi;
}
int *ifirst_function(void){
    int ilocal_to_first = 11;
    return &ilocal_to_first;
}
int *isecond_function(void){
    int ilocal_to_second = 44;
```

```
return &ilocal_to_second;}
```

Запустив программу, вы убедитесь, что она выводит значение 44, а не 11. Как такое может быть?

При вызове функции `ifirst_function()` в программном стеке резервируется место для локальной переменной `ilocal_to_first` и по соответствующему адресу записывается значение 11. После этого функция `ifirst_function()` возвращает адрес этой локальной переменной в основную программу (сам по себе малоприятный факт — компилятор выдает по этому поводу лишь предупреждение, но, увы, не ошибку). В следующей строке программы вызывается функция `isecpnd_function()`, которая, в свою очередь, резервирует место для переменной `ilocal_to_second` и присваивает ей значение 44. Почему же оператор `cout` выводит на экран значение 44, если ему был передан адрес переменной `ilocal_to_first`?

А произошло вот что. Указатель `pi` получил адрес ячейки, в которой хранилось значение локальной переменной `ilocal_to_first`, присвоенное ей во время выполнения функции `ifirst_function()`. Указатель `pi` сохранил этот адрес даже после того, как переменная `ilocal_to_first` вышла за пределы своей области видимости по завершении функции. При этом ячейка памяти, которую занимала переменная `ilocal_to_first` адрес которой записан в указателе `pi`, оказалась свободной, и ее тут же использовала функция `isecpnd_function()` для сохранения своей локальной переменной `ilocal_to_second`. Но поскольку адрес ячейки в указателе `pi` не изменился, то на экран будет выведено значение переменной `ilocal_to_second`, а не `ilocal_to_first`. Мораль такова: нельзя допускать, чтобы функции возвращали адреса локальных переменных.

Глава 10. Ввод-вывод в языке C

- Потокные функции
 - Открытие файлов и потоков
 - Переадресация ввода-вывода
 - Изменение буфера потока
 - Закрытие файлов и потоков
- Низкоуровневый ввод-вывод
- Ввод-вывод символов
 - Функции `getc()`, `putc()`, `fgetc()` и `fputc()`
 - Функции `getchar()`, `putchar()`, `_fgetc()` и `_fputc()`
 - Функции `_getch()`, `_getche()` и `_putch()`
- Ввод-вывод строк
 - Функции `gets()`, `puts()`, `fgets()` и `fputs()`
- Ввод-вывод целых чисел
 - Функции `getw()` и `putw()`
- Форматный вывод данных
 - Функция `printf()`
- Поиск в файлах с помощью функций `fseek()`, `ftell()` и `rewind()`
- Форматный ввод
 - Функции `scanf()`, `fscanf()` и `sscanf()`

Большинство популярных языков программирования высокого уровня имеет довольно ограниченные средства ввода-вывода. В результате программистам часто приходится разрабатывать изощренные алгоритмы, чтобы добиться необходимых результатов при вводе или выводе данных. К счастью, это не относится к языку C, который снабжен мощной подсистемой ввода-вывода, представленной обширной библиотекой соответствующих функций, хотя исторически эта подсистема не была частью языка. Как бы то ни было, те программисты, которые привыкли в языке Pascal использовать только два оператора — `readln` и `writeln`, — будут поражены обилием функций ввода-вывода в языке C. В настоящей главе мы рассмотрим более 20-ти различных способов ввода и вывода информации, существующих в C.

Стандартные библиотечные функции ввода-вывода языка C позволяют считывать и записывать данные, связанные с файлами и различными устройствами. В то же время в самом языке C не предусмотрены какие-то предопределенные структуры файлов. Любые данные рассматриваются как цепочка байтов. В целом же все функции ввода-вывода можно разделить на три основные категории: потоковые, консольные и низкоуровневые.

Все потоковые функции воспринимают данные в виде потока символов. С их помощью можно передавать блоки символов определенного размера и формата, оперировать как отдельными символами, так и достаточно большими и сложными структурами данных.

На практике, когда программа открывает файл, используя потоковую функцию, организуется связь между файлом и структурой типа `file`, которая описана в файле `STDIO.H` и содержит основные сведения об открываемом файле. Одновременно программа получает указатель на эту структуру, называемый еще указателем потока или просто потоком. Данный указатель

используется в дальнейшем при осуществлении всех операций ввода-вывода, связанных с этим файлом.

Потоковые функции обеспечивают возможность буферного, форматного и неформатного ввода-вывода. Буферные потоки позволяют временно сохранять данные в буфере как при записи данных в поток, так и при чтении их из потока. Поскольку прямая запись на диск и считывание с диска требуют много времени, использование в этих целях буферной области памяти значительно ускоряет процесс. Обмен данными потоковые функции всегда осуществляют не символ за символом, а целыми блоками. Если приложению необходимо прочесть блок информации, оно обращается прежде всего к буферу как к наиболее доступной области памяти. Если буфер оказывается пустым, то запрашивается очередной блок данных с диска. Обратный процесс происходит во время буферизованного вывода данных. Вместо того чтобы посылать устройству вывода всю информацию сразу, выводимые данные сначала записываются в буфер. Когда буфер оказывается заполненным, данные "выталкиваются" из него.

Следует учитывать, что многие языки высокого уровня испытывают проблемы с реализацией буферного ввода-вывода. Например, если программа выводит данные в буфер, но не заполняет его целиком (что привело бы к записи данных на диск), то эта информация будет утеряна после завершения программы.

Решение проблемы состоит в использовании специальных функций, которые "выталкивают" данные из буфера. В отличие от других языков высокого уровня, в C предусмотрена автоматическая выгрузка данных из буфера при завершении работы программы. Хотя, конечно, профессионально написанное приложение не должно полагаться на автоматическое решение этой проблемы: всегда следует явно указывать всю последовательность действий, которую должна выполнить программа для предотвращения потери данных. Еще одно замечание: если используются потоковые функции, то в случае аварийного завершения программы вся информация, хранящаяся в выходном буфере, может быть утеряна.

Ввод-вывод может также осуществляться через консоль или порт (например, порт принтера). Во втором случае соответствующие функции просто читают и записывают данные в байтах. Функции работы с консолью предоставляют ряд дополнительных возможностей. Например, можно определить момент ввода символа с клавиатуры, а также включить или отменить режим эха введенных символов на экране.

Последняя категория функций — низкоуровневые. Ни одна из них не осуществляет промежуточной записи данных в буфер и какого бы то ни было форматирования. Все они напрямую используют системные средства ввода-вывода, открывая доступ к файлам и периферийным устройствам на более низком уровне, чем потоковые функции. При открытии файла с помощью низкоуровневых функций возвращается его дескриптор, который представляет собой целое число, употребляемое в дальнейших операциях в качестве идентификатора файла.

Стоит отметить, что довольно порочной практикой является смешение в одной программе потоковых и низкоуровневых функций. Поскольку потоковые функции заносят данные в буфер, а низкоуровневые — обращаются к данным напрямую, то последовательный доступ к одному и тому же файлу двумя функциями разного типа может привести к конфликтам и даже к потере данных в буфере.

Потоковые функции

Чтобы получить доступ к потоковым функциям, приложение должно подключить файл `STDIO.H`. Этот файл содержит объявления констант, типов данных и структур, используемых этими функциями, а также прототипы самих функций и описания служебных макросов.

Многие константы, содержащиеся в файле `STDIO.H`, находят достаточно широкое применение в приложениях. Например, константа `EOF` возвращается функциями ввода при достижении конца файла, а константа `null` служит нулевым ("пустым") указателем. Другие примеры: тип данных `file` представляет собой структуру, содержащую информацию о потоке, а константа `BUFSIZ` задает стандартный размер в байтах буфера, используемого потоком.

Открытие файлов и потоков

Чтобы открыть файл для ввода или вывода данных, воспользуйтесь одной из следующих функций: `fopen()`, `fdopen()` или `freopen()`. Файл открывается для чтения, записи или для

выполнения обеих операций. Кроме того, файл может быть открыт в текстовом или в двоичном режиме.

Все три функции возвращают указатель файла, который используется для обращения к потоку. Например:

```
pfinfile = fopen("input.dat", "r");
```

При запуске приложения автоматически открываются сразу пять стандартных потоков: ввода (stdin), вывода (stdout), ошибок (stderr), печати (stdprn) и внешнего устройства (stdaux). По умолчанию стандартные потоки ввода, вывода и ошибок связаны с консолью. Например, данные, записываемые в стандартный вывод, выводятся на терминал. Любое сообщение об ошибке, сгенерированное библиотечной функцией, также выводится на терминал. Потоки stdprn и stdaux направляют данные соответственно в порт принтера и порт внешнего устройства.

Во всех функциях, где в качестве аргумента требуется указатель на поток, можно использовать любой из перечисленных выше указателей. Некоторые функции, такие как getchar() и putchar(), работают только с потоками stdin и stdout. Поскольку указатели stdin, stdout, stderr, stdprn и stdaux являются константами, а не переменными, не пытайтесь переадресовать их на другие потоки.

Переадресация ввода-вывода

Современные операционные системы рассматривают клавиатуру и экран компьютера как файловые устройства. Это имеет смысл, поскольку система считывает данные с клавиатуры точно так же, как из файла на диске или с магнитной ленты. Аналогично, вывод данных на экран реализуется подобно записи в файл.

Предположим, что ваше приложение считывает данные с клавиатуры и выводит их на экран. И вот, для автоматизации работы вы решили поместить входные данные в файл SAMPLE.DAT и организовать чтение этого файла программой. С этой целью необходимо дать системе указание вместо клавиатуры, которая рассматривается как файл, использовать другой файл — SAMPLE.DAT. Подобный процесс называется переадресацией.

Например, в командной строке MS-DOS переадресация выполняется очень просто. Вы используете оператор < для переадресации ввода и > для переадресации вывода. Допустим, исполняемый файл вашего приложения называется REDIRECT. Тогда следующая команда сначала запустит программу REDIRECT, а затем инициирует ввод данных из файла SAMPLE.DAT вместо стандартного ввода с клавиатуры:

```
redirect< sample.dat
```

Следующая строка одновременно осуществит переадресацию ввода из файла SAMPLE.DAT и переадресацию вывода в файл SAMPLE.BAK:

```
redirect < sample.dat > sample.bak
```

И наконец, последняя строка переадресует только вывод данных:

```
redirect > sample.bak
```

Стандартный поток ошибок stderr не может быть переадресован.

Существует два способа организации связи между стандартными потоками и реальными файлами или периферийными устройствами: переадресация и создание каналов. Под каналом понимают установление прямой связи между стандартным вводом одной программы и стандартным выводом другой. Контроль за переадресацией и каналами обычно осуществляется извне программы, поскольку сама программа, как правило, не отслеживает, откуда приходят и куда уходят данные.

Для того чтобы связать каналом стандартный ввод одной программы и стандартный вывод другой, следует поставить символ вертикальной черты (|):

```
process1 | process2
```

Все детали по организации взаимодействия систем ввода-вывода двух программ PROCESS1 и PROCESS2 берет на себя операционная система.

Изменение буфера потока

Потоки `stdin`, `stdout` и `stderr` буферизуются по умолчанию: данные из них извлекаются, как только буфер переполняется. Потоки `stderr` и `stdaux` не буферизуются, если только они не используются в функциях семейств `printf()` и `scanf()`: в этих случаях им назначается временный буфер. Буферизацию потоков `stderr` и `stdaux` также можно осуществлять с помощью функций `setbuf()` и `setvbuf()`.

Следует отметить, что системные буферы недоступны для пользователей, но буферы, создаваемые функциями `setbuf()` и `setvbuf()`, являются именованными и могут модифицироваться наподобие переменных. Именованные буферы удобно использовать для того, чтобы проконтролировать вводимые или выводимые данные, прежде чем они будут переданы системным функциям и смогут вызвать появление сообщений об ошибках.

Размер создаваемого буфера может быть произвольным. При использовании функции `setbuf()` размер буфера неявно задается константой `BUFSIZ`, объявленной в файле `STDIO.H`. Синтаксис функции выглядит следующим образом:

```
void setbuf(FILE *имя_потока, char *имя_буфера) ;
```

В следующей программе с помощью данной функции потоку `stderr` назначается буфер.

```
/*
 *   setbuf.c
 *   Эта программа на языке C демонстрирует, как назначить
 *   буфер небуферизованному потоку stderr.
 */
#include <stdio.h>
char cmyoutputbuffer[BUFSIZ];
void main(void)
{
    /* Назначение буфера небуферизованному потоку stderr*/
    setbuf(stderr, cmyoutputbuffer); /* попробуйте превратить эту строку в
    комментарий*/
    /* Вставка данных в выходной поток */
    fputs("Строка данных\n", stderr);
    fputs("вставляется в выходной буфер.\n", stderr);
    /* "Выталкивание" буфера потока на экран */
    fflush(stderr); }
```

Запустите программу в режиме отладки, и вы увидите, что выводимые строки появятся на экране только после выполнения последней строки, когда содержимое буфера принудительно "выталкивается". Если заключить строку с вызовом функции `setbuf()` в символы комментария и снова запустить программу в отладчике, то данные, записываемые в поток `stderr`, не будут буферизоваться, и тогда результат выполнения каждой функции `fputs()` будет появляться на экране немедленно.

В приводимой ниже программе используется функция `setvbuf()`, синтаксис которой таков:

```
int setvbuf(FILE *имя_потока, char *имя_буфера, int тип_буфера,
size_t размер_буфера) ;
```

С помощью этой функции размер буфера задается явно, что демонстрируется в следующей программе.

```
/*
 *   setvbuf.c
 *   Эта программа на языке C демонстрирует использование функции
    Setvbuf() .
 */
#include <stdio.h>
#define MYBUFSIZE 512
void main(void) {
    char ichar, cmybuffer[MYBUFSIZE]; FILE *pfinfile, <<pfoutfile;
    pfinfile = fopen("sample.in", "r");
```

```

pfoutfile = fopen("sample.out", "w");
if(setvbuf(pfinfile, cmybuffer, _IOFBF, MYBUFSIZE) != 0)
printf("Ошибка выделения буфера для потока pfinfile.\n");
else
printf("Буфер потока pfinfile создан.\n");
if(setvbuf(pfoutfile, NULL, _IONBF, 0) != 0)
printf("Ошибка выделения буфера для потока pfoutfile.\n") ;
else
printf("Поток pfoutfile не имеет буфера.\n");
while (fscanf (pfinfile, "%c", &ichar,) != EOF)
fprintf(pfoutfile, "%c",ichar);
fclose(pfinfile);
fclose (pfoutfile);
}

```

Программа создает именованный буфер для потока pfinfile и запрещает буферизацию потока pfoutfile. В первом случае указан тип буфера _IOFBF (полная буферизация). Если при этом не задать имя буфера, он будет создан автоматически в динамической памяти и так же автоматически удален по завершении работы программы. Во втором случае указан тип _IONBF (нет буфера). В такой ситуации имя буфера и его размер, даже если они заданы, игнорируются.

Закрывание файлов и потоков

Функция fclose () закрывает указанный файл, тогда как функция _fcloseall() закрывает сразу все открытые потоки, кроме стандартных: stdin, stdout, stderr, stdprn и stdaux. Если в программе явно не закрыть поток, он все равно будет автоматически закрыт по завершении программы. Поскольку одновременно можно открыть только ограниченное число потоков, следует своевременно закрывать те из них, которые больше не нужны.

Низкоуровневый ввод-вывод

При низкоуровневом вводе-выводе не происходит ни буферизации, ни форматирования данных. Файлы, открытые на низком уровне, представляются в программе дескрипторами — целочисленными значениями, которые операционная система использует в качестве ссылок на файлы. Для открытия файла предназначена функция _open (). Чтобы открыть файл в режиме совместного доступа, следует воспользоваться функцией _sopen().

В табл. 10.1 перечислены наиболее часто употребляемые в приложениях функции ввода-вывода низкого уровня. Все они объявлены в файле IO.H.

Данная подсистема ввода-вывода, ориентированная на работу с дисковыми файлами, изначально была создана для операционной системы UNIX. Поскольку комитет ANSI C отказался стандартизировать ее, мы не рекомендуем применять эти функции. В новых проектах предпочтительнее работать со стандартными потоковыми функциями, которые детально рассматриваются в этой главе.

Таблица 10.1. Наиболее часто используемые низкоуровневые функции ввода-вывода	
Функция	Описание
_close()	Закрывает файл на диске
_lseek()	Перемещает указатель файла на заданный байт
_open ()	Открывает файл на диске
_read()	Считывает блок данных из файла
_unlink()	Удаляет файл из каталога
_write()	Записывает блок данных в файл

Ввод-вывод символов

В стандарте ANSI C описан ряд функций, предназначенных для ввода-вывода символов и входящих в стандартный комплект поставки всех компиляторов языка C. Таков общий принцип C: реализовывать ввод-вывод посредством внешних библиотечных функций, а не ключевых слов, являющихся частью языка.

Функции `getc()`, `putc()`, `fgetc()` и `fputc()`

Функция `getc()` читает один символ из указанного файлового потока:

```
int ic;  
ic = getc(stdin);
```

Вас может удивить, почему переменная `ic` не объявлена как `char`. Дело в том, что в прототипе функции `getc()` указано, что она возвращает значения типа `int`. Это связано с необходимостью обрабатывать также признак конца файла, который не может быть сохранен в обычной переменной типа `char`.

Функция `getc()` преобразовывает читаемый символ из типа `char` в тип `unsigned char` и только затем — в `int`. Такой способ обработки данных гарантирует, что символы с ASCII-кодами больше 127 не будут представлены отрицательными числами. Это позволяет зарезервировать отрицательные значения для нестандартных ситуаций — признаков ошибок или конца файла. Так, например, признаком конца файла традиционно служит значение -1. Правда, стандарт ANSI C гарантирует лишь то, что константа EOF содержит некое отрицательное значение.

Хотя может показаться странным, что функция, предназначенная для ввода символов, возвращает целые числа, в действительности язык C не делает больших различий между типами `char` и `int`. Существует четкий алгоритм преобразования целых чисел в символы и наоборот.

Функция `getc()` читает данные из буфера. Это означает, что управление не будет обратно передано программе до тех пор, пока в указанном потоке не встретится символ новой строки. Функция возвращает только самый первый из обнаруженных ею в буфере символов, другие остаются невоображенными. Таким образом, функцию `getc()` нельзя использовать для последовательного ввода символов с клавиатуры, не нажимая при этом каждый раз клавишу [Enter].

Функция `putc()` записывает символ в файловый поток, представленный указателем файла. Например, чтобы отобразить тот символ, который был введен в предыдущем примере, задайте такую строку:

```
putc(ic, stdout);
```

Особенность функций семейства `putc()` состоит в том, что они возвращают константу `eof` (обычно она обозначает конец файла) всякий раз при возникновении ошибочных ситуаций. Это может вызвать некоторые недоразумения, хотя, с технической точки зрения, следующий фрагмент совершенно корректен:

```
if (putc (ic, stdout) == EOF)  
    printf("Обнаружена ошибка записи в stdout");
```

И последнее замечание: функции `getc()` и `putc()` реализованы и как макросы, и как функции. Макроверсии имеют более высокий приоритет и выполняются в первую очередь. Чтобы изменить такой порядок, следует с помощью директивы препроцессора `#undef` отменить определение макроса:

```
#undef getc
```

Существуют "чистые" функции `fgetc()` и `fputc()`, которые выполняют аналогичные действия, но не имеют макросов-"двойников".

Функции `getchar()`, `putchar()`, `_fgetc()` и `_fputc()`

Функции `getchar()` и `putchar()` являются модификациями рассмотренных выше функций `getc()` и `putc()`, работающими только со стандартными потоками ввода (`stdin`) и вывода (`stdout`). Рассмотренные в предыдущем параграфе примеры могут быть переписаны с использованием функций `getchar()` и `putchar()` следующим образом:

```
int ic;
```

```
ic = getchar();
```

И

```
putchar(ic) ;
```

Эти функции тоже реализованы в виде макросов и в виде функций, а аналогичные им функции, `fgetchar()` и `fputchar()`, не имеют макродубликатов.

Функции `_getch()`, `_getche()` и `_putch()`

Функции `_getch()`, `_getche()` и `_putch()`, объявленные в файле `CONIO.H`, не соответствуют стандарту ANSI C, поскольку взаимодействуют напрямую с консолью или портом ввода-вывода. Они не работают с буферами, т.е., например, все символы, вводимые с клавиатуры, немедленно возвращаются функцией `_getch()` в программу. Вывод функции `_putch()` всегда направляется на консоль.

Функция `_getch()` воспринимает нажатия тех клавиш, которые игнорируются функцией `getchar()`, например `[PageUp]`, `[PageDown]`, `[Home]` и `[End]`, и при этом не требует последующего нажатия клавиши `[Enter]`. Она работает в режиме без эха, а ее аналог `_getche()` — с эхом. В случае функциональных и управляющих клавиш эти функции следует вызывать дважды: сначала возвращается 0, а затем — непосредственно код клавиши.

Ввод-вывод строк

Для большинства приложений более важным является ввод-вывод целых строк, а не отдельных символов. Ниже мы познакомимся с основными потоковыми функциями, предназначенными для этих целей. Все они объявлены в файле `STDIO.H`.

Функции `gets()`, `puts()`, `fgets()` и `fputs()`

Предположим, в компании, торгующей лодками, разрабатывается программа, оперирующая строками, каждая из которых состоит из четырех полей: фамилия торгового представителя, отпускная цена, комиссионные и число проданных лодок. Каждое поле отделено от другого символом пробела. С учетом организации данных в файле лучше всего будет рассматривать каждую запись как отдельную строку. Для решения этой задачи наилучшим образом подходит функция `fgets()`, которая считывает всю строку сразу. Противоположна ей функция `fputs()`, осуществляющая вывод строки.

Функция `fgets()` принимает три аргумента: адрес массива, в котором будет сохранена строка, максимальное число символов в строке и указатель потока, из которого выполняется чтение данных. Функция будет считывать символы до тех пор, пока их количество не станет на единицу меньше, чем указанный предел. Последним всегда записывается символ `\0`. Если функция `fgets()` встретит символ новой строки (`\n`), дальнейшее чтение будет прекращено, а сам символ помещен в массив. Если обнаруживается конец потока, чтение также прекращается.

Предположим, у нас есть файл базы данных `SALESMAN.DAT` со следующими строками:

```
Иванов 32767 0.1530
Сергеев 35000 0.1223
Кузьмин 40000 0.1540
```

Допустим также, что максимальная длина строки с учетом символа `\n` не должна превышать 40 символов. Приводимая ниже программа считывает записи из файла и направляет их на стандартное устройство вывода:

```
/*
 * fgets.c
 * Эта программа на языке C демонстрирует процесс считывания строк с
 * помощью функции fgets() и вывода их с помощью функции fputs()
 */
#include <stdio.h>
#define INULL_CHAR 1
#define IMAX_REC_SIZE 40
void main ()
```

```

{
FILE *pfinfile;
char crecord[IMAX_REC_SIZE + INULL_CHAR];
pfinfile = fopen("salesman.dat", "r") ;
while(fgets(crecord, IMAX_REC_SIZE + INULL_CHAR, pfinfile) !=
NULL)
fputs(crecord, stdout);
fclose(pfinfile);}

```

Поскольку максимальная длина строки равна 40 символам, следует создать для нее массив, содержащий 41 элемент. Дополнительный элемент необходим для хранения признака конца строки \0. Программа не генерирует самостоятельно никаких разрывов строк при выводе строк на терминал. Тем не менее, структура строк сохраняется такой же, что и в исходном файле, поскольку символы \n читаются и сохраняются в массиве функцией fgets(). Функция fputs() выводит содержимое массива crecord в поток stdout без каких-либо изменений.

Функция gets() отличается от функции fgets () тем, что читает данные только из потока stdin, причем до тех пор, пока не будет нажата клавиша [Enter], не проверяя, достаточно ли в указанном массиве места для размещения всех введенных символов. Символ новой строки \n, генерируемый клавишей [Enter], заменяется символом \0.

Функция puts() выводит данные в стандартный поток вывода stdout и добавляет в конец выводимой строки символ \n, чего не делает функция fputs().

Пример совместной работы всех перечисленных функций был дан в главе "Массивы".

Ввод-вывод целых чисел

В некоторых приложениях бывает необходимо считывать и записывать потоки (в том числе буферизованные) целых чисел. Для этих целей в языке C существуют две функции: getw() и putw().

Функции getw() и putw()

Дополняющие друг друга функции getw() и putw() очень похожи по своему действию на функции getc() и putc() за тем исключением, что работают с целыми числами, а не символами, и могут использоваться только с файлами, открытыми в двоичном режиме. Следующая программа открывает двоичный файл, записывает в него ряд целых чисел, закрывает файл, а затем вновь открывает его для чтения данных с одновременным выводом их на экран:

```

/*
*      badfile.c
*      Эта программа на языке C демонстрирует использование функций
getw() и
*      putw() для ввода-вывода данных из двоичного файла .
*/
#include <stdio.h>
#include <stdlib.h>
#define ISIZE 10
void main () {
FILE *pfi;
int ivalue, ivalues [ISIZE], i;
pfi = fopen ("integer.dat","wb") ;
if (pfi== NULL) {
printf("Неудалось открыть файл.");
exit(1);
}
for(i = 0; i < ISIZE; i++) {
ivalues[i] = i + 1;
putw(ivalues[i], pfi); } fclose(pfi);

```

```

pfi = fopen("integer.dat", "wb");
if(pfi == NULL) {
printf("Неудалось открыть файл.");
exit(1); } while(!feof(pfi)) {
ivalue = getw(pfi);
printf("%3d",ivalue); } }

```

Посмотрите, какие данные будут выведены на экран при выполнении этой программы, и попытайтесь определить, что было сделано неправильно:

```
1 2 3 4 5 6 7 8 9 10 -1
```

Поскольку в цикле while может встретиться признак конца файла EOF, в программе используется функция feof() , сигнализирующая об обнаружении конца файла. Но особенностью этой функции является то, что она не выполняет упреждающего чтения, а лишь проверяет состояние специального флага, который, в свою очередь, устанавливается только после того, как будет непосредственно выполнена операция чтения признака конца файла.

Чтобы исправить ошибку, допущенную в предыдущем примере, применим метод упреждающего чтения.

```

/*
*   getwputw.c
*   Это исправленная версия предыдущего примера .
*/
#include <stdio.h>
#include <stdlib.h>
#define ISIZE 10
void main () {
FILE *pfi;
int ivalue, ivalues [ISIZE], 1;
pfi = fopen ("integer .dat","wb"); if (pfi == NULL) {
printf("Неудалось открыть файл.");
exit(1); } for(i = 0; i < ISIZE; i++) {
ivalues [i]= i + 1;
putw (ivalues [i], pfi); } fclose (pfi);
pfi = fopen ("integer .dat","rb");
if (pfi== NULL) {
printf("Неудалось открыть файл.");
exit(1);
}
ivalue = getw(pfi); while(!feof(pfi)) {
printf("%3d",ivalue) ivalue = getw(pfi); }

```

Прежде чем приступить к выполнению цикла while, программа осуществляет упреждающее чтение файла, для того чтобы проверить, не является ли он пустым. Если файл не пуст, то в переменную ivalue будет записано целочисленное значение. Если же файл окажется пустым, то это будет обнаружено функцией feof() .

Также обратите внимание, что метод упреждающего чтения потребовал изменения последовательности инструкций в цикле while. Предположим, цикл выполнен уже девять раз. На девятой итерации переменная ivalue приняла значение 9. На следующей итерации на экран будет выведено 9, а переменной будет присвоено значение 10. Цикл выполнится еще раз, в результате чего на экране отобразится 10 и переменная ivalue примет значение -1, соответствующее константе EOF. Это вызовет завершение цикла while, поскольку функция feof() определит конец файла.

Форматный вывод данных

Для вывода форматированных данных, как правило, применяются функции printf() и fprintf(). Первая записывает данные в поток stdout, а вторая — в указанный файл или поток. Как уже говорилось ранее, аргументами функции printf() являются строка форматирования и список выводимых переменных. (В функции fprintf() первый аргумент — указатель файла.) Для каждой

переменной из списка задание формата вывода осуществляется с помощью следующего синтаксиса:

`% [флаги] [ширина] [.точность] [{h | l}] спецификатор`

В простейшем случае указывается только знак процента и спецификатор, например %f. Обязательное поле спецификатор указывает на способ интерпретации переменной: как символа, строки или числа (табл. 10.2). Необязательное поле флаги определяет дополнительные особенности вывода (табл. 10.3).

Необязательное поле ширина задает минимальную ширину поля вывода. Если количество выводимых символов меньше указанного значения, поле дополняется слева или справа пробелами или нулями в зависимости от установленных флагов.

Необязательное поле точность интерпретируется следующим образом:

- при выводе чисел формата e, E и f определяет количество цифр после десятичной точки (последняя цифра округляется);
- при выводе чисел формата g и G определяет количество значащих цифр (по умолчанию 6);
- при выводе целых чисел определяет минимальное количество цифр (если цифр недостаточно, число дополняется ведущими нулями);
- при выводе строк определяет максимальное количество символов, лишние символы отбрасываются (по умолчанию строка выводится, пока не встретится символ '\0').

Поля ширина и точность могут быть заданы с помощью символа-заменителя *. В этом случае в списке переменных должны быть указаны их настоящие значения.

Необязательные модификаторы h и l определяют размерность целочисленной переменной: h соответствует ключевому слову short, l — long.

Таблица 10.2. Спецификаторы типа переменной в функциях printf() и fprintf()		
Спецификатор	Тип	Формат вывода
c	int	Символ (переменная приводится к типу unsignedchar)
d, i	int	Знаковое десятичное целое число
o	int	Беззнаковое восьмеричное целое число
u	int	Беззнаковое десятичное целое число
x	int	Беззнаковое шестнадцатеричное целое число (в качестве цифр от 10 до 15 используются буквы "abcdef")
X	int	Беззнаковое шестнадцатеричное целое число (в качестве цифр от 10 до 15 используются буквы "ABCDEF")
e	double	Знаковое число с плавающей запятой в формате [-] t. ddde xxx, где t — одна десятичная цифра, ddd — ноль или более десятичных цифр (количество определяется значением поля точность, нулевая точность подавляет вывод десятичной точки, по умолчанию точность — 6), 'e' — символ экспоненты, xxx — ровно три десятичные цифры (показатель экспоненты)
E	double	То же, что и e, только вместо символа 'e' применяется 'E'
f	double	Знаковое число с плавающей запятой в формате [-] mmm.ddd, где mmm — одна или более десятичных цифр, ddd — ноль или более десятичных цифр (количество определяется значением поля точность, нулевая точность подавляет вывод десятичной точки, по умолчанию точность — 6)
g	double	Знаковое число с плавающей запятой в формате f или e; формат e выбирается, если показатель экспоненты меньше -4 или больше либо равен значению поля точность; десятичная точка не ставится, если за ней не следуют значащие цифры; хвостовые нули не выводятся
G	double	То же, что и g, но при необходимости выбирается формат E, а не e
n	int *	Ничего не выводится; количество символов, выведенных к данному моменту функцией, записывается в переменную

p	void *	Адрес, содержащийся в указателе (отображается в формате x)
s	char *	Строка символов; вывод осуществляется до тех пор, пока не будет обнаружен символ \0 или число символов не станет равным значению поля точность

Таблица 10.3. Флаги в функциях printf() и fprintf()	
Флаг	Назначение
-	Если число выведенных символов оказывается меньше указанного, результат выравнивается по левому краю поля вывода (по умолчанию принято правостороннее выравнивание)
+	При выводе знаковых чисел знак отображается всегда (по умолчанию знак устанавливается только перед отрицательными числами)
0	Если значению поля ширина предшествует символ '0', выводимое число дополняется ведущими нулями до минимальной ширины поля вывода (по умолчанию в качестве заполнителей применяются пробелы); при левостороннем выравнивании игнорируется
пробел	Если выводится положительное знаковое число, перед ним ставится пробел (по умолчанию пробел в таких случаях не ставится); игнорируется при наличии флага +
#	Для чисел формата o, x и X означает добавление ведущих 0, 0x и 0X соответственно (по умолчанию отсутствуют); для чисел формата e, E, g, G и f задает присутствие десятичной точки, даже когда за ней не следуют значащие цифры (по умолчанию точка в таких случаях не ставится); для чисел формата g и G предотвращает отбрасывание хвостовых нулей (по умолчанию отбрасываются)

Функция printf()

В следующей программе демонстрируется, как правильно применять различные спецификаторы форматирования к переменным четырех типов: символу, массиву символов, целому числу и числу с плавающей запятой. Программа содержит достаточно подробные комментарии, а, кроме того, выводимые строки пронумерованы, чтобы легче было обнаружить, какая из функций их сгенерировала.

```

/*
 *      printf.c
 *  Эта программа на языке C демонстрирует применение
 *  спецификаторов форматирования функции printf().
 */
#include <stdio.h>
void main () {
    char    c      = 'A',
    psz[] = "Строка для экспериментов";
    int    iln    = 0,
    ivalue = 1234;
    double dPi    = 3.14159265;
    /* 1 — вывод символа c */
    printf("\n[%2d] %c",++iln, c) ;
    /* 2 — вывод ASCII-кода символа c */
    printf("\n[%2d] %d",++iln, c);
    /* 3 — вывод символа c ASCII-кодом 90 */
    printf("\n[%2d] %c",++iln, 90);
    /* 4 — вывод значения ivalue в восьмеричной системе */
    printf("\n[%2d] %o",++iln, ivalue);
    /* 5 — вывод значения ivalue в шестнадцатеричной */
    /* системе с буквами в нижнем регистре */
    printf("\n[%2d] %x",++iln, ivalue);
    /* 6 — вывод значения ivalue в шестнадцатеричной */
    /* системе с буквами в верхнем регистре */
    printf("\n[%2d] %X",++iln, ivalue);
    /* 7 — вывод одного символа, минимальная ширина поля равна 5, */
    /* выравнивание вправо с дополнением пробелами */
    printf("\n[%2d] %5c",++iln, c); .

```

```

/* 8 -- вывод одного символа, минимальная ширина поля равна 5, */
/* выравнивание влево с дополнением пробелами */
printf("\n[%2d]%-5c",++iln, c);
/* 9 -- вывод строки, отображаются 24 символа */
printf("\n[%2d]s",++iln, psz);
/* 10 -- вывод минимум 5-ти символов строки, отображаются 24 символа */
printf("\n[%d]5s",-n-iln, psz);
/* 11 -- вывод минимум 38-ми символов строки, */
/* выравнивание вправо с дополнением пробелами */
printf("\n[%d]38s",++iln, psz);
/* 12 -- вывод минимум 38-ми символов строки, */
/* выравнивание влево с дополнением пробелами */
printf("\n[%d]-38s",++iln, psz);
/* 13 -- вывод значения ivalue, по умолчанию отображаются 4 цифры */
printf("\n[%d]d",++iln, ivalue);
/* 14 -- вывод значения ivalueso знаком */
printf("\n[%d]+d",++iln, ivalue);
/* 15 -- вывод значения ivaluemинимум из 3-х цифр, */
/* отображаются 4 цифры*/
printf("\n[%d]3d",++iln, ivalue);
/* 16 -- вывод значения ivaluemинимум из 10-ти цифр, */
/* выравнивание вправо с дополнением пробелами */
printf("\n[%d]10d",++iln, ivalue);
/* 17 -- вывод значения ivaluemинимум из 10-ти цифр, */
/* выравнивание влево с дополнением пробелами */
printf("\n[%d]-10d",++iln, ivalue);
/* 18 -- вывод значения ivalue минимум из 10-ти цифр, */
/* выравнивание вправо с дополнением нулями */
printf("\n[%d]010d",-n-iln, ivalue);
/* 19 -- вывод значения dPic форматированием по умолчанию */
printf("\n[%d]f",++iln, dPi);
/* 20 -- вывод значения dPi, минимальная ширина поля равна 20, */
/* выравнивание вправо с дополнением пробелами */
printf("\n[%d]20f",++iln, dPi);
/* 21 -- вывод значения dPi, минимальная ширина поля равна 20, */
/* выравнивание вправо с дополнением нулями */
printf("\n[%d]020f",++iln, dPi);
/* 22 -- вывод значения dPi, минимальная ширина поля равна 20, */
/* выравнивание влево с дополнением пробелами */
printf("\n[%d]-20f",++iln, dPi);
/* 23 -- вывод 19-ти символов строки, */
/* минимальная ширина поля равна 19 */
printf("\n[%d]19.19s",++iln, psz);
/* 24 -- вывод первых двух символов строки */
printf("\n[%d],2s",++iln, psz);
/* 25 -- вывод первых двух символов строки, минимальная ширина поля */
/* равна 19, выравнивание вправо с дополнением пробелами */
printf("\n[%d]119.2s",++iln, psz);
/* 26 -- вывод первых двух символов строки, минимальная ширина поля */
/* равна 19, выравнивание влево с дополнением пробелами */
printf("\n[%d]-19.2s",++iln, psz);
/* 27 -- вывод первых шести символов строки, минимальная ширина поля */
/* равна 19, выравнивание вправо с дополнением пробелами */

```

```

printf ("\n[%d]%.s",++iln, 19,6, psz);
/* 28 – вывод значения dPi, минимальная ширина поля */
/*      равна 10, 8 цифр после десятичной точки      */
printf("\n[%d] %10.8f",++iln, dPi);
/* 29 – вывод значения dPi, минимальная ширина поля */
/* равна 20, 2 цифры после десятичной точки, */
/* выравнивание вправо с дополнением пробелами */
printf("\n[%d]%20.2f",++iln, dPi) ;
/*' 30 – вывод, значения dPi, минимальная ширина поля */
/*      равна 20, 4 цифры после десятичной точки, */
/*      . выравнивание влево с дополнением пробелами */
printf("\n[%d]%-20.4f",++iln, dPi);
/* 31 – вывод значения dPi, минимальная ширина поля */
/* равна 20, 4 цифры после десятичной точки, */
/* выравнивание вправо с дополнением пробелами */
printf("\n[%d]%20.4f",++iln, dPi);
/* 32 – вывод значения dPi, минимальная ширина поля */
/* равна 20, 2 цифры после десятичной точки, */
/* выравнивание вправо с дополнением пробелами, */
/* научный формат (с экспонентой) */
printf("\n[%d]%20.2e",++iln, dPi);

```

Результат работы программы будет выглядеть следующим образом:

```

[ 1] A
[ 2] 65
[ 3] Z
[ 4] 2322
[ 5] 4d2
[ 6] 4D2
[ 7]      A
[ 8] A
[ 9] Строка для экспериментов
[10] Строка для экспериментов
[11] Строка для экспериментов
[12] Строка для экспериментов
[13] 1234          .
[14] +1234
[15] 1234
[16]      1234
[17] 1234
[18] 0000001234
[19] 3.141593
[20]                3.141593
[21] 00000000000003.141593
[22] 3.141593
[23] Строка для эксперим
[24] Ст
[25]                Ст
[26] Ст
[27]                Строка
[28] 3.14159265
[29]                3.14
[30] 3.1416

```

```
[31]          3.1416
[32]          3.14e+000
```

Поиск в файлах с помощью функций `fseek()`, `ftell()` и `rewind()`

Функции `fseek()`, `ftell()` и `rewind()` предназначены для определения и изменения положения маркера текущей позиции файла. Первая из них, `fseek()`, перемещает маркер в файл, заданном указателем `pf`, на требуемое число байтов, определенное аргументом `ibytes`. Перемещение осуществляется от начала файла (аргумент `ifrom` равен 0), от текущего положения маркера (аргумент `ifrom` равен 1) или от конца файла (аргумент `ifrom` равен 2). В языке C предусмотрены три константы, которые можно указывать в качестве аргумента `ifrom`: `seek_set` (сдвиг от начала файла), `seek_cur` (сдвиг от текущей позиции) и `seek_end` (сдвиг от конца файла). Функция `fseek()` возвращает ноль при успешном завершении и EOF в противном случае. Синтаксис функции таков:

```
fseek(pf,  ibytes, ifrom) ;
```

Функция `ftell()` возвращает текущую позицию маркера в файле, которая определяется величиной смещения в байтах от начала файла. Возвращаемое значение имеет тип `long`.

Функция `rewind()` просто перемещает маркер в начало файла.

В следующей программе на языке C продемонстрировано использование всех трех функций: `fseek()`, `ftell()` и `rewind()`.

```
/*
 *   fseek.c
 *   Эта программа на языке C демонстрирует использование
 *   функций fseek(), ftell() и rewind() .
 */
#include <stdio.h>
void main ( ) {
    FILE *pf;
    char c;
    long llocation;
    pf = fopen("test.dat","r+t");
    c = fgetc(pf) ;
    putchar (c) ;
    c = fgetc(pf) ;
    putchar (c);
    llocation = ftell (pf) ;
    c = fgetc (pf) ;
    putchar (c);
    fseek(pf,llocation, 0) ;
    c = fgetc(pf) ;
    putchar(c);
    fseek(pf, llocation, 0);
    fputc('E', pf) ;
    fseek(pf,llocation, 0) ;
    c = fgetc(pf); putchar(c);
    rewind (pf);
    c = fgetc (pf) ;
    putchar(c);
}
```

Переменная `llocation` имеет тип `long`. Это связано с тем, что язык C поддерживает работу с файлами, размер которых превышает 64 Кб. Файл TEST.DAT содержит строку "ABCD". Первая из функций `fgetc()` возвращает букву 'A', после чего программа выводит ее на экран. В следующих двух строках выводится буква 'B'.

Далее функция `ftell()` записывает в переменную `llocation` значение маркера текущей позиции в файле. Поскольку буква 'B' уже прочитана, переменная `llocation` будет содержать 2. Это означает, что в данный момент маркер указывает на третий символ, который смещен на 2 байта от первого символа 'A'.

В следующей паре строк считывается и отображается буква 'C'. После выполнения данной операции маркер сдвинется на четвертый символ — 'D'.

В этом месте программа вызывает функцию `fseek()`, которая перемещает маркер на 2 байта от начала файла (отсчет идет от начала файла, поскольку третий аргумент функции равен 0). После выполнения функции маркер вновь "указывает на третий символ в файле. Поэтому в результате работы следующих двух строк на экран будет выведена буква 'C'.

При втором вызове функции `fseek()` устанавливаются те же параметры, что и в первом случае. Но на этот раз с помощью функции `fputc()` в файл записывается буква 'E' на место буквы 'C'. Чтобы убедиться, что новая буква действительно была вставлена в файл, программа в очередной раз обращается к функции `fseek()` для перемещения к третьей позиции, считывает символ в этой позиции и отображает его. Этим символом будет 'E'.

Затем вызывается функция `rewind()`, которая перемещает маркер в начало файла. При следующем вызове функции `fgetc()` из файла считывается буква 'A', которая и выводится на экран. В результате работы программы на экране будет получена следующая последовательность букв:

ABCSEA

Форматный ввод

Форматирование вводимых данных в языке C можно осуществлять с помощью достаточно мощных функций `scanf()` и `fscanf()`. Различие между ними заключается в том, что последняя требует указания файла, из которого читаются данные. Функция `scanf()` принимает данные из стандартного входного потока `stdin`.

Функции `scanf()`, `fscanf()` и `sscanf()`

Функция `scanf()`, как и ее "родственники" `fscanf()` и `sscanf()`, в качестве аргумента принимает такого же рода строку форматирования, что и функция `printf()`, осуществляющая вывод данных, хотя имеются и некоторые отличия. Для примера рассмотрим следующее выражение:

```
scanf("%2d%5s%4f", sival, psz, &fvalue);
```

Данная функция считывает целое число, состоящее из двух цифр, строку из пяти символов и число с плавающей запятой, образованное не более чем четырьмя символами (2,97, 12,5 и т.п.). Все аргументы, перечисленные после строки форматирования, должны быть указателями, т.е. содержать адреса переменных. А теперь попробуем разобрать более сложное выражение:

```
scanf (" \\%[^A-Za-z-] %*[-] %[^\\"]", ps1, ps2);
```

Первым в строке форматирования стоит пробел. Он служит указанием пропустить все ведущие пробелы, символы табуляции и новой строки, пока не встретится двойная кавычка ("). В строке форматирования этот символ защищен обратной косой чертой (\), так как в противном случае он означал бы завершение самой строки! Таким образом, обратная косая черта является своего рода командой отмены специального назначения следующего за ней символа.

Управляющая последовательность `%[^A-Za-z-]` говорит о том, что в строку `ps1` нужно вводить все, кроме букв и символа дефиса (-). Квадратные скобки, в которых на первом месте стоит знак крышки (], определяют диапазон символов, из которых не должна состоять вводимая строка. В данном случае в диапазон входят буквы от 'A' до 'Z' (дефис между ними означает "от и до") и от 'a' до 'z', а также сам дефис. Если убрать знак ^, то, наоборот, будут ожидаться только буквы и символы дефиса. В конец прочитанной строки добавляется символ \0. Если во входном потоке вслед за открывающей кавычкой первыми встретятся буква или дефис, выполнение функции `scanf()` завершится ошибкой и в переменные `ps1` и `ps2` ничего не будет записано, а сама строка останется в буфере потока. Чтение строки `ps1` продолжится до тех пор, пока не встретится один из символов, указанных после знака ^.

Вторая управляющая последовательность `%*[-]` говорит о том, что после чтения строки `ps1` должна быть обнаружена группа из одного или нескольких дефисов, которые необходимо

пропустить. Символ звездочки (*) указывает на то, что данные должны быть прочитаны, но не сохранены. Отметим два важных момента.

- Если после строки ps1 первой встретится буква, а не дефис, выполнение функции scanf() завершится ошибкой, в переменную ps2 ничего не будет записано, а сама строка останется в буфере потока.
- Если бы в первой управляющей последовательности символ дефиса не был отмечен как пропускаемый, функция scanf() работала бы неправильно, так как символ дефиса, являющийся "ограничителем" строки ps1, воспринимался бы как часть этой строки! В этом случае чтение было бы прекращено только после обнаружения буквы, а это, в свою очередь, привело бы к возникновению ошибки, описанной в первом пункте.

Последняя управляющая последовательность %[^\"] указывает на то, что в строку ps2 нужно считывать все символы, кроме двойных кавычек, которые являются признаком завершения ввода.

Для полной ясности приведем пример входной строки:

```
"65---AAA"
```

Вот что будет получено в результате: *ps1= 65, *ps2 = aaa.

Все вышесказанное справедливо и в отношении функций fscanf () и sscanf (). Функция sscanf() работает точно так же, как и scanf(), но читает данные из указанного символьного массива, а не из потока stdin. В следующем примере показано, как с помощью функции sscanf() преобразовать строку цифр в целое число:

```
sscanf (psz, "%d", &ivalue);
```

Глава 11. Основы ввода-вывода в языке C++

- Подсистема ввода-вывода в C++
 - Стандартные потоки `cin`, `cout` и `cerr`
 - Операторы ввода (`>>`) и вывода (`<<`) данных
- Флаги и функции форматирования
- Файловый ввод-вывод
- Определение состояния потока

Хотя в целом C++ является расширенной версией языка C, это не означает, что для создания эффективной программы достаточно взять каждое выражение на C и записать его синтаксический эквивалент на C++. Очень часто C++ предлагает не только дополнительные операторы, но и новые способы решения традиционных задач. Одним из новшеств языка является уникальная подсистема ввода-вывода, знакомство с которой мы начнем в этой главе, а завершим в главе 15. Разделение данной темы на две главы необходимо из-за разнообразия средств ввода-вывода, используемых в C++, что связано с внедрением в язык концепции объектно-ориентированного программирования. Только после того, как вы уясните принципы создания объектов и манипулирования ими, вы сможете в полной мере освоить новую методику ввода-вывода данных посредством объектов. Пока же вам следует быть готовыми к тому, что некоторые термины и понятия, упоминаемые в настоящей главе, могут оказаться для вас неизвестными.

Подсистема ввода-вывода в C++

Стандартные функции ввода-вывода, используемые в языке C и объявленные в файле `STDIO.H`, доступны также и в C++. В то же время C++ располагает своим собственным файлом заголовков `IOSTREAM.H`, содержащим набор средств ввода-вывода, специфичных для этого языка.

Потоковый ввод-вывод в C++ организуется посредством комплекта стандартных классов, подключаемых с помощью файла `IOSTREAM.H`. Эти классы содержат перегруженные операторы ввода `>>` и вывода `<<`, которые поддерживают работу с данными всевозможных типов. Чтобы лучше понять, почему легче работать с потоками в C++, чем в C, давайте вспомним, как вообще в языке C реализуется ввод и вывод данных.

Прежде всего вспомним о том, что язык C не имеет встроенных средств ввода-вывода. Все функции, такие как `printf()` или `scanf()`, предоставляются через внешние библиотеки, хоть и считающиеся стандартными, но не являющиеся частью самого языка. В принципе, это позволяет гибко решать проблемы, возникающие в различных приложениях, в соответствии с их особенностями и назначением.

Трудности появляются в связи с тем, что подобного рода функций слишком много, они по-разному возвращают значения и принимают разные аргументы. Программисты полагаются главным образом на функции форматного ввода-вывода, `printf()`, `scanf()` и им подобные, особенно если приходится работать с числами, а не текстом. Эти функции достаточно универсальны, но зачастую, из-за обилия всевозможных спецификаторов форматирования, становятся чересчур громоздкими и трудно читаемыми.

Язык C++ точно так же не располагает встроенными средствами ввода-вывода, но предлагает модульный подход к решению данной проблемы, группируя возможности ввода-вывода в трех основных потоковых классах:

`istream` содержит средства ввода
`ostream` содержит средства вывода
`iostream` поддерживает двунаправленный ввод-вывод, является производным от первых двух классов

Во всех этих классах реализованы операторы `<<` и `>>`, оптимизированные для работы с конкретными данными.

Библиотека IOSTREAM.H содержит также классы, с помощью которых можно управлять вводом-выводом данных из файлов:

`ifstream` Порожден от `istream` и подключает к программе файл, предназначенный для ввода данных

`ofstream` Порожден от `ostream` и подключает к программе файл, предназначенный для вывода данных

`fstream` Порожден от `iostream` и подключает к программе файл, предназначенный как для ввода, так и для вывода данных

Стандартные потоки `cin`, `cout` и `cerr`

Стандартным потокам языка C `stdin`, `stdout` и `stderr`, объявленным в файле `STDIO.H`, в C++ соответствуют объекты-потоки `cin`, `cout`, `cerr` и `clog`, подключаемые посредством файла `IOSTREAM.H`.

`cin` Объект класса `istream`, связанный со стандартным потоком ввода

`cout` Объект класса `ostream`, связанный со стандартным потоком вывода

`cerr` Объект класса `ostream`, не поддерживающий буферизацию и связанный со стандартным потоком ошибок

`clog` Объект класса `ostream`, поддерживающий буферизацию и связанный со стандартным потоком ошибок

Все они открываются автоматически во время запуска программы и обеспечивают интерфейс между программой и пользователем.

Операторы ввода (>>) и вывода (<<) данных

Ввод-вывод в C++ значительно усовершенствован и упрощен благодаря универсальным операторам >> (ввод) и << (вывод). Их универсальность стала возможной благодаря появившемуся в C++ понятию перегрузки операторов, которая заключается в создании функций, имена которых совпадают с именами стандартных операторов языка. Компилятор различает вызов настоящего и "функционального" операторов на основании типов передаваемых им операндов. Операторы >> и << перегружены таким образом, чтобы поддерживать все стандартные типы данных C++, включая классы. Рассмотрим пример вывода данных с помощью функции `printf()` в языке C:

```
printf("Целое число: %d, число с плавающей запятой: %f", ivalue, fvalue);
```

А теперь запишем это же выражение на C++:

```
cout<< "Целое число: " << ivalue<< ", число с плавающей запятой: " << fvalue;
```

Обратите внимание, что один и тот же перегруженный оператор используется для вывода данных трех разных типов: строк, целых чисел и чисел с плавающей запятой. Оператор << самостоятельно анализирует тип данных и выберет подходящий формат их представления. Аналогично работает оператор ввода. Сравним два эквивалентных выражения на языках C и C++:

```
/* на языке C */
scanf("%d%f%c",&ivalue, &fvalue, &c);
// на языке C++.
cin >> ivalue >> fvalue >> c;
```

Нет необходимости при вводе данных ставить перед именами переменных оператор взятия адреса &. В C++ оператор >> берет на себя задачу вычисления адреса, определения формата и прочих особенностей записи значения переменной.

Прямым следствием перегрузки является возможность расширения операторов << и >> для обработки данных нестандартных типов. Ниже показано, как перегрузить оператор вывода, чтобы он мог принимать данные нового типа `tclient`:

```
ostream& operator <<(ostream& osout, tclient client)
{
    osout<< " " << client.pszname;
    osout<< " " << client.pszaddress;
```



```
osout<< " " << client .pszphone;
}
```

Тогда для вывода содержимого структуры client на экран достаточно будет задать такое выражение:

```
cout << client;
```

Эффективность этих операторов объясняется компактностью поддерживающего их программного кода. При обращении к функциям типа printf() и scanf() задействуется большой объем кода, основная часть которого не используется при выполнении конкретной операции ввода или вывода данных. Даже если в языке C вы оперируете только целыми числами, все равно будет подгружаться весь код функции с блоками обработки данных самых разных типов. Напротив, в языке C++ реально подгружаться будут только те подпрограммы, которые необходимы для решения конкретной задачи.

В следующей программе оператор >> применяется для чтения данных различных типов:

```
//
// insert.cpp
// Эта программа на языке C++ демонстрирует применение оператора >>
// для
// ввода данных типа char, int и double, а также строк.
//
#include <iostream.h>
#define INUMCHARS. 45
#define INULL_CHAR 1
void main(void) {
char canswer;
int ivalue;
double dvalue;
char pszname[INUMCHARS + INULL_CHAR];
pout<< "Эта программа позволяет вводить данные различных типов.\n";
cout<< "Хотите попробовать? (Y- да, N - нет) ";
cin >> canswer;
if (canswer == 'Y'){
cout << "\n"<< "Введите целое число: ";
cin >> ivalue;
cout<< "\n\nВведите число с плавающей запятой: "; cin >> dvalue;
cout<< "\n\nВведите ваше имя: "; cin >> pszname; cout<< "\n\n"; } }
```

В данном примере оператор << используется в простейшем виде — для вывода текста приглашений. Обратите внимание, что оператор >> выглядит во всех случаях одинаково, если не считать имен переменных.

Теперь приведем пример программы, в которой оператор << применяется для вывода данных различных типов:

```
//
I/ extract.cpp
// Эта программа на языке C++ демонстрирует применение оператора <<
// для
// вывода данных типа int, float, а также строк.
//
#include <iostream.h>
void main (void)
{
char description[] = "Магнитофон";
int quantity = 40;
float price = 45.67;
cout<< "Наименование товара: " << description<< endl;
```

```
cout<<      "Количество на складе: " << quantity<< endl;
cout<<      "Цена в долларах: " << price<< endl;
}
```

Программа выведет на экран следующую информацию:

Наименование товара: Магнитофон

Количество на складе: 40

Цена в долларах: 45.67

Здесь следует обратить внимание на манипулятор endl. Манипуляторами называются специальные функции, которые в выражениях с потоковыми объектами, такими как cout и cin, записываются подобно обычным переменным, но в действительности выполняют определенные действия над потоком. Манипулятор endl широко применяется при выводе данных в интерактивных программах, так как помимо записи символа новой строки он очищает буфер потока, вызывая "выталкивание" содержащихся в нем данных. Эту же операцию, но без вывода символа \n, можно выполнить с помощью манипулятора flush.

Рассмотрим некоторые особенности ввода-вывода строк:

```
//
//  string.cpp
//  Эта программа на языке C++ иллюстрирует особенности
//  работы оператора >> со строками.
//
#include <iostream.h>
#define INUMCHARS 45
#define INULL_CHARACTER 1
void main (void) {
char ps zname[ INUMCHARS + INULL_CHARACTER] ;
cout<< "Введите ваше имя и фамилию: ";
cin >> pszname;
cout << "\n\nСпасибо, " << pszname;
```

В результате выполнения программы будет выведена следующая информация:

Введите ваши имя и фамилию: Александр Иванов

Спасибо, Александр

Почему не была выведена вся строка? Это связано с особенностью оператора >>: он прекращает чтение строки, как только встречает первый пробел, знак табуляции или символ новой строки (кроме ведущих). Поэтому в переменной pszname сохранилось только имя, но не фамилия. Чтобы решить эту проблему, следует воспользоваться функцией cin.get (), как показано ниже:

```
//
//  cinget.cpp
//  Эта программа на языке C++ демонстрирует применение оператора >>
//  совместно с функцией get() для ввода строк с пробелами.
//
#include <iostream.h>
#define INUMCHARS 45
#define INULL_CHARACTER 1
void main(void)
char pszname[INUMCHARS + INULL_CHARACTER];
cout<< "Введите ваше имя и фамилию: ";
cin.get(pszname, INUMCHARS);
cout << "\n\nСпасибо, " << pszname;
```

Теперь программа отобразит данные следующим образом:

Введите ваше имя и фамилию: Александр Иванов

Спасибо, Александр Иванов

Функция `cin.get()`, помимо имени переменной, принимающей данные, ожидает два дополнительных аргумента: максимальное число вводимых символов и символ, служащий признаком конца ввода. В качестве последнего по умолчанию используется символ `\n`. Функция `cin.get()` считывает все символы в строке, включая пробелы и знаки табуляции, пока не будет прочитано указанное число символов или не встретится символ-ограничитель. Если в качестве ограничителя необходимо, к примеру, использовать знак `*`, следует записать такое выражение:

```
cin.get(pszname, INUMCHARS, '*');
```

Флаги и функции форматирования

Работа всех потоковых объектов из библиотеки `IOSTREAM.H` контролируется флагами форматирования, определяющими такие параметры, как, например, основание системы счисления при выводе целых чисел и точность представления чисел с плавающей запятой.

Флаги можно устанавливать с помощью функции `setf()`, а сбрасывать — с помощью функции `unsetf()`. Есть два варианта функции `setf()` с одним аргументом типа `long` и с двумя. Первым аргументом является набор флагов, объединенных с помощью операции побитового ИЛИ (`|`). Возможные флаги перечислены в таблице 11.1.

Таблица 11.1. Флаги форматирования	
Флаг	Назначение
<code>skipws</code>	при вводе пробельные литеры пропускаются
<code>left</code>	выводимые данные выравниваются по левому краю с дополнением символами-заполнителями по ширине поля
<code>right</code>	выводимые данные выравниваются по правому краю с дополнением символами-заполнителями по ширине поля (установлен по умолчанию)
<code>internal</code>	при выравнивании символы-заполнители вставляются между символом знака или префиксом основания системы счисления и числом
<code>dec</code>	целые числа выводятся по основанию 10 (установлен по умолчанию); устанавливается также манипулятором <code>dec</code>
<code>oct</code>	целые числа выводятся по основанию 8; устанавливается также манипулятором <code>oct</code>
<code>hex</code>	целые числа выводятся по основанию 16; устанавливается также манипулятором <code>hex</code>
<code>showbase</code>	при выводе целых чисел отображается префикс, указывающий на основание системы счисления
<code>showpoint</code>	при выводе чисел с плавающей запятой всегда отображается десятичная точка, а хвостовые нули не отбрасываются
<code>uppercase</code>	шестнадцатеричные цифры от A до F, а также символ экспоненты E отображаются в верхнем регистре
<code>showpos</code>	при выводе положительных чисел отображается знак плюс
<code>scientific</code>	числа с плавающей запятой отображаются в научном формате (с экспонентой)
<code>fixed</code>	числа с плавающей запятой отображаются в фиксированном формате (без экспоненты)
<code>unitbuf</code>	при каждой операции вывода буфер потока должен очищаться
<code>stdio</code>	при каждой операции вывода буферы потоков <code>stdout</code> и <code>stderr</code> должны очищаться

Вторым аргументом является специальная битовая маска, определяющая, какую группу флагов можно модифицировать. Имеются три стандартные маски:

```
adjustfield = internal | left | right
```

```
basefield = dec | oct | hex
```

```
floatfield = fixed | scientific
```

Оба варианта функции `setf()` возвращают значение типа `long`, содержащее предыдущие установки всех флагов.

Все перечисленные флаги, а также константы битовых масок и упоминавшиеся манипуляторы `dec`, `hex` являются членами класса `ios` - базового в иерархии всех классов ввода-вывода. В этот класс входят, помимо прочего, функции `fill()`, `precision()` и `width()`, тоже связанные с форматированием выводимых данных!

Функция `fill()` устанавливает переданный ей символ в качестве символа-заполнителя. Аналогичные действия выполняет манипулятор `setfill()`. Вызванная без аргументов, функция возвращает текущий символ-заполнитель. По умолчанию таковым служит пробел.

Когда установлен флаг `scientific` или `fixed`, функция `precision()` задает точность представления чисел с плавающей запятой, в противном случае определяет общее количество значащих цифр. Аналогичные действия выполняет манипулятор `setprecision()`. По умолчанию точность равна 6. Вызванная без аргументов, функция возвращает текущее значение точности.

Функция `width()` определяет минимальную ширину поля вывода в символах. Если при выводе количество символов оказывается меньше ширины поля, оно дополняется специальными символами-заполнителями. После каждой операции записи значение ширины поля сбрасывается в 0. Аналогичные действия выполняет манипулятор `setw()`. Вызванная без аргументов, функция возвращает текущую ширину поля.

Манипуляторы `setfill()`, `setprecision()` и `setw()`, также являющиеся членами класса `ios`, относятся к категории параметризованных, и для работы с ними необходимо дополнительно подключить к программе файл `IOMANIP.H`.

Следующая программа является написанным на C++ аналогом программы `printf.c`, рассмотренной в предыдущей главе.

```
// ioflags.cpp
// Эта программа на языке C++ демонстрирует применение
// флагов и функций форматирования.
#include <strstream.h>
#define MAX_LENGTH 20 void row (void);
main() {
    char c = 'A', psz[] = "Строка для экспериментов",
    strbuffer[MAX_LENGTH];
    int ivalue = 1234;
    double dPi = 3.14159265;
    // вывод символа c
    row(); // [ 1]
    cout << c;
    // вывод ASCII-кода символа c
    row(); // [2]
    cout << (int)c;
    // вывод символа c ASCII-кодом 90
    row(); // [ 3]
    cout << (char)90;
    // вывод значения ivalue в восьмеричной системе
    row(); // [4]
    cout << oct << ivalue;
    // вывод значения ivalue в шестнадцатеричной системе
    // с буквами в нижнем регистре
    row(); // [5]
    cout << hex << ivalue;
    // вывод значения ivalue в шестнадцатеричной системе
    // с буквами в верхнем регистре
    row(); // [ 6]
    cout.setf(ios::uppercase);
    cout << hex << ivalue;
    cout.unsetf(ios::uppercase); // отмена верхнего регистра символов
    cout << dec; // возврат к десятичной системе
    // вывод одного символа, минимальная ширина поля равна 5,
    // выравнивание вправо с дополнением пробелами
    row(); // [ 7]
    cout.width(5); cout << c;
    // вывод одного символа, минимальная ширина поля равна 5,
    // выравнивание влево с дополнением пробелами
    row(); // [ 8]
    cout.width(5);
    cout.setf(ios::left);
    cout << c;
    cout.unsetf(ios::left);
}
```

```

// вывод строки, отображаются 24 символа
row();// [9] cout<< psz;
// вывод минимум 5-ти символов строки
row();// [10] cout.width(5); cout<< psz;
// вывод минимум 38-ми символов строки,
// выравнивание вправо с дополнением пробелами
row();// [11]
cout.width(38); cout << psz;
// вывод минимум 38-ми символов строки,
// выравнивание влево с дополнением пробелами
row ();// [12]
cout.width(38);
cout.setf(ios::left);
cout << psz;
cout.unsetf(ios::left);
// вывод значения ivalue, по умолчанию отображаются 4 цифры
row();// [13] cout<< ivalue;
// вывод значения ivalue со знаком
row ();// [14]
cout.setf(ios::showpos);
cout << ivalue;
cout.unsetf(ios::showpos) ;
// вывод значения ivalue минимум из 3-х цифр, отображаются 4 цифры
row ();// [15]
cout.width(3); cout << ivalue;
// вывод значения ivalue минимум из 10-ти цифр,
} // выравнивание вправо с дополнением пробелами
row();// [16]
cout.width(10); cout<< ivalue;
// вывод значения ivalue минимум из 10-ти цифр,
// выравнивание влево с дополнением пробелами
row();// [17]
cout.width(10);
cout.setf(ios::left) ;
cout << ivalue;
cout.unsetf(ios::left);
// вывод значения ivalue минимум из 10-ти цифр,
// выравнивание вправо с дополнением нулями
row (); // [18]
cout.width (10);
cout.fill ('0');
cout << ivalue;
cout.fill (' ');
// вывод значения dPi форматированием по умолчанию
row(); // [19]
cout<< dPi;
// вывод значения dPi, минимальная ширина поля равна 20,
// выравнивание вправо с дополнением пробелами
row(); // [20]
cout.width (20); cout<< dPi;
// вывод значения dPi, минимальная ширина поля равна 20,
// выравнивание вправо с дополнением нулями
row();// [21]
cout.width(20);
cout.fill('0');
cout << dPi;
cout. fill (' ');
// вывод значения dPi, минимальная ширина поля равна 20,
// выравнивание влево с дополнением пробелами

```

```

row();// [22]
cout.width(20);
cout.setf(ios::left) ;
cout<< dPi;
// вывод значения dPi, минимальная ширина поля равна 20,
// выравнивание влево с дополнением нулями
row();// [23]
cout.width(20);
cout. fill ('0');
cout << dPi;
cout.unsetf(ios::left);
cout.fill(' ');
// вывод 19-ти символов строки, минимальная ширина поля равна 19
row();// [24]
ostream(strbuffer, 20).write(psz,19)<< ends;
cout.width(19);
cout << strbuffer;
// вывод первых двух символов строки
row(); // [25]
ostream(strbuffer, 3).write(psz,2) << ends;
cout<< strbuffer;
// вывод первых двух символов строки, минимальная ширина поля равна 19,
// выравнивание вправо с дополнением пробелами
row();// [26] cout.width(19); cout << strbuffer;
// вывод первых двух символов строки, минимальная ширина поля равна 19,
// выравнивание влево с дополнением пробелами row();// [27] cout.width(19);
cout .setf (ios :: left) ; cout << strbuffer; cout .unsetf (ios :: left) ;
// вывод значенияdPi из5-ти значащих цифр
row ();// [28]
cout .precision (9); cout << dPi;
// вывод значения dPiиз 2-х значащих цифр, минимальная ширина поля
// равна 20, выравнивание вправо с дополнением пробелами
row ();// [29]
cout. width (20) ;
cout .precision (2);
cout << dPi;
// выводзначенияdPi из 4-хзначащихцифр
row () ; // [30]
cout .precision (4); cout << dPi;
// вывод значения dPi из 4-х значащих цифр, минимальная ширина поля
// равна 20, выравнивание вправо с дополнением пробелами
row();// [31]
cout. width(20); cout<< dPi;
// вывод значения dPi, минимальная ширина поля равна 20,
// 4 цифры после десятичной точки, выравнивание вправо
IIs дополнением пробелами, научный формат (с экспонентой)
row ();// [32]
cout . setf (ios :: scientific) ;
cout. width ( 20 );
cout << dPi;
cout. unset f (ios: : scientific );
return (0);
}

void row(void) {
static int In = 0;
cout << " \n [";
cout.width(2) ;
cout<< ++In<< "]" "; }

```

Результат работы программы будет выглядеть следующим образом:

```

[ 1]  A
[ 2]  65
[ 3]  z
[ 4]  2322
[ 5]  4d2
[ 6]  4D2
[ 7]A
[ 8]A
[ 9]  Строка для экспериментов
[10]  Строка для экспериментов
[11]'    Строка для экспериментов
[12]Строка для экспериментов
[13]1234
[14]+1234
[15]1234
[16]1234
[17] 1234
[18] 0000001234
[19] 3.14159
[20] 3.14159
[21] 000000000000003.14159
[22] 3.14159
[23] 3.141590000000000000
[24] Строка для эксперим
[25] Ст
[26] Ст
[27] Ст
[28] 3.14159265
[29] 3.1
[30] 3.142
[31]3.142
[32] 3.1416e+000

```

В этой программе следует обратить внимание на использование класса `ostrstream` в пунктах 24 и 25, а также неявно в пунктах 26 и 27. Этот класс управляет выводом строк. Необходимость в нем возникла в связи с тем, что флаги форматирования оказывают влияние на работу оператора `<<`, но не функции `write()` класса `ostream` (класс `ostrstream` является его потомком и наследует данную функцию), которая записывает в поток указанное число символов строки. Выводимые ею данные всегда прижимаются к левому краю. Поэтому мы поступаем следующим образом:

```

ostrstream(strbuffer, 20).write(psz,19)<< ends;
cout.width(19); cout << strbuffer;

```

Разберем этот фрагмент шаг за шагом. Сначала вызывается конструктор класса `ostrstream`, создающий временный безымянный объект данного класса, автоматически уничтожаемый по завершении строки. Если вы еще не знакомы с понятием конструктора, то поясним, что это особая функция, предназначенная для динамического создания объектов своего класса. Конструктор класса `ostrstream` требует указания двух аргументов: буферной строки, в которую будет осуществляться вывод, и размера буфера. Особенность заключается в том, что реальный размер буферного массива может быть больше указанного, но все данные, записываемые сверх лимита, будут отсекаются.

Итак, конструктор создал требуемый объект, связанный с буфером `strbuffer`. Этот объект является обычным потоком в том смысле, что им можно управлять как и любым другим потоком вывода, только данные будут направляться не на экран, а в буфер. Вторым действием вызывается функция `write()` объекта, записывающая в поток первые 19 символов строки `psz`. Оператор точка (.) в данном случае обозначает операцию доступа к члену класса. Здесь тоже есть своя особенность: функция `write()` возвращает адрес потока, из которого она была вызвана. А это, в свою очередь, означает, что мы можем тут же применить к потоку оператор

вывода <<. Итого, три действия в одном выражении! Манипулятор `ends` класса `ostream` вставляет в поток символ `\0`, служащий признаком завершения строки. Именно поэтому в буфере было зарезервировано 20 ячеек — на единицу больше, чем количество выводимых символов строки `psz`. В результате выполненных действий в буфере оказалась сформированная готовая строка, которая, в конце концов, и направляется в поток `cout`.

Теперь рассмотрим, что происходит в пункте 25:

```
ostream (strbuffer, 3) .write (psz, 2) << ends;
cout << strbuffer;
```

Последовательность действий та же, но резервируется не весь буфер, а только первые три ячейки. Трюк в том, что объект `cout` воспринимает массив `strbuffer` как обычную строку, но поскольку в третьей ячейке стоит символ `\0`, то получается, что строка состоит из двух символов! В пунктах 26 и 27 используется содержимое буфера, полученное в пункте 25.

Важно также помнить, что функция `width()` оказывает влияние только на следующий за ней оператор вывода, тогда как, например, действие функции `precision()` останется в силе до тех пор, пока она не будет выполнена снова либо не будет вызван манипулятор `setprecision()`.

Файловый ввод-вывод

Во всех программах на языке C++, рассмотренных до сих пор, для ввода и вывода данных применялись стандартные потоки `cin` и `cout`. Если для этих же целей удобнее работать с файлами, следует воспользоваться классами `ifstream` и `ofstream`, которые порождены от классов `istream` и `ostream` и унаследовали от них функции, соответственно, чтения и записи. Необходимо также подключить файл `FSTREAM.H` (он, в свою очередь, включает файл `IOSTREAM.H`). В следующей программе демонстрируется работа с файлами посредством классов `ifstream` и `ofstream`:

```
//
//  fstream. cpp
//  Эта программа на языке C++ демонстрирует, как создавать
//  потоки ifstream и ofstream для обмена данными с файлом.
//
#include <fstream.h>
int main(void) {
    char c;
    ifstream ifsin("text.in", ios::in); if(!ifsin)
        cerr<< " \nНевозможно открыть файл text.in для чтения.";
    ofstream ofout("text.out", ios::out);
    if(!ofout)
        cerr<< " \nНевозможно открыть файл text.out для записи.";
    while (ofout && ifsin.get(c) )
        ofout.put (c);
    if ifsin.close ();
    ofout.close ();
    return(0); }
```

Программа создает объект `ifsin` класса `ifstream` и связывает с ним файл `TEXT. IN`, находящийся в текущем каталоге. Всегда следует проверять доступность указанного файла. В данном случае проверка осуществляется достаточно оригинальным образом. Оператор `!` в потоковых классах перегружен таким образом, что, будучи примененным к соответствующему объекту, при наличии каких-либо ошибок в потоке возвращает ненулевое значение. Аналогичным образом создается и объект `ofout` класса `ofstream`, связываемый с файлом `TEXT.OUT`.

В цикле `while` осуществляется считывание и запись отдельных символов в выходной файл до тех пор, пока в ряду символов не попадется `EOF`. При завершении программы закрываются оба файла. Особенно важно закрыть файл, в который записывались данные, чтобы предупредить потерю информации, еще находящейся в буфере.

Иногда возникают ситуации, когда необходимо отложить процесс спецификации файла или когда с одним объектом нужно последовательно связать сразу несколько потоков. В следующем фрагменте показано, как это сделать:

```
ifstream ifsin;
.
.
.
ifsin.open("week1.in");
.
.
.
if sin.close () ;
ifsin.open("week2.in")
.
.
.
if sin. close () ;
```

Если необходимо изменить режим доступа к файлу, то это можно сделать путем модификации второго аргумента конструктора соответствующего файлового объекта, Например:

```
ofstream ofsout("file.out", ios::app | ios::nocreate);
```

В данном выражении делается попытка создать объект ofsout и связать его с файлом FILE.OUT. Поскольку указан флаг ios::nocreate, подключение не будет создано, если файл FILE.OUT не существует. Флаг ios::app означает, что все выводимые данные будут добавляться в конец файла. В следующей таблице перечислены флаги, которые можно использовать во втором аргументе конструкторов файловых потоков (допускается объединение флагов с помощью операции побитового ИЛИ):

Флаг	Назначение
ios: : in	Файл открывается для чтения, его содержимое не очищается
ios: : out	Файл открывается для записи
ios: : ate	После создания объекта маркер текущей позиции устанавливается в конец файла
ios: : app	Все выводимые данные добавляются в конец файла
ios: : trunc	Если файл существует, его содержимое очищается (автоматически устанавливается при открытии файла для записи)
ios: : nocreate	Объект не будет создан, если файл не существует
ios: : noreplace	Объект не будет создан, если файл существует
ios: : binary	Файл открывается в двоичном режиме (по умолчанию — в текстовом)

Для обмена данными с файлом можно также использовать объект класса fstream. Например, в следующем выражении файл UPDATE.DAT открывается для чтения и записи данных:

```
fstream io("update.dat", ios::in | ios::app);
```

К объектам класса iostream (а класс fstream является его потомком) можно применять функции seekg() и seekp(), позволяющие управлять положением маркера текущей позиции файла. Функция seekg() задает положение маркера, связанного с чтением данных из файла, а функция seekp() — с записью. Обе функции требуют указания одного или двух аргументов. Если указан один аргумент, он считается абсолютным адресом маркера. Если два, то первый задает относительное смещение, а второй — направление перемещения. Существует также функция tellg(), возвращающая текущее положение маркера чтения, и функция tellp(), возвращающая текущее положение маркера записи. Рассмотрим небольшой фрагмент программы:

```
streampos current_position = io.tellp();
io << obj1 << obj2 << obj3;
io.seekp(current_position);
io.seekp(sizeof(obj1), ios::cur);
io<< newobj2;
```

Сначала создается указатель current_position типа streampos, который инициализируется текущим адресом маркера записи. Во второй строке в поток io записываются три объекта. В третьей строке с помощью функции seekp() указатель перемещается в сохраненную позицию.

Далее с помощью оператора sizeof() вычисляется объем памяти в байтах, занимаемый в файле объектом obj1, и функция seekp() "перескакивает" через него. В результате по верх объекта obj2 записывается объект newobj2.

Вторым аргументом функций seekg() и seekp() может быть один из следующих флагов: ios::beg(смещение на указанную величину от начала файла), ios::cur(смещение на указанную величину от текущей позиции) и ios::end (смещение на указанную величину от конца файла). Например, данное выражение переместит маркер чтения на 5 байтов от текущей позиции:

```
io.seekg(5, ios::cur);
```

Следующее выражение переместит маркер на 7 байтов от конца файла:

```
io.seekg(-7, ios::end);
```

Определение состояния потока

С каждым потоком связана внутренняя переменная состояния. В случае возникновения ошибки устанавливаются определенные биты этой переменной в зависимости от категории ошибки. Обычно принято, что если поток класса ostream находится в ошибочном состоянии, то функции записи не выполняются и не меняют состояние потока. Существует ряд функций, позволяющих определить состояние потока:

Функция	Действие
eof()	Возвращает ненулевое значение при обнаружении конца файла
fail()	Возвращает ненулевое значение в случае возникновения какой-либо ошибки в потоке, возможно не фатальной; если функция bad() при этом возвращает 0, то, скорее всего, можно продолжать работу с потоком, предварительно сбросив флаг ios::failbit
bad()	Возвращает ненулевое значение в случае возникновения серьезной ошибки ввода-вывода; в этом случае продолжать работу с потоком не рекомендуется
good()	Возвращает ненулевое значение, если биты состояния не установлены
rdstate()	Возвращает текущее состояние потока в виде одной из констант: ios::goodbit(нет ошибки), ios::eofbit(достигнут конец файла), ios::failbit(возможно, некритическая ошибка форматирования или преобразования), ios::badbit(критическая ошибка)
clear()	Задаёт состояние потока; принимает аргумент типа int, который по умолчанию равен 0, что соответствует сбросу всех битов состояния, в противном случае содержит одну или несколько перечисленных в предыдущем пункте констант, объединенных с помощью операции побитового ИЛИ ()

Приведем пример:

```
ifstream pfsinfile("sample.dat", ios::in);
if (pfsinfile.eof ())
pfsinfile.clear() ; // состояние потока pfsinfile сбрасывается
if (pfsinfile.fail () )
cerr<< ">>> ошибка при создании файла sample.dat<<<";
if (pfsinfile.good ()) cin >> my_object;
if(!pfsinfile) // другой способ обнаружения ошибки
cout<< ">>> ошибка при создании файла sample.dat<<<";
```

Глава 12. Дополнительные типы данных

- Структуры
 - Синтаксис
 - Доступ к членам структур
 - Создание простейшей структуры
 - Передача структур в качестве аргументов функции
 - Массивы структур
 - Указатели на структуры
 - Передача массива структур в качестве аргумента функции
 - Структуры в C++
 - Вложенные структуры
- Битовые поля
- Объединения
 - Синтаксис
 - Создание простейшего объединения
- Ключевое слово `typedef`
- Перечисления

В этой главе рассматриваются некоторые дополнительные типы данных, используемые в C и C++: структуры, объединения, битовые поля и ряд других. Структуры являются очень важным средством, так как служат основой для построения классов — фундаментального понятия объектно-ориентированного программирования. Разобравшись в принципах создания структур, вы легко поймете, как работают классы в C++. Поэтому начнем мы главу с того, что выясним, как формировать простые структуры и массивы структур, передавать их в функции и получать доступ к элементам структур посредством указателей. Объединения и битовые поля являются близкими к структурам типами данных, и после знакомства со структурами вам будет несложно понять их особенности.

Структуры

Понятию структуры можно легко найти аналог в повседневной жизни. Обычная записная книжка, содержащая адреса друзей, телефоны, дни рождений и прочую информацию, по сути своей является структурой взаимосвязанных элементов. Список файлов и папок в окне Windows — тоже структура. Точнее сказать, это все примеры использования структур, но что такое структура сама по себе? Можно дать следующее определение: структура — это группа переменных разных типов, объединенных в единое целое.

Синтаксис

Структура создается с помощью ключевого слова `struct`, за которым следует необязательный тег, а затем — список членов структуры. Тег становится именем нового типа данных и может использоваться для создания переменных этого типа. Описание структуры в общем виде выглядит следующим образом:

```
struct тег { тип1 имя1;  
тип2 имя2;  
тип3 имя3;  
.
```

```
.
.
тип-п имя-п; };
```

В примерах программ этой главы мы будем работать с такой структурой:

```
struct stboat {
char szmodel[15 + 1];
char szserial[20 + 1];
int iyear;
long lmotor_hours;
float fsaleprice; };
```

В данном случае создается структура с именем `stboat`, содержащая описание моторной лодки. Массив `szmodel` хранит название модели лодки, а массив `szserial` — ее регистрационный номер. В переменной `iyear` записан год изготовления лодки, в переменной `lmotor_hours` — наработанный ресурс мотора в часах, в переменной `fsaleprice` — цена.

Создать переменную на основании описанной структуры можно следующим образом:

```
struct stboat stused_boat;
```

В этом выражении объявляется переменная `stused_boat` типа `structstboat`. Допускается и такое объявление:

```
struct stboat {
char szmodel[15 + 1];
char szserial[20 + 1];
int iyear;
long lmotor_hours;
float fsaleprice; } stused_boat;
```

В этом случае переменная создается одновременно с описанием структуры. Если больше не предполагается создавать переменные данного типа, то тег структуры можно не указывать:

```
struct {
char szmodel[15 + 1];
char szserial[20 + 1];
int iyear;
long lmotor_hours;
float fsaleprice;
} stused_boat;
```

Созданная структура называется безымянной. Больше нигде в программе нельзя будет создать ее экземпляры, разве только придется полностью повторить ее описание. Но зато во время описания структуры можно объявить сразу несколько переменных:

```
struct {
char szmodel[15 + 1];
char szserial[20 + 1];
int iyear;
long lmotor_hours;
float fsaleprice; } stboat1, stboat2, stboat3;
```

Компилятор зарезервирует для членов структуры необходимый объем памяти, точно так же, как при объявлении обычных переменных.

Дополнительный синтаксис структур в C++

В C++ при создании экземпляра структуры, описанной ранее, ключевое слово `struct` можно не указывать:

```
/* действительно как в C, так и в C++ */
struct stboat stused_boat;
// действительно только в C++
```

```
stboat stused_boat;
```

Доступ к членам структур

Доступ к отдельному члену структуры можно получить с помощью оператора точки:

имя_переменной. член_структуры

Например, в языке C запросить содержимое поля `szmodel` описанной выше структуры `stused_boat` можно с помощью следующего выражения:

```
gets(stused_boat.szmodel);
```

Вывести полученное значение на экран можно так:

```
printf("%s", stused_boat.szmodel);
```

В C++ применяется аналогичный синтаксис:

```
cin >> stused_boat.szmodel; cout << stused_boat.szmodel;
```

Создание простейшей структуры

В следующем примере используется структура `stboat`, описание которой мы рассматривали выше.

```
/*
 *      struct, c
 *  Эта программа на языке C демонстрирует работу со структурой.
 */
#include <stdio.h>
#define iSTRING15 15
#define lSTRING20 20 #define iNULL_CHAR 1
struct stboat {
    char szmodel[iSTRING15 + iNULL_CHAR];
    char szserial[iSTRING20 + iNULL_CHAR];
    int iyear;
    long lmotor_hours;
    float fsaleprice; } stused_boat;

int main(void)
*{
    printf("\nВведите модель судна: "); gets(stused_boat.szmodel) ;
    printf ("\nВведите регистрационный номер судна: ")/'
    gets(stused_boat.szserial) ;
    printf("\nВведите год изготовления судна: ");
    scanf("%d",&stused_boat.iyear) ;
    printf("\nВведите число моточасов, наработанных двигателем: ");
    scanf("%d",&stused_boat.lmotor_hours) ;
    printf("\nВведите стоимость судна: ");
    scanf("%a",Sstused_boat.fsaleprice);
    printf("\n\n");
    printf("Судно %s  %dгода выпуска с регистрационным номером #s,\n",
    stused_boat.szmodel,  stused_boat.iyear,
    stused_boat.szserial);
    printf("отработавшее%d моточасов,",  stused_boat.lmotor_hours);
    printf(" было продано за $%.2f.\n",  stused_boat.fsaleprice); return(0);
}
```

При выполнении этой программы на экран будет выведена информация примерно следующего содержания:

Судно "Чайка" 1982 года выпуска с регистрационным номером #XA1011, отработавшее 34187 моточасов, было продано за \$18132.00.

Передача структур в качестве аргументов функции

Часто бывает необходимо передать структуру в функцию. При этом аргумент-структура передается по значению, то есть в функции модифицируются лишь копии исходных данных. В прототипе функции следует задать структуру в качестве параметра (в C++ указывать ключевое слово `struct` необязательно):

```
/* синтаксис объявления функции в C и C++ */
void vprint_data (struct stboat stused_boat) ;
// синтаксис, доступный только в C++
void vprint_data (stboat stused_boat) ;
```

Следующая программа является модифицированной версией предыдущего примера. В ней структура `stused_boat` передается в функцию `vprint_data()`.

```
/*
 *      structfn.c
 *  Эта программа на языке C демонстрирует передачу структуры в
 *  функцию.
 */
#include <stdio.h>
#define ISTRING15 15
#define ISTRING20 20
#define INULL_CHAR 1
struct stboat {
    char szmodel[ISTRING15 + INULL_CHAR] ;
    char szserial[ISTRING20 + INULL_CHAR] ;
    int iyear;
    long lmotor_hours;
    float fsaleprice;
};
void vprint_data(struct stboat stused_boat);
int main(void) {
    struct stboat stused_boat;
    printf("\nВведите модель судна: "); gets(stused_boat.szmodel);
    printf("\nВведите регистрационный номер судна: ");
    gets(stused_boat.szserial);
    printf("\nВведите год изготовления судна: ");
    scanf("%d",&stused_boat.iyear);
    printf("\nВведите число моточасов, наработанных двигателем: ");
    scanf("%d",&stused_boat.lmotor_hours);
    printf("\nВведите стоимость судна: ");
    scanf("%f",&stused_boat.fsaleprice);
    vprint_data(stused_boat); return(0);
}
void vprint_data(struct stboat stused_boat) {
    printf("\n\n");
    printf("Судно  %s %d года выпуска с регистрационным номером#%s,\n",
        stused_boat.szmodel, stused_boat.iyear,
        stused_boat.szserial);
    printf("отработавшее%d моточасов,", stused_boat.lmotor_hours);
    printf(" было продано за$%8.2f\n", stused_boat.fsaleprice);
}
```

Массивы структур

В следующей программе создается массив структур с информацией о лодках.

```
/*
 *      structar.c
 *  Эта программа на языке C демонстрирует работу с массивом структур.
```

```

*/
#include <stdio.h>
#define iSTRINGIS 15
#define iSTRING20 20
#define iNULL_CHAR 1
#define iMAX_BOATS 50
struct stboat {
char szmodel[iSTRINGIS + iNULL_CHAR];
char szserial[iSTRING20 + iNULL_CHAR];
char szcomment[80];
int iyear;
long lmotor_hours;
float fsaleprice;
};
int main (void)
{
int i, iinstock;
struct stboat astBoats [ iMAX_BOATS ];
printf( "Информацию о скольких лодках следует ввести в базу 'данных?'
") ; scanf("%d",&iinstock) ;
for (i =0;i < iinstock; i++) (
_flushall());
/* очистка буфера */
printf ("\nВведите модель судна: ") ; gets (astBoats [i]. szmodel) ;
printf ("\nВведите регистрационный номер судна: "); gets (astBoats
[i]. szserial) ;
printf ("\nВведите строку заметок о судне: "); gets (astBoats [i].
szcomment) ;
printf ("\nВведите год изготовления судна: "); scanf("%d",SastBoats
[i].iyear) ;
printf ("\nВведите число моточасов, наработанных двигателем: "); scanf
("%d",SastBoats [i] . lmotor_hours) ;
printf ("\nВведите стоимость судна: "); scanf ("%f",SastBoatsfi
.fsaleprice) ;
}
printf("\n\n");
for(i=0;i < linstock; i++) {
printf("Судно%s %d года выпуска с регистрационным номером#%s,\n"),
astBoats[i].szmodel, astBoats[i].iyear, astBoats[i].szserial);
printf("отработавшее%d моточасов.\n"),
astBoafcs [i].lmotor_hours); printf("%s\n",astBoats[i].szcomment);
printf ("ВСЕГО$%8.2f\n\n",astBoats[i].fsaleprice); }
return(0); }

```

В этой программе мы добавили в структуру stboat новую переменную — массив szcomment[80], содержащий строку с краткой характеристикой судна.

Использование функции _flushall() внутри первого цикла for необходимо, чтобы удалить из входного потока символ новой строки, оставшийся после выполнения предшествующей функции scanf() (либо той, что стоит перед циклом, либо той, что стоит в конце цикла). Вспомните, что функция gets() считывает все символы вплоть до \n. В паре с функцией scanf(), которая оставляет в потоке символ новой строки, следующая за ней функция gets() прочитает пустую строку. Чтобы предотвратить подобный нежелательный ход событий, вызывается функция flushall (), помимо прочего, очищающая буферы всех открытых входных потоков.

Вот как примерно будут выглядеть результаты работы программы:

Судно "Чайка" 1982 гола выпуска с регистрационным номером #XA1011,отработавшее 34187 моточасов.
 Великолепное судно для морских прогулок.
 ВСЕГО \$18132.00

Судно "Метеорит" 1988 года выпуска с регистрационным номером #КИ8096, отработавшее 27657 моточасов.
 Отлично выглядит, прекрасные ходовые качества.
 ВСЕГО \$20533.00

Судно "Спрут" 1993года выпуска с регистрационным номером #ДД7845, отработавшее 1000 моточасов. Надежно и экономично. ВСЕГО \$36234.00

Указатели на структуры

Следующая программа является вариантом предыдущего примера, в котором для получения доступа к отдельным членам структуры применяются указатель и оператор ->.

```
/*
 *      structpt.c
 *   Эта программа на языке C демонстрирует применение оператора ->.
 */
#include <stdio.h>
#define iSTRING15 15
#define lSTRING20 20
#define iNULL_CHAR 1
#define iMAX_BOATS 50
struct stboat {
char szmodel[iSTRING15 + iNULL_CHAR];
char szserial[iSTRING20 + iNULL_CHAR] ;
char szcomment[80];
int iyear;
long lmotor_hours;
float fsaleprice; } ;
int main(void)
int i, iinstock;
struct stboat astBoats[iMAX_BOATS], *pastBoats;
pastBoats= sastBoats[0];
printf("Информацию о скольких лодках следует ввести в базу данных? ");
scanf("%d",&iinstock);
for (i =0;i < linstock; i++) {
    _flushall(); /* очисткабуфер */
printf("\nВведите модель судна: "); gets (pastBoats->szmodel) ;
printf("\nВведите регистрационный номер судна: "); gets(pastBoats->szserial) ;
printf("\nВведите строку заметок о судне: ");
gets(pastBoats->szcomment);,
printf("\nВведите год изготовления судна: "); scanf("%d",&pastBoats->iyear) ;
printf("\nВведите число моточасов, наработанных двигателем: ");
scanf("%d",&pastBoats->lmotor_hours);
printf("\nВведите стоимость судна: "); scanf("%f",&pastBoats->fsaleprice) ;
pastBoats++;
}
pastBoats = SastBoats [0]; printf ("\n\n");
for (d = 0; i < linstock; i++) {
printf ("Судно%s %d года выпуска с регистрационным номером%s,\n",
pastBoats->szmodel, pastBoats->iyear, pastBoats->szserial) ;
```



```
printf ("отработавшее%d моторчасов . \n",pastBoats->lmotor_hours) ;
printf ("%s\n",pastBoats->szcomment) ;
printf ("ВСЕГО$%8.2f .\n\n",pastBoats->fsaleprice) ;
pastBoats++;
}
return(0);
}
```

К отдельному члену структуры, представленной указателем, можно обратиться и таким способом:

```
gets ( (*pastBoats) .szmodel) ;
```

Дополнительная пара скобок необходима, поскольку оператор прямого доступа к члену структуры (.) имеет больший приоритет, чем оператор раскрытия указателя (*). Использование оператора косвенного доступа (->) делает запись проще и аккуратнее:

```
gets (pastBoats->szmodel) ;
```

Передача массива структур в качестве аргумента функции

Ранее в этой главе уже говорилось о том, что структуры передаются в функции по значению. Если же вместо самой структуры передавать указатель на нее, то это может существенно ускорить выполнение программы.

```
/*
 * structaf.c
 * В этой программе на языке C в функцию передается указатель на
 * структуру.
 */
#include <stdio.h>
#define i STRING 15 15
#define lSTRING20 20
#define iNULL_CHAR 1
#define iMAX_BOATS 50
int linstock;
struct stboat {
char szmodel[iSTRING15 + iNULL_CHAR] ;
char szserial[iSTRING20 + iNULL_CHAR] ;
char szcomment [80];
int iyear;
long lmotor_hours;
float fsaleprice;
};
void vprint_data(struct stboat *stused_boatptr);
int main(void) {
int i;
struct stboat astBoats[iMAX_BOATS], *pastBoats;
pastBoats= SastBoats[0];
printf("Информацию о скольких лодках следует ввести в базу данных? ");
scanf("%d",&iinstock);
for(i= 0; i < iinstock; i++) {
_flushall(); /* очищаетбуфер */ printf("\nВведитемодельсудна: ");
gets(pastBoats->szmodel);
printf("\nВведите регистрационный номер судна: "); gets(pastBoats-
>szserial);
printf("\nВведите строку заметок о судне: "); gets (pastBoats-
>szcoiranent) ;
printf("\nВведите год изготовления судна: ");
scanf("%d",&pastBoats->iyear);
```

```

printf("\nВведите число моточасов, наработанных двигателем: ");
scanf("%d",&pastBoats->lmotor_hours);
printf("\nВведите стоимость судна: ");
scanf("%f",spastBoats->fsaleprice);
pastBoats++;
)
pastBoats = SastBoats[0]; vprint_data(pastBoats);
return(0);      }
void vprint_data (struct stboat *stused_boatptr) {
int i;
printf ("\n\n");
for(i=0;i < linstock; i++){
printf ("Судно%s %d года выпуска с регистрационным номером#%s,\n",
stused_boatptr->szmodel, stused_boatptr->iyear, stused_boatptr->szserial) ;
printf("отработавшее %dмоточасов. \n",stused_boatptr->lmotor_hours) ,
printf ("%s\n",stused_boatptr->szcomment) ;
printf ("БЦЕГО$%8.2f\n\n",stused_boatptr->fsaleprice) ;
stused_boatptr++;
}
}
}

```

Структуры в С++

Ниже показана программа на С++, являющаяся аналогом рассмотренного в предыдущем параграфе примера на языке С. В обоих случаях для работы со структурой применяется одинаковый синтаксис.

```

//
//  struct. cpp
//  В этой программе на языке С++ в функцию передается
//  указатель на структуру.
//
#include <iostream.h>
#define lSTRING15 15
#define lSTRING20 20
#define iNULL_CHAR 1
#define iMAX_BOATS 50
int linstock;
struct stboat {
char szmodel[iSTRING15 + iNULL_CHAR];
char szserial[iSTRING20 + iNULL_CHAR];
char szcomment[80];
int iyear;
long lmotor_hours;
float fsaleprice; };
void vprint_data (stboat *stused_boatptr) ;
int main (void) {
int i;
char newline;
stboat astBoats [iMAX_BOATS] , *pastBoats;
pastBoats = SastBoats [0];
cout<< "Информацию о скольких лодках следует ввести в базу данных?
cin >> linstock;
ford = 0; i < linstock; i++) {
cout << "\nВведите модель судна: "; cin >> pastBoats->szmodel;

```

```

cout<< "\nВведите регистрационный номер судна: "; cin>> pastBoats-
>szserial;
cout<< "\nВведите год изготовления судна: "; cin>> pastBoats-
>iyear;
cout<< "\nВведите число моточасов, наработанных двигателем: "; cin >>
pastBoats->lmotor_hqurs;
cout<< "\nВведите стоимость судна: ";
cin >> pastBoats->fsaleprice;
cout<< "\nВведите строку заметок о судне: ";
cin.get (newline) ; // пропуск символа новой строки
cin.get (pastBoats->szcomment, 80, '.');
cin.get (newline) ; // пропуск символа новой строки
pastBoats++;
}
pastBoats = sastBoats[0]; vprint_data(pastBoats) ;
return(0); }
void vprint_data(stboat *stused_boatptr) {
int i ;
cout << "\n\n";
for(i=0;i < linstock; i++) {
cout << "Судно " << stused_boatptr->szmodel << " " << stused_boatptr-
>iyear << " года выпуска с регистрационным номером " <<
stused_boatptr->szserial << ",\n" << "отработавшее " <<
stused_boatptr->lmotor_hours << " моточасов.\n";
cout << stused_boatptr->szcomment << ".\n";
cout << "ВСЕГО $" << stused_boatptr->fsaleprice << "\n\n";
stused_boatptr++; .} }

```

Считывание строки заметок о судне реализуется несколько иначе, чем в случае других членов структуры. Вспомните, что оператор >> читает строку символов до первого пробела. Поэтому он не подходит для считывания строки комментариев, состоящей из слов, разделенных пробелами. Для получения всей строки применяется следующая конструкция:

```

cout<< "\nВведите строку заметок о судне: ";
cin.get (newline) ; // пропуск символа новой строки
cin.get (pastBoats->szcomment, 80, ' . ' ) ;
cin.get (newline) ; // пропуск символа новой строки

```

Первая функция `cin.get(newline)` выполняет то же действие, что и функция `_£ lushall()` в одной из предыдущих программ на языке С. В системах, где данные, вводимые с клавиатуры, буферизируются, часто бывает необходимо удалять из буфера символ новой строки. Существует несколько способов сделать это, но мы показали наиболее лаконичный. Прочитанный символ сохраняется в переменной `newline`, которая больше нигде в программе не используется.

В третьей строке функция `cin . get ()` считывает максимум 80 символов, причем признаком конца строки задана точка. Таким образом, функция завершается, когда длина строки превышает 79 символов (80-я позиция резервируется для символа `\0`) или если была введена точка. Сам символ точки не будет прочитан. Поэтому, чтобы строка имела заверченный вид, при выводе на печать точку нужно добавить.

Вложенные структуры

Структуры могут быть вложены друг в друга. То есть некоторая структура может содержать переменные, которые, в свою очередь, являются структурами. Предположим, описана такая структура:

```

structstowner { /* информация о владельце судна */
charcname[50];/* имя и фамилия */
intiage; /* возраст */
int isailing_experience; /* опыт мореплавания */ };

```

В следующем примере структура stowner становится вложенной в рассмотренную ранее структуру stboat:

```
struct stboat {
char szmodel[iSTRING15 + iNULL_CHAR];
char szserial[iSTRING20 + iNULL_CHAR];
char szcomment[80]; . struct stowner stowner_record;
int iyear;
long lmotor_hours;
float fsaleprice; } astBoats[iMAX_BOATS];
```

Чтобы, например, вывести на экран возраст владельца судна, необходимо воспользоваться таким выражением:

```
printf("%d\n", astBoats[0].stowner_record.iage);
//
```

Битовые поля

В C/C++ существует весьма удобная возможность — "запаковать" набор битовых флагов в единую структуру, называемую битовым полем и расположенную в системном слове. Предположим, например, что в программе, осуществляющей чтение данных с клавиатуры, используется группа флагов, отображающих состояние специальных клавиш, таких как [CapsLock], [NumLock] и т.п. Эти флаги можно организовать так:

```
struct stkeybits {
unsigned char
rshift      1,          /* нажата правая клавиша [Shift] */
lshift      1,          /* нажата левая клавиша [Shift] */
Ctrl        1,          /* нажата клавиша [Ctrl] */
alt         1,          /* нажата клавиша [Alt] */
scroll      1,          /* включен режим [Scroll Lock] */
numlock     1,          /* включен режим [NumLock] */
capslock    1,          /* включен режим [Caps Lock] */
insert      1,          /* включен режим [Insert] */
};
```

После двоеточия указано число битов, занимаемых флагом. Их может быть более одного. Можно не указывать имя флага, а лишь двоеточие и ширину битовой ячейки. Такие безымянные ячейки предназначены для пропуска нужного количества битов в пределах поля. Если указана ширина 0, следующий флаг будет выровнен по границе типа данных. Разрешается создавать битовые поля только целого типа (char, short, int, long), причем желательно явно указывать модификатор unsigned, чтобы поле не интерпретировалось как знаковое число.

При работе с битовыми полями применяется синтаксис структур. Например, чтобы установить флаг capslock, следует воспользоваться таким выражением:

```
struct stkeybits flags; flags.capslock = 1;
```

К битовым полям нельзя применять оператор взятия адреса (&).

Объединения

Объединением называется специального рода переменная, которая в разные моменты времени может содержать данные разного типа. Одно и то же объединение может в одних операциях участвовать как переменная типа int, а в других — как переменная типа float или double. Для всех членов такой переменной резервируется единая область памяти, которую они используют совместно. Размерность объединения определяется размерностью самого "широкого" из указанных в нем типов данных.

Синтаксис

Объединение создается с помощью ключевого слова union:

```
union ter {
```

```

тип1 имя1;
тип2 имя2;
тип3 имя3;
.
.
.
тип-п имя_п; };

```

Обратите внимание на схожесть описания структуры и объединения:

```

union unmany_types {
char c;
int ivalue;
float fvalue;
double dvalue; } my_union;

```

Необязательный тег играет ту же роль, что и тег структуры: он становится именем нового типа данных. Также по аналогии со структурами к члену объединения можно обратиться с помощью оператора точки (.):

имя_переменной.член_объединения

Создание простейшего объединения

Проиллюстрируем работу с объединением на следующем, довольно простом примере:

```

//
// union.cpp
// Эта программа на языке C++ демонстрирует, как работать с
// объединением.
//
#include <iostream.h>
union unmany_types {
char c;
int ivalue;
float fvalue;
double dvalue; } my_union;
int main (void) {
// корректные операции ввода/вывода
my_union.c = 'b';
cout << my_union.c << endl;
my_union. ivalue = 1990;
cout << my_union. ivalue << endl;
my_union. fvalue =19.90; , cout << my_union. fvalue << endl;
my_union. dvalue = 987654 . 32E+13; cout << my_union. dvalue << endl;
// некорректные операции ввода/вывода
cout << my_union.c << endl; cout << my_union. ivalue << endl; cout <<
my_union. fvalue << endl; cout << my_union. dvalue << endl;
// вычисление размера объединения
cout<< "Размер объединения составляет " << sizeof (unmany_types) << "
байт.";
return(0);
}

```

Первая часть программы выполняется без ошибок, поскольку данные разных типов присваиваются объединению не одновременно, а последовательно. Во второй части правильно отобразится лишь значение типа double, поскольку оно было занесено последним. Вот что будет выведено на экран:

b

1990
19.9
9.87654e+018
Ш
-154494568
-2.05461e+018
9.87654e+018

Размер объединения составляет: 8 байт.

Ключевое слово typedef

С помощью ключевого слова typedef можно создавать новые типы данных на основании уже существующих. Как правило, это необходимо для упрощения текста программы. Например, выражение

```
typedef struct stboat* stboatptr;
```

делает имя stboatptr синонимом конструкции stboat*. Теперь создать указатель на структуру stboat можно будет следующим образом:

```
stboatptr my_boat;
```

Перечисления

Перечисления, создаваемые с помощью ключевого слова enum, также служат цели сделать программный код более удобочитаемым. Они представляют собой множества именованных целочисленных констант. Для описания перечисления используется следующий синтаксис:

```
enum необязательный_тег (константа1 [= значение!],  
.  
.  
.  
,  
константа-п [= значение-п]);
```

Как вы уже догадались, необязательный тег служит тем же целям, что и теги структур и объединений. Если тег не указан, то сразу после закрывающей фигурной скобки должен находиться список имен переменных. При наличии тега переменные данного перечисления можно будет создавать в любом месте программы (в C++ в этом случае нет необходимости повторно указывать ключевое слово enum).

Константы, входящие в перечисление, имеют тип int. По умолчанию первая константа равна 0, следующая — 1, потом — 2 и т.д. в арифметической прогрессии. Таким образом, значения констант можно не указывать: они будут вычислены автоматически путем формирования прогрессии с шагом 1 от последнего специфицированного значения. В то же время, для каждой константы после знака равенства можно указывать собственное значение.

Все константы одного перечисления должны иметь разные имена, но их значения могут совпадать. Эти имена должны также отличаться от имен обычных переменных в той же области видимости.

Например, в следующем фрагменте программы можно организовать цикл от 0 до 4, а можно — от понедельника до пятницы:

```
enum weekdays { /* будние дни */  
Monday, /* понедельник */  
Tuesday, /* вторник */  
Wednesday, /* среда */  
Thursday, /* четверг */  
.Friday } /* пятница */  
/* описание перечисления в языке C */  
enum weekdaysewToday;  
/* описание перечисления в C++ */  
weekdaysewToday;  
/* задание цикла for без использования перечисления */
```

```

for(i=0;i<= 4; i++)
.
.
.
/* и с использованием перечисления */
for(ewToday = Monday; ewToday <= Friday; ewToday++)

```

Компилятор не видит разницы между типами данных `int` и `enum`. Поэтому переменным типа перечисления в программе могут присваиваться целочисленные значения. Но в языке C++, если такое присваивание не сопровождается явным приведением типа, компилятор выдаст предупреждение:

```

/* допустимо в C, но не в C++ */ ewtoday = 1;
/* устранение ошибки в C++ */ ewToday = (eweekdays) 1;

```

Перечисления часто используются в тех случаях, когда данные можно представить в виде пронумерованного списка, например содержащего названия месяцев года или дней недели. В следующем примере создается перечисление `emonths`, включающее названия месяцев.

```

/*
 *      enum. c
 *  Эта программа на языке C демонстрирует работу с перечислением.
 */
#include <stdio.h>
enum emonths {
January = 1 ,
February,
March,
April,
May,
June,
July,
August,
September,
October,
November,
December
} months;
int main(void)
{
int ipresent_month;
int idiff;
{
printf("Введите номер текущего месяца (от 1 до 12): ") ;
scanf("%d",&ipresent_month);
months = December;
idiff = (int)months - ipresent_month;
printf("\nДо конца года осталось %d месяцев (ев)An", idiff);
return (0);
}
}

```

В данной программе перечисление в действительности представляет собой ряд целых чисел от 1 до 12. Например, значение переменной `months`, после того как ей была присвоена константа `December`, стало равным 12. Поскольку названию каждого месяца соответствует определенное числовое значение, то элементы перечисления могут участвовать в арифметических операциях.

В результате выполнения программы на экран будет выведена примерно следующая информация:

Введите номер текущего месяца (от 1 до 12): 4
До конца года осталось 8 месяца(ев).

Глава 13. Основы ООП

- Все новое — это хорошо забытое старое
- C++ и ООП
- Основная терминология
 - Инкапсуляция
 - Иерархия классов
- Первое знакомство с классом
 - Структура в роли примитивного класса
 - Синтаксис описания классов
 - Простейший класс

В данной главе будут рассмотрены основные термины и понятия объектно-ориентированного программирования (ООП). В чем, вообще говоря, особенность объектно-ориентированных языков, таких как C++? Одним словом на этот вопрос можно ответить так: компактность! Методы ООП позволяют лучше организовать и эффективнее использовать все те функции, с которыми вы познакомились в предыдущих главах книги. Если вы новичок в ООП, то основная проблема для вас будет состоять не в освоении каких-то новых ключевых слов, а в изучении принципиально иных подходов к построению и структурированию программ. Единственное серьезное препятствие на вашем пути — это новая терминология. Многие средства процедурного программирования, которые вы изучали до сих пор, в ООП носят другие названия. Например, вы узнаете о превращении знакомого вам понятия структуры в понятие класса — нового типа данных, предназначенного для создания особого рода динамических переменных — объектов.

Все новое — это хорошо забытое старое

Специалисты по маркетингу хорошо знают, что товар продается лучше, если где-нибудь на упаковке вставить слово "новинка". Поэтому не стоит полагать, что ООП является такой уж новой концепцией. Скотт Гудери (Scott Guthery) вообще утверждал в 1989, что ООП берет свое начало с появления подпрограмм в 40-х годах (см. статью "Является ли новое платье короля объектно-ориентированным?" ("Are the Emperor's New Clothes Object Oriented?"), Dr. Dobb's Journal, декабрь, 1989 г.).

В этой же статье утверждалось, что объекты, как основа ООП, появились еще в языке FORTRAN II.

Если это верно, то почему же об ООП все говорят как о новинке последнего десятилетия нашего века? Все дело в компьютерной технике. Совершенно верно, концепция ООП могла быть сформулирована еще в 40-х годах, но реализовать эту идею в полной мере удалось только в последние десять лет.

Первые программы на языке BASIC часто представляли собой многостраничные листинги, в которых не прослеживалось четкой структуры. Огромные программные блоки связывались друг с другом посредством одно- или двухбуквенных меток, по которым осуществлялись переходы с помощью многочисленных команд `goto`. Ночным кошмаром были анализ и отладка таких программ, не говоря уже об их модификации. Сопровождение их тоже было нелегким делом.

В 60-х годах была реализована методика структурного программирования, подразумевавшая использование переменных с осмысленными именами, существование глобальных и локальных переменных и процедурное проектирование приложений по принципу "сверху вниз". Благодаря внедрению в жизнь этих концепций тексты программ стали понятнее и читабельнее, в результате чего упростился процесс отладки программ. Стало проще сопровождать программы, так как появилась возможность контролировать выполнение не отдельных команд, а целых процедур. Примерами языков, ориентированных на подобный процедурный подход, могут быть Ada, C и Pascal.

Бьярна Страуструпа (Bjarne Stroustrup) можно считать отцом ООП в том виде, в каком эта концепция представлена в языке C++, разработанном им в начале 80-х в компании BellLabs. С точки зрения Джеффа Дантемманна (JeffDuntemann), "ООП представляет собой то же структурное программированное, но с еще большей степенью структуризации. Это вторая производная от исходной теории построения программ." (См. статью "Лавирование среди айсбергов" ("DodgingSteamships"), Dr. Dobb'sJournal, июль, 1989 г.) Действительно, как вы убедитесь, ООП в C++ в значительной степени основано на концепциях и средствах структурного программирования языка C. И хотя язык C++ ориентирован на работу с объектами, при желании на нем можно писать как традиционные процедурные приложения, так и неструктурированные программы. Выбор за вами.

C++ и ООП

Все предыдущие главы книги были посвящены изучению методик традиционного структурного программирования. Структура процедурных программ в общем такова: имеется функция `main()`, а из нее вызываются другие функции программы. Тело функции `main()`, как правило, невелико. Ее роль обычно сводится к распределению задач между остальными функциями, которые описывают действия, выполняемые над данными.

Основным недостатком такого подхода являются проблемы, возникающие при сопровождении программ. Например, если в программу нужно добавить какой-нибудь программный блок, то чаще всего ее приходится полностью перекомпилировать. Это не только отнимает много времени, но и чревато ошибками.

В основу ООП заложены совершенно иные принципы и другая стратегия написания программ, что часто становится камнем преткновения для многих разработчиков, привыкших к традиционному программированию. Теперь программы представляют собой алгоритмы, описывающие взаимодействие групп взаимосвязанных объектов. В C++ объекты создаются путем описания класса как нового типа данных. Класс содержит ряд констант и переменных (данных), а также операций (функций-членов, или методов), выполняемых над ними. Чтобы произвести какое-либо действие над данными объекта, ему необходимо, выражаясь в новых терминах, послать сообщение, т.е. вызвать один из его методов. Подразумевается, что к данным, хранящимся в объекте, нельзя получить доступ иначе, как путем вызова того или иного метода. Таким образом, программный код и оперируемые данные объединяются в единой "виртуальной" структуре.

ООП дает программистам три важных преимущества. Первое состоит в упрощении программного кода и улучшении его структуризации. Программы стали проще для чтения и понимания. Код описания классов, как правило, отделен от кода основной части программы, благодаря чему над ними можно работать по отдельности. Второе преимущество заключается в том, что модернизация программ (добавление и удаление программных блоков) становится несравнимо более простой задачей. Чаще всего она сводится к добавлению нового класса, который наследует все свойства одного из имеющихся классов и содержит требуемые дополнительные методы. Третье преимущество состоит в том, что одни и те же классы можно много раз использовать в разных программах. Удачно созданный класс можно сохранить отдельно в библиотечном файле, и его добавление в программу, как правило, не требует внесения серьезных изменений в ее текст.

При изучении предыдущих глав книги у вас могло сложиться впечатление, что преобразование текстов программ с языка C на C++ и обратно не составляет труда. Достаточно только поменять некоторые ключевые слова, например `printf()` на `cout`. В действительности различия между этими языками гораздо более глубоки и лежат как раз в области ООП. Например, язык C не поддерживает классы. Поэтому преобразовать процедурную программу в объектно-ориентированную намного труднее, чем может показаться. Зачастую проще выбросить старую программу и написать новую с нуля. В отсутствии преемственности между двумя подходами заключается определенный недостаток ООП.

Идеи ООП в той или иной форме проявились во всех современных языках. Почему же сейчас мы говорим о C++ как об основном средстве создания объектно-ориентированных программ? Все дело в том, что, в отличие от многих других языков, классы в нем являются особым типом данных, развившимся из типа `struct` языка C. Кроме того, в C++ добавлены некоторые дополнительные средства создания объектов, которых нет в других языках программирования. В данном контексте в числе преимуществ языка C++ следует указать строгий контроль за типами данных, возможность перегрузки операторов и меньшую зависимость от препроцессора. Хотя объектно-ориентированные программы можно писать и на других языках, только в C++ вы можете в полной мере реализовать всю мощь данной концепции, так как это язык, который не адаптировался к новому веянию, а специально создавался для ООП.

Основная терминология

Поскольку идея ООП была достаточно глобальной и не связанной с конкретным языком программирования, потребовалось определить термины, общие для всех языков программирования, поддерживающих данную концепцию. То есть те термины, с которыми мы сейчас познакомимся, являются общими как для Pascal, так и для C++ и прочих языков. Впрочем, наряду с общей терминологией существуют и термины, специфичные для отдельных языков.

ООП представляет собой уникальный подход к написанию программ, когда задачи и решения формулируются путем описания схемы взаимодействия связанных объектов. С помощью объектов можно смоделировать реальный процесс, а затем проанализировать его программными средствами. Каждый объект является определенной структурой данных, поэтому переменную типа `struct` в C/C++ можно рассматривать как простейший объект. Взаимодействие между объектами осуществляется путем отправки им сообщений, как уже говорилось ранее. Сообщение по сути своей — это то же, что и вызов функции в процедурном программировании. Когда объект получает сообщение, выполняется хранящийся в нем метод, который возвращает результат вычислений в программу. Методы также называют функциями-членами, и они напоминают обычные функции за тем исключением, что являются неразрывной частью объекта и не могут быть вызваны отдельно от него.

Класс языка C++ является расширенным вариантом структуры и служит для создания объектов. Он содержит функции-члены (методы), связанные некоторыми общими атрибутами. Объект — это экземпляр класса, доступный для выполнения над ним требуемых действий.

Инкапсуляция

Под инкапсуляцией понимается хранение в одной структуре как данных (констант и переменных), так и функций их обработки (методов). Доступ к отдельным частям класса регулируется с помощью специальных ключевых слов `public`(открытая часть), `private`(закрытая часть) и `protected`(защищенная часть). Методы, расположенные в открытой части, формируют интерфейс класса и могут свободно вызываться всеми пользователями класса. Считается, что переменные-члены класса не должны находиться в секции `public`, но могут существовать интерфейсные методы, позволяющие читать и модифицировать значение каждой переменной. Доступ к закрытой секции класса возможен только из его собственных методов, а к защищенной — также из методов классов-потомков.

Иерархия классов

В C++ класс выступает в качестве шаблона, на основе которого создаются объекты. От любого класса можно породить один или несколько подклассов, в результате чего сформируется иерархия классов. Родительские классы обычно содержат методы более общего характера, тогда как решение специфических задач поручается производным классам.

Наследование

Под наследованием понимают передачу данных и методов от родительских классов производным. Если класс наследует свои атрибуты от одного родительского класса, то такое наследование называется одиночным. Если же атрибуты наследуются от нескольких классов, то говорится о множественном наследовании. Наследование является важнейшей концепцией программирования, поскольку позволяет многократно использовать одни и те же классы, не переписывая их, а лишь подстраивая для решения конкретных задач и расширяя их возможности.

Полиморфизм и виртуальные функции

Другая важная концепция ООП, связанная с иерархией классов, заключается в возможности послать одинаковое сообщение сразу нескольким классам в иерархии, предоставив им право выбрать, кому из них надлежит его обработать. Это называется полиморфизмом. Методы, содержащиеся в разных классах одной иерархии, но имеющие общее имя и объявленные с ключевым словом `virtual`, называются виртуальными.

Благодаря полиморфизму можно делать в программе запрос к объекту, даже если тип его не известен заранее. В C++ эта возможность реализуется за счет подсистемы позднего связывания, под которым понимается динамическое определение адресов функций во время выполнения программы в противоположность традиционному статическому (раннему)

связыванию, осуществляемому во время компиляции. В процессе связывания имена функций заменяются их адресами.

При вызове виртуальных функций используется специальная таблица адресов функций, называемая виртуальной таблицей. Она инициализируется в ходе выполнения программы в момент создания объекта конструктором класса. Роль конструктора заключается в том, чтобы связать виртуальную функцию с правильной таблицей адресов. Во время компиляции адрес виртуальной функции не известен, но известна ячейка виртуальной таблицы, где этот адрес будет записан во время выполнения программы.

Первое знакомство с классом

В этом параграфе мы на простых примерах детально разберем, чем же все-таки класс отличается от структуры.

Структура в роли примитивного класса

Структуры в C++ можно рассматривать как примитивные классы, поскольку они могут содержать не только данные, но и функции. Рассмотрим следующую программу:

```
//
//  sqroot.cpp
//  В этой программе на языке C++ создается структура,
//  содержащая, кроме данных, также функции.
//
#include <iostream.h>
# include <math.h>
struct math_operations { double data_value;
void  set_value (double value) { data_value = value; }
double get_square (void)
{ return (data_value * data_value) ; }
double get_square_root (void) {
return (sqrt (data_value) ); } } math;
main ()
{
// записываем в структуру число 35.63
math.set_value (35.63);
cout << "Квадрат числа равен " << math.get_square () << endl;
cout << "Корень числа равен " << math.get_square_root () << endl;
return (0); }
```

В первую очередь обратите внимание на то, что помимо переменной структура содержит также несколько функций. Это первый случай, когда внутри структуры нам встретились описания функций. Такая возможность существует только в C++. Функции-члены структуры могут выполнять операции над переменными этой же структуры. Неявно подразумевается, что все члены структур являются открытыми, как если бы они были помещены в секцию public.

При выполнении программы на экране отобразится следующая информация:

```
Квадрат числа равен 1269.5 Корень числа равен  5.96909
```

В структуре math_operation объявлена единственная переменная-член data_value и три функции, описание которых дано тут же внутри структуры. Первая функция отвечает за инициализацию переменной data_value, другие возвращают соответственно квадрат и квадратный корень числа, хранимого в переменной. Обратите внимание, что последние две функции не принимают никаких значений, поскольку переменная data_value доступна для них как член структуры.

В следующей программе вновь создается структура, содержащая функции, но на этот раз они описаны вне структуры.

```
//
//  trigon.cpp
```

```
// В этой программе на языке C++ создается структура,
// содержащая тригонометрические функции.
//
#include <iostream.h>
#include <math.h>
const double.DEG_TO_RAD = 0.0174532925;
struct degree {
double data value;
void set_value (double angle);
double get_sine (void) ;
double get_cosine (void) ;
double get_tangent (void) ;
double get_cotangent (void) ;
double get_secant (void) ;
double get_cosecant (void) ; ) deg;
void degree: :set_value (double angle) {
data_value = angle;
}
double degree: :get_sine (void) {
return (sin(DEG_TO_RAD * data_value) )
}
double degree::get_cosine(void) {
return(cos(DEG_TO_RAD * data_value)); }
double degree : : get_tangent (void) {
return ( tan (DEG_TO_RAD * data_value) ) ; }
double degree: :get_secant (void). {
return (1.0/ sin(DEG_TO_RAD * data_value) ) ; }
double degree: :get_cosecant (void) (
return (1.0/ cos(DEG_TO_RAD * data_value) ) ; )
double degree : : get_cotangent (void) <
return (1.0/ tan(DEG_TO_RAD * data_value) ) ; }
main() {
// устанавливаем значение угла равным 25 градусов deg . set_value (
25.0 ) ;
cout << "Синус угла равен " << deg.get_sine () << endl;
cout << "Косинус угла равен " << deg.get_cosine () << endl;
cout << "Тангенс угла равен " << deg.get_tangent () << endl;
cout << "Секанс угла равен " << deg.get_secant () << endl;
cout << "Косеканс угла равен " << deg.get_cosecant () << endl;
cout << "Котангенс угла равен " << deg.get_cotangent () << endl;
return (0 ) ; }
```

В этой программе структура содержит прототипы семи функций, но сами они описаны отдельно. Тригонометрические функции вычисляют синус, косинус, тангенс, котангенс, секанс и косеканс угла, значение которого в градусах передается в функцию `set_value()`. Здесь следует обратить внимание на синтаксис заголовка функции, например:

```
void degree::set_value(double angle)
```

Имя функции состоит из имени структуры, за которым расположен оператор `::`.

При вызове функции используется оператор точка (`.`), как и при доступе к переменным-членам структуры. Если бы структура была представлена указателем, доступ к функциям необходимо было бы осуществлять с помощью оператора `->`.

Синтаксис описания классов

Синтаксис описания класса подобен синтаксису описания структуры. Оно начинается с ключевого слова `class`, за которым следует имя класса, становящееся именем нового типа данных. В простейшем случае описание класса можно представить так:

```
class имя {
тип1 переменная1 тип2 переменная2 тип3 переменная3
public: метод1; метод2; метод3;
};
```

По умолчанию все члены класса считаются закрытыми и доступ к ним могут получить только функции-члены этого же класса. Это именно то, что в C++ отличает классы от структур: все члены структур по умолчанию являются открытыми. Если необходимо изменить тип доступа к членам класса, перед ними следует указать один из спецификаторов доступа: `public`, `protected` или `private`.

Ниже показано описание класса, который будет использоваться в следующем примере программы:

```
class degree (
double data_value; public:
void set_value(double);
double get_sine(void);
double get_cosine(void);
double get_tangent(void);
double get_secant(void);
double get_cosecant(void);
double get_cotangent(void);
} deg;
```

Здесь создается новый тип данных `degree`. Закрытая переменная-член `data_value` доступна только функциям-членам класса. Одновременно создается и объект данного класса — переменная `deg`. Не показалось ли вам описание класса знакомым? Это, по сути, та же самая структура, которую мы рассматривали в предыдущем примере, лишь ключевое слово `class` превращает структуру в настоящий класс.

Простейший класс

Рассмотрим следующий пример программы:

```
//
// class.cpp
// В этой программе на языке C++ создается простейший
// класс, имеющий открытые и закрытые члены.
//
#include <iostream.h>
#include <math.h>
const double DEG_TO_RAD = 0.0174532925;
class degree (
double data_value;
public:
void set_value(double angle);
double get_sine(void) ;
double get_cosine(void) ;
double get_tangent(void) ;
double get_secant(void) ;
double get_cosecant(void) ;
double get_cotangent(void); ) deg;
void degree::set_value(double angle) <
data_value = angle;
I
```

```

double degree::get_sine(void)
{
return(sin(DEG_TO_RAD * data_value));
}
double degree::get_cosine(void)
return(cos(DEG_TO_RAD * data_value));
double degree::get_tangent(void)
return(tan(DEG_TO_RAD * data_value)); }
double degree::get_secant(void)
return(1.0 / sin(DEG_TO_RAD * data_value)); )
double degree::get_cosecant(void)
return (1.0/ cos(DEG_TO_RAD * data_value));
}
double degree::get_cotangent(void)
return(1.0/ tan(DEG_TO_RAD * data_value));
main()
// устанавливаем значение угла равным 25.0градусов
deg.set_value(25.0);
cout << "Синус угла равен, " << deg.get_sine() << endl;
cout << "Косинус угла равен " << deg.get_cosine() << endl;
cout << "Тангенс угла равен " << deg.get_tangent() << endl;
cout << "Секанс угла равен " << deg.get_secant() << endl;
cout << "Косеканс угла равен " << deg.get_cosecant() << endl;
cout << "Котангенс угла равен " << deg.get_cotangent() << endl;
return(0); }

```

Как видите, тело программы сохранилось прежним. Просто описание структуры было преобразовано в описание настоящего класса C++ с открытой и закрытой секциями.

Программа выведет на экран следующую информацию:

```

Синус угла равен      0.422618
Косинус угла равен   0.906308
Тангенс угла равен    0.466308
Секанс угла равен     2.3662
Косеканс угла равен  1.10338
Котангенс угла равен  2.14451

```

Глава 14. Классы

- Особенности классов
 - Конструкторы и деструкторы
 - Перегрузка функций-членов класса
 - Дружественные функции
 - Указатель this
- Перегрузка операторов
- Производные классы

Как было показано в предыдущей главе, примитивный класс можно создать с помощью ключевого слова `struct`, хотя логичнее все же воспользоваться ключевым словом `class`. В любом случае получается структура, состоящая из переменных и функций-членов, которые выполняют действия над переменными этой структуры. В данной главе приводятся дополнительные сведения о классах в C++: рассказывается об использовании конструкторов и деструкторов, перегрузке функций-членов и операторов, "дружественных" функциях, наследовании классов и др.

Особенности классов

Синтаксис создания простейшего класса был рассмотрен в предыдущей главе. Но, конечно же, возможности классов значительно шире, и в следующих параграфах мы подробнее познакомимся с данной темой.

Конструкторы и деструкторы

Конструктор представляет собой особого рода функцию-член класса, предназначенную в первую очередь для инициализации переменных класса и резервирования памяти. Имя конструктора совпадает с именем класса, которому он принадлежит. Конструкторы могут принимать аргументы и быть перегруженными. При создании объекта класса нужный конструктор вызывается автоматически. Если при описании класса конструктор не был задан, то компилятор сгенерирует для класса стандартный конструктор.

Деструктором называется еще одна специальная функция-член класса, которая служит в основном для освобождения динамической памяти, занимаемой удаляемым объектом. Деструктор, как и конструктор, носит имя класса, которое в качестве префикса содержит знак тильды (~). Деструктор вызывается автоматически, когда в программе встречается оператор `delete` с указателем на объект класса или когда объект выходит за пределы своей области видимости. В отличие от конструкторов, деструкторы не принимают никаких аргументов и не могут быть перегружены. Если деструктор не задан явно, компилятор предоставит классу стандартный деструктор.

В следующей программе продемонстрировано создание простейших конструктора и деструктора. В данном случае они лишь сигнализируют соответственно о создании и удалении объекта класса `coins`. Обратите внимание на то, что обе функции вызываются автоматически: конструктор — в строке

```
coins cash_in_cents;  
а деструктор — после завершения функции main().  
//  
// coins.cpp  
// В этой программе на языке C++ демонстрируется создание  
конструктора  
// и деструктора. Данная программа вычисляет, каким набором монет  
// достоинством 25,10,5 и 1 копейка можно представить заданную  
денежную  
// сумму.  
//
```



```

#include <iostream.h>
const int QUARTER = 25;
const int DIME = 10;
const int NICKEL = 5;
class coins {    int number;
public: coins ()
{ cout << "Начало вычислений.\n"; } // конструктор
~coins ()
{ cout << "\nКонец вычислений."; } // деструктор
void get_cents(int); int quarter_conversion(void) ;
int dime_conversion(int); int nickel_conversion(int);
};

void coins::get_cents(int cents) {
number = cents;          cout << number << " копеек состоит из таких
монет:" << endl; }

int coins::quarter_conversion()
{
cout << number / QUARTER << " — достоинством 25, return (number %
QUARTER) ;
}

int coins::dime_conversion(int d) {
cout<< d/ DIME<< " — достоинством 10,
return(d % DIME);
}

int coins::nickel_conversion(int n) {
cout<< n/ NICKEL<< " — достоинством 5 и
return(n % NICKEL); }

main ()
int c, d, n, p;
cout<< "Задайте денежную сумму в копейках: "; cin>> c;
// создание объекта cash_in_cents класса coins
coins cash_in_cents;
cash_in_cents.get_cents(c); d = cash_in_cents.quarter_conversion(); n
= cash_in_cents .dime_conversion (d); p =
cash_in_cents.nickel_conversion(n); cout << p << " — достоинством 1.";
return(0); }

```

Вот как будут выглядеть результаты работы программы:

Задайте денежную сумму в копейках: 159

Начало вычислений.

159 копеек состоит из таких монет:

6 — достоинством 25, 0 — достоинством 10,

1 — достоинством 5 и 4 — достоинством 1.

Конец вычислений. В функции `get_cents()` заданная пользователем денежная сумма записывается в переменную `number` класса `coins`. Функция `quarter_conversion()` делит значение `number` на 25 (константа `QUARTER`), вычисляя тем самым, сколько монет достоинством 25 копеек "умещается" в заданной сумме. В программу возвращается остаток от деления, который далее, в функции `dime_conversion()`, делится уже на 10 (константа `dime`). Снова возвращается остаток, на этот раз он передается в функцию `nickel_conversion()`, где делится на 5 (константа `NICKEL`). Последний остаток определяет количество однокопеечных монет.

Инициализация переменных-членов с помощью конструктора

Основное практическое применение конструктора состоит в инициализации закрытых переменных-членов класса. В предыдущем примере переменная `number` инициализировалась с помощью специально предназначенной для этого функции-члена `get_cents()`, записывавшей

в переменную значение, полученное от пользователя. Помимо этого, в конструкторе можно присвоить данной переменной значение по умолчанию, например:

```
coins() {  
    cout<< "Начало вычислений.\n";  
    number= 431; } // конструктор
```

Тут будет уместно упомянуть о том, что переменные-члены классов, в отличие от обычных переменных, не могут быть инициализированы напрямую с помощью оператора присваивания:

```
classcoins {  
    intnumber= 431; // недопустимая инициализация  
}
```

Вот почему все подобные операции приходится помещать в конструктор.

Резервирование и освобождение памяти

Другой важной задачей, решаемой с помощью конструктора, является выделение динамической памяти. В следующем примере конструктор с помощью оператора `new` резервирует блок памяти для указателя `string1`. Освобождение занятой памяти выполняет деструктор при удалении объекта. Этой цели служит оператор `delete`.

```
class string_operation { char *string1; int string_len;  
public:  
    string_operation(char*)  
    { string1 = new char[string_len]; } ~string_operation()  
    { delete string1; } ' void input_data(char*); void  
    output_data(char*); );
```

Память, выделенная для указателя `string1` с помощью оператора `new`, может быть освобождена только оператором `delete`. Поэтому если конструктор класса выделяет память для какой-нибудь переменной, важно проследить, чтобы в деструкторе эта память обязательно освобождалась.

Области памяти, занятые данными базовых типов, таких как `int` или `float`, освобождаются системой автоматически и не требуют помощи конструктора и деструктора.

Перегрузка функций-членов класса

Функции-члены класса можно перегружать так же, как и любые другие функции в C++, т.е. в классе допускается существование нескольких функций с одинаковым именем. Правильный вариант функции автоматически выбирается компилятором в зависимости от количества и типов аргументов, заданных в прототипе функции. В следующей программе создается перегруженная функция `number()` класса `absolute_value`, которая возвращает абсолютное значение как целочисленных аргументов, так и аргументов с плавающей запятой. В первом случае для этого используется библиотечная функция `abs()`, принимающая аргумент типа `int`, во втором — `fabs()`, принимающая аргумент типа `double`.

```
//  
// absolute. cpp  
// Эта программа на языке C++ демонстрирует использование  
// перегруженных  
// функций-членов класса. Программа вычисляет абсолютное значение  
// чисел  
// типа int и double.  
#include <iostream.h>  
#include <math.h> // содержит прототипы функций abs() и fabs()  
class absolute_value { public:  
    int number (int);  
    double number (double) ;  
    int absolute_value::number(int test_data) return(abs(test_data));  
    double absolute_value::number(double test_data)  
    return(fabs(test_data)); }
```

```

main()
absolute_value neg_number;
cout<< "Абсолютное значение числа -583 равно << neg_number.number(-
583) << endl;
cout<< "Абсолютное значение числа -583.1749 равно "
<< neg_number. number (-583.1749)<< endl; return (0); )

```

Вот какими будут результаты работы программы:

Абсолютное значение числа -583 равно 583 Абсолютное значение числа -583.1749 равно 583.175

В приводимой далее программе в функцию trig_calc() передается значение угла в одном из двух форматов: числовом или строковом. Программа вычисляет синус, косинус и тангенс угла.

```

//
// overload. cpp
// Эта программа на языке C++ содержит пример перегруженной функции,
// принимающей значение угла как в числовом виде, так и в формате
// градусы/минуты/секунды.
//
#include <iostream.h>
#include <math.h>
#include <string.h>
const double DEG_TO_RAD = 0.0174532925;
class trigonometric { double angle;
public:
void trig_calc(double);
void trig_calc(char *); };
void trigonometric::trig_calc(double degrees) !
angle = degrees;
cout << "\nДля угла " << angle << " градусов:" << endl;
cout << "синус равен " << sin (angle * DEG__TO_RAD) << endl;
cout << "косинус равен " << cos(angle * DEG_TO_RAD) << endl;
cout << "тангенс равен " << tan(angle * DEG_TO_RAD) << endl; }
void trigonometric::trig_calc(char *dat) (
char *deg, *min, *sec;
deg = strtok(dat, "d");
min = strtok(0, "m");
sec = strtok(0,"s");
angle = atof(deg) + atof(min)/60.0 + atof (sec)/360.0;
cout<< "\nДля угла". << angle << " градусов:" << endl;
cout << "синус равен ' " << sin(angle * DEG_TO_RAD) << endl;
cout << "косинус равен " << cos(angle * DEG_TO_RAD) << endl;
cout << "тангенс равен " << tan (angle * DEG_TO_RAD) << endl; }
main ()
(   trigonometric data;
data.trig_calc(75.0) ;
char str1[]   = "35d 75m 20s"; data.trig_calc(str1) ;
data.trig_calc(145.72);
char str2[1]= "65d45m 30s"; data.trig_calc (str2) ;
return(0); }

```

В программе используется библиотечная функция strtok() , прототип которой находится в файле STRING.H. Эта функция сканирует строку, разбивая ее на лексемы, признаком конца которых служит один из символов, перечисленных во втором аргументе. Длина каждой лексемы может быть произвольной. Функция возвращает указатель на первую обнаруженную лексему. Лексемы можно читать одну за другой путем последовательного вызова функции

`strtok()`. Дело в том, что после каждой лексемы она вставляет в строку символ `\0` вместо символа-разделителя. Если при следующем вызове в качестве первого аргумента указать 0, функция продолжит чтение строки с этого места. Когда в строке больше нет лексем, возвращается нулевой указатель.

Рассмотренная программа позволяет задавать значение угла в виде строки с указанием градусов, минут и секунд. Признаком конца первой лексемы, содержащей количество градусов, служит буква `d`, второй лексемы — `t`, третьей — `s`. Каждая извлеченная лексема преобразуется в число с помощью стандартной библиотечной функции `atof()` из файла `MATH.H`.

Ниже представлены результаты работы программы:

```
Для угла 75 градусов:
синус равен 0.965926
косинус равен 0.258819
тангенс равен 3.73205
Для угла 36.3056 градусов:
синус равен 0.592091
косинус равен 0.805871
тангенс равен 0.734722
Для угла 145.72 градусов:
синус равен 0.563238
косинус равен -0.826295
тангенс равен -0.681642
Для угла 65.8333 градусов:
синус равен 0.912358
косинус равен 0.409392
тангенс равен 2.22857
```

Дружественные функции

Переменные-члены классов, как правило, являются закрытыми, и доступ к ним можно получить только посредством функций-членов своего же класса. Может показаться странным, но существует категория функций, создаваемых специально для того, чтобы преодолеть это ограничение. Эти функции, называемые дружественными и объявляемые в описании класса с помощью ключевого слова `friend`, получают доступ к переменным-членам класса, сами не будучи его членами.

```
//
// friend.cpp
// Эта программа на языке C++ демонстрирует использование
// дружественных
// функций. Программа получает от системы информацию о текущей дате и
// времени и вычисляет количество секунд, прошедших после полуночи.
//
#include <iostream.h>
#include <time.h>          // содержит прототипы функций
time(),localtime(),
// asctime(), а также описания структур tm и time_t
class time_class {
long secs;
friend long present_time(time_class);
// дружественная функция public:
time_class(tm *);
time_class::time_class(tm *timer) {
secs = timer->tm_hour*3600 + timer->tm_min*60 + timer->tm_sec; }
long present_time(time_class); // прототип main()
{
// получение данных о дате и времени от системы
```

```

time_t ltime; tm *ptr;
time(&ltime);
ptr = localtime(&ltime);
time_class tz(ptr);
cout<< "Текущие дата и время: " << asctime(ptr) << endl;
cout<< "Число секунд, прошедших после полуночи: "
<< present_time(tz) << endl; return (0);
}
long present_time (time_class tz)
{
return(tz.secs);
}

```

Давайте подробнее рассмотрим процесс получения данных о дате и времени.

```

time_t ltime;
tm *ptr;
time (&ltime) ; ptr = localtime(&ltime) ;

```

Сначала создается переменная `ltime` типа `time_t`, которая инициализируется значением, возвращаемым функцией `time()`. Эта функция вычисляет количество секунд, прошедших с даты 1-е января 1970 г. 00:00:00 по Гринвичу, в соответствии с показаниями системных часов. Тип данных `time_t` специально предназначен для хранения подобного рода значений. Он лучше всего подходит для выполнения такой операции, как сравнение дат, поскольку содержит, по сути, единственное число, которое легко сравнивать с другим аналогичным числом.

Далее создается указатель на структуру `tm`, предназначенную для хранения даты в понятном для человека виде:

```

struct tm {
int tm_sec; // секунды (0–59)
int tm_min; // минуты (0–59)
int tm_hour; // часы (0–23)
int tm_mday; // день месяца (1–31)
int tm_mon; // номер месяца (0–11; январь = 0)
int tm_year; // год (текущий год минус 1900)
int tm_wday; // номер дня недели (0–6; воскресенье = 0)
int tm_yday; // день года (0–365; 1-е января = 0)
int tm_isdst; // положительное число, если осуществлен переход на
летнее время;
// 0, если летнее время еще не введено;
// отрицательное число, если информация о летнем времени
// отсутствует };

```

Функция `localtime()`, в качестве аргумента принимающая адрес значения типа `time_t`, возвращает указатель на структуру `tm`, содержащую информацию о текущем времени в том часовом поясе, который установлен в системе. Имеется схожая с ней функция `gmtime()`, вычисляющая текущее время по Гринвичу.

Полученная структура передается объекту `tz` класса `time_class`, а точнее — конструктору этого класса, вызываемому в строке

```
time_class tz(ptr);
```

В конструкторе из структуры `tm` извлекаются поля `tm_sec`, `tm_min` и `tm_hour` и по несложной формуле вычисляется количество секунд, прошедших после полуночи.

```
secs = timer->tm_hour*3600 + timer->tm_min*60 + timer->tm_sec;
```

Функция `asctime()` преобразует структуру `tm` в строку вида

```
день месяц число часы:минуты:секунды год\n\n0
```

Например:

```
MonAug10 13:12:21 1998
```

Дружественная классу `time_class` функция `present_time()` получает доступ к переменной `secs` этого класса и возвращает ее в программу. Программа выводит на экран две строки следующего вида:

Текущие дата и время: MonAug10 09:31:141998
Число секунд, прошедших после полуночи: 34274

Указатель `this`

Ключевое слово `this` выступает в роли неявно заданного указателя на текущий объект:

`имя_класса *this;`

Указатель `this` доступен только в функциях-членах классов (а также структур) и позволяет сослаться на объект, для которого вызвана функция. Чаще всего он находит применение в конструкторах, выполняющих резервирование памяти:

```
class имякласса {
.
.
.
public:
имя_класса(int size)    (    this = new(size);    }
~имя_класса (void) ; };
```

Перегрузка операторов

В C++ разрешается осуществлять перегрузку не только функций, но и операторов. В создаваемом классе можно изменить свойства большинства стандартных операторов, таких как `+`, `-`, `*` и `/`, заставив их работать не только с данными базовых типов, но также с объектами.

Концепция перегрузки операторов реализована в большинстве языков программирования, пусть и в неявном виде. Например, оператор суммирования `+` позволяет складывать как целочисленные значения, так и значения с плавающей запятой. Это и есть перегрузка, так как один и тот же оператор применяется для выполнения действий над данными разных типов.

Для перегрузки операторов применяется ключевое слово `operator`:

`тип operator оператор (список_параметров)`

В таблице 14.1 перечислены операторы, перегрузка которых допустима в C++.

Таблица 14.1. Операторы, которые могут быть перегружены					
<code>+</code>	<code>-</code>	<code>*</code>	<code>/</code>	<code>%</code>	<code>^</code>
<code>!</code>	<code>=</code>	<code><</code>	<code>></code>	<code>+=</code>	<code>-=</code>
<code>^=</code>	<code>&=</code>	<code> =</code>	<code><<</code>	<code>>></code>	<code><<=</code>
<code><=</code>	<code>>=</code>	<code>&&</code>	<code> </code>	<code>++</code>	<code>--</code>
<code>()</code>	<code>[]</code>	<code>new</code>	<code>delete</code>	<code>&</code>	<code> </code>
<code>~</code>	<code>*=</code>	<code>/=</code>	<code>%=</code>	<code>>>=</code>	<code>==</code>
<code>!=</code>	<code>,</code>	<code>-></code>	<code>->*</code>		

На перегрузку операторов накладываются следующие ограничения: невозможно изменить приоритет оператора и порядок группировки его операндов;

- невозможно изменить синтаксис оператора: если, например, оператор унарный, т.е. принимает только один аргумент, то он не может стать бинарным;
- перегруженный оператор является членом своего класса и, следовательно, может участвовать только в выражениях с объектами этого класса;
- невозможно изменить смысл стандартного оператора применительно к базовым типам данных;
- невозможно создавать новые операторы; запрещается переопределять такие операторы:
- `..* :: ?:`

Следующая программа наглядно иллюстрирует концепцию перегрузки операторов, осуществляя суммирование углов, заданных с помощью строк формата

градусыd минутm секундs

Выделение из строки значащих частей осуществляет уже знакомая нам функция strtok(). Признаками конца лексем служат буквы d, m и s.

```
//
// angles.cpp
// Эта программа на языке C++ содержит пример перегрузки оператора +.
//
#include <iostream.h>
#include <string.h>
include <stdlib.h> // функция atoi()
class angle_value {
int degrees, minutes, seconds;
public:
angle_value () { degrees =0,
minutes =0,
seconds = 0; } // стандартный конструктор
angle_value(char *);
int get_degrees () ;
int get_minutes();
int get_seconds();
angle_value operator +(angle_value); // перегруженный оператор
};
angle_value::angle_value(char *angle_sum) {
// выделение значащих частей строки и преобразование их в целые числа
degrees = atoi(strtok(angle_sum, "d"));
minutes = atoi(strtok(0,"m"));
seconds = atoi(strtok(0,"s"));
}
int angle_value::get_degrees() return degrees;
int angle_value::get_minutes()
return minutes;
}
int angle_value::get_seconds() return seconds;
angle_value angle_value::
operator+(angle_value angle_sum) angle_value ang;
ang.seconds = (seconds + angle_sum.seconds) % 60; ang.minutes =
((seconds+ angle_sum.seconds) / 60 +
minutes + angle_sum.minutes) % 60; ang.degrees = ((seconds+
angle_sum.seconds) / 60 +
minutes + angle_sum.minutes) / 60;
ang.degrees += degrees + angle_sum.degrees;
return ang;
main()
char str1[]= "37d15m 56s";          angle_value angle1(str1);
char str2[]= "10d44m 44s"; angle_value angle2(str2);
char str3[]= "75d 17m 59s";
angle_value angles(str3);
char str4[]= "130d 32m 54s";
angle_value angle4(str4);
angle_value sum_of_angles;
sum_of_angles = angle1 + angle2 + angle3 + angle4;
```

```

cout<< "Сумма углов равна "
<< sum_of_angles.get_degrees() << "d "
<< sum_of_angles.get_minutes() << "m "
<< sum_of_angles.get_seconds() << "s"
<< endl;
return(0);
}

```

Следующий фрагмент программы отвечает за раздельное суммирование градусов, минут и секунд. Здесь применяется еще не перегруженный оператор +:

```

angle_value ang;
ang.seconds = (seconds + angle_sum.seconds) % 60; ang.minutes =
((seconds + angle_sum.seconds) / 60 +
minutes + angle_sum.minutes) % 60;
ang.degrees = ((seconds + angle_sum.seconds) / 60 +
minutes + angle_sum.minutes) / 60;
ang.degrees += degrees + angle_sum.degrees;

```

Обратите внимание на то, что когда сумма секунд или минут превышает 60, должен осуществляться перенос соответствующего числа разрядов. Поэтому каждая предыдущая операция повторяется в следующей, только оператор деления по модулю (%) заменяется оператором деления без остатка (/).

Программа выдаст на экран такую строку:

```
Сумма углов равна 253d 51m 33s
```

Проверьте, правильно ли вычислен результат.

Производные классы

Порождение новых классов от уже существующих — одна из наиболее важных особенностей объектно-ориентированного программирования. Исходный класс называют базовым или родительским, а производный от него — подклассом или дочерним, классом. Порождение нового класса является достаточно удобным способом расширения возможностей родительского класса, так как класс-потомок наследует все его свойства, но может добавлять и свои собственные. У класса может быть множество потомков, как прямых, так и косвенных. Если в подклассе переопределяется какая-либо функция родительского класса, то такие функции называются виртуальными.

Когда новый класс создается на основе существующего, применяется следующий синтаксис:

```

class имя_производного_класса : (public/private/protected)
имя_базового_класса
{...};

```

Тип наследования `public` означает, что все открытые и защищенные члены базового класса становятся таковыми производного класса. Тип наследования `private` означает, что все открытые и защищенные члены базового класса становятся закрытыми членами производного класса. Тип наследования `protected` означает, что все открытые и защищенные члены базового класса становятся защищенными членами производного класса.

В следующей программе показан пример создания открытых производных классов. Родительский класс `consumer` содержит общие данные о клиенте: имя, адрес, город, штат и почтовый индекс. От базового класса порождаются два производных класса: `airlinei` и `rental_car`. Первый из них хранит сведения о расстоянии, покрытом клиентом на арендованном самолете, а второй — сведения о расстоянии, покрытом на арендованном автомобиле.

```

//
// derclass.cpp
// Эта программа на языке C++ демонстрирует использование производных
классов.
//
#include <iostream.h>

```



```

#include <string.h>
char newline;
class consumer { char name[60],
street[60], city[20], state[15], zip[10]; public:
void data_output(void); void data_input(void); };
void consumer::data_output() {
cout << "Имя:      " << name << endl;
cout << "Улица*   " << street << endl;
cout << "Город:    " << city << endl;
cout << "Штат:     " << state << endl;
cout << "Индекс:   " << zip << endl;
void consumer::data_input () {
cout<< "Введите полное имя клиента: ";
cin.get (name, 59, '\n')>>'
cin.get(newline); // пропуск символа новой строки
cout << "Введите адрес: "; cin.get(street, 59, '\n'); cin.get(newline);
cout << "Введите город: ";
cin.get(city, 19, '\n');
cin.get(newline) ;
cout<< "Введите название штата:   ";
cin.get(state, 14, '\n');
cin.get (newline);
cout << "Введите почтовый индекс: ";
    cin.get(zip, 9, '\n');
    cin.get(newline);
}
class airline : public consumer {
char airline_type[20];
float acc_air_miles; public:
void airline_consumer();
void disp_air_mileage(); };
void airline : :airline_consumer () {
data_input ( ) ;
cout << "Введите тип авиалиний : " ;
cin.get (airline_type, 19, '\n'); cin.get (newline) ;
cout << "Введите расстояние, покрытое в авиарейсах: ";
cin >> acc_air_miles; cin.get (newline) ; }
void airline : :disp_air_mileage () {
data_output ( ) ;
cout << "Тип авиалиний: "" << airline_type << endl;
cout << "Суммарное расстояние: " << acc_air_miles << endl;
class rental_car : public consumer {
char rental_car_type[20] ;
float acc_road_miles; public:
void rental_car_consumer ( ) ;
void disp_road_mileage ( ) ;
};
void rental_car::rental_car_consumer() {
data_input();
cout<< "Введите марку автомобиля: ";
cin.get(rental_car_type, 19, '\n');
cin.get (newline) ;
cout << "Введите суммарный автопробег: ";
cin >> acc_road_miles;

```

```

cin.get(newline);
}
void rental_car::disp_road_mileage() {
data_output();
cout << "Марка автомобиля: "
<< rental_car_type << endl; cout << "Суммарный автопробег: " <<
acc_road_miles << endl;
}
main()
airline cons1; rental_car cons2;
cout << "\n-Аренда самолета-\n";          cons1.airline_consumer();
cout << "\n-Аренда автомобиля-\n";
cons2.rental_car_consumer();
cout<< "\n-Аренда самолета-\n";
cons1.disp_air_mileage();
cout<< "\n-Аренда автомобиля-\n";
cons2 . disp_road_mileage();
return(0);
}

```

Переменные родительского класса `consumer` недоступны дочерним классам, так как являются закрытыми. Поэтому для работы с ними созданы функции `data_input()` и `data_output()`, которые помещены в открытую часть класса и могут быть вызваны в производных классах.

Глава 15. Классы ввода-вывода в языке C++

- Иерархия классов ввода-вывода
- Файловый ввод
- Файловый вывод
 - Двоичные файлы
- Буферы потоков
 - Строковые буферы
- Несколько примеров форматного вывода данных

В главе "Основы ввода-вывода в языке C++" были даны общие представления о потоковых объектах `cin`, `cout` и `cerr` и операторах потокового ввода-вывода `<<` и `>>`. В настоящей главе мы поговорим о стандартных классах C++, управляющих работой этих объектов.

Иерархия классов ввода-вывода

Все потоковые классы порождены от одного класса, являющегося базовым в иерархии, — `ios`. Исключение составляют лишь классы буферизованных потоков, базовым для которых служит класс `streambuf`. Всех их можно разделить на четыре категории, как показано в табл. 15.1.

На рис. 15.1 схематически изображена иерархия классов ввода-вывода, порожденных от класса `ios`.

Классы семейства `ios` предоставляют программный интерфейс и обеспечивают необходимое форматирование обрабатываемых данных, тогда как непосредственную обработку данных выполняют классы семейства `streambuf`, управляющие обменом данными между буфером потока и конечным устройством (рис. 15.2).

Таблица 15.1. Категории классов ввода-вывода	
Класс	Описание
<code>ios</code>	Содержит базовые средства управления потоками, является родительским для других классов ввода-вывода (файл <code>IOSTREAM.H</code>)
Потоковый ввод	
<code>istream</code>	Содержит общие средства потокового ввода, является родительским для других классов ввода (файл <code>IOSTREAM.H</code>)
<code>ifstream</code>	Предназначен для ввода данных из файлов (файл <code>FSTREAM.H</code>)
<code>istream_withassign</code>	Поддерживает операцию присваивания; существует предопределенный объект <code>cin</code> данного класса, по умолчанию читающий данные из стандартного входного потока, но благодаря операции присваивания этот объект может быть переадресован на различные объекты класса <code>istream</code> (файл <code>IOSTREAM.H</code>)
<code>istrstream</code>	Предназначен для ввода данных из строковых буферов (файл <code>STRSTREA.H</code>)
Потоковый вывод	
<code>ostream</code>	Содержит общие средства потокового вывода, является родительским для других классов вывода (файл <code>IOSTREAM.H</code>)
<code>ofstream</code>	Предназначен для вывода данных в файлы (файл <code>FSTREAM.H</code>)
<code>ostream_withassign</code>	Поддерживает операцию присваивания; существуют предопределенные объекты <code>cout</code> , <code>cerr</code> и <code>clog</code> данного класса, по умолчанию выводящие данные в стандартный выходной поток, но благодаря операции присваивания их можно переадресовать на различные объекты класса <code>ostream</code> (файл <code>IOSTREAM.H</code>)
<code>ostrstream</code>	Предназначен для вывода данных в строковые буферы (файл <code>STRSTREA.H</code>)
Потоковый ввод-вывод	
<code>iostream</code>	Содержит общие средства потокового ввода-вывода, является родительским для других классов ввода-вывода (файл <code>IOSTREAM.H</code>)
<code>fstream</code>	Предназначен для организации файлового ввода-вывода (файл <code>FSTREAM.H</code>)

strstream	Предназначен для ввода-вывода строк (файл STBLSTREA.H)
stdiostream	Поддерживает работу с системными средствами стандартного ввода-вывода, существует для совместимости со старыми функциями ввода-вывода (файл STDIOSTR.H)
Буферизованные потоки	
streambuf	Содержит общие средства управления буферами потоков, является родительским для других буферных классов (файл IOSTREAM.H)
filebuf	Предназначен для управления буферами дисковых файлов (файл FSTREAM.H)
strstreambuf	Предназначен для управления строковыми буферами, хранящимися в памяти (файл STRSTREA.H)
stdiobuf	Осуществляет буферизацию дискового ввода-вывода с помощью стандартных системных функций (файл STDIOSTR.H)

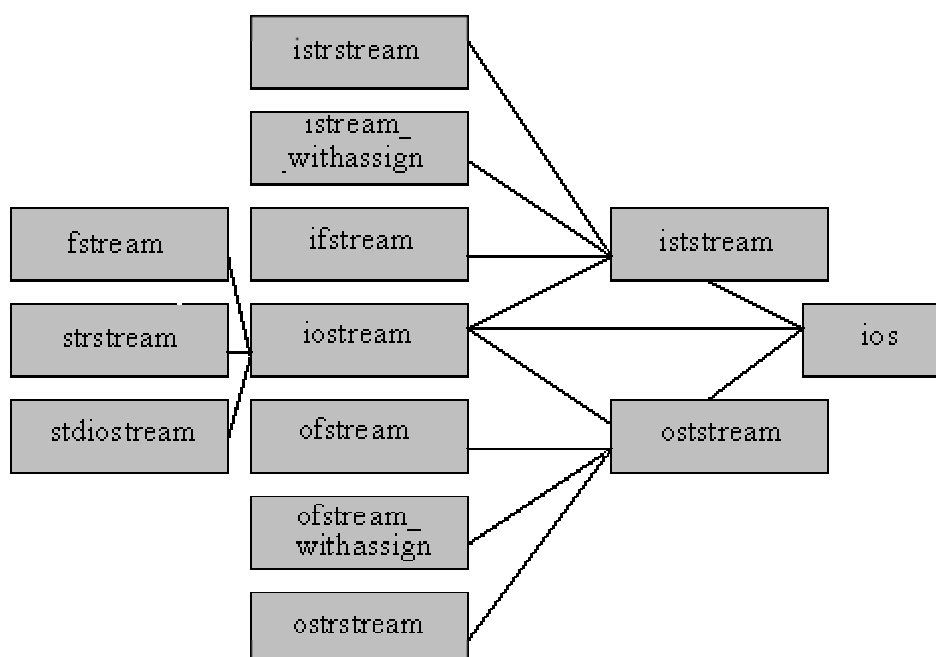


Рис. 15.1. Иерархия классов ввода-вывода порожденных от ios

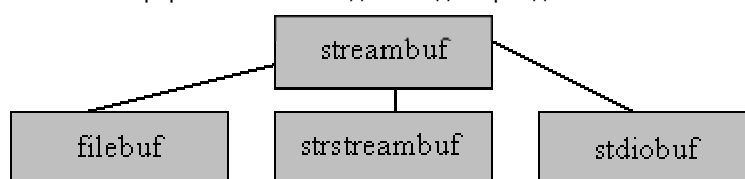


Рис. 15.2. Иерархия классов, порожденных от streambuf

Файловый ввод

Основные функции управления потоковым вводом сосредоточены в классе istream. С каждым из объектов этого класса и его производных связан объект класса streambuf. Эти классы работают в связке: первый осуществляет форматирование, а второй управляет низкоуровневым буферизованным вводом. Функции класса istream, доступные его потомкам, перечислены в табл. 15.2.

Функция	Описание
ipfx	Вызывается перед операцией чтения для проверки наличия ошибок в потоке
isfx	Вызывается после каждой операции чтения
get	Извлекает из потока требуемое число символов; если указан символ-ограничитель, он не извлекается
getline	Извлекает из потока требуемое число символов; если указан символ-ограничитель, он извлекается, но не сохраняется в буфере
read	Извлекает из потока требуемое число байтов; применяется при работе с двоичными потоками

ignore	Выбрасывает из потока требуемое число символов вплоть до символа-ограничителя
peek	Возвращает текущий символ, сохраняя его в потоке
gcount	Определяет число символов, извлеченных из потока во время последней операции чтения
eatwhite	Извлекает из потока ведущие пробельные символы; аналогичное действие выполняет манипулятор ws
putback	Возвращает в поток символы, извлеченные из него во время последней операции чтения
sync	Синхронизирует внутренний буфер потока с внешним источником символьных данных
seekg	Перемещает маркер, обозначающий текущую позицию чтения, на требуемую позицию в потоке
tellg	Возвращает позицию маркера чтения
Таблица 15.2. Функции класса istream	

Класс ifstream является потомком класса istream, ориентированным на чтение данных из файлов. Его конструктор автоматически создает объект класса filebuf, управляющий низкоуровневой работой с файлом, включая поддержку буфера чтения. Функции класса ifstream перечислены в табл. 15.3.

Таблица 15.3. Функции класса ifstream	
ФУНКЦИЯ	Описание
open	Открывает файл для чтения, связывая с ним объект класса filebuf
close	Закрывает файл
setbuf	Передаёт указанный символьный буфер в распоряжение объекта класса filebuf
setmode	Задаёт режим доступа к файлу: двоичный (константа filebuf:: binary) или текстовый (константа filebuf:: text)
attach	Связывает указанный открытый файл с объектом класса filebuf
rdbuf	Возвращает указатель на объект класса filebuf
fd	Возвращает дескриптор файла
is_open	Проверяет, открыт ли файл, связанный с потоком

В следующей программе из файла читаются данные блоками по 80 символов, которые затем выводятся на экран:

```
//
// ifstream.cpp
// В этой программе на языке C++ демонстрируется
// использование класса ifstream для чтения данных из файла.
//
#include <fstream.h>
#define ICOLUMNS 80
void main(void)
{
    char cOneLine[ICOLUMNS];
    fstream ifMyInputStream("IFSTREAM.CPP");
    while(ifMyInputStream) {
        ifMyInputStream.getline(cOneLine, ICOLUMNS);
        cout << '\n' << cOneLine;
    }
    ifMyInputStream.close();
}
```

Конструктор класса ifstream создает объект ifMyInputStream, связывая с ним файл IFSTREAM.CPP, который открывается для чтения (по умолчанию в текстовом режиме). Этот объект можно использовать в условных операторах, проверяя его на равенство нулю, что означает достижение конца файла.

Функция getline(), унаследованная от класса istream, читает в массив cOneLine строку текста длиной ICOLUMNS(80 символов). Ввод будет прекращен при обнаружении символа новой строки \n, конца файла или, если ни одно из этих событий не произошло, 79-го по счету символа (последним записывается символ \0).

После окончания вывода всего файла он закрывается командой `close()`.

Файловый вывод

Основные функции управления потоковым выводом сосредоточены в классе `ostream`. С каждым из объектов этого класса и его производных связан объект класса `streambuf`. Эти классы работают в связке: первый осуществляет форматирование, а второй управляет низкоуровневым буферизованным выводом. Функции класса `ostream`, доступные его потомкам, перечислены в табл. 15.4.

Таблица 15.4. Функции класса <code>ostream</code>	
Функция	Описание
<code>orfx</code>	Вызывается перед каждой операцией записи для проверки наличия ошибок в потоке
<code>osfx</code>	Вызывается после каждой операции записи для очистки буфера
<code>put</code>	Записывает в поток одиночный байт
<code>write</code>	Записывает в поток требуемое число байтов
<code>flush</code>	Очищает буфер потока; аналогичное действие выполняет манипулятор <code>flush</code>
<code>seekp</code>	Перемещает маркер, обозначающий текущую позицию записи, на требуемую позицию в потоке
<code>tellp</code>	Возвращает позицию маркера записи

Класс `ofstream` является потомком класса `ostream`, ориентированным на запись данных в файлы. Его конструктор автоматически создает объект класса `filebuf`, управляющий низкоуровневой работой с файлом, включая поддержку буфера записи. Класс `ofstream` содержит такой же набор функций, что и класс `ifstream` (см. табл. 15.3).

В следующей программе в файл дважды записывается одна и та же строка — посимвольно и вся целиком.

```
//
// ofstream. cpp
// В этой программе на языке C++ демонстрируется
// использование класса ofstream для записи данных в
// файл, открытый в текстовом режиме.
#include <fstream.h>
#include <string.h>
#define iSTRING_MAX 40
void main (void) {
    int i = 0;
    char pszString [iSTRING_MAX] = "Записываемая строка\n";
    // файл по умолчанию открывается в текстовом режиме
    ofstream ofMyOutputStream("OFSTREAM.OUT") ;
    // строка выводится символ за символом;
    // обратите внимание, что символ '\n' .
    // преобразуется в два символа
    while (pszString[i] != '\0'){
        ofMyOutputStream.put (pszString[i] ) ;
        cout<< !'\nПозиция маркера записи: " << ofMyOutputStream.tellp() ;
        i++; }
    // запись всей строки целиком
    ofMyOutputStream.write(pszString, strlen(pszString));
    cout<< "\nНовая позиция маркера записи: " << ofMyOutputStream.tellp();
    ofMyOutputStream.close() ; -}
```

Вот результаты работы программы:

```
Позиция маркера записи:      1
Позиция маркера записи:      2
Позиция маркера записи:      3
```

```

.
.
.
Позиция маркера записи: 18
Позиция маркера записи: 19
Позиция маркера записи: 21
Новая позиция маркера записи: 42

```

Цикл while последовательно, символ за символом, с помощью функции put() записывает содержимое строки pszstring в выходной поток. После вывода каждого символа Вызывается функция tellp(),возвращающая текущую позицию маркера записи. Результатам работы этой функции следует уделить немного внимания.

Строка pszString содержит 20 символов плюс концевой нулевой символ (\0), итого — 21. На хотя, если судить по выводимой информации, в поток записывается 21 символ, последним из них будет не \0. Его вообще не окажется в файле. Дело в том, что при выводе данных в файл, открытый в текстовом режиме, автоматически выполняется преобразование \n = CR/LF, т.е. символ новой строки преобразуется в пару символов возврата каретки и перевода строки. (При чтении происходит обратное преобразование.) Именно поэтому наблюдается "скачок" счетчика: после 19 идет 21. Функция put(), записывающая в файл символ \n, на самом деле помещает в файл два других символа.

Функция write() выполняет такое же преобразование, поэтому конечное значение счетчика равно 42, а не 41.

Двоичные файлы

Следующая программа аналогична предыдущему примеру и иллюстрирует, что произойдет, если ту же самую строку вывести в тот же самый файл, только открытый в двоичном режиме.

```

//
//      binary.cpp
//      Эта программа является модификацией предыдущего примера
//      и демонстрирует работу с файлом,открытым в двоичном режиме.
//
#include <fstream. h>
#include <string.h>
#define iSTRING_MAX 40
void main (void) <<
int i = 0;
char pszString [iSTRING_MAX] = "Записываемая строка\n";
// файл открывается в двоичном режиме
ofstream ofMyOutputStream("OFSTREAM.OUT", ios :: binary)
// строка выводится символ за символом;
// обратите внимание, что символ ' \n '
// никак не преобразуется
while (pszString [i]!='\0') {
ofMyOutputStream.put (pszString [i]);
cout <<' '\nПозиция маркера записи: << ofMyOutputStream.tellp0 ;
i++> 1
// запись всей строки целиком
ofMyOutputStream.write(pszString, strlen(pszString)),
cout<< "\nНовая позиция маркера записи: "
<< ofMyOutputStream.tellp() ;
ofMyOutputStream.close() ;
}

```

Вот результаты работы программы:

```

Позиция маркера записи: 1

```

Позиция маркера записи: 2
 Позиция маркера записи: 3
 Позиция маркера записи: 18
 Позиция маркера записи: 19
 Позиция маркера записи: 20
 Новая позиция маркера записи: 40

При выводе строки pszString не происходит замены конечного символа \n парой символов возврата каретки и перевода строки, поэтому в поток записывается 20 символов — ровно столько, сколько содержится в строке.

Буферы потоков

В основе всех буферизованных потоков ввода-вывода в C++ лежит класс streambuf. В этом классе описаны основные операции, выполняемые над буферами ввода-вывода (табл. 15.5). Любой класс, порожденный от ios, наследует указатель на объект класса streambuf. Именно последний выполняет реальную работу по вводу и выводу данных.

Объекты класса streambuf управляют фиксированной областью памяти, называемой также областью резервирования. Она, в свою очередь, может быть разделена на область ввода и область вывода, которые могут перекрывать друг друга.

Класс streambuf является абстрактным, т.е. создать его объект напрямую нельзя (его конструктор является защищенным и не может быть вызван из программы). В то же время, имеются три производных от него класса, предназначенные для работы с потоками конкретного типа: filebuf (буферы дисковых файлов), strstreambuf (строковые буферы, хранящиеся в памяти) и stdiobuf (буферизация дискового ввода-вывода с помощью стандартных системных функций). Кроме того, можно создавать собственные классы, порожденные от streambuf, переопределяя его виртуальные функции (табл. 15.6) и настраивая, таким образом, его работу в соответствии с потребностями конкретного приложения. Только из производных классов можно вызывать и многочисленные защищенные функции этого класса, управляющие областью резервирования (табл. 15.7).

Таблица 15.5. Открытые функции класса streambuf	
Открытая функция	Описание
in_avail	Возвращает число символов в области ввода
sgetc	Возвращает символ, на который ссылается указатель области ввода; при этом указатель не перемещается
snextc	Перемещает указатель области ввода на одну позицию вперед, после чего возвращает текущий символ
sbumpc	Возвращает текущий символ и затем перемещает указатель области ввода на одну позицию вперед
stossc	Перемещает указатель области ввода на одну позицию вперед, но не возвращает символ
sputbaqkc	Перемещает указатель области ввода на одну позицию назад, возвращая символ в буфер
sgetn	Читает требуемое количество символов из буфера
out_waiting	Возвращает число символов в области вывода
sputc	Записывает символ в буфер и перемещает указатель области вывода на одну позицию вперед
sputn	Записывает требуемое количество символов в буфер и перемещает указатель области вывода на соответствующее число позиций
dbp	В текстовом виде записывает в стандартный выходной поток различного рода информацию о состоянии буфера

Таблица 15.6. Виртуальные функции класса streambuf	
Виртуальная функция	Описание
sync	Очищает области ввода и вывода
setbuf	Добавляет к буферу указанную зарезервированную область памяти
seekoff	Перемещает указатель области ввода или вывода на указанное количество байтов относительно текущей позиции

seekpos	Перемещает указатель области ввода или вывода на указанную позицию относительно начала потока
overflow	Очищает область вывода
underflow	Если область ввода пуста, заполняет ее данными из источника
pbackfail	Вызывается функцией sputbackc() в случае неудачной попытки вернуться назад на один символ

Таблица 15.7. Защищенные функции класса streambuf	
Защищенная функция	Описание
base	Возвращает указатель на начало области резервирования
ebuf	Возвращает указатель на конец области резервирования
blen	Возвращает размер области резервирования
phase	Возвращает указатель на начало области вывода
pptr	Возвращает значение указателя области вывода
epptr	Возвращает указатель на конец области ввода
eback	Возвращает указатель на начало области ввода
gptr	Возвращает значение указателя области ввода
egptr	Возвращает указатель на конец области вывода
setp	Задаст значения указателей, связанных с областью вывода
setg	Задаст значения указателей, связанных с областью ввода
pbump	Перемещает указатель области вывода на указанное число байтов относительно текущей позиции
gbump	Перемещает указатель области ввода на указанное число байтов относительно текущей позиции
setb	Задаст значения указателей, связанных с областью резервирования
unbuffered	Задаст или возвращает значение переменной, определяющей состояние буфера
allocate	Вызывает виртуальную функцию deallocate для создания области резервирования
deallocate	Выделяет память для области резервирования (виртуальная функция)

Классы файловых потоков, такие как ifstream, ofstream и fstream, содержат встроенный объект класса filebuf, который вызывает низкоуровневые системные функции для управления буферизованным файловым вводом-выводом. Для объектов класса filebuf области ввода и вывода, а также указатели текущей позиции чтения и записи всегда равны друг другу. Функции этого класса перечислены в табл. 15.8.

Таблица 15.8. Функции класса filebuf	
Функция	Описание
open	Открывает файл, связывая с ним объект класса filebuf
close	Закрывает файл, "выталкивая" все содержимое области вывода
setmode	Задаст режим доступа к файлу: двоичный (константа filebuf:: binary) или текстовый (константа filebuf:: text)
attach	Связывает указанный открытый файл с объектом класса filebuf
fd	Возвращает дескриптор файла
is_open	Проверяет, открыт ли файл, связанный с потоком

В следующей программе создаются два файловых объекта: fbMyInputBuf и fbMyOutputBuf. Оба они открываются в текстовом режиме с помощью функции open(): первый — для чтения, второй — для записи. Если при открытии файлов не возникнет никаких ошибок, то каждый из них связывается с соответствующим объектом класса istream и ostream. Далее в цикле while символы читаются из файла fbMyInputBuf с помощью функции get() класса istream и записываются в файл fbMyOutputBuf с помощью функции put() класса ostream. Подсчет числа строк осуществляется с помощью выражения

```
iLineCount += (ch == '\n');
```

Когда в поток помещается символ \n, выражение в скобках возвращает 1 и счетчик iLineCount увеличивается на единицу, в противном случае он остается неизменным.

Оба файла закрываются с помощью функции close().

```
//
//  filebuf.cpp
//  Эта программа на языке C++ демонстрирует, как работать
//  с объектами класса filebuf.
//
#include <fstream.h>
#include <process.h>
// содержит прототип функции exit()
void main (void)
{
    char ch;
    int iLineCount = 0;
    filebuf fbMyInputBuf, fbMyOutputBuf;
    fbMyInputBuf.open ("FILEBUF.CPP", ios::in); if (fbMyInputBuf
.is_open() == 0) {
        cerr<< "Невозможно открыть файл для чтения"
        exit (1);
    }
    istream is(fbMyInputBuf) ;
    fbMyOutputBuf.open ("output.dat", ios::out); if (fbMyOutputBuf.is_open()
    == 0) {
        cerr<< "Невозможно открыть файл для записи";
        exit(2); }
    ostream os(fbMyOutputBuf);
    while (is) {
        is.get(ch);
        os.put(ch);
        iLineCount += (ch== '\n');
    }
    fbMyInputBuf.close ();
    fbMyOutputBuf.close();
    cout << "Файл содержит " << iLineCount << " строк";
}
```

Строковые буферы

Класс `stringstream` управляет символьным буфером, расположенным в динамической памяти.

Несколько примеров форматного вывода данных

В первом примере на экран выводится список факториалов чисел от 1 до 25 с выравниванием влево:

```
//
//  fact.cpp
//  Эта программа на языке C++ выводит список факториалов чисел от 1
//  до 25.
//
#include <iostream.h>
main ()
{
    double number = 1.0, factorial = 1.0;
    cout.precision (0); //0 цифр после запятой
    cout.setf (ios::left) ; // выравнивание влево
```

```

cout .setf (ios:: fixed) ;    //      фиксированный формат
(безэкспоненты)
for(int i <= 0; i < 25;i++) {
factorial *= number++;
cout << factorial << endl; }
return(0);
}

```

Результаты работы программы будут такими:

```

1
2
6
24
120
720
5040
40320
362880
3628800
39916800
479001600
6227020800
87178291200
1307674368000
20922789888000
355687428096000
6402373705728000
121645100408832000
2432902008176640000
51090942171709440000
1124000727777607680000
25852016738884976640000
620448401733239439360000
15511210043330985984000000

```

Во втором примере на экран выводится таблица квадратов и квадратных корней целых чисел от 1 до 15.

```

//
//      sqrt.cpp
//  Эта программа на языке C++ строит таблицу квадратов и квадратных
//  корней чисел от 1 до 15.
//
#include <iostream.h>
# include <math.h>
main () {
double number =1.0,square, sqroot;
cout << "Число\tКвадрат\t\tКвадратный корень\n";
cout << " _____ \n";
cout .setf (ios:: fixed) ; // фиксированный формат (без экспоненты)
for(int i = 1; i < 16;i++) {
square = number * number; // вычисление квадрата числа
sqroot = sqrt(number);    // нахождение корня числа
cout.fill('0');          // заполнение недостающих позиций нулями
cout.width(2);            // ширина столбца – минимум 2 символа
cout.precision(0);        // 0 цифр после запятой

```

```

cout << number << "\t";
cout.fill(' '); // использовать пробел в качестве заполнителя cout
<< square << "\t\t";
cout.precision(6); // 6 цифр после запятой
cout<< sqroot<< endl;
number++; }
return(0); }

```

Программа выведет следующую таблицу чисел:

число	корень	квадратный корень
01	1	1.000000
02	4	1.414214
03	9	1.732051
04	16	2.000000
05	25	2.236068
06	36	2.449490
07	49	2.645751
08	64	2.828427
09	81	3.000000
10	100	3.162278
11	121	3.316625
12	144	3.464102
13	169	3.605551
14	196	3.741657
15	225	3.872983

Глава 16. Концепции и средства программирования в Windows

- Основные понятия
 - Среда Windows
 - Преимущества Windows
 - Формат исполняемых файлов
- Базовые концепции программирования
 - Что представляет собой окно
 - Компоненты окна
 - Классы окон
 - Графические объекты, используемые в окнах
 - Принципы обработки сообщений
 - Вызов системных функций
 - Файл WINDOWS.H
 - Этапы создания приложения
- Создание ресурсов приложений средствами Visual C++
 - Файлы проектов
 - Редакторы ресурсов

Языки C и C++ являются основными средствами программирования 32-разрядных приложений Windows. Раньше, когда решающим фактором была скорость выполнения программ, основным языком программирования считался ассемблер, но с появлением Windows ситуация коренным образом изменилась. В настоящей главе мы познакомимся с существующими подходами к созданию традиционных 32-разрядных приложений Windows.

Главу условно можно разбить на три части. В первой из них рассматриваются терминология и основные концепции программирования в Windows. Затем, во второй части, речь пойдет об окнах и графических компонентах приложений, таких как значки, шрифты и прочее. В третьей части мы поговорим о ресурсах Windows и редакторах ресурсов, предоставляемых компилятором Visual C++.

Примечание

Далее в книге под Windows подразумеваются Windows 95, Windows 98 и WindowsNT. Если описываемая возможность реализуется лишь одной из указанных версий, это оговаривается отдельно.

Основные понятия

Приложения Windows могут создаваться как традиционными методами процедурного программирования на языках C и C++, так и с помощью мощных средств объектно-ориентированного программирования, предоставляемых языком C++. В данном параграфе раскрываются основные концепции программирования в Windows и объясняется используемая терминология.

Среда Windows

Windows, как известно, представляет собой графическую многозадачную операционную систему. Все программы, разработанные для этой среды, должны соответствовать определенным стандартам и требованиям. Это касается прежде всего внешнего вида окна программы и принципов взаимодействия с пользователями. Благодаря стандартам, общим для всех приложений Windows, пользователю не составляет труда разобраться в принципах работы любого приложения.

Чтобы помочь программистам в разработке приложений для Windows, были созданы многочисленные системные функции, позволяющие легко добавлять в создаваемые программы контекстные меню, полосы прокрутки, диалоговые окна, значки и многие другие элементы пользовательского интерфейса. А существующее многообразие шрифтов и простота их инсталляции значительно облегчают работу по форматированию выводимых текстовых данных.

Windows позволяет работать с различными периферийными устройствами, такими как монитор, клавиатура, мышь, принтер и т.д., вне зависимости от типа самих устройств. Это дает возможность запускать одни и те же приложения на компьютерах с разной аппаратной конфигурацией.

Преимущества Windows

Можно долго перечислять все открывающиеся перед пользователями преимущества среды Windows по сравнению с устаревшей операционной системой MS-DOS. Среди наиболее важных следует указать стандартизированный графический интерфейс пользователя, многозадачность, совершенные средства управления памятью, аппаратную независимость и возможность широкого применения библиотек динамической компоновки (DLL).

Графический интерфейс пользователя

Первое, что бросается в глаза при знакомстве с приложениями Windows, — это стандартизированный графический интерфейс. Графические стандарты со времени появления Windows 95 не претерпели кардинальных изменений. Для представления дисков, файлов, папок и других системных объектов используются специальные растровые изображения, называемые значками. Окно типичного приложения Windows показано на рис. 16.1.

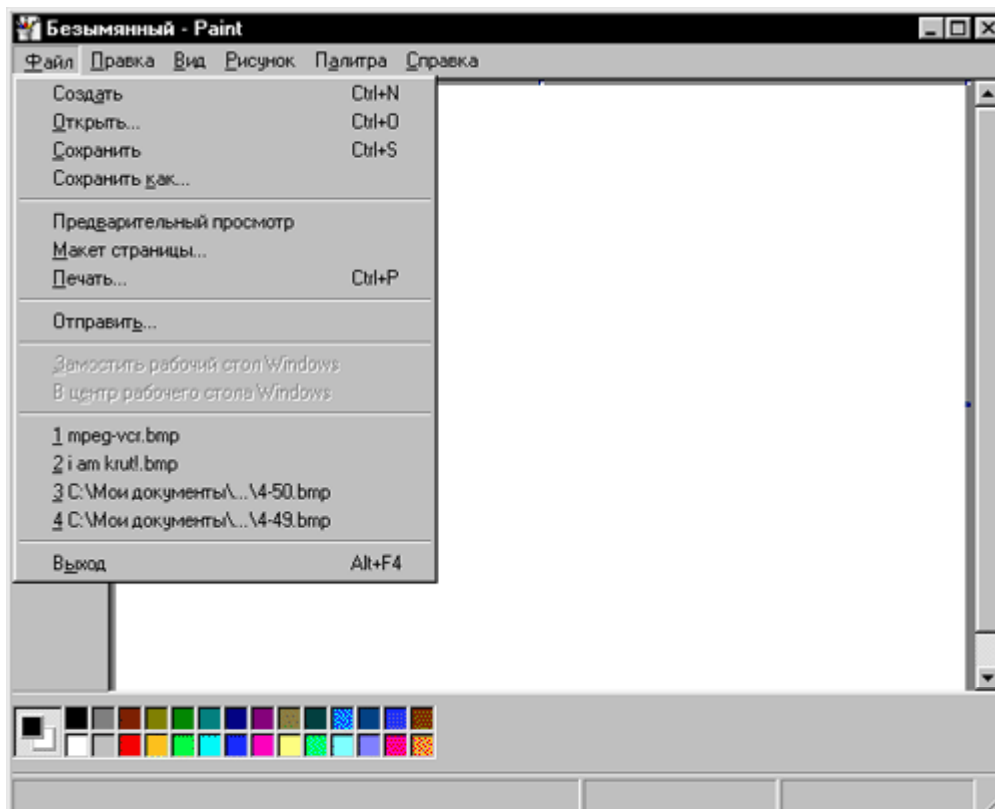


Рис.16.1. Окно типичного Windows-приложения

Название приложения появляется в строке заголовка его окна. Все функции, выполняемые программой, перечислены в меню. Чтобы выполнить команду, достаточно выбрать в меню соответствующий пункт и щелкнуть на нем мышью. В большинстве приложений вызовы команд меню дублируются также комбинациями клавиш.

Как правило, программы имеют сходный внешний вид окон и похожие наборы команд меню. Пользователю достаточно на примере одного приложения изучить основные операции по открытию и манипулированию файлами и данными, чтобы чувствовать себя вполне уверенно с любым другим приложением Windows. В качестве примера к сказанному давайте сравним внешний вид окон Microsoft Excel и Microsoft Word, представленных соответственно на рис. 16.2 и рис. 16.3.

Нетрудно заметить, что окна построены по одному и тому же принципу и в строке меню присутствуют стандартные пункты — File, Edit и другие. Эти же пункты меню вы найдете в окне редактора Paint, показанном на рис. 16.1.

Стандартный интерфейс облегчает работу не только пользователям, но и программистам. Для добавления в приложение меню или диалогового окна достаточно воспользоваться одной из стандартных функций Windows. Поскольку за реализацию меню и диалоговых окон отвечает сама система, а не программист, это гарантирует правильность работы интерфейса приложения.

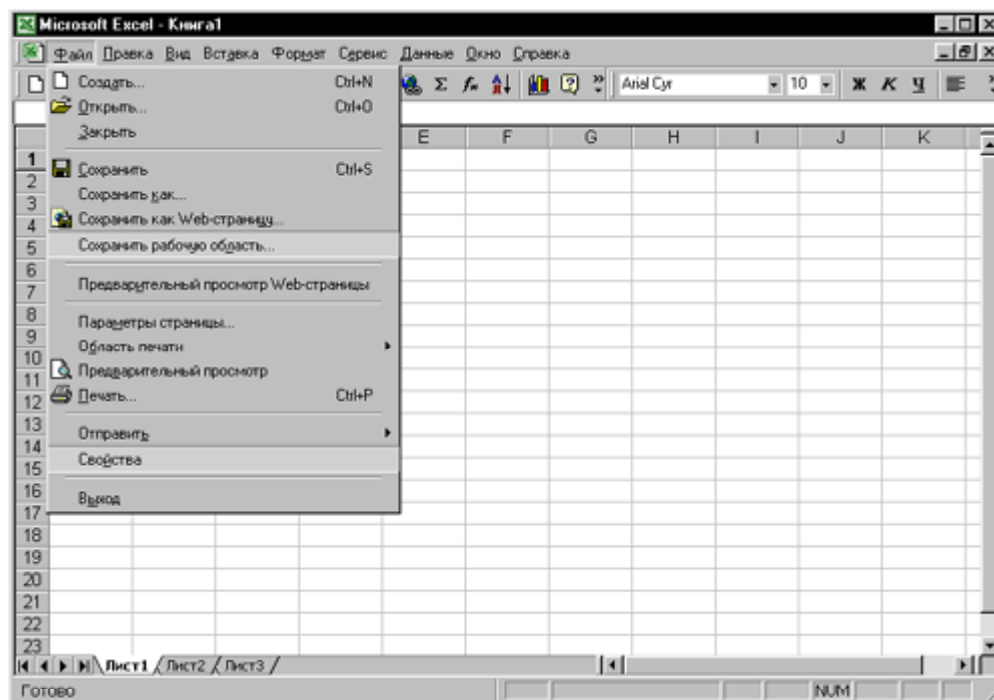


Рис. 16.2. Окно приложения Microsoft Excel

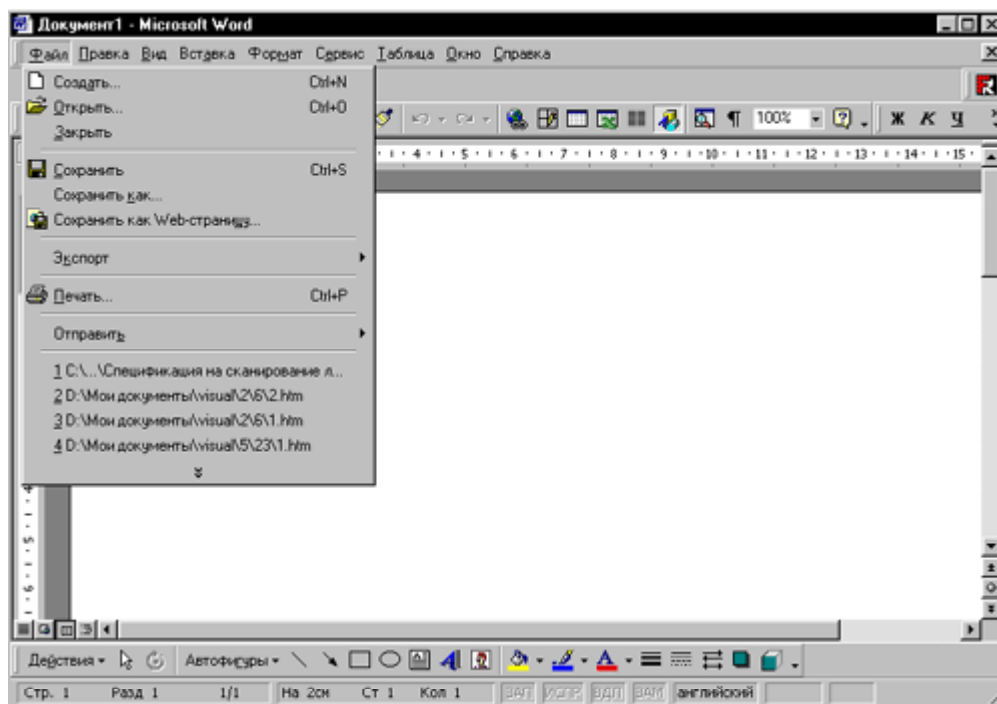


Рис. 16.3. Окно приложения Microsoft Word

Многозадачная среда

Многозадачность Windows состоит в том, что одновременно можно запустить несколько приложений или открыть сразу несколько сеансов работы с одним приложением. На рис. 16.4 показаны окна двух приложений, запущенных одновременно. Каждая программа разместила свое окно поверх рабочего стола Windows. В любой момент времени пользователь может переместить одно из окон в иное место экрана, перейти от одного окна к другому, изменить их размер или произвести обмен данными между приложениями.

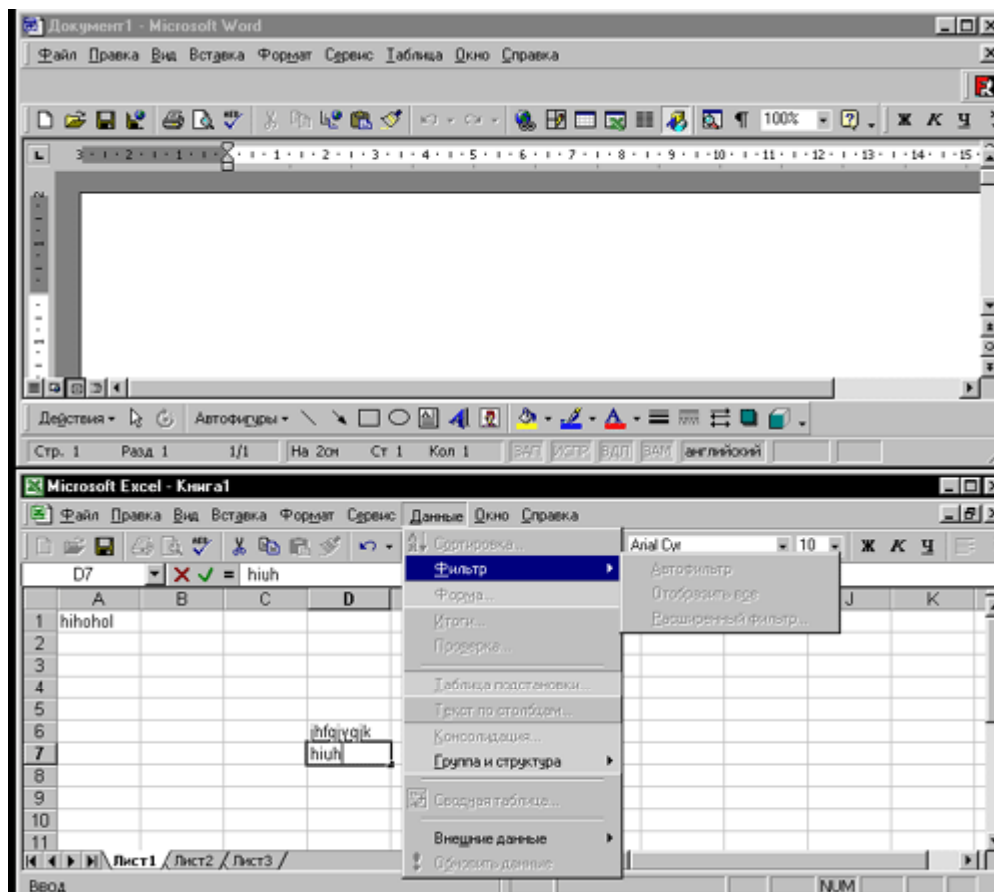


Рис. 16.4. Windows позволяет запускать несколько приложений одновременно

Хотя считается, что приложения выполняются одновременно, в действительности это не так. В текущий момент времени только одно приложение может использовать ресурсы процессора. Многозадачность реализуется за счет того, что процессор переключается между выполняющимися заданиями в течение очень коротких промежутков времени. Приоритеты выполнения одновременно запущенных программ также не одинаковы. В текущий момент времени активным, т.е. принимающим данные от пользователя, может быть только одно приложение, хотя инструкции для процессора могут поступать сразу от всех открытых приложений, независимо от их состояния. Задачу определения приоритетов приложений и распределения времени работы процессора берет на себя Windows.

До того как многозадачность в Windows была реализована, приложения получали полный контроль над предоставляемыми им ресурсами компьютера, включая устройства ввода/вывода, память, монитор и центральный процессор. В среде Windows все эти ресурсы динамически распределяются между запущенными приложениями.

Ввод данных посредством очередей

Как вы уже знаете, в среде Windows память компьютера представляет собой совместно используемый ресурс. Таковыми являются и большинство устройств ввода, в частности клавиатура и мышь. Поэтому при разработке Windows-приложений становятся недоступными функции наподобие `getchar()` языка C, считывающие символы непосредственно с клавиатуры, равно как и потоки ввода/вывода языка C++. В среде Windows приложение не может обращаться напрямую к клавиатуре или мыши и получать данные непосредственно от них. Подобная задача выполняется самой Windows, которая заносит данные в системную очередь. Из очереди введенные данные распределяются между запущенными программами. Это осуществляется путем копирования сообщений из системной очереди в очереди соответствующих приложений. Затем, как только приложение оказывается готовым принять данные, оно считывает сообщения из своей очереди и распределяет их между открытыми окнами.

В Windows данные от устройств ввода поступают в виде сообщений. В каждом сообщении указывается системное время, состояние клавиатуры, код нажатой клавиши, позиция указателя мыши и состояние кнопок мыши, а также идентификатор устройства, пославшего сообщение.

Все сообщения от клавиатуры, мыши и системного таймера имеют одинаковый формат и обрабатываются схожим образом. В случае сообщений клавиатуры передается виртуальный код нажатой клавиши, который идентифицирует клавишу независимо от имеющейся в наличии клавиатуры и генерируется Windows, а также аппаратно-зависимый скан-код, генерируемый самой клавиатурой. Передается также информация о состоянии ряда других клавиш, таких как [NumLock], [Alt], [Shift] и [Ctrl].

Клавиатура и мышь, повторяем, являются совместно используемыми ресурсами. Они посылают сообщения всем приложениям, открытым в данный момент в среде Windows. Система переадресовывает все сообщения клавиатуры текущему активному окну. Сообщения мыши обрабатываются иначе. Они направляются тому приложению, над чьим окном в данный момент находится указатель мыши. О таком окне говорят, что оно получает фокус.

Другую группу составляют сообщения системного таймера. Формат их такой же, как у сообщений клавиатуры и мыши. Windows позволяет приложениям инициализировать таймер таким образом, чтобы одно из окон приложения регулярно принимало сообщения от таймера. Такие сообщения поступают непосредственно в очередь данного приложения.

Сообщения

В основе Windows лежит механизм генерации и обработки сообщений. С точки зрения приложений, сообщения являются формой уведомления о произошедших событиях, на которые приложение должно (или не должно) каким-то образом реагировать. Пользователь может инициировать событие щелчком или перемещением указателя мыши, изменением размера окна или выбором команды из меню. Другие события могут быть инициализированы самим приложением. Например, в электронной таблице завершение вычислений в ячейках может сопровождаться автоматическим обновлением диаграммы, основанной на данном блоке ячеек. В таком случае при завершении вычислений генерируется сообщение для этого же приложения о необходимости обновления диаграммы.

Сообщения могут генерироваться автоматически самой Windows, например в случае завершения работы системы, когда каждому открытому приложению посылается уведомление о необходимости проверить сохранность данных и закрыть свои окна.

Рассматривая роль сообщений в Windows, необходимо отметить, что именно благодаря подсистеме сообщений становится возможной многозадачность Windows. Подсистема сообщений позволяет ей распределять время работы процессора между всеми открытыми приложениями. Каждый раз, когда Windows посылает сообщение приложению, она на некоторое время открывает этому приложению доступ к процессору. Единственная возможность для приложения получить доступ к процессору — это получить сообщение от Windows. Кроме того, сообщения позволяют приложению прореагировать определенным образом на событие, произошедшее в системе. Это событие могло быть вызвано самим приложением, другим приложением, выполняющимся в это же время в Windows, пользователем или операционной системой. Каждый раз, когда происходит то или иное событие, Windows оповещает о нем все заинтересованные приложения, открытые в настоящий момент.

Управление памятью

Одним из наиболее важных ресурсов системы является память компьютера. Если в системе одновременно запущено несколько приложений, то они должны иметь возможность координировать свою работу таким образом, чтобы не вызывать конфликтов при распределении системных ресурсов. Так, в результате многократного открытия и закрытия программ память компьютера может оказаться фрагментированной. Windows обладает встроенной подсистемой дефрагментации памяти, организующей перемещение и объединение блоков памяти.

Если же размер кода запускаемого приложения превышает объем свободной памяти, появляется опасность исчерпания памяти компьютера. Windows способна автоматически выгружать из памяти неиспользуемые в данный момент фрагменты приложений и загружать их опять с диска при обращении к ним.

Приложения Windows могут совместно использовать функции, хранящиеся в отдельных исполняемых файлах общего доступа. Такие файлы называются библиотеками динамической компоновки (DLL — dynamic link libraries). Windows содержит встроенный механизм компоновки таких библиотек с программами на этапе выполнения. Для работы самой Windows необходим достаточно большой набор DLL-файлов. С целью упрощения работы с динамическими библиотеками в Windows применяется новый формат исполняемых файлов. Такие файлы содержат информацию, позволяющую системе управлять блоками кода и данных, а также выполнять динамическую компоновку.

Аппаратная независимость

Windows обеспечивает также независимость пользовательской среды от типа используемых аппаратных устройств. Благодаря этому программисты избавляются от необходимости наперед учитывать, какие именно монитор, принтер или устройство ввода будут подключены к конкретному компьютеру. Ранее, при написании приложений для MS-DOS, необходимо было снабжать программы драйверами всех устройств, которые могут повстречаться программе на разных компьютерах. Например, для того чтобы приложение MS-DOS могло выводить данные на принтеры любого типа, программист должен был позаботиться о том, чтобы программа содержала драйверы принтеров всех известных марок и типов. Поэтому при написании программы уйма времени уходила на переписывание практически одинаковых программ драйверов. Скажем, один вариант драйвера принтера LaserJet предназначался для приложения Microsoft Word for DOS, другой вариант этого же драйвера — для Microsoft Works и т.д.

В среде Windows драйверы для определенного устройства создаются и устанавливаются только один раз. Причем они могут устанавливаться сразу при установке Windows, поступать вместе с программными продуктами или добавляться самим пользователем.

Аппаратная независимость приложений в Windows реализуется за счет того, что приложения не обмениваются данными с устройствами напрямую, а используют в качестве посредника Windows. Для приложения нет необходимости знать, какой именно принтер подключен к данному компьютеру. Программа передает команду Windows напечатать прямоугольную область с заливкой, после чего уже операционная система разбирается, как реализовать эту задачу на имеющемся оборудовании. Аппаратная независимость облегчает жизнь не только программистам, но и пользователям, поскольку избавляет их от необходимости выяснять, с какими типами внешних устройств может работать то или иное приложение.

Аппаратная независимость достигается также за счет определения минимального набора базовых операций, которые должны выполняться любыми устройствами. Этот минимальный набор включает элементарные операции, необходимые для правильного решения любых задач, которые могут быть поставлены перед данным устройством. Какой бы сложности ни была задача, она может быть сведена к последовательности элементарных операций, входящих в набор минимальных требований к устройству. Например, далеко не все графопостроители обладают функцией рисования окружностей. Тем не менее, даже если выведение окружностей не входит в набор функций графопостроителя, вы все равно можете создавать программы, которые могут ставить перед графопостроителем подобную задачу. И поскольку, в соответствии с минимальными требованиями, все графопостроители, устанавливаемые в Windows, должны уметь рисовать линии, Windows автоматически преобразует окружность в набор линий и передаст массив векторов графопостроителю.

Чтобы избежать конфликтов при вводе данных, в Windows также определен минимальный набор кодов клавиш, которые должны распознаваться любым приложением. Стандартный набор кодов в целом соответствует раскладке ПК-совместимой клавиатуры. Если какая-нибудь фирма выпустит клавиатуру с дополнительными клавишами, не входящими в стандартный набор, то она должна позаботиться о специальном программном обеспечении, преобразующем коды этих клавиш в последовательности стандартных кодов, распознаваемых Windows. Минимальному набору кодов должны соответствовать команды, поступающие не только от клавиатуры, но и от всех других устройств ввода, в том числе от мыши. Таким образом, если какая-нибудь фирма выпустит четырехкнопочную мышь, то это не должно вас беспокоить, поскольку производитель сам позаботится о том, чтобы команды от дополнительных кнопок соответствовали стандартным кодам Windows.

Библиотеки динамической компоновки

Функциональные возможности Windows в большой мере обеспечиваются за счет использования библиотек динамической компоновки (DLL). В частности, благодаря этим библиотекам к ядру операционной системы добавляется графический пользовательский

интерфейс. DLL-файлы содержат функции, которые подключаются к программе во время ее выполнения (динамически), а не во время компиляции (статически).

Динамическая компоновка программ с внешними файлами, содержащими часто используемые процедуры, не является чем-то уникальным для Windows. Этот принцип широко применяется в языках C/C++, где все стандартные функции предоставляются внешними библиотеками. Компоновщик автоматически копирует из библиотек и вставляет в исполняемый файл такие функции, как `getchar()` и `printf()`. Благодаря использованию подобного подхода программист избавляется от необходимости переписывать каждый раз базовые функции, чем достигается стандартность выполнения таких операций, как считывание символов, вводимых с клавиатуры, и форматирование данных, выводимых на печать. Программист легко может создать собственную библиотеку часто задеиствуемых функций, например функций изменения шрифта и выравнивания текста. Предоставление доступа к функциям и методам как к глобальным инструментам среды программирования обеспечивает возможность повторного использования кода — ключевая концепция объектно-ориентированного программирования.

Все библиотеки Windows компонуются динамически. То есть функции из этих библиотек не копируются в исполняемые файлы, а вызываются во время выполнения программы, что позволяет экономить ресурсы памяти. Не важно, сколько приложений одновременно запущено, — в ОЗУ хранится только одна копия библиотеки, используемая всеми приложениями.

Если в программе вызывается стандартная функция Windows, то компилятор должен сгенерировать по месту вызова машинный код, содержащий обращение по адресу, находящемуся в другом сегменте кода. Такое положение представляет собой определенную проблему, поскольку до тех пор, пока программа не будет запущена, невозможно угадать, каким будет адрес библиотечной функции. Решение этой проблемы в Windows называется отложенным связыванием, или динамической компоновкой. Начиная с Windows 3.0 и Microsoft C 6.0, компоновщик позволяет обращаться к функциям, адреса которых не известны в момент компоновки. Окончательная компоновка функции происходит лишь после того, как программа запускается и загружается в память.

В компиляторах C/C++ используются специальные импортируемые библиотеки, назначение которых состоит в подготовке приложений к динамической компоновке в среде Windows. Например, в библиотеке USER32.LIB перечислены все стандартные функции Windows, к которым можно обращаться в программах. В записях этой библиотеки определяются модули Windows, где хранятся соответствующие функции, а также, в большинстве случаев, порядковый номер функции в этом модуле.

Например, многие приложения Windows вызывают функцию `PostMessage()`. Во время компиляции программы компоновщик получает из файла USER32.LIB информацию о модуле и номере функции `PostMessage()` и встраивает эти данные в код программы. При запуске программы Windows считывает имеющуюся информацию и связывает программу с реальным кодом функции `PostMessage()`.

Формат исполняемых файлов

Для Windows был разработан новый формат исполняемых файлов. В частности, изменения коснулись заголовка файла, который теперь может содержать информацию об импортируемых функциях библиотек динамической компоновки. Чаще всего используются функции модулей KERNEL, USER и GDI, содержащих множество подпрограмм, связанных с различными рутинными операциями, в том числе с обработкой сообщений. Функции, хранящиеся в DLL-модулях, еще называют экспортируемыми. В соответствии с новым форматом исполняемых файлов, экспортируемые функции определяются по имени модуля и порядковому номеру функции в нем. Все DLL-файлы содержат таблицу точек входа, в которой перечислены адреса всех экспортируемых функций,

С точки зрения приложений, эти же функции называются импортируемыми. Импорт функций осуществляется посредством рассмотренного выше механизма отложенного связывания и всевозможных таблиц компоновки. Обычно приложения Windows включают не менее одной экспортируемой функции, называемой оконной процедурой и отвечающей за прием сообщений.

Другая особенность нового формата исполняемых файлов состоит в том, что с каждым сегментом кода и данных любого приложения и библиотеки связывается дополнительная информация. Обычно сегменты кода помечаются специальными флагами как перемещаемые и выгружаемые, а сегменты данных — как перемещаемые. В соответствии с установленными

флагами, Windows может автоматически перемещать загруженные сегменты в память и даже временно выгружать их на диск, если для выполнения других программ в многозадачной среде требуется дополнительная память. Впоследствии, когда выгруженные сегменты вновь понадобятся для выполнения программы, Windows автоматически загрузит их обратно. Другой интересной особенностью является загрузка по вызову. При использовании данного механизма сегмент кода загружается в память только при условии, что в программе встречается вызов функции, находящейся в этом сегменте. Благодаря такого рода средствам в Windows можно работать с несколькими приложениями одновременно, даже если свободной памяти хватает лишь для одного приложения.

Базовые концепции программирования

Особенность создания приложений в среде Windows заключается в том, что здесь применяются специальные методики программирования и используется своя терминология. Последнюю можно разделить на две большие категории: терминология, связанная с пользовательским интерфейсом (меню, диалоговые окна, значки и т.д.), и терминология, относящаяся непосредственно к программированию (сообщения, вызовы функций и т.д.). Чтобы вы могли эффективно общаться и легко понимать друг друга, необходимо внимательно изучить все описанные ниже термины и понятия.

Что представляет собой окно

Окно — это специальная прямоугольная область экрана. Все элементы окна, его размер и внешний вид контролируются открывающей его программой. Каждый щелчок пользователя на каком-нибудь элементе окна вызывает ответные действия приложения. Многозадачность в Windows заключается, в частности, в возможности одновременно открывать окна сразу нескольких приложений или нескольких окон одного приложения. Активизируя с помощью мыши или клавиатуры то или иное окно, пользователь дает системе понять, что последующие команды и данные следует направлять именно этому окну.

Компоненты окна

Стандартный внешний вид окон всех приложений Windows и предсказуемость работы различных их компонентов позволяют пользователям чувствовать себя уверенно с новыми приложениями и легко разбираться в принципах их работы. Основные компоненты любого окна показаны на рис. 16.5.

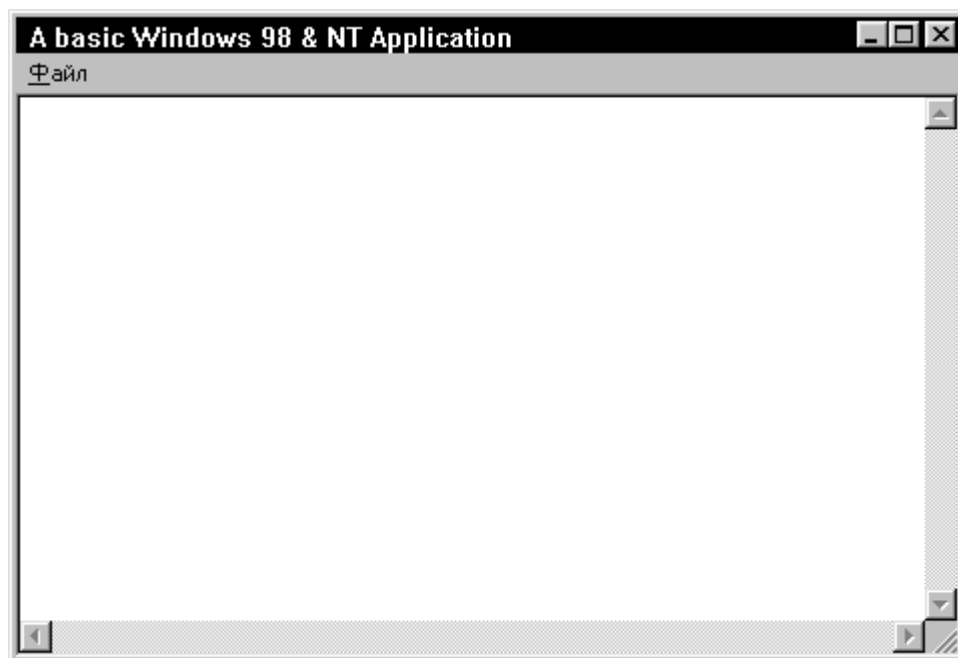


Рис. 16.5. Стандартное окно приложения Windows

Рамка

Обычно каждое окно заключается в небольшую рамку. Новичку может показаться, что функции рамки сводятся только к отделению окна от остальных частей экрана. Но в действительности это не так. Как правило, рамка является и средством масштабирования. Размер окна приложения можно изменить, если поместить указатель мыши на рамку и перетащить его, удерживая нажатой левую кнопку мыши.

Строка заголовка

Имя приложения, которому принадлежит открытое окно, отображается в строке заголовка в верхней части окна. Строка заголовка является обязательным элементом всех окон приложений и позволяет пользователю легко определить, какому приложению принадлежит какое окно, если в системе запущено сразу несколько приложений. Строка заголовка активного окна выделяется альтернативным цветом, чтобы его легко можно было отличить от неактивных окон.

Значок приложения

Другим обязательным элементом любого окна является расположенный в его верхнем левом углу значок приложения. Этот значок обычно представляет собой маленький логотип приложения. Щелчок на значке приводит к открытию системного меню.

Системное меню

Системное меню открывается при щелчке мышью на значке приложения. В нем представлены стандартные команды управления окном: **Restore**(Восстановить), **Move**(Переместить), **Size**(Размер), **Minimize**(Свернуть), **Maximize**(Развернуть) и **Close**(Закрыть).

Кнопка свертывания

В правом верхнем углу большинства окон приложений имеется три кнопки. Крайняя левая кнопка предназначена для свертывания окна в кнопку на панели задач.

Кнопка развертывания/восстановления

Средняя кнопка в правом верхнем углу либо разворачивает окно на весь экран, либо восстанавливает прежние его размеры, если окно уже развернуто.

Кнопка закрытия

Крайняя правая кнопка в углу предназначена для закрытия приложения. После закрытия окна активным автоматически становится окно следующего приложения.

Вертикальная полоса прокрутки

В некоторых случаях окно приложения может содержать вертикальную полосу прокрутки, которая располагается по правому краю окна. В верхней и нижней частях полосы находятся кнопки со стрелками, направленными соответственно вверх и вниз. Вдоль самой полосы располагается бегунок. Положение бегунка показывает, какая часть окна или документа отображается в данный момент на экране. Перемещая бегунок, можно выбрать нужную часть многостраничного документа. Щелчок мышью на кнопке со стрелкой приведет к смещению содержимого окна на одну строку вверх или вниз, а щелчок на свободном пространстве выше или ниже бегунка — на одну экранную страницу вверх или вниз.

Горизонтальная полоса прокрутки

Окно может также быть оснащено горизонтальной полосой прокрутки, которая размещается по нижнему краю окна и работает аналогично вертикальной полосе прокрутки. Горизонтальная полоса предназначена для выведения на экран частей документов, состоящих из большого числа столбцов. Щелчок мышью на кнопках со стрелками приводит к смещению содержимого окна на один столбец влево или вправо. Щелчок на областях между стрелками и бегунком смещает изображение на одну экранную страницу влево или вправо.

Строка меню

В окнах большинства приложений сразу под строкой заголовка находится строка меню, содержащая наборы команд и опций программы. Обычно для выбора команд меню

используется мышь, хотя эти действия можно выполнить и с помощью клавиатуры. Каждому элементу меню, как правило, соответствует клавиша быстрого вызова, выделенная подчеркиванием в названии элемента. Чтобы выбрать данный элемент, нужно нажать соответствующую клавишу в сочетании с клавишей [Alt]. Так, комбинация клавиш [Alt+F] открывает меню **File**.

Рабочая область

Рабочая область обычно занимает большую часть окна. Именно в эту область программа выводит результаты своей работы.

Классы окон

Чтобы два окна одной и той же программы выглядели и работали совершенно одинаково, они оба должны базироваться на общем классе окна. В приложениях, написанных на C, где используются традиционные вызовы функций, класс окна регистрируется программой в процессе инициализации. При создании окна класс указывается в качестве параметра функции `CreateWindow()`. Зарегистрированный новый класс становится доступным для всех программ, запущенных в данный момент в системе. В случае использования библиотеки MFC вся работа по регистрации классов окон выполняется автоматически, что существенно облегчает работу программиста.

Благодаря тому, что окна приложения создаются на основе общего базового класса, значительно сокращается объем информации, которую при этом следует указывать. Поскольку класс окна содержит в себе описания элементов, общих для всех окон данного класса, нет необходимости повторять эти описания при создании каждого нового окна. К тому же все окна одного класса используют одну оконную процедуру, что также позволяет избежать дублирования кода.

Графические объекты, используемые в окнах

Примерами графических объектов, с которыми можно обращаться как с единым целым и которые выступают элементами пользовательского интерфейса, являются строка меню, полосы прокрутки, кнопки и т.д. Наиболее широко используемые графические объекты Windows описаны ниже.

Значки

Значками называются маленькие графические изображения, выполняющие опознавательную функцию. Так, значки приложений на панели задач позволяют легко определить, какие программы в настоящий момент запущены, даже если названия программ не отображаются целиком. Значки могут быть очень полезны в самих приложениях, поскольку с их помощью можно привлекать внимание пользователей к сообщениям об ошибках и различным предупреждениям. В Windows входит набор стандартных значков, в частности стилизованные знак вопроса и восклицательный знак, а также ряд других. С помощью встроенного в компилятор Visual C++ редактора ресурсов вы можете создавать собственные значки.

Указатели мыши

Указатели мыши также являются графическими объектами, используемыми для отслеживания перемещения мыши. Вид указателя может меняться в зависимости от выполняемого задания и состояния системы. Например, стандартный указатель-стрелка меняет свой вид на изображение песочных часов в том случае, если система занята. С помощью встроенного в компилятор Visual C++ редактора ресурсов вы можете создавать собственные указатели мыши.

Текстовые курсоры

Курсоры предназначены для указания места, куда следует осуществлять ввод текстовых данных. Отличительной особенностью курсоров является их мерцание. Поведение курсоров напоминает работу курсоров в MS-DOS. В большинстве текстовых редакторов и в полях диалоговых окон в качестве курсора применяется 1-образный указатель мыши. Причем в Windows нет (в отличие от значков и указателей) коллекции готовых курсоров.

Окна сообщений

Окна сообщений представляют собой разновидность диалоговых окон, содержащих строку заголовка, значок и текст сообщения. На рис. 16.6 показано стандартное окно сообщения, которое появляется при закрытии окна программы Notepad в том случае, если в нем содержатся несохраненные данные.

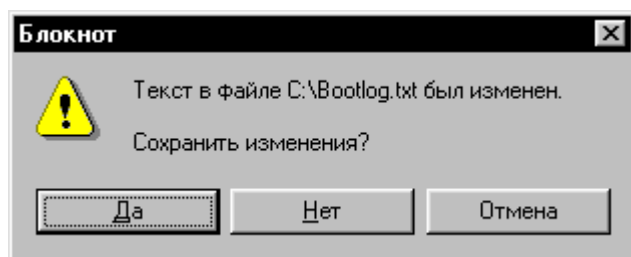


Рис. 16.6. Типичное окно сообщения

В приложении указывается текст заголовка окна, текст сообщения, какой из стандартных значков Windows использовать (если это необходимо) и какой набор кнопок выводить. В частности, можно вызывать окна с такими комбинациями кнопок: Abort/Retry/Ignore, OK, Yes/No, Yes/No/Cancel, OK/Cancel и Retry/Cancel. Обычно в окнах сообщений отображаются такие стандартные значки, как IconHand, IconQuestion, IconExclamation, IconAsterisk и некоторые другие.

Диалоговые окна

Диалоговые окна содержат наборы различных элементов управления и предназначены для предоставления пользователю возможности устанавливать опции и параметры программы, которой принадлежит данное окно. Внешний вид диалогового окна разрабатывается с помощью редактора ресурсов компилятора Visual C++.

Шрифты

Шрифт в Windows — это графический ресурс, содержащий набор символов определенного типа. Существует набор функций, с помощью которых можно манипулировать начертанием символов для получения форматированного текста. В приложениях можно использовать различные стандартные шрифты, включая System, Courier и TimesNewRoman, а также пользовательские шрифты. Встроенные функции позволяют на базе основного шрифта получать полужирное начертание, курсив, подчеркнутый текст, изменять размер шрифта. Причем внешний вид шрифта не зависит от типа устройства, на которое выводится текст. Благодаря внедрению технологии TrueType было достигнуто, начиная с Windows 3.1, соответствие между внешним видом шрифта на экране и шрифта, выводимого при печати.

Точечные рисунки

Точечные рисунки представляют собой точную копию части экрана, снятую попиксельно. Тот факт, что изображение является точным образом экрана, устраняет необходимость в каких бы то ни было дополнительных преобразованиях, что существенно сокращает время вывода изображения на экран. В Windows точечные рисунки наиболее широко применяются для двух целей. Во-первых, они служат изображениями всевозможных кнопок и значков, например стрелок полос прокрутки и кнопок панелей инструментов. Другой областью применения точечных рисунков являются кисти, с помощью которых рисуются и заполняются цветом различные геометрические фигуры на экране.

Точечные рисунки можно создавать и модифицировать с помощью встроенного редактора ресурсов.

Перья

Перья предназначены для рисования геометрических фигур и различных контуров и характеризуются тремя основными параметрами: ширина линии, стиль (точечный, штрихпунктирный, непрерывный) и цвет. Существует два готовых пера: одно для рисования черных линий, другое — для рисования белых. С помощью специальных функций вы можете создавать собственные перья.

Кисти

Кисти предназначены для заливки объектов цветом, выбранным из заданной палитры. Минимальный размер кисти — 8x8 пикселей. Кисть также характеризуется тремя параметрами: размером, шаблоном заливки и цветом. Заливка может быть сплошной, штриховой, диагональной или представлять собой узор, заданный пользователем.

Принципы обработки сообщений

Как вы уже знаете, ни одно приложение в Windows не отображает свои данные непосредственно на экране. Точно так же ни одно приложение не обрабатывает напрямую прерывания устройств и не выводит данные непосредственно на печать. Вместо этого приложение вызывает встроенные функции Windows и ожидает от системы соответствующих сообщений.

Подсистема сообщений в Windows — это средство распределения информации в многозадачной среде. С точки зрения приложения, сообщение — это уведомление о некотором произошедшем событии, на которое оно, приложение, должно ответить определенным образом. Такое событие может быть инициировано пользователем, скажем путем нажатия клавиши или перемещения мыши, изменением размера окна или выбором команды из меню. Но события могут порождаться и самим приложением.

Особенность этого процесса состоит в том, что приложение должно быть полностью ориентировано на прием и обработку сообщений. Программа должна быть готова в любой момент принять сообщение, определить его тип, выполнить соответствующую обработку и вновь перейти в режим ожидания до поступления следующего сообщения.

Приложения Windows существенно отличаются от приложений, написанных для MS-DOS. Windows открывает приложениям доступ к сотням встроенных функций, которые можно вызывать напрямую или косвенно, посредством библиотек типа MFC. Эти функции содержатся в ряде модулей, таких как KERNEL, GDI и USER. Функции модуля KERNEL отвечают за управление памятью, загрузку и выполнение приложений, а также за распределение системных ресурсов. Модуль GDI содержит функции создания и отображения графических объектов. Модуль USER отвечает за выполнение всех других функций, обеспечивающих взаимодействие приложений с пользователями и средой Windows.

Ниже мы более детально проанализируем работу подсистемы сообщений, изучим формат сообщений и источники их появления, а также разберем некоторые механизмы организации взаимодействия между приложениями и Windows посредством сообщений.

Формат сообщений

Сообщения используются для информирования приложения о том, что в системе произошло то или иное событие. На практике сообщение направляется не столько самому приложению, сколько определенному окну, открытому этим приложением.

Реально в Windows существует только один механизм обработки сообщений — системная очередь сообщений. Но каждое выполняющееся приложение организует и свою очередь. Функции модуля USER, в частности, ответственны за передачу сообщений из системной очереди в очередь конкретного приложения. В последней накапливаются сообщения, адресованные любому окну, открытому данным приложением.

Независимо от типа все сообщения характеризуются четырьмя параметрами: дескриптором окна, которому адресуется данное сообщение, типом сообщения и еще двумя 32-разрядными параметрами.

Примечание

Речь идет о параметрах сообщений в 32-разрядных версиях Windows. Для Windows 3.x характерны другие параметры.

Дескрипторы широко используются в приложениях Windows. Напомним, что дескриптором называется уникальный номер, который присваивается всем системным объектам, таким как окна, элементы управления, меню, значки, перья и кисти, а также областям памяти, устройствам вывода и т.д.

Поскольку Windows позволяет одновременно открывать несколько копий одного приложения, операционная система должна иметь возможность отслеживать каждую копию в отдельности. Это достигается путем присвоения каждому экземпляру программы дескриптора.

Дескрипторы обычно служат в качестве индексов системной таблицы объектов. Благодаря тому что доступ к объектам осуществляется по индексам таблицы, а не по их непосредственным адресам в памяти, Windows может динамически перераспределять ресурсы за счет обновления адресов в таблице. Например, если Windows связала некоторый ресурс приложения с 16-й строкой таблицы, то независимо от того, куда Windows переместит впоследствии этот ресурс, его текущий адрес всегда будет представлен в 16-й строке.

Вторым параметром, передаваемым с каждым сообщением, является его тип. Тип сообщения задается идентификатором, который определен в одном из файлов заголовков Windows. Для работы с идентификаторами сообщений в программу включается файл `WINDOWS.H`. Как правило, идентификаторы начинаются с двухсимвольного префикса, за которым следует символ подчеркивания. Так, оконные сообщения начинаются с префикса `m_`: `wm_create`, `wm_paint`, `wm_close`, `wm_copy`, `wm_paste`. Сообщения кнопок имеют префикс `BM_`, полей — `EM_`, списков — `LB_`. В приложении можно также создать и зарегистрировать собственный тип сообщения, предназначенного для частных целей.

Последние два параметра сообщения несут дополнительную информацию. Их содержание может изменяться в зависимости от типа сообщения. Например, посредством этих параметров может передаваться информация о том, какая клавиша была нажата, где находился указатель мыши или бегунок полосы прокрутки и т.д.

Генерирование сообщений

Именно реализация концепции обмена сообщениями позволяет Windows быть многозадачной средой. Работа Windows основана на передаче, приеме и обработке сообщений. Существует четыре основных источника, от которых приложение может получить сообщение: пользователь, Windows, само приложение, другие приложения.

Сообщения пользователей включают информацию о вводе текста, перемещении и нажатии кнопок мыши, выборе команд меню, перемещении бегунка полосы прокрутки и т.д. Большую часть времени приложение занято обработкой именно таких сообщений. Получение сообщения от пользователя означает, что человек, запустивший программу, хочет изменить ход ее выполнения.

Системные сообщения посылаются программе при изменении ее состояния. Например, щелчок на значке приложения означает, что пользователь хочет сделать данное приложение активным. В этом случае Windows сообщает приложению, что на экране открывается его окно, размер и положение окна изменяются и т.д. В зависимости от текущего состояния приложения сообщение от Windows может быть принято и обработано либо проигнорировано.

В следующей главе мы поговорим о том, как создаются простейшие приложения Windows. Вы поймете, что, по сути, приложение разбивается на ряд процедур, каждая из которых отвечает за обработку определенного типа сообщений для определенного окна. Например, одна из процедур может отвечать за изменение размеров окна. Причем совсем не обязательно, чтобы это происходило только по команде пользователя, — решение об изменении может принимать и сама программа.

Сообщения четвертого типа в настоящее время применяются достаточно редко. Примером межзадачного обмена сообщениями может служить протокол DDE (DynamicDataExchange—динамический обмен данными).

Обработка сообщений

Традиционные приложения Windows, написанные на C/C++, содержат специальные процедуры для обработки всех типов сообщений, которые могут быть посланы программе. Разные окна программы могут по-разному реагировать на одни и те же сообщения. Это достигается за счет того, что обработчики сообщений пишутся отдельно для каждого окна, но Windows знает, какому именно окну адресовано сообщение. Таким образом, приложения содержат процедуры обработки не только сообщений разных типов, но и сообщений для разных окон.

Цикл обработки сообщений

Основным компонентом любого приложения Windows является цикл обработки сообщений. В процедурном приложении местонахождение этого цикла нетрудно определить. В программах, написанных с использованием библиотеки MFC, цикл обработки сообщений скрыт в классе `cWinApp`.

Если говорить в целом, работа приложения организуется следующим образом. Приложение сначала создает и открывает свои окна, затем запускается цикл обработки сообщений и, в конце концов, при получении сообщения типа `WM_CLOSE`, работа программы завершается. Цикл обработки сообщений ответственен за передачу поступающих от Windows сообщений соответствующим оконным процедурам.

На последовательность обработки сообщений влияют два фактора: очередь, в которой находится сообщение, и приоритет его самого. Сообщения могут поступать из двух очередей: системной и очереди приложения. Но даже если сообщение поступило от самого приложения, оно все равно будет помещено в системную очередь. Когда подойдет черед сообщения, оно будет направлено в очередь соответствующего приложения. Такая организация работы системной очереди позволяет Windows контролировать прохождение всех сообщений и ограничивает ответственность приложений обработкой только тех сообщений, которые относятся непосредственно к ним.

Сообщения обычно помещаются в очередь по принципу FIFO (*first in, first out* — первым поступил, первым обслужен). Такие сообщения называются синхронными. Но иногда Windows вставляет сообщение сразу в голову очереди. Сообщения такого типа, изменяющие нормальный ход выполнения программы, называются асинхронными.

Существует несколько видов асинхронных сообщений, среди них — сообщения о перерисовке, сообщения таймера и сообщения о завершении программы. Например, сообщение таймера может запускать некоторое действие в определенный момент времени независимо от того, какие сообщения сейчас находятся в очереди; оно имеет наивысший приоритет и передается раньше всех других сообщений.

У вас может возникнуть вопрос, как Windows выходит из положения в том случае, если сообщения поступают одновременно нескольким приложениям. Эта проблема решается одним из двух способов. Первый состоит в задании приоритетов. Когда загружается некоторое приложение, для него автоматически устанавливается нулевой приоритет. В процессе работы приложения его приоритет может быть изменен. Любые конфликты разрешаются в пользу того приложения, чей приоритет выше.

Изменять приоритет приложения, заданный по умолчанию, обычно не рекомендуется. Тем более что в распоряжении Windows есть еще один метод, определяющий очередность передачи сообщений группе приложений с одинаковым приоритетом. Если Windows видит, что у приложения образовался длинный список необработанных сообщений, то она автоматически приостанавливает передачу этому приложению новых сообщений, хотя продолжает передавать сообщения другим, более свободным приложениям.

Вызов системных функций

Как известно, Windows содержит сотни встроенных функций, таких как `DispatchMessage()`, `PostMessage()`, `RegisterWindowMessage()`, `SetActiveWindow()` и т.д. При создании приложений на C++ с использованием библиотеки MFC большинство этих функций инкапсулируются в базовых классах и выполняются автоматически.

Соглашение о вызове функций

В Windows 3.x описания функций включали спецификатор PASCAL, который в 32-разрядных версиях Windows не используется. Параметры функций здесь передаются посредством системных стеков и считываются в обычном для C/C++ режиме — в направлении Справа налево. После завершения функции вызвавшая ее процедура должна модифицировать указатель стека с учетом количества байтов, занимаемых параметрами функции.

Файл WINDOWS.H

Файл заголовков `WINDOWS.H` открывает доступ к тысячам описаний констант, структур, типов данных и прототипов функций. Он включается в большинство приложений Windows и содержит директивы включения множества других файлов заголовков. (Одна из причин, почему приложения Windows компилируются так долго, состоит не только в относительно больших размерах самих программ, но и в том, что к ним подключаются большие файлы заголовков.) В

случае использования библиотеки MFC файл WINDOWS.H подключается к программе косвенно, Через файл AFXWIN.H.

Обычно в директивах #define, встречающихся в файлах заголовков, за текстовым идентификатором следует числовая константа. Например:

```
#define WM_CREATE 0x0001
```

Для VisualC++ данная запись означает, что на этапе компиляции идентификатор сообщения WM_CREATE следует заменить шестнадцатеричной константой 0x0001. Некоторые директивы #define могут выглядеть довольно необычно:

```
#define NEAR near
```

```
#define FAR far
```

Данные команды позволяют использовать в приложении ключевые слова near и far, набираемые как в верхнем, так и в нижнем регистре, что дает программе ряд преимуществ. Не стоит забывать, что проще изменить объявления в файле заголовков, чем отыскивать и менять ключевые слова по всей программе.

Этапы создания приложения

Разработку приложений Windows можно разбить на несколько общих этапов. Перечислим их.

- Создание функции winMain() и других базовых функций. При использовании MFC эти действия выполняются автоматически в классе cWinApp.
- Создание меню и других необходимых ресурсов, включение их в файл сценария ресурсов.
- Создание с помощью редактора ресурсов уникальных указателей мыши, значков и других растровых изображений. (Не обязательный.)
- Создание с помощью редактора ресурсов дополнительных диалоговых окон. (Не обязательный.)
- Компиляция проекта.

Создание ресурсов приложений средствами VisualC++

Компилятор VisualC++ содержит ряд встроенных редакторов ресурсов. Чтобы открыть список доступных редакторов, в меню **Insert** нужно выбрать команду **Resource**. Редакторы ресурсов позволяют быстро создавать и изменять собственные значки, указатели мыши, точечные рисунки, диалоговые окна и многое другое.

Каждый ресурс представляет собой блок данных, который добавляется в исполняемый файл приложения. Но если разобраться в технических деталях, то окажется, что эти данные размещаются не в сегменте данных программы. Когда программа загружается в память для выполнения, ресурсы остаются на диске. В качестве примера приведем ситуацию, когда пользователь первый раз вызывает окно About. Прежде чем Windows сможет отобразить это окно, система должна обратиться к жесткому диску и загрузить соответствующие данные из выполняемого файла в память.

Если в приложении планируется использовать такие ресурсы, как диалоговые окна, меню и значки, то каждый из них предварительно должен быть описан в файле сценария ресурсов. Этот файл создается, модифицируется и дополняется с помощью редакторов ресурсов, о которых говорилось выше. Посредством компилятора ресурсов файл сценария преобразуется в файл скомпилированных ресурсов, данные из которого вставляются затем в исполняемый файл приложения. Такой метод обработки ресурсов позволяет приложениям хранить информацию обо всех своих ресурсах непосредственно в исполняемом файле.

Файлы проектов

Файлы проектов служат средством контроля за ходом компиляции ресурсов и программного кода приложения. Они позволяют отслеживать взаимосвязь программы и исходных файлов, учитывать время создания файлов.

В файлах проектов содержится информация о компиляции и компоновке текущего приложения с учетом подключаемых библиотек, аппаратной платформы и системной среды. В большинстве случаев с целью сокращения времени создания нового проекта можно выбрать один из готовых шаблонов, предлагаемых компилятором VisualC++.

С помощью файлов проектов можно также организовать ступенчатую компиляцию и компоновку, когда перекомпиляции подвергаются только те части программы, которые были модифицированы. Это позволяет существенно сократить время построения исполняемого файла.

Редакторы ресурсов

В VisualC++ встроен целый набор редакторов ресурсов, каждый из которых ориентирован на определенный тип ресурсов. В следующих параграфах вы узнаете, как с их помощью создавать значки, указатели мыши, меню и диалоговые окна.

Значки, указатели мыши и точечные рисунки

Хотя ресурсы данного типа разрабатываются с помощью разных редакторов, в основу их создания положен один принцип, так как все они относятся к растровым изображениям. Отличие состоит лишь в том, что редакторы значков и указателей мыши позволяют создавать аппаратно-независимые изображения — в том смысле, что их внешний вид не будет зависеть от разрешения монитора.

Так, значок может иметь четыре описания: одно — для черно-белого монитора, другое — для CGA-системы, третье — для EGA, четвертое — для VGA. Обращение к значку во всех случаях производится одинаково — по имени. При этом Windows автоматически выбирает то описание, которое в наибольшей степени соответствует реальному типу монитора, установленного на данном компьютере. Окно редактора в процессе создания нового значка показано на рис. 16.8.

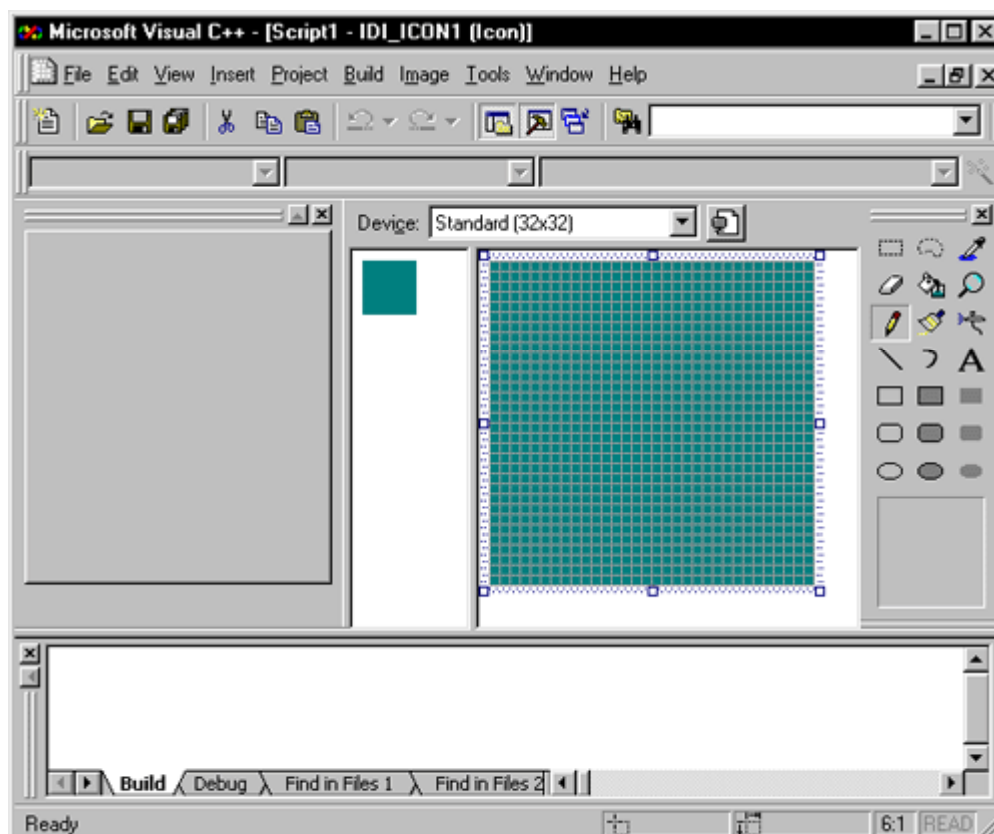


Рис. 16.8. Окно разработки нового значка

В центре окна находится область редактирования, где можно создать значок, указатель мыши или точечный рисунок. Изначально размер этой области составляет 32x32 ячейки. В левой

части окна находится область просмотра. Здесь пользователь может увидеть, как изображение будет выглядеть в системе.

Создание пользовательских значков и указателей мыши

Создать собственный значок или указатель мыши очень просто. Перейдите в меню **Insert** и выберите команду **Resource**. Теперь выберите из списка необходимый вам тип ресурса (**Icon**, **Cursor**), в результате чего откроется пустая область редактирования.

Редактор предоставляет в ваше распоряжение панель инструментов рисования и палитру цветов. Чтобы сделать нужный цвет активным, достаточно щелкнуть мышью на образце этого цвета в палитре. Нарисовав изображение, сохраните его, выбрав в меню **File** команду **Save** или **Save As**.

На рис. 16.9 приведено окно редактора ресурсов, в котором только что был создан новый указатель. Большое изображение в центре показывает точечную структуру рисунка, тогда как маленькое изображение слева отображает рисунок в том виде, в каком он в действительности будет представлен на экране.

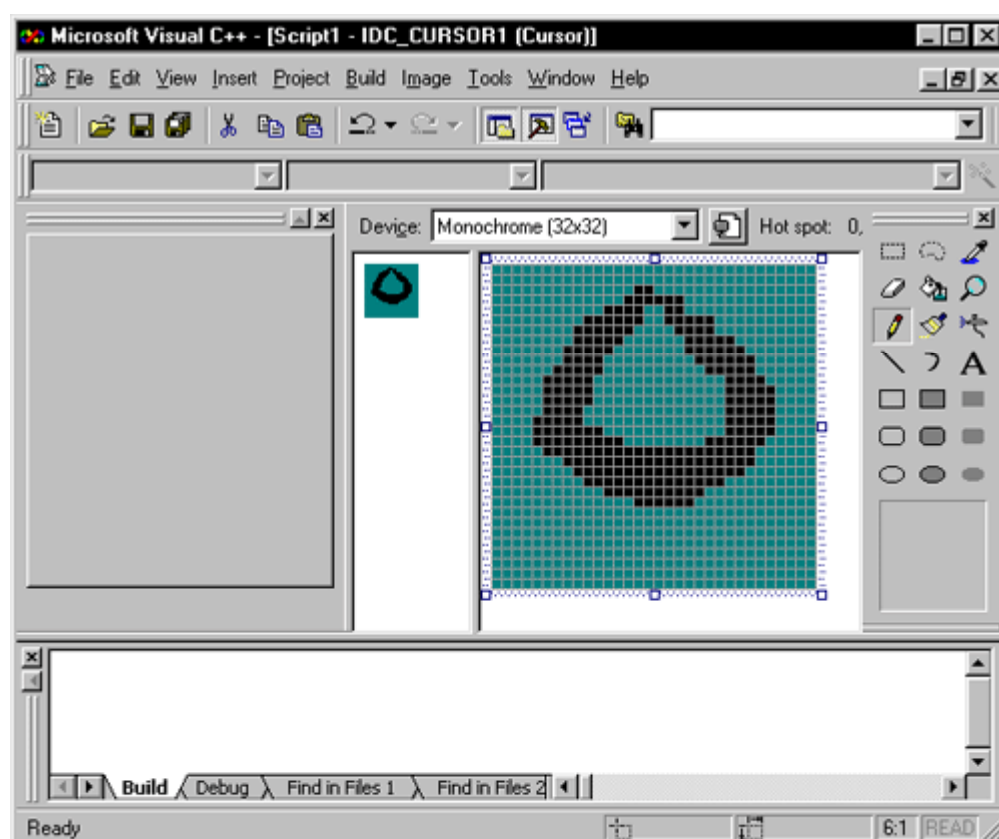


Рис. 16.9. Окно создания указателя мыши

Когда вы первый раз выберете в меню **File** команду **Save**, редактор предложит вам ввести имя файла. Если вы создавали значок, то система по умолчанию предложит для файла расширение **ICO**. Расширение **CUR** используется для указателей мыши.

С Помощью кнопки **SetHotspot** вы можете выбрать в создаваемом указателе мыши активную точку, по положению которой определяются координаты указателя на экране. Щелкните на кнопке **SetHotspot**, наведите появившийся указатель в виде маленького перекрестия на тот пиксель изображения, который вы хотите сделать активной точкой, и просто щелкните мышью. Слева от кнопки отобразятся выбранные координаты. Любой указатель мыши может иметь только одну активную точку.

Меню

Одним из важнейших элементов приложений Windows является меню, которое представляет простой и понятный способ группировки команд и опций программы. Для выбора команды достаточно щелкнуть на требуемом элементе меню. Некоторые команды приводят к последующему появлению диалоговых окон, содержащих дополнительные наборы опций и команд.

Меню, создаваемое в этом параграфе, будет использовано в следующей главе при разработке готового приложения.

Что такое меню

Меню — это список команд и опций программы. Выбрать элемент меню можно с помощью мыши или различных сочетаний клавиш. В ответ на это Windows пошлет щриложению сообщение с указанием того, какой элемент был задействован.

Создание меню

Разработать меню приложения можно с помощью соответствующего редактора ресурсов, который позволяет проектировать меню в визуальной форме. Альтернативный вариант — использование текстового редактора, в котором набирается описание меню на языке описания ресурсов.

Редактор ресурсов может читать описания меню из файла сценария ресурсов (расширение RC) или файла скомпилированных ресурсов (расширение RES). Первый файл содержит описания ресурсов в текстовом формате. Если имеется файл заголовков с объявлениями констант, используемых в описании меню, этот файл можно включить в файл ресурсов. Например, встречающаяся ниже константа `idm_about` может быть определена в файле заголовков равной числу 40.

Меню `pie_menu`, разработанное с помощью редактора ресурсов, показано на рис. 16.10.

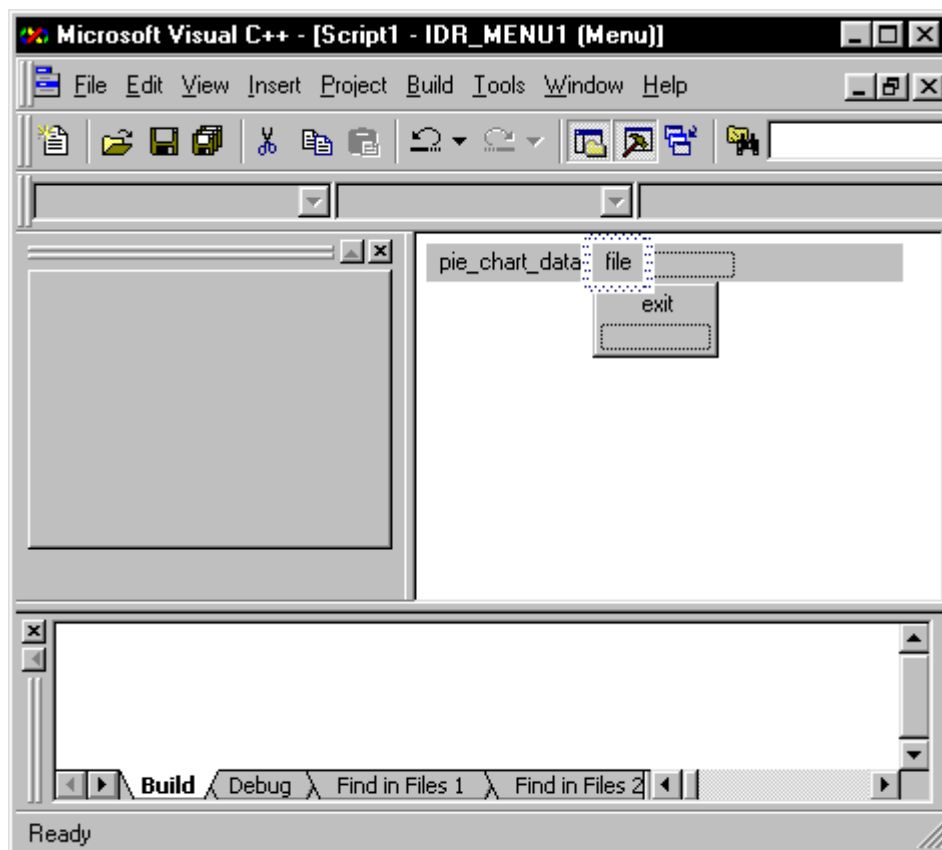


Рис. 16.10. Создание нового меню

В описание меню можно включать всевозможные стили и атрибуты. Например, можно задать маркировку выбранных элементов специальными метками, указать, что определенный пункт меню изначально недоступен (выделен серым цветом) или является простой линией-разделителем и т.д.

Компиляция ресурса меню

На основании RC-файла компилятор ресурсов скомпилирует описание меню в файл ресурсов с расширением RES. Этот файл будет использован компоновщиком для связывания ресурсов с исполняемым файлом приложения.

Описание простейшего меню не сложно понять. Вот как выглядит файл сценария ресурса:

```
PIEMENU MENU DISCARDABLE BEGIN
POPUP "Pie_Chart_Data"
BEGIN
MENUITEM "About...", IDM_ABOUT
      MENUITEM "Input...", IDM_INPUT
MENUITEM "Exit...", IDM_EXIT
END
END
```

Изучая этот листинг, вы заметите ряд ключевых слов, используемых в описаниях меню: `menu` (означает новый ресурс меню), `popup` (подменю) и `menuitem` (команда меню). Вместо операторов `begin` и `end` можно использовать фигурные скобки (`{}`). Нетрудно также понять, какие команды будут представлены в нашем меню: **About...**, **Input...** и **Exit**. Трехточие после названия команды означает, что ее выбор приведет к открытию диалогового окна.

После того как Windows получит от вас все сведения о том, каким вы хотели бы видеть меню своего приложения, она автоматически создаст его в соответствии со своими стандартами и правилами. Таким образом обеспечивается стандартность интерфейсов всех приложений Windows.

Ключевые слова редактора ресурсов

Вернемся еще раз к описанию меню `piemenu`, представленном выше. За именем меню следует ключевое слово `menu`. В данном примере описывается ниспадающее меню `pie_Chart_Data`, которое открывается щелчком на соответствующем пункте в строке меню. Элементы в строке располагаются в порядке их описания. Если элементы не помещаются в одной строке, то автоматически добавляется новая. В текущий момент времени на экране может быть раскрыто только одно подменю.

Для того чтобы связать с элементом меню клавишу быстрого вызова, в его описании перед соответствующей буквой ставится символ амперсанда (&). В нашем случае такие клавиши не задаются, но если бы мы, к примеру, написали `& About. . .`, то данную команду можно было бы выбрать путем нажатия клавиш `[Alt+A]`.

При выборе подменю Windows отображает список находящихся в нем команд, которые были указаны в описании меню с помощью ключевого слова `menuitem`. Справа от названия команды идет ее идентификатор или константа, содержащиеся в одном из включаемых файлов заголовков. `idm` — это общепринятый (но не обязательный) префикс идентификаторов команд меню.

Горячие клавиши

В большинстве приложений для быстрого вызова команд меню используются определенные сочетания клавиш. Представим себе меню, в котором содержится 12 команд, задающих различный цвет фона окна. Пользователю можно предоставить возможность изменять цвет обычным способом, т.е. посредством меню, или — как альтернативный путь — нажатием различных комбинаций клавиш, скажем от `[F1]` до `[F12]`.

Диалоговые окна

Меню предоставляет пользователям простейший способ взаимодействия с программой. Более совершенным методом передачи данных в программу является использование диалоговых окон. В диалоговом окне пользователь может выбирать опции из списка, устанавливать

различные флажки, вводить в поля текст и числовые значения и многое другое. Диалоговые окна не только служат ключевым средством ввода данных в программу, но и существенно облегчают программирование, так как многие операции с элементами управления диалоговых окон в Windows автоматизированы.

Чтобы открыть диалоговое окно, достаточно, как правило, выбрать соответствующую команду меню. Подобные команды обозначаются символами троеточия после их названия. Так, в предыдущем примере таковыми были **About...** и **Input....**

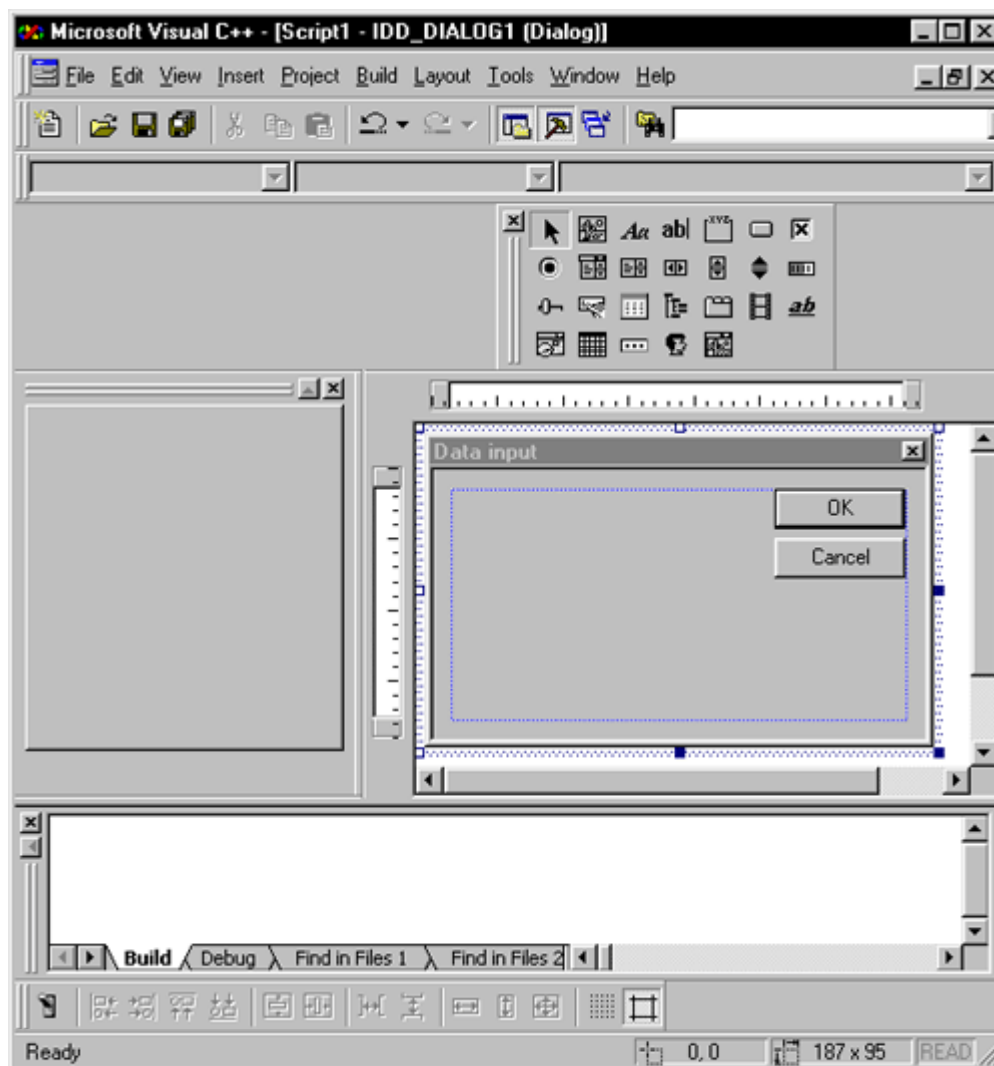


Рис. 16.12. Заготовка диалогового окна

Ниже приведено описание диалогового окна из файла ресурсов.

```
PIEDLGBOX DIALOG DISCARDABLE 93,37,195,159
STYLE DS_MODALFRAME I WS_POPOP I WS_VISIBLE | WS_CAPTION I WS_SYSMENU
CAPTION "Pie Chart Data"
FONT 8, "MS Sans Serif"
BEGIN
    GROOPBOX "Chart Title:",100, 5, 3, 182, 30,WSJTABSTOP
    GROUPBOX "PieWedge Sizes:",101,3, 34,187,95,WSJTABSTOP
    LTEXT "Title:", -1,10,21, 30, 8
    EDITTEXT DMJTITLE, 40, 18,140, 12
    LTEXT "Wedge #1:", -1,10,50,40,8, NOT WS_GROUP
```

```

LTEXT "Wedge #2:", -1,10,65,40,8, NOT WS_GROUP
LTEXT "Wedge #3:", -1,10,80,40,8, NOT WS_GROUP
LTEXT "Wedge #4:", -1,10,95,40,8, NOT WS_GROUP
LTEXT "Wedge #5:", -1,10,110, 40,8, NOT WS_GROUP
LTEXT "Wedge #6:", -1,106,50, 40,8, NOT WS_GROUP
LTEXT "Wedge #7:", -1,106,65,40,8, NOT WS_GROUP
LTEXT "Wedge #8:", -1,106,80, 40,8, NOT WS_GROUP
LTEXT "Wedge #9:", -1,106,95,40,8, NOT WS_GROUP
LTEXT "Wedge #10:", -1,102, 110, 45,8, NOT WS_GROUP
EDITTEXT DM_P1, 55,45,30,12
EDITTEXT DM_P2, 55,60, 30, 12
EDITTEXT DM_P3, 55,75,30,12
EDITTEXT DM_P4, 55,90,30,12
EDITTEXT DM_P5, 55,105, 30,12
EDITTEXT DM_P6, 150, 44,30, 12
EDITTEXT DM_P7, 150, 61,30,12
EDITTEXT DM_P8, 150, 76,30,12
EDITTEXT DM_P9, . 149,91, 30, 12
EDITTEXT DM_P10, 149,106,30,12
PUSHBUTTON "OK", IDOK, 39,135, 24,14
PUSHBUTTON "Cancel", IDCANCEL, 122,136,34,14
END

```

Описания диалоговых окон обычно создаются редактором ресурсов автоматически. Причем редактор может читать описания окон как в текстовом (RC-файл), так и в скомпилированном (RES-файл) виде.

Принципы работы диалоговых окон

Диалоговое окно представляет собой дочернее окно программы, которое появляется при выборе пользователем соответствующей команды из меню. Диалоговые окна разделяются на два основных типа: модальные и немодальные. В большей мере распространены первые. В случае открытия модального диалогового окна все другие окна и команды приложения становятся недоступными до тех пор, пока пользователь не закончит работу с этим окном, обычно щелчком на кнопке **OK** или **Cancel**. При нажатии кнопки **OK** запускается процедура обработки новых данных, введенных пользователем, тогда как нажатие кнопки **Cancel** возвращает программу к исходному состоянию, а все введенные данные отменяются. В Windows с кнопками **OK** и **Cancel** связаны стандартные идентификаторы `idok` и `idcancel` со значениями 1 и 2 соответственно.

Немодальные диалоговые окна больше напоминают обычные окна приложений. Пользователь может свободно переходить между таким диалоговым окном и его родителем. Немодальные диалоговые окна используют в том случае, когда необходимо параллельно работать с несколькими окнами. Примером подобного окна может служить палитра цветов в редакторе растровых изображений.

Разработка диалоговых окон

Существует два способа создания диалогового окна. Если вы нашли описание окна в каком-нибудь листинге, приведенном в книге, то проще всего с помощью текстового редактора ввести это описание в том виде, в каком оно есть, после чего сохранить его в файле сценария ресурсов с расширением RC. Затем с помощью компилятора ресурсов преобразуйте данный файл в RES-файл. Если же вы создаете для своего проекта совершенно новое диалоговое окно, следует использовать соответствующий редактор ресурсов.

Чтобы убедиться в целесообразности использования редактора ресурсов, еще раз обратимся к примеру диалогового окна, описание которого мы рассматривали чуть раньше. Сразу возникает множество вопросов: откуда взялись все эти ключевые слова? что означают эти ряды чисел? как по данному описанию можно представить себе внешний вид диалогового окна? С помощью специальных ключевых Слов и параметров нужно задать размер окна, разместить его на экране, установить в нем необходимые элементы управления и сделать еще много других действий. Вместо того чтобы набирать с клавиатуры непонятные строки, гораздо

проще визуально создать требуемое диалоговое окно, используя графический интерфейс редактора ресурсов. Редактор автоматически проделает всю работу за вас.

Чтобы вызвать редактор диалоговых окон, выберите в меню **Insert** команду **Resource**, а затем в списке **Resource type**— элемент **Dialog**. Начальный вид окна редактора показан на рис. 16.12.

В данный момент в нашем окне содержится первичный макет создаваемого диалогового окна. На рис. 16.13 показан промежуточный этап его редактирования. Расположение кнопок было изменено, как и размер окна.

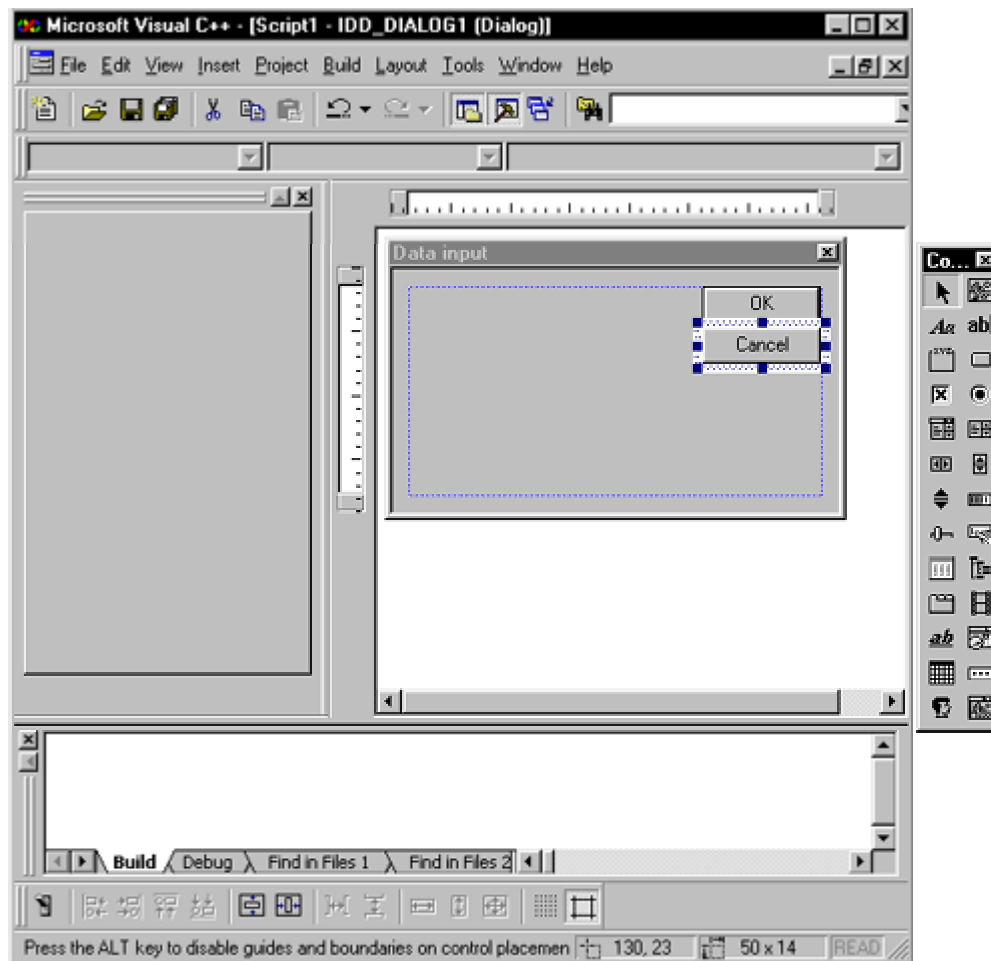


Рис. 16.13. Диалоговое окно в процессе редактирования

Размещение элементов управления

Прежде чем приступить к разработке диалогового окна с помощью редактора ресурсов, следует разобраться в многочисленных элементах управления, которые можно использовать в диалоговых окнах (рис. 16.14).

Ниже перечислены все доступные элементы управления и дано их краткое описание.

- Надпись (StaticText) представляет собой произвольный текст, который можно разместить в любом месте диалогового окна, например возле текстового поля, чтобы указать его назначение.
- Рамка (GroupBox) окружает группу логически связанных элементов управления. В верхнем левом углу рамки автоматически добавляется надпись.

- Флажок (CheckBox) представляет собой маленькое квадратное поле, в котором можно устанавливать и снимать метки. Справа от флажка автоматически добавляется надпись. Обычно используются группы флажков, представляющих наборы взаимосвязанных опций.
- Поле со списком (ComboBox) — это комбинация двух элементов управления: поля и списка. С его помощью пользователь может либо выбирать элемент из списка, либо добавлять в список новый элемент.
- Горизонтальная полоса прокрутки (HorizontalScrollBar) обычно используется для просмотра больших блоков текста или графических изображений, не помещающихся в отведенной им области окна.
- Счетчик (Spin) представляет собой комбинацию двух кнопок со стрелками. Обычно со счетчиком связано поле для ввода числовых значений. Щелчок на соответствующей кнопке счетчика приводит к увеличению или уменьшению значения в поле.
- Регулятор (Slider) состоит из горизонтальной или вертикальной полосы и бегунка на ней. Регулятор обычно используется для пошагового изменения связанного с ним значения в заданном диапазоне.
- Четырехрежимный список (ListControl) представляет собой прямоугольную область со списком значков, которые можно отображать в увеличенном или уменьшенном виде, выстраивать друг под другом или рядом, располагать в виде простого списка или в виде таблицы с возможностью сортировки по различным столбцам.
- Вкладки (TabControl) используются в тех случаях, когда диалоговое окно содержит слишком много различных опций. Вместо того чтобы бесконечно увеличивать размер окна, пытаясь вывести на экран все опции, можно создать многостраничное окно, каждая страница которого будет представлена своим ярлычком вкладки. Таким образом, опции диалогового окна будут разбиты на категории и помещены на разных страницах.
- Расширенное поле (RichEdit) позволяет вводить многострочные блоки текста. Введенный текст можно форматировать, внедрять в него OLE-объекты.
- Рисунок (Picture) — это прямоугольная область, куда может быть вставлено графическое изображение.
- Поле (EditBox) — это прямоугольная область, в которую пользователь может вводить текст. Поступающий текст может интерпретироваться как набор символов или как число (в этом случае автоматически производится проверка правильности ввода).
- Кнопки (Button) обычно служат средством выдачи команд наподобие закрытия окна или отмены выполненных установок. По умолчанию они содержат только надпись, но могут также содержать значок или небольшое изображение.
- Переключатель (RadioButton) представляет собой маленький кружок, справа от которого добавляется надпись. Переключатели, как и флажки, обычно располагаются группами, но особенностью группы переключателей является то, что в ней можно выбрать только один переключатель.
- Список (ListBox) представляет собой прямоугольную область с набором текстовых элементов. Таковым, в частности, является список файлов текущего каталога.
- Вертикальная полоса прокрутки (VerticalScrollBar) аналогична по назначению горизонтальной полосе прокрутки.
- Индикатор (Progress) предназначен для визуального отображения хода выполнения программой какого-либо задания. Он представляет собой полосу, которая по мере выполнения задания заполняется квадратами.

- Горячими клавишами (HotKey) называются сочетания клавиш, посредством которых можно выполнять команды меню, не прибегая к помощи мыши. В диалоговом окне данный элемент управления представляется полем, в котором отображаются обозначения нажимаемых клавиш.
- Дерево (TreeControl) отображает список элементов в виде древовидной структуры. С помощью дерева удобно отображать иерархические зависимости между объектами.
- Анимацией (Animate) называется элемент управления, позволяющий отображать видеоклипы в формате AVI (audio video interleaved).
- Пользовательский элемент управления (CustomControl) служит оболочкой для внедрения в диалоговое окно любого элемента управления, разработанного пользователем. После появления технологии ActiveX такая методика считается устаревшей, поэтому пользовательский элемент управления существует просто для обеспечения совместимости со старыми проектами.

Чтобы добавить в макет диалогового окна новый элемент управления, нужно выбрать на панели соответствующий инструмент, установить указатель мыши в нужное место окна и, удерживая нажатой левую кнопку мыши, очертить контур элемента управления. Впоследствии такой элемент управления можно будет переместить в другое место, изменить его размер.

Компоновка диалогового окна

Рассмотрим кратко процесс создания простого диалогового окна About. Окно с данным названием является "визитной карточкой" приложения, содержит номер его версии и указывает обладателей авторских прав. Обычно в таком окне имеется единственная кнопка ОК, и его создание не составляет труда. Заготовка диалогового окна показана на рис. 16.15. Остается только добавить соответствующий текст и правильно разместить элементы управления.

В нашем окне About имеются элементы управления только двух типов — надписи и кнопки. Используя мышь, разместите в окне первую надпись и определите для нее подходящий размер. Затем выполните двойной щелчок внутри надписи, чтобы в открывшемся окне свойств задать ее текст (см. рис. 16.16).

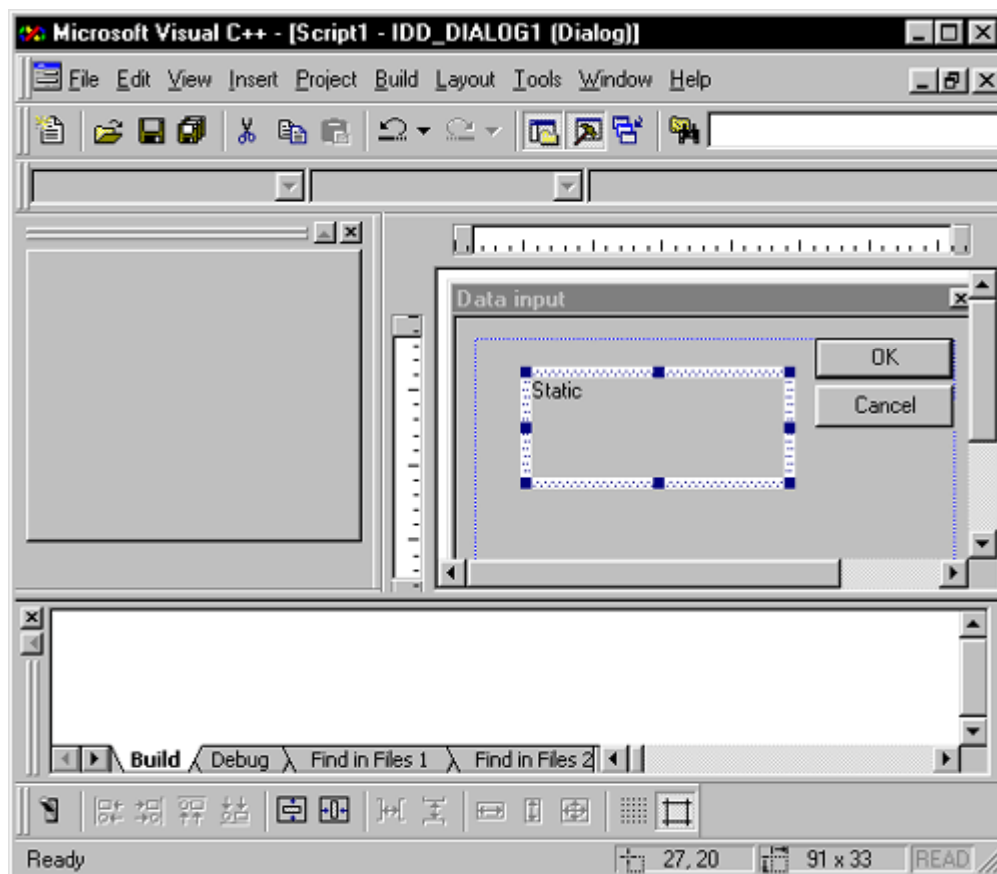


Рис. 16.15. В макет диалогового окна добавлена надпись, указывающая, куда нужно ввести соответствующий текст

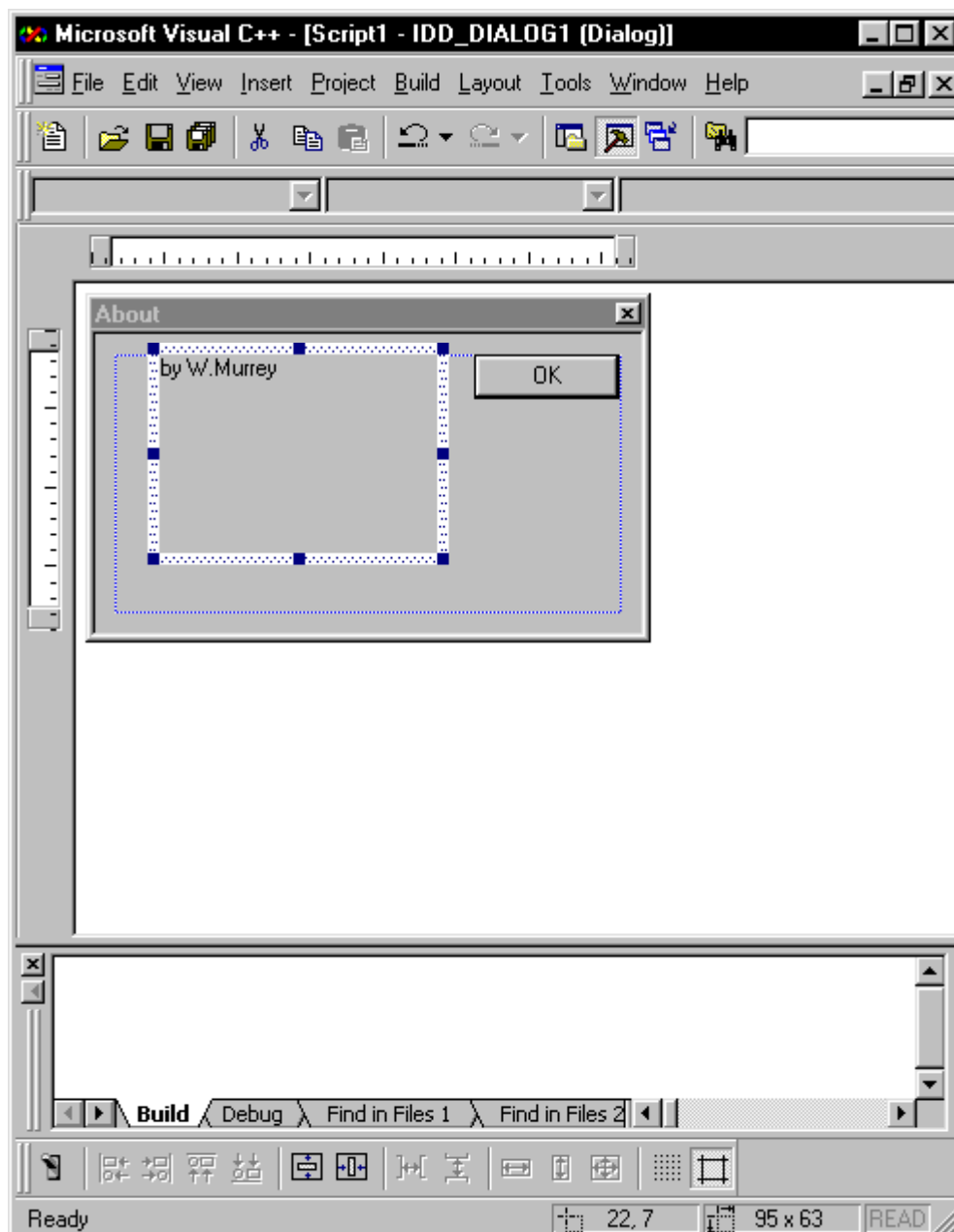


Рис. 16.16. Окно About после добавления текста надписи

Чтобы удалить ненужную кнопку Cancel, щелкните на ней мышью и нажмите клавишу [Del], после чего разместите кнопку OK там, где хотите ее видеть. На рис. 16.17 показан окончательный вариант макета создаваемого диалогового окна.

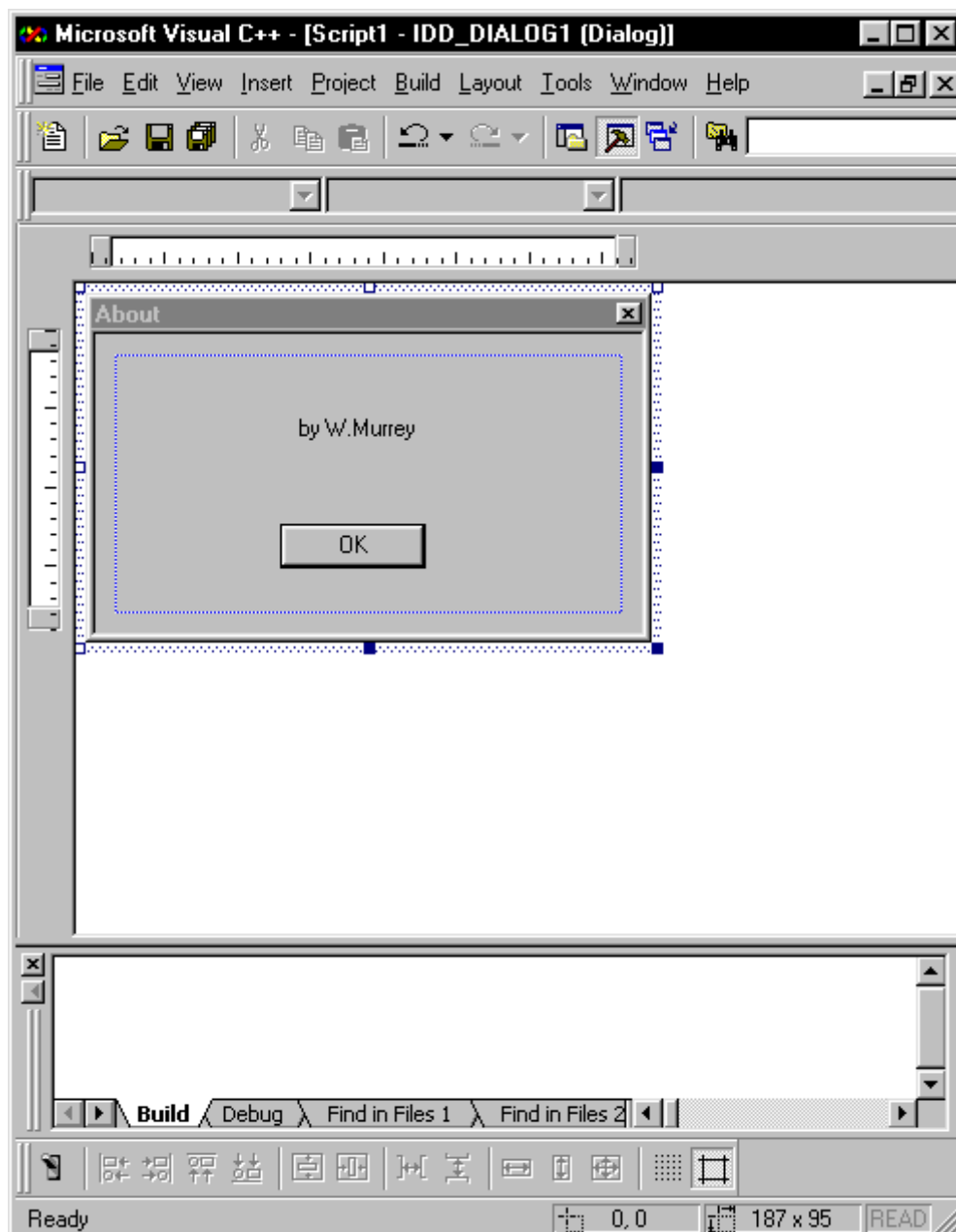


Рис. 16.17. Окончательный вид окна About

Чтобы сохранить новое диалоговое окно, выберите в меню File команду Save. В результате будет создан файл в формате RC. Для просмотра файла сценария ресурсов можно использовать любой текстовый редактор, в том числе редактор компилятора VisualC++. Как будет выглядеть сценарий созданного нами диалогового окна About, показано ниже.

```
ABOUTDLGBOX DIALOG DISCARDABLE 50,300, 180, 84
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "About"
FONT 8, "MS Sans Serif"
BEGIN
    CTEXT "Microsoft C Pie Chart Program", -1,3, 29,176,10
    CTEXT "by William H. Murray and Chris H. Pappas", -1, 3, 16,176,10
    PUSHBUTTON "OK", IDOK, 74,51,32, 14
```


END

Диалоговому окну присвоено имя `aboutdlgbox`. Редактор ресурсов автоматически рассчитал размеры самого окна и всех элементов управления. В описании указано, что данное окно является модальным (`ds_modalframe`) и всплывающим (`HS_POPUP`). Далее следуют описания трех используемых в нем элементов управления. Первые два элемента управления представляют собой надписи. В кавычках указан текст надписи, после чего через запятую следуют числовые значения, задающие идентификатор надписи, ее начальные координаты и размер.

Третьим элементом управления является кнопка ОК. Текст в кавычках является надписью на кнопке. Для данной кнопки используется системный идентификатор

`IDOK`.

Впрочем, вам нет необходимости просматривать все эти описания. Редактор ресурсов автоматически скомпилирует описание ресурсов в файл с расширением `RES`. Единственный случай, когда, возможно, придется работать со сценариями ресурсов, — это при переписывании их из журнальных публикаций или книг.

Подключение внешних ресурсов

В `RC`-файл можно также включать ресурсы, содержащиеся во внешних файлах, например значки, указатели мыши, точечные рисунки и т.д. При этом указывается уникальное имя ресурса, ключевое слово, определяющее тип ресурса, и имя файла ресурса. Например:

```
myicon ICON myicon.ico
mycursor CURSOR mycursor.cur
mybitmap BITMAP mybitmap.bmp
```

С помощью приведенных строк в файл ресурсов добавляется три новых ресурса: `myicon`, `mycursor` и `mybitmap`. Литералы `ICON`, `CURSOR` и `BITMAP` являются зарезервированными ключевыми словами, определяющими тип ресурса: значок, указатель мыши и точечный рисунок соответственно. Затем следуют непосредственные имена файлов: `myicon.ico`, `mycursor.cur` и `mybitmap.bmp`.

Глава 17. Процедурные приложения для Windows

- Структура приложения
 - Основные компоненты приложения
 - Функция WinMain()
 - Структура WNDCLASS
 - Структура WNDCLASSEX
 - Определение класса окна
 - Создание окна
 - Отображение и обновление окна
 - Цикл обработки сообщений
 - Оконная процедура
 - Обработка сообщения WM_PAINT
 - Обработка сообщения WM_DESTROY
 - Функция DefWindowProc()
- Создание проекта
- Текст программы
 - Рисование эллипса
 - Рисование сегмента
 - Рисование сектора
 - Рисование прямоугольника
- Использование файла SWP.C в качестве шаблона
- Создание диаграмм
 - Файл PIE.H
 - Файл PIE.RC
 - Файл PIE.C

В предыдущей главе мы рассмотрели терминологию и основные концепции программирования в среде Windows. Наиболее важными и привлекательными особенностями всех приложений Windows следует считать стандартность интерфейса, аппаратную независимость и возможность одновременного выполнения. В этой главе мы, воспользовавшись полученными знаниями, попытаемся написать несложное процедурное приложение.

Структура приложения

В данном параграфе мы рассмотрим компоненты, из которых состоит программа SWP.C (SimpleWindowsProgram). Будет показано, что необходимо включить в программу для создания и отображения на экране окна приложения (с рамкой, строкой заголовка, системным меню и кнопками свертывания/развертывания), а также для корректного завершения приложения.

Вы также узнаете, как программу SWP.Си связанные с ней файлы можно использовать в качестве шаблона при создании других приложений Windows. Четкое понимание структуры нашего примера позволит вам сэкономить много времени в будущем, так как рассматриваемые компоненты составляют основу практически любого приложения.

Но для начала в качестве справки перечислим наиболее часто используемые типы данных Windows, которые могут встретиться нам в процессе работы (табл. 17.1).

Таблица 17.1. Часто используемые типы данных	
Тип	Описание
callback	Замещает спецификацию farpascal в функциях обратного вызова
handle	32-разрядное беззнаковое целое число, используемое в качестве дескриптора
HDC	Дескриптор контекста устройства
hwnd	Дескриптор окна
LONG	32-разрядное целое число со знаком
lparam	Используется для описания младшего аргумента сообщения
LPCSTR	То же, что и lpstr, но используется для указания константных строк
lpstr	32-разрядный указатель на строку
lvoid	Обобщенный тип указателя, эквивалентен void*
lresult	Используется для возвращения значений из оконных процедур
uint	32-разрядное беззнаковое целое число
wchar	16-разрядный символ UNICODE; используется для представления печатных символов всех языков мира
winapi	Замещает спецификацию farpascal в описаниях API-функций
wparam	Используется для описания старшего параметра сообщения

В табл. 17.2 приведено несколько стандартных структур Windows.

Таблица 17.2. Часто используемые стандартные структуры	
Структура	Что определяет
msg	Параметры сообщения
PAINTSTRUCT	Область окна, требующую перерисовки
rect	Прямоугольную область
WNDCLASS	Класс окна

Основные компоненты приложения

Приложения Windows содержат два общих элемента: функцию winMain() и оконную процедуру. Функция winMain() составляет основу любого приложения. Она служит как бы точкой входа в приложение и выполняет ту же роль, что и функция main() в обычных программах на С и С++.

Оконная процедура выполняет роль диспетчера сообщений. Приложения никогда не получают к ней прямого доступа. Для этого им сначала нужно сделать запрос к Windows на выполнение соответствующего действия. Поэтому все оконные процедуры должны быть функциями обратного вызова. Подобная функция регистрируется в Windows и вызывается всякий раз, когда выполняется некоторое действие над окном.

Функция WinMain()

Функция winMain(), как уже было сказано, является обязательным компонентом любого приложения Windows. С нее начинается выполнение программы и ею же обычно заканчивается. Функция WinMain() отвечает за следующее:

- создание и инициализацию цикла обработки сообщений (который имеет доступ к очереди сообщений приложения);
- начальную инициализацию приложения;

- регистрацию класса окна приложения;
- завершение выполнения программы, обычно в результате получения сообщения WM_QUIT.

В функцию WinMain() передаются четыре параметра:

```
int WINAPI WinMain(HINSTANCE
hInst, HINSTANCE
hPreInst, LPSTR
lpCmdLine, int nCmdShow)
```

Первый параметр, hInst, представляет собой дескриптор экземпляра приложения. Вторым, hPreInst, всегда содержит значение null, что указывает на отсутствие других, запущенных ранее, экземпляров приложения.

Примечание

В ранних версиях Windows (вплоть до 3.x) параметр hPreInst позволял определить, были ли запущены другие копии приложения. В Windows 95/98/NT для каждого запущенного модуля выделяется отдельное адресное пространство, в связи с чем параметр hPreInst всегда устанавливается равным NULL.

Третий параметр, lpCmdLine, служит указателем на строку, завершающуюся нулевым символом и содержащую командную строку программы.

Последний параметр, nCmdShow, задает одну из многочисленных констант, определяющих возможные варианты отображения окна, например SW_SHOWNORMAL, SWSHOWMAXIMIZED или SWSHOWMINIMIZED.

Структура WNDCLASS

Одна из задач функции WinMain() заключается в регистрации класса окна приложения. Класс окна содержит комбинацию выбранных пользователем свойств окна, дескрипторы значка и указателя мыши, а также прочие атрибуты и выступает в роли шаблона при создании всех экземпляров окон приложения.

Примечание

В ранних версиях Windows зарегистрированные классы окон становились доступными для всех программ, выполняющихся в системе. В связи с этим программисты должны были при регистрации классов следить за тем, чтобы используемые ими имена не конфликтовали с другими зарегистрированными классами окон. В Windows 95/98/NT требуется, чтобы не только приложения, но и все экземпляры одного приложения регистрировали собственные классы окон.

Для описания всех классов окон используется единая структура. Следующий пример взят непосредственно из файла WINUSER.H, включаемого в файл WINDOWS.H с помощью директивы #include.

```
typedef struct tagWNDCLASSW (
UINT style
WNDPROC lpfnWndProc
int cbClsExtra;
int cbWndExtra;
HANDLE hInstance;
HICON hIcon;
HCURSOR hCursor
HBRUSH hbrBackground;
LPCWSTR lpszMenuName;
LPCWSTR lpszClassName;
) WNDCLASSW, *PWNDCLASSW, NEAR *NPWNDCLASSW, FAR *LPWNDCLASSW;
```

Структура wndclassw предназначена для работы с UNICODE-строками. Имеется также аналогичная ей структура wndclassa, ориентированная на работу с ASCII-строками. С помощью

typedef-определений от этих структур порождается универсальная Структура `wndclass`, позволяющая программисту не заботиться о формате используемых строк.

Чтобы определить класс окна, приложение должно объявить переменную следующего типа:

```
WNDCLASS wcApp;
```

Впоследствии переменная `wcApp` заполняется информацией о классе окна. Ниже описываются различные поля структуры `wndclass`.

style

Определяет свойства окна. С помощью операции побитового ИЛИ можно задать комбинацию из нескольких свойств. Возможные свойства поля `style` перечислены в табл. 17.3.

Свойство	Описание
CS_BYTEALIGNCLIENT	Задаёт выравнивание по границе байта размера рабочей области по горизонтали
CS_BYTEALIGNWINDOW	Задаёт выравнивание по границе байта размера окна по горизонтали
CS_CLASSDC	Указывает, что все окна данного класса должны использовать один и тот же контекст устройства
CS_DBLCLKS	При установке этого флага окну будут посылаться сообщения о двойном щелчке
CS_GLOBALCLASS	Указывает, что класс окна является глобальным и должен регистрироваться независимо от значения параметра <code>hInstance</code>
CS_HREDRAW	Задаёт операцию перерисовки окна при изменении горизонтального размера
CS_NOCLOSE	Делает неактивной команду <code>close</code> в системном меню
CS_OWNDC	Задаёт выделение каждому окну собственного контекста устройства
CS_PARENTDC	Указывает, что область отсечения дочернего окна будет равна области отсечения родительского окна
CS_SAVEBITS	Служит указанием системе сохранять часть окна, перекрываемую другим окном, чтобы впоследствии не посылать перекрытому окну сообщения <code>WM_PAINT</code> , а восстанавливать его содержимое самостоятельно
CS_VREDRAW	Задаёт операцию перерисовки окна при изменении вертикального размера

lpfnWndProc

Содержит указатель на оконную процедуру, которая будет обрабатывать все сообщения, посылаемые окну.

cbClsExtra

Содержит информацию о количестве байтов, выделяемых системой после создания структуры класса окна. Значение этого параметра может быть равным `NULL`.

cbWndExtra

Содержит информацию о количестве байтов, выделяемых системой после создания экземпляра окна. Этот параметр может быть равным `NULL`.

hInstance

Содержит дескриптор экземпляра приложения, регистрирующего класс окна.

hIcon

Содержит дескриптор значка приложения. Этот параметр может быть равным `NULL`.

hCursor

Содержит дескриптор указателя мыши, используемого в окне приложения. Этот параметр может быть равным `null`.

hbrBackground

Определяет кисть, используемую для заливки фона окна. Может содержать дескриптор реальной кисти или одну из следующих констант (к задаваемой константе следует прибавить 1):

```
COLOR_ACTIVEBORDER  
COLOR_ACTIVECAPTION  
COLOR_APPWORKSPACE  
COLOR_BACKGROUND  
COLOR_BTNSHADOW  
COLORJBTNTTEXT  
COLOR_CAPTIONTEXT  
COLORJ3RAYTEXT  
COLOR_HIGHLIGHT  
COLOR_HIGHLIGHTTEXT  
COLOR_INACTIVEBORDER  
COLOR_INACTIVECAPTION  
COLOR_MENU  
COLOR_MENUTEXT  
COLOR_SCROLLBAR  
COLOR_WINDOW  
COLOR_WINDOWFRAME  
COLOR_WINDOWTEXT
```

Если значение параметра `hbrBackground` равно `NULL`, то приложение должно самостоятельно управлять заливкой фона окна при получении сообщения

```
WM_ERASEBKGD.
```

IpszMenuItemName

Содержит указатель на строку, завершающуюся нулевым символом и представляющую собой имя ресурса меню. Этот параметр может быть равным `null`.

IpszClassName

Содержит указатель на строку, завершающуюся нулевым символом и представляющую собой имя класса окна.

Структура WNDCLASSEX

Существует расширенный вариант структуры `wndclass`, получивший название `wndclassex`. От `wndclass` отличается тем, что дополнительно позволяет использовать в приложении значок малого размера.

```
typedef struct _WNDCLASSEX {  
    UINT    cbSize  
    UINT    style;  
    WNDPROC  lpfnWndProc;  
    int      cbClsExtra;  
    int      cbWndExtra;  
    HANDLE  hInstance;  
    HICON    hIcon;  
    HCURSOR  hCursor  
    HBRUSH  hbrBackground;  
    LPCTSTR  lpszMenuName;  
    LPCTSTR  lpszClassName;  
    HICON    hIconSm  
} WNDCLASSEX;
```

В поле `cbSize` задается размер всей структуры. Это обычно делается с помощью выражения `sizeof(WNDCLASSEX)`. В поле `hIconSm` содержится дескриптор малого значка.

Определение класса окна

В приложении можно описать собственный класс окна, создав для него структуру соответствующего типа и заполнив поля структуры атрибутами разрабатываемого приложения.

Следующий фрагмент взят из файла SWP.C и демонстрирует, как можно создать и проинициализировать структуру `wndclass`:

```
char szProgName[]="ProgName";
WNDCLASS wcApp;
wcApp.lpszClassName = szProgName; wcApp.hInstance = hInst;
wcApp.lpfnWndProc = WndProc;
wcApp.hCursor = LoadCursor(NULL, IDC_ARROW);
wcApp.hIcon = NULL; wcApp.lpszMenuName = 0;
wcApp.hbrBackground = GetStockObject(WHITE_BRUSH);
wcApp.style = CS_HREDRAW | CS_VREDRAW;
wcApp.cbClsExtra = 0;
wcApp.cbWndExtra = 0; if (IRegisterClass(SwcApp))
return 0;
```

Имя программы хранится в переменной `szProgName` и записывается в класс окна в поле `wcApp.lpszClassName`.

Второму полю структуры `wndclass`, называемому `wcApp.hInstance`, передается аргумент `hInst` функции `winMain()`, хранящий дескриптор текущего экземпляра приложения. Полю `lpfnWndProc` присваивается указатель на оконную процедуру, которая будет обрабатывать все сообщения, посылаемые окну. В файле SWP.C эта функция называется `WndProc()`.

Имя `WndProc()` задано программистом, т.е. не является стандартным. Прототип функции должен быть описан до выполнения операции присваивания.

В поле `wcApp.hCursor` хранится дескриптор указателя мыши текущего экземпляра приложения. В нашем примере создается указатель `idc_arrow` (соответствует обычному указателю мыши в виде наклоненной стрелки), дескриптор которого возвращается функцией `LoadCursor()`. Поскольку в приложении нет устанавливаемого по умолчанию значка, полю `wcApp.hIcon` присваивается значение `null`.

Если поле `wcApp.lpszMenuName` содержит `NULL`, то Windows считает, что у приложения нет собственного меню. В противном случае должно быть указано название ресурса меню, взятое в кавычки. Функция `GetStockObject()` возвращает дескриптор кисти, которая используется для заливки фона окна, созданного на базе данного класса. В нашем случае применяется стандартная белая кисть — `WHITEBRUSH`.

В поле `wcApp.style` свойства окна задаются как `cs_hredraw` в сочетании с `cs_vredraw`. Константы с префиксом `cs_` определены в файле `WINUSER.H` и могут объединяться с помощью операции побитового ИЛИ (`|`). Константы `cs_hredraw` и `cs_vredraw` указывают Windows, что рабочую область окна нужно перерисовывать всякий раз, когда размер окна изменяется по вертикали или горизонтали.

Последние два поля, `wcApp.cbClsExtra` и `wcApp.cbWndExtra`, обычно устанавливают равными 0. Их предназначение — указание дополнительного количества байтов памяти, которые должны быть зарезервированы после создания класса окна или экземпляра самого окна.

Ранее мы уже говорили о том, что в 16-разрядных версиях Windows приложение регистрировало класс окна только в том случае, если не существовало ранее загруженных копий этой же программы. Зарегистрированный класс окна становился доступным для всех остальных программ, поэтому необходимость в повторной его регистрации не возникала. Вот как это делалось:

```
if (IhPreInst)
{
if (IRegisterClass(SwcApp))
return FALSE; }
```

В 32-разрядных версиях Windows параметр `hPreInst` всегда равен `NULL`, поэтому подобная проверка избыточна.

В функцию `Register-Class()` передается указатель на только что созданную структуру класса окна. Если Windows не может зарегистрировать новый класс (скажем, в случае нехватки памяти), функция возвращает 0, вследствие чего выполнение программы завершается.

Создание окна

Регистрация класса окна еще не свидетельствует о создании самого окна. Окно создается в результате вызова функции `CreateWindow()`. Этот процесс одинаков во всех версиях Windows. В то время как класс окна определяет общие свойства всех окон данного семейства, параметры функции `CreateWindow` задают описание конкретного экземпляра окна. Если функция `CreateWindow()` выполняется успешно, она возвращает дескриптор созданного окна, в противном случае — `null`.

В файле `SWP.C` функция `CreateWindow()` вызывается следующим образом:

```
hWnd = CreateWindow(szProgName, "SimpleWindowsProgram",
WSJDVERLAPPEDWINDOW, CW_USEDEFAULT,
CW_USEDEFAULT, CW_DSEDEFAULT,
CWJSEDEFAULT, (HWND) NULL, (HMENU) NULL,
(HANDLE) hInst, (LPSTR) NULL);
```

Первый параметр, `szProgName`, содержит название класса окна. За ним в кавычках указывается содержимое строки заголовка окна — `"SimpleWindowProgram"`. Третий параметр задает стиль окна — `ws_overlappedwindow`. Это стандартный стиль Windows, определяющий обычное масштабируемое окно со строкой заголовка, системным меню, кнопками свертывания, разворачивания и закрытия, а также с рамкой.

Следующие шесть параметров (`sw_usedefault` или `null`) определяют координаты начальной точки окна, его размеры, а также дескрипторы родительского окна и меню. Всем этим параметрам в нашем примере присвоены значения по умолчанию. Десятый параметр, `hinst`, содержит дескриптор экземпляра приложения. Окну не передаются никакие дополнительные значения, поэтому последний параметр равен `NULL`.

Отображение и обновление окна

Для отображения окна на экране предназначена функция `ShowWindow()`:

```
ShowWindow(hWnd, nCmdShow);
```

Дескриптор окна `hWnd` был создан функцией `CreateWindow()`. Второй параметр, `nCmdShow`, устанавливает режим начального отображения окна. Так, задание константы `sw_showminnoactive`, определенной в файле `WINUSER.H`, приведет к отображению окна в виде значка на панели задач:

```
ShowWindow(hWnd, SW_SHOWMINNOACTIVE);
```

Константа `sw_showmaximized` отвечает за разворачивание окна на весь экран. Режим `sw_showminimized` аналогичен режиму `sw_showminnoactive` за исключением того, что окно остается активным, т.е. имеет фокус.

Последний этап вывода окна на экран реализует функция `UpdateWindow()`:

```
UpdateWindow(hWnd);
```

Вызов функции `ShowWindow()` с установленным параметром `sw_shownormal` приводит к заливке фона окна выбранной кистью. А функция `UpdateWindow` генерирует сообщение `WM_PAINT`, указывающее на формирование содержимого рабочей области окна.

Цикл обработки сообщений

После отображения окна программа готова к выполнению своей непосредственной задачи — обработке сообщений. Помните, что в Windows информация от мыши или клавиатуры не передается напрямую приложению, а в виде сообщений помещается в очередь. Очередь

может содержать сообщения как сгенерированные самой системой, так и поступившие от других приложений.

Как правило, цикл организуется с помощью такой конструкции:

```
while (GetMessage(&lpMsg, NULL, 0, 0) ) {  
    TranslateMessage(SlpMsg) ;  
    DispatchMessage(slpMsg);}
```

Функция GetMessage()

Функция GetMessage() читает очередное сообщение из очереди и записывает его в структуру msg, адресуемую указателем lpMsg.

Установка второго параметра равным NULL свидетельствует о том, что читаться должны сообщения, адресованные любому окну приложения. Два других параметра формируют фильтр сообщений, который ограничивает получаемые сообщения определенным диапазоном (например, только сообщения мыши). Если эти параметры равны нулю, как в нашем случае, то принимаются любые сообщения, адресованные данному приложению.

После запуска цикла только одно сообщение может остановить его выполнение. Как только в цикл передается сообщение wm_quit, функция GetMessage() возвращает значение false, в результате чего цикл завершается, выполняются заключительные операции функции winMain() и работа приложения прекращается.

Функция TranslateMessage()

Сообщения о нажатии виртуальных клавиш могут быть преобразованы в символьные сообщения с помощью функции TranslateMessage(), которая формирует сообщение wm_charиз пары wm_keydown/wm_keyup. Эта функция необходима только тем приложениям, которые обрабатывают ввод данных с клавиатуры. Важность функции заключается в том, что она позволяет пользователям выбирать команды меню путем нажатия клавиш, а не только щелчками мыши.

Функция DispatchMessage()

Функция DispatchMessage() направляет полученное сообщение соответствующей оконной процедуре, автоматически определяя окно, которому адресовано сообщение.

Оконная процедура

Оконная процедура регистрируется в системе и вызывается всякий раз, когда Windows выполняет какую-либо операцию над окном приложения. В следующем фрагменте показан текст этой процедуры из файла SWP.C (за исключением нескольких специфичных для приложения строк):

```
LRESULT CALLBACK WndProc(HWND hWnd, UINT messg,  
    WPARAM wParam, LPARAM lParam) {  
    HDC hdc;  
    PAINTSTRUCT ps;  
    switch (messg) {  
        case WM_PAINT:  
            hdc = BeginPaint (hWnd, &ps);  
            ValidateRect(hWnd, NULL);  
            EndPaint(hWnd, &ps); break;  
        case WM_DESTROY:  
            PostQuitMessage (0);  
            break;  
        default:  
            return(DefWindowProc(hWnd, messg, wParam, lParam));  
    }  
    return(0);  
}
```

Windows предполагает, что имя этой процедуры указано в поле `lpfnWndProc` структуры `WcApp`, описывающей класс окна. Все окна, созданные на базе данного класса, будут управляться процедурой `WndProc()`.

В Windows существует более двухсот различных сообщений, которые могут посылааться окнам. Все они имеют префикс `WM_` (`Window Message` — оконное сообщение), например: `WM_CREATE`, `WM_SIZE`, `WM_PAINT`.

Первым параметром процедуры `WndProc()` является дескриптор окна, которому посылается сообщение. Поскольку одна процедура может обрабатывать сообщения сразу нескольких окон, этот дескриптор позволяет определить, какому именно окну адресовано сообщение. Второй параметр содержит идентификатор сообщения. Два оставшихся параметра, `WPARAM` и `LPARAM`, несут дополнительную информацию, зависящую от конкретного сообщения.

В нашей процедуре `WndProc()` определяются две локальные переменные: `hdc`, содержащая дескриптор контекста устройства (экрана), и `ps`, задающая структуру `PAINTSTRUCT`, которая необходима для хранения информации, отображаемой в рабочей области окна.

Обработка самих сообщений осуществляется с помощью стандартной конструкции `switch`, которая может быть довольно большой (ее размер зависит от числа обрабатываемых сообщений).

Обработка сообщения `WM_PAINT`

Первое сообщение, которое в нашем примере обрабатывает процедура `WndProc()`, — это `WM_PAINT`. Поскольку Windows является многозадачной системой, то одно приложение может открыть свое окно поверх окна другого приложения. Это создает определенную проблему, так как при закрытии или перемещении верхнего окна нижнее не знает о происходящем, в результате чего область, которую занимало верхнее окно, становится некорректной. Windows решает данную проблему путем направления сообщения `WM_PAINT` нижнему окну, информируя о том, что его содержимое требует обновления.

Помимо самого первого сообщения `WM_PAINT`, которое генерируется функцией `UpdateWindow()` внутри функции `WinMain()`, сообщение `WM_PAINT` также посылается в следующих ситуациях:

- при вызове функции `InvalidateRect()` или `InvalidateRgn()`;
- при изменении размеров окна;
- при вызове функции `ScrollWindow()`;
- каждый раз, когда рабочая область окна перекрывается ниспадающим меню или диалоговым окном, после их закрытия.

Любая часть окна, которая была закрыта другим объектом, например диалоговым окном, помечается как недействительная. Наличие недействительной области заставляет Windows послать приложению сообщение `WM_PAINT`, причем система отслеживает координаты такой области. Если окно оказалась закрыто несколькими объектами, Windows вычислит координаты суммарной области. Другими словами, Windows не станет посылать несколько сообщений `WM_PAINT` для каждого отдельно взятого прямоугольного участка, а сгенерирует одно сообщение сразу для всей недействительной области сложной формы.

При обработке сообщения `WM_PAINT` вначале выполняется функция `BeginPaint()`, которая подготавливает указанное окно к операции рисования, заполняет структуру `PAINTSTRUCT` данными об обновляемой области и возвращает дескриптор контекста устройства, связанного с этим окном.

Вызов функции `InvalidateRect()` заставляет Windows пометить указанную область окна как недействительную и сгенерировать для нее сообщение `WM_PAINT`. Приложение может получить координаты этой области, вызвав функцию `GetUpdateRect()`. Функция `ValidateRect()` удаляет указанный фрагмент из определения области, требующей перерисовки.

Функция `EndPaint()` вызывается всякий раз по завершении операций вывода данных в рабочую область окна. Функция ставит Windows известность, что приложение закончило обработку сообщений о перерисовке и имеющийся контекст устройства можно удалить.

Обработка сообщения WM_DESTROY

Когда пользователь выбирает в системном меню команду **Close** или нажимает кнопку закрытия окна, Windows посылает приложению сообщение WM_DESTROY. В ответ на это приложение завершает свою работу, вызывая функцию PostQuitMessage (), которая направляет в очередь сообщений приложения сообщение WM_QUIT, вследствие чего цикл обработки завершается.

Функция DefWindowProc ()

Функция DefWindowProc обычно вызывается в блоке default оператора switch. Она возвращает все нераспознанные или необработанные сообщения обратно системе, гарантируя таким образом, что все посылаемые приложению сообщения будут корректно обработаны.

Создание проекта

Чтобы начать новый проект, выберите в окне компилятора в меню **File** команду **New**, а затем в открывшемся диалоговом окне **New** на вкладке Projects активизируйте элемент Win32 Application (рис. 17.1).

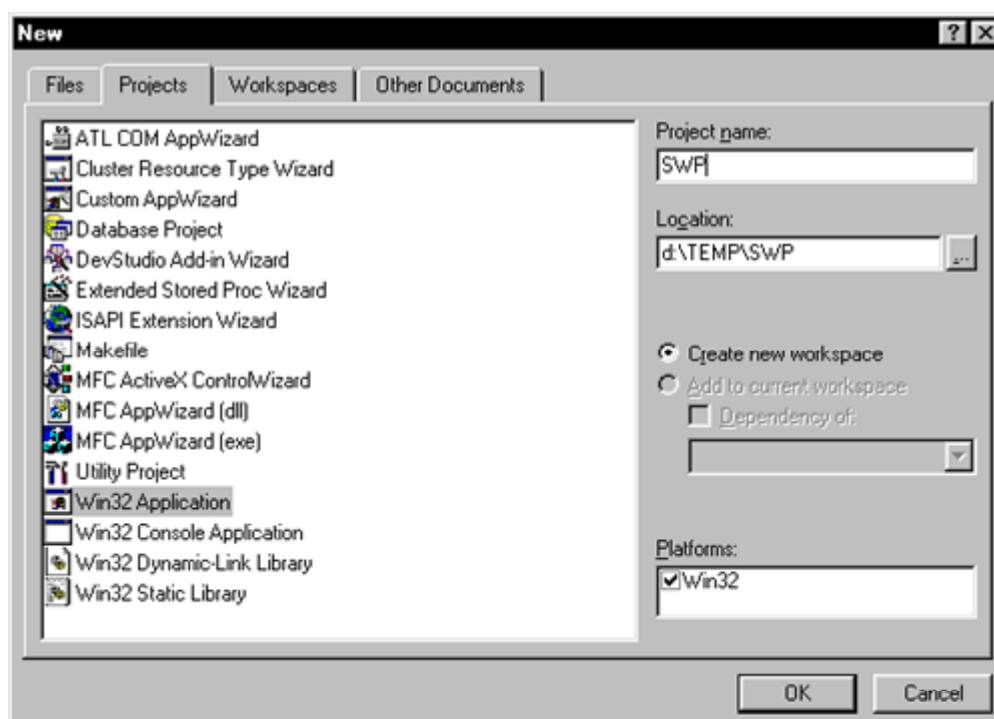


Рис. 17.1. Выбор типа проекта

Задайте для проекта имя SWP, щелкните на кнопке OK, в появившемся окне мастера выберите опцию **An empty project** и вновь нажмите **OK**, после чего будет создан пустой проект.

Чтобы добавить в проект программный файл, выберите в меню **Project** подменю **Add To Project**, а в нем — команду **New** (рис. 17.2).

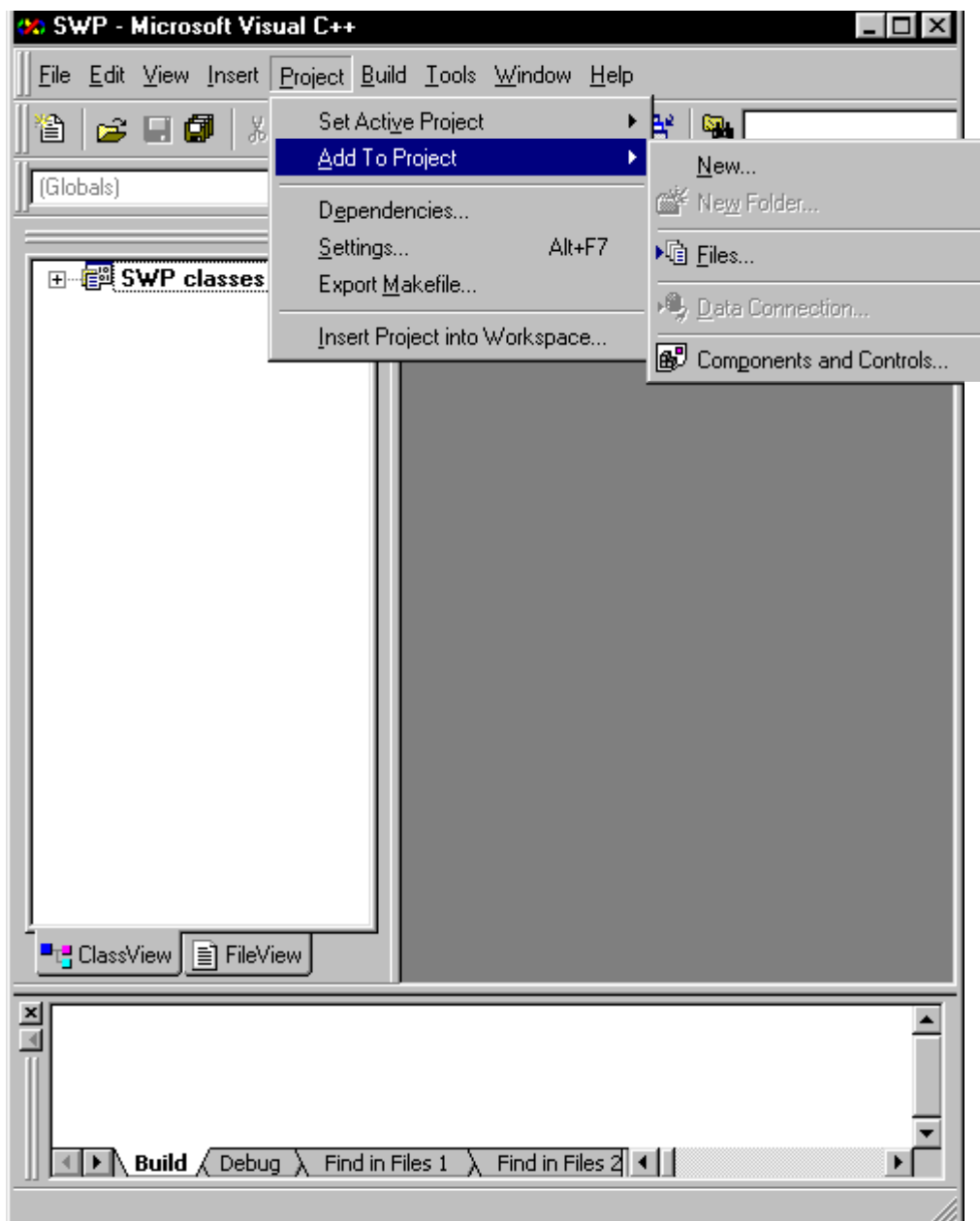


Рис. 17.2. Для добавления в проект нового файла используется команда **New** из подменю **Add To Project**

Воспользовавшись командой New, откройте диалоговое окно New(рис. 17.3), выделите в нем элемент C++ **SourceFile** и введите имя файла (в нашем примере это SWP.C).

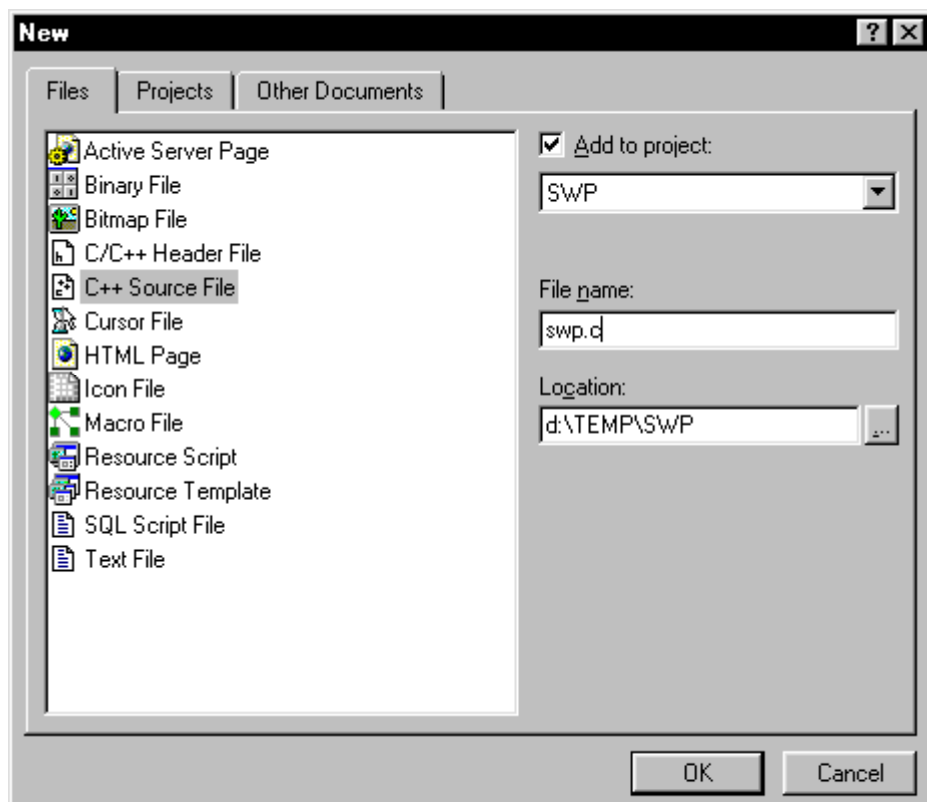


Рис.17.3. Выбор типа добавляемого файла

Далее в окне редактора наберите текст программы (рис. 17.4).

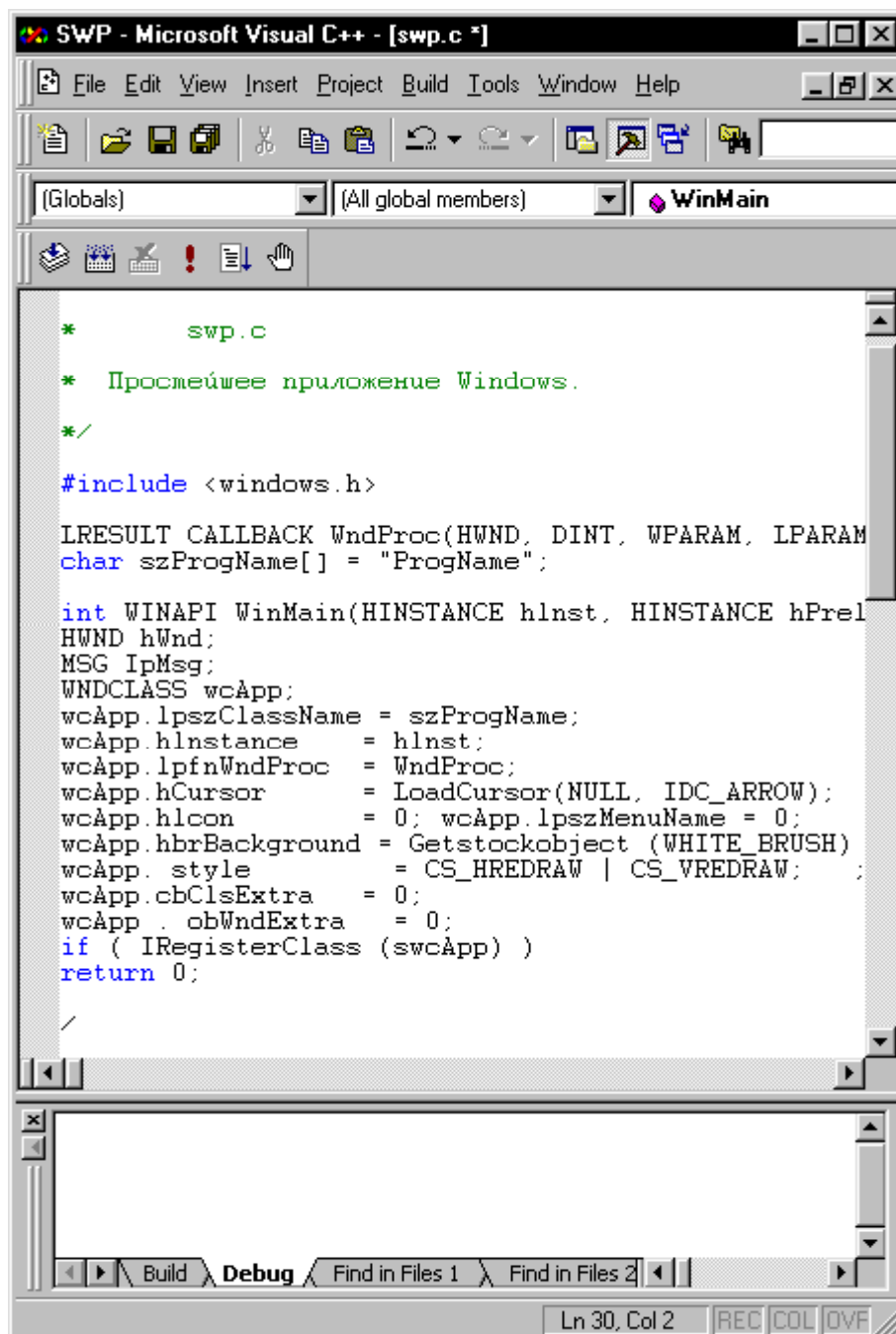


Рис. 17.4. Исходный текст программы в окне редактора

Завершив ввод текста, сохраните файл, выбрав в меню **File** команду **Save**.

Следующий этап создания нового приложения состоит в построении исполняемого файла. Но для начала важно правильно установить опции проекта. Выберите в меню **Project** команду **Settings....** Открывающееся при этом диалоговое окно **ProjectSettings** содержит множество вкладок, на которых представлены всевозможные опции сборки приложения. Убедитесь в том, что на вкладке **General** в списке **MicrosoftFoundationClasses** выделена опция **NotUsingMFC** (рис. 17.5).

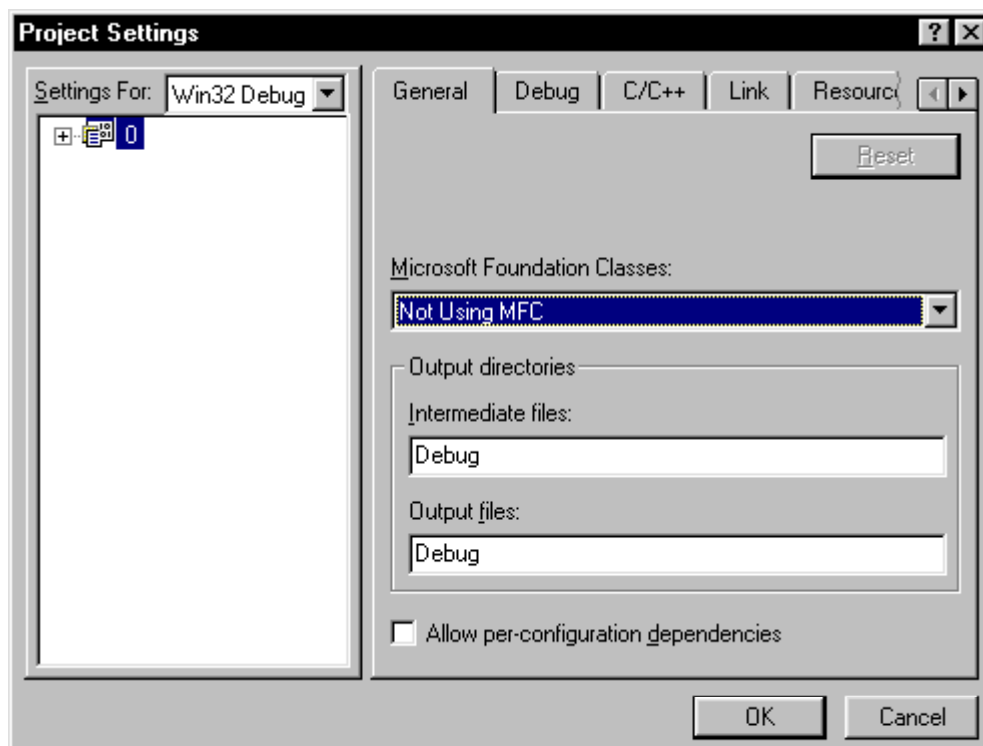


Рис. 17.5. Установки вкладки **General**

По умолчанию для проекта создаются две конфигурации — Debug(отладочная) и Release(финальная), из которых активной является первая, поэтому все файлы создаются и сохраняются в папке DEBUG. Когда вы будете уверены в корректности работы приложения и решите создать окончательную версию, выберите в меню **Build** команду **Set Active Configuration** и укажите конфигурацию Release.

На вкладке **Link** в поле **Output file name** указано, в какую папку будет помещен исполняемый файл и как он будет называться (рис. 17.6). Обратите внимание, что по умолчанию выходной папкой является DEBUG.

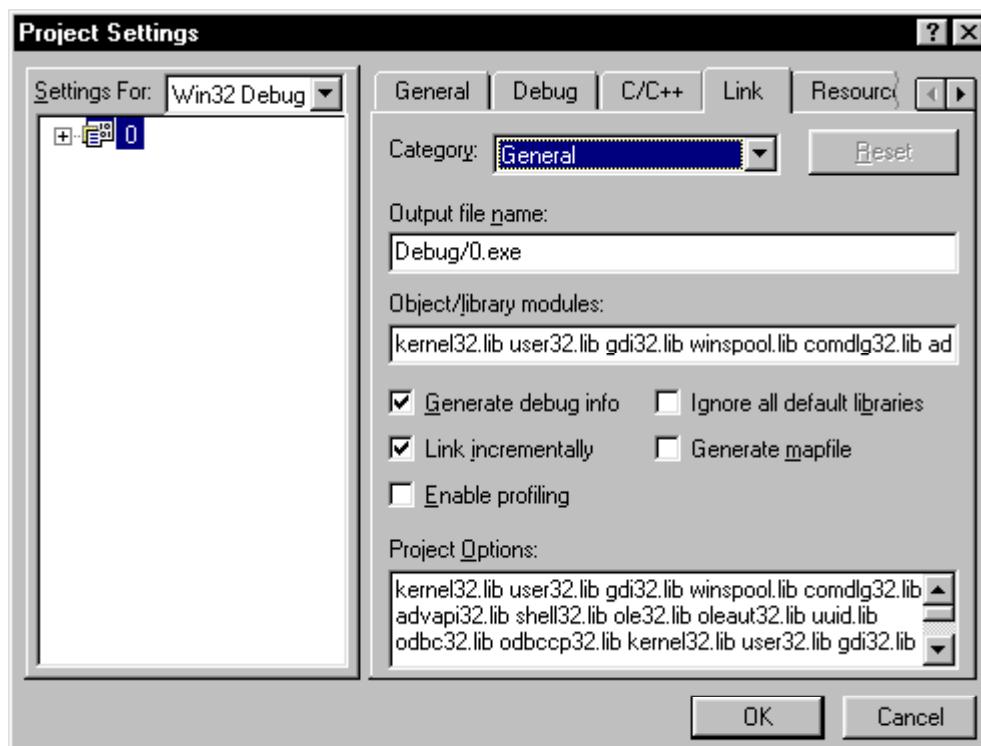


Рис. 17.6. Установки вкладки **Link**

Текст программы

Мы уже рассмотрели большую часть файла SWP.C. Ниже показан полный его текст.

```

/*
 *      swp.c
 *      Простейшее приложение Windows.
 */
#include <windows.h>
LRESULT CALLBACK WndProc(HWND, DINT, WPARAM, LPARAM); char
szProgName[] = "ProgName";
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst,
LPSTR lpszCmdLine, int nCmdShow) {
    HWND hWnd;
    MSG IpMsg;
    WNDCLASS wcApp;
    wcApp.lpszClassName = szProgName;
    wcApp.hInstance = hInst;
    wcApp.lpfnWndProc = WndProc;
    wcApp.hCursor = LoadCursor(NULL, IDC_ARROW);
    wcApp.hIcon = 0; wcApp.lpszMenuName = 0;
    wcApp.hbrBackground = Getstockobject (WHITE_BRUSH) ;
    wcApp.style = CS_HREDRAW | CS_VREDRAW;
    wcApp.cbClsExtra = 0;
    wcApp.obWndExtra = 0;
    if ( IRegisterClass (swcApp) )
    return 0;
    {
        hWnd = CreateWindow(szProgName, "Simple Windows Program",
        WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,, CWJSEDEFAULT,

```



```

CW_USEDEFAULT, (HWND) NULL, (HMENU) NULL, (HANDLE) hInst, (LPSTR) NULL)
; ShowWindow(hWnd, nCmdShow) ; UpdateWindow(hWnd) ;
while (GetMessage (SlpMsg, 0, 0, 0)){ TranslateMessage(SlpMsg) ;
DispatchMessage (SlpMsg) ; }
return (IpMsg.wParam) ;
}
LRESULT CALLBACK WndProc(HWND hWnd, UINT messg,
WPARAM wParam, LPARAM lParam) {
HDC hdc;
PAINTSTRUCT ps;
switch (messg)
{case WM_PAINT:
hdc = BeginPaint(hWnd, ps) ;
MoveToEx(hdc, 0, 0, NULL); LineTo(hdc, 639,429); MoveToEx(hdc, 300, 0,
NULL); LineTo(hdc, 50, 300);
TextOut(hdc, 120,-30,"<-a few lines ->",17);.
ValidateRect(hWnd, NULL);
EndPaint(hWnd, ps);
break; case WM_DESTROY:
PostQuitMessage(0);
break; default:
return(DefWindowProc(hWnd, messg, wParam, lParam));
break; }
return(0); }

```

Вспомните, что основная часть приведенного кода необходима для регистрации класса окна и создания самого окна. Блок программы, отвечающий за вывод содержимого окна, достаточно прост:

```

MoveToEx(hdc, 0, 0, NULL);
LineTo(hdc, 639,429);
MoveToEx(hdc, 300, 0, NULL);
LineTo(hdc, 50, 300);
TextOut(hdc, 120, 30, "<- a few lines ->",17);

```

В нескольких следующих параграфах мы будем модифицировать текст этого блока, чтобы поэкспериментировать с различными встроенными функциями Windows, предназначенными для рисования графических объектов GDI, которые называются графическими примитивами.

А пока что скомпилируйте и скомпонуйте приложение с помощью команды **Build** или **RebuildAll** меню **Build**, после чего посредством команды **Execute** запустите программу на выполнение. Окно приложения показано на рис. 17.7. Программа осуществляет вывод в окно двух линий и небольшого сообщения.

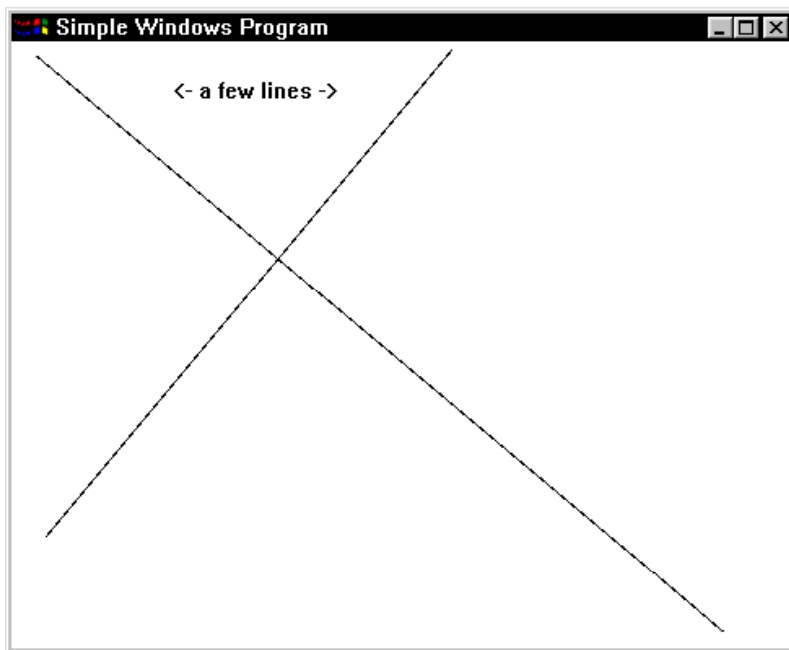


Рис. 17.7. Окно программы

Рисование эллипса

Функция `Ellipse()` используется для построения эллипса или окружности. Центр фигуры будет одновременно и центром воображаемого прямоугольника, в который вписывается эллипс. Координаты этого прямоугольника: `x1,y1` `x2,y2`.

Эллипс представляет собой замкнутую фигуру. Внутренняя область эллипса закрашивается цветом текущей кисти. Deskриптор контекста устройства задается параметром `hdc`. Все остальные параметры имеют тип `int`. Функция в случае успешного завершения возвращает значение `true`.

Синтаксис функции `Ellipse()` таков:

```
Ellipse (hdc,    x1,    y1,    x2,    y2)
```

В приведенном ниже фрагменте программы создаются небольшой эллипс и сопровождающая его надпись (рис. 17.9):

```
Ellipse(hdc, 200, 200, 275, 250);
TextOut(hdc, 210, 215, "<-an ellipse", 13);
```

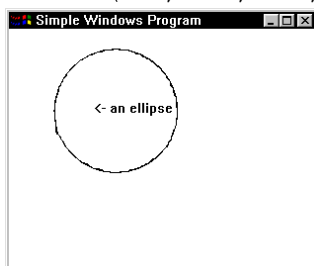


Рис. 17.9. Эллипс, нарисованный в окне приложения

Рисование сегмента

Функция `chord()` рисует замкнутую дугу, конечные точки которой соединены линией с координатами `x3,y3` и `x4,y4`. Построение фигуры осуществляется против часовой стрелки (от точки `x3,y3` до точки `x4,y4`). Внутренняя область фигуры закрашивается цветом активной кисти. Deskриптор контекста устройства задается параметром `hdc`. Все остальные параметры имеют тип `int`. В случае успешного завершения функция возвращает значение

`TRUE`

Синтаксис функции Chord():

```
Chord (hdc, x1, y1, x2, y2, x3, y3, x4, y4)
```

В приведенном ниже фрагменте программы создается сегмент и сопровождающая его надпись (рис. 17.11):

```
Chord(hdc, 550, 20, 630, 80, 555, 25, 625, 70);
```

```
TextOut(hdc, 470, 30, "A Chord ->", 11);
```

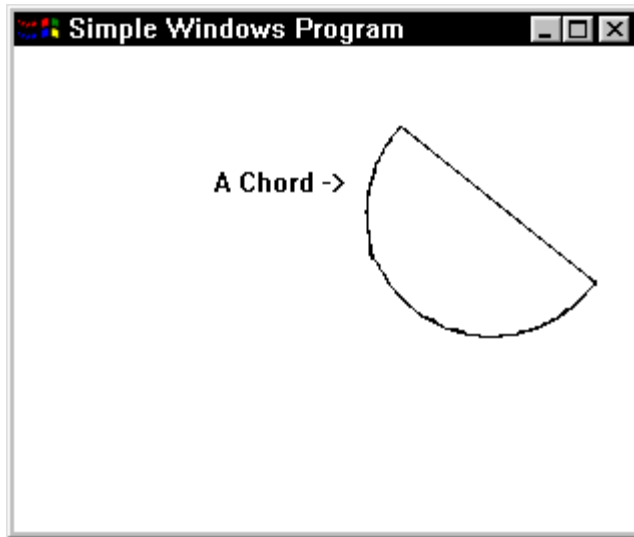


Рис. 17.11. Сегмент, нарисованный в окне приложения

Рисование сектора

Для рисования сектора предназначена функция Pie(). Центр эллипса, из которого вырезается сектор, будет одновременно и центром воображаемого прямоугольника, в который вписывается эллипс. Координаты этого прямоугольника — $x1, y1$ и $x2, y2$. Концы сектора обозначаются точками с координатами $x3, y3$ и $x4, y4$. Построение фигуры осуществляется против часовой стрелки (от точки $x3, y3$ до точки $x4, y4$). Внутренняя область сектора закрашивается цветом активной кисти. Дескриптор контекста устройства задается параметром `hdc`. Все остальные параметры имеют тип `int`. В случае успешного завершения функция возвращает значение `true`.

Синтаксис функции Pie():

```
Pie (hdc, x1, y1, x2, y2, x3, y3, x4, y4)
```

В приведенном ниже фрагменте программы создается сектор и сопровождающая его надпись (рис. 17.13):

```
Pie(hdc, 300, 50, -400, 150, 300, 50, 300, 100);
```

```
TextOut(hdc, 350, 80, "<-A Pie Wedge", 14);
```

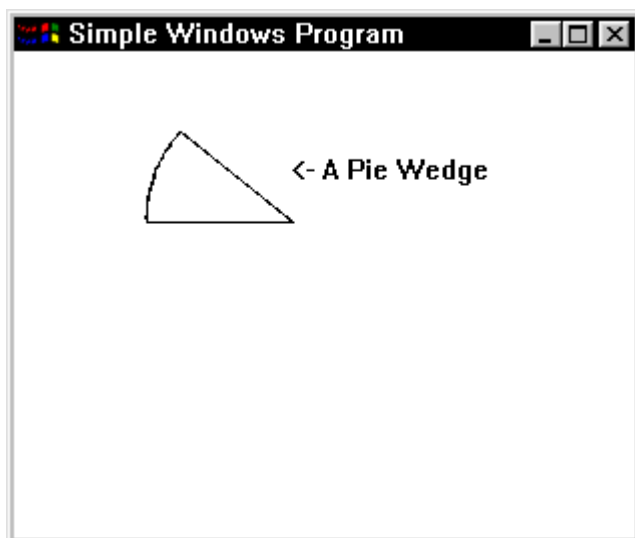


Рис. 17.13. Сектор, нарисованный в окне приложения

Рисование прямоугольника

Функция `Rectangle()` предназначена для построения прямоугольника или квадрата с использованием двух точек с координатами `x1,y1` и `x2,y2`. Получаемая замкнутая фигура закрашивается цветом текущей кисти. Дескриптор контекста устройства задается параметром `hdc`. Все остальные параметры имеют тип `int`. В случае успешного завершения функция возвращает значение `true`.

Синтаксис функции `Rectangle ()`:

```
Rectangle (hdc, x1, y1, x2, y2)
```

В приведенном ниже фрагменте программы создается прямоугольник и сопровождающая его надпись (рис. 17.14):

```
Rectangle(hdc, 50, 300, 150, 400);
TextOut(hdc, 160,350, "<- A Rectangle", 14);
```

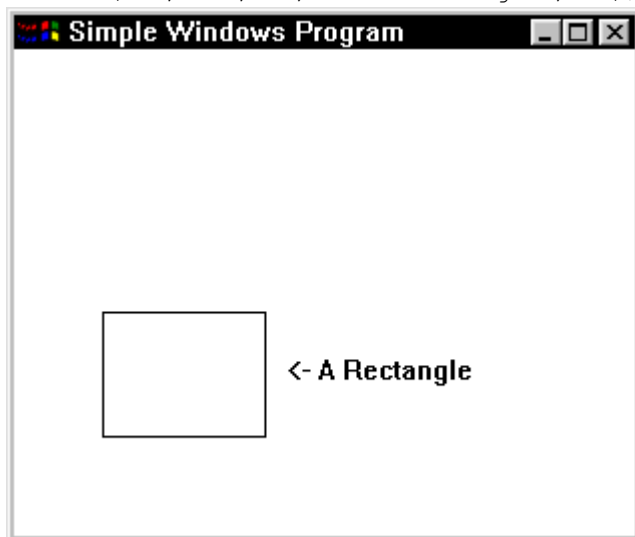


Рис. 17.14. Прямоугольник, нарисованный в окне приложения

Использование файла `SWP.C` в качестве шаблона

В предыдущих параграфах описаны принципы разработки простейшего Windows-приложения, которое теперь может послужить шаблоном для создания других приложений. Ниже показан текст приложения, строящего синусоиду.

```

/*
 * Sine.c
 * Программа, которая рисует синусоиду.
 * Построена на базе файла swp.c.
 */
#include <windows.h>
#include <math.h>
#define pi 3.14159265359
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);    : char
szProgName[]="ProgName";
int WINAPI WinMain (HINSTANCE hInst, HINSTANCE hPreInst, , , LPSTR
IpszCmdLine, int nCmdShow)
{
    HWND hWnd;
    MSG IpMsg;
    WNDCLASS wcApp;
    wcApp.lpszClassName = szProgName; wcApp.hInstance = hInst;
    wcApp.lpfnWndProc = WndProc;
    wcApp.hCursor = LoadCursor(NULL, IDC_ARROW); wcApp.hIcon =
    NULL; wcApp.lpszMenuName = NULL;
    wcApp.hbrBackground = GetStockObject(WHITE_BRUSH); wcApp.
    style = CS_HREDRAW | CS_VREDRAW; wcApp.cbClsExtra = 0;
    wcApp.cbWndExtra = 0; if (RegisterClass(&wcApp)) return 0;
    hWnd = CreateWindow(szProgName, "A Sine Wave",
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, (HWND)NULL, (HMENU)NULL, (HANDLE)hInst, (LPSTR)NULL);
    ShowWindow(hWnd, nCmdShow); UpdateWindow(hWnd);
    while (GetMessage(&IpMsg, 0, 0, 0)) { TranslateMessage(&IpMsg);
    DispatchMessage(&IpMsg);
    }
    return (IpMsg.wParam) ;
}

LRESULT CALLBACK WndProc(HWND hWnd, DINT messg,
WPARAM wParam, LPARAM lParam) {
    HDC hdc;
    PAINTSTRUCT ps;
    double y; int i;
    {
    switch (messg) {
    case WM_PAINT:
        hdc=BeginPaint(hWnd, &ps) ;
        /* построение осей координат */
        MoveToEx(hdc, 100, 50, NULL);
        LineTo(hdc, 100, 350);
        MoveToEx(hdc, 100, 200, NULL);
        LineTo(hdc, 500, 200);
        MoveToEx(hdc, 100, 200, NULL);
        /* построение синусоиды */
        for (i=0; i < 400; i++) {
            y = 120.*sin(pi*i*(360.0/400.0)/180.0) LineTo(hdc, i+100,
            (int)(200.0-y));
        }
        ValidateRect (hWnd, NULL);
        EndPaint (hWnd, &ps);
        break; case WM_DESTROY:

```

```

PostQuitMessage (0);
break;
default:
return (DefWindowProc (hWnd, messg, wParam, lParam) ) ;
break; } return (0);
}

```

Если проанализировать текст файла SINE.C и сравнить его с текстом рассмотренного ранее файла SWP.C, то можно увидеть, что внесенные изменения действительно незначительны.

Обратите внимание, что в процедуре WndProc () определяются две новые переменные:

```
double y; int i;
```

Оси координат строятся с помощью функций MoveToEx () и LineTo () :

```

/* построение осей координат */
MoveToEx(hdc, 100, 50, NULL);
LineTo (hdc, 100, 350);
MoveToEx (hdc, 100, 200, NULL) ;
LineTo (hdc, 500, 200);
MoveToEx (hdc, 100, 200, NULL) ;

```

Синусоида рисуется в цикле for. Ее амплитуда составляет 120 пикселей относительно горизонтальной оси. Для построения синусоиды используется функция $\sin()$, объявленная в файле MATH.H. Константа π необходима для преобразования значений углов из градусов в радианы.

```

/* построение синусоиды */
for(i=0; i < 400; i++)
{
y = 120.0 * sin(pi * i * (360.0 / 400.0) / 180.0);
LineTo(hdc, i+100, (int)(200.0 - y));
}

```

Окно программы показано на рис. 17.15. Поскольку в приложении не делалось никаких предположений о возможных размерах экрана, на мониторах с разной разрешающей способностью физические размеры изображения могут оказаться разными, что, как правило, нежелательно. Ниже, на примере приложения PIE.C, вы увидите, как можно избежать этого недостатка.

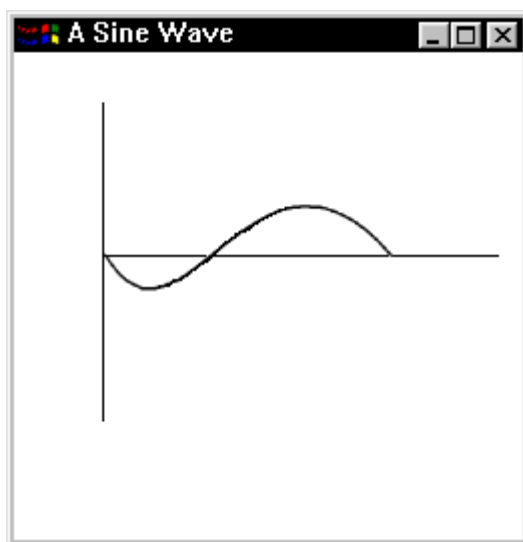


Рис. 17.15. Синусоида, выведенная на экран программой SINE.C

Создание диаграмм

Круговые диаграммы широко применяются в коммерческих приложениях при создании различного рода презентаций. В представленной ниже программе будут объединены многие приемы, рассмотренные в этой и предыдущей главах. В частности, в программе задействованы меню и разработанные нами диалоговые окна About и PieChartData. В окне ввода данных пользователю будет предложено задать до десяти значений, определяющих размеры секторов. На основании введенных целочисленных значений будут вычислены угловые размеры секторов в пропорции к общему размеру круга — 360 градусов. Минимальный размер сектора — 1 градус. Для закрашивания секторов определен массив IColor[] с набором цветов. Пользователь может также ввести подпись к диаграмме.

Для работы над данным приложением необходимо иметь четыре файла: PIE.H, PIE.RC, PIE.C и PIE.CUR. Последний, содержащий изображение указателя мыши, вы можете разработать самостоятельно с помощью редактора ресурсов.

Ниже показан текст файла заголовков PIE.H, в котором содержатся идентификаторы команд меню и элементов управления диалоговых окон:

```
#define IDM_ABOUT 10
#define IDM_INPUT 20
#define IDM_EXIT 30
#define DM_TITLE 160
#define DM_P1161
#define DM_P2162
#define DM_P3163
#define DM_P4164
#define DM_P5165
#define DM_P6166
#define DM_P7167
#define DM_P8168
#define DM_P9169
#define DM_P10170
```

В файле ресурсов PIE.RC описываются меню и два диалоговых окна (более подробно об этих ресурсах говорилось в предыдущей главе), а также ряд вспомогательных ресурсов.

```
#include "resource.h"
#define APSTUDIO_READONLY_SYMBOLS
////////////////////////
#include "pie.h"
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS
////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
////////////////////////
// Ресурсы для английского (США) языка
#if !defined(AFX_RESOURCE_DLL) || defined (AFX_TARG_END)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page (1252)
#endif // _WIN32
////////////////////////
//
// Указатель мыши
//
PIECURSOR CURSOR_DISCARDABLE "pie.cur"
////////////////////////
//
// Меню
```

```

//
PIEMENU MENU DISCARDABLE BEGIN
POPUP "Pie_Chart_Data"
BEGIN
MENUITEM      "About...",      IDM_ABOUT
MENUITEM      "Input...",      IDM_INPUT
MENUITEM      "Exit",          IDM_EXIT
END
END
////////////////////////////////////
// Диалоговые окна
//
ABOUTDLGBOX DIALOG DISCARDABLE 50,300, 180, 84
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE I WS_CAPTION I WS_SYSMENU
CAPTION "About"
FONT 8, "MS Sans Serif"
BEGIN
CTEXT "Microsoft CPie Chart Program", -1,3, 29,176,10
CTEXT "by William H. Murray and Chris H. Pappas", -1, 3, 16,176,10
PUSHBUTTON "OK", IDOK, 74,51,32,14
END
PIEDLGBOX DIALOG DISCARDABLE 93,37,195,159
STYLE DS_MODALFRAME | WS_POPUP I WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Pie Chart Data"
FONT 8, "MS Sans Serif"
BEGIN
GROUPBOX "Chart Title:",100, 5, 3, 182, 30, WSJTABSTOP
GROUPBOX "Pie Wedge Sizes:",101, 3, 34,187, 95,WS_TABSTOP
LTEXT "Title: ", -1,10,21,30,8
EDITTEXT DMJTITLE, 40,18,140, 12
LTEXT "Wedge #1:", -1, 10, 50, 40,8, NOT WS_GROUP
LTEXT "Wedge #2:", -1, 10, 65, 40,8. NOT WS_GROUP
LTEXT "Wedge #3:", -1, 10, 80, 40, 8, NOT WS_GROUP
LTEXT "Wedge #4:", -1, 10, 95, 40, 8, NOT WS_GROUP
LTEXT "Wedge #5:", -1, 10, 110, 40,8, NOT WS_GROUP
LTEXT "Wedge #6:", -1, 106,50, 40,8, NOT WS_GROUP
LTEXT "Wedge #7:", -1, 106,65, 40,8. NOT WS_GROUP
LTEXT "Wedge #8:", -1, 106,80, 40,8, NOT WS_GROUP
LTEXT "Wedge #9:", -1, 106,95, 40, 8, NOT WS_GROUP
LTEXT "Wedge #10:", -1,102, 110, 45,8, NOT WS_GROUP
EDITTEXT DM_P1, 55, 45, 30,12
EDITTEXT DM_P2, 55, 60, 30, 12
EDITTEXT DM_P3, 55, 75, 30, 12
EDITTEXT DM_P4, 55, 90, 30, 12
EDITTEXT DM_P5, 55, 105, 30, 12
EDITTEXT DM_P6, 150, 44, 30,12
EDITTEXT DM_P7, 150, 61, 30,12
EDITTEXT DM_P8, 150, 76, 30,12
EDITTEXT DM_P9, 149,91, 30,12
EDITTEXT DM_P10, 149,106,30, 12
PUSHBUTTON "OK", IDOK, 39,135, 24,14
PUSHBUTTON "Cancel", IDCANCEL, 122, 136,34,14
END
tifdef APSTUDIO_INVOKED

```



```

////////////////////////////////////
1 TEXTINCLUDE DISCARDABLE
BEGIN
"resource.h\0" END
2 TEXTINCLUDE DISCARDABLE BEGIN
#include "pie.h"\r\n"
#define APSTUDIO_HIDDEN_SYMBOLS\r\n"
#include "windows.h"\r\n".
#undef APSTUDIO_HIDDEN_SYMBOLS\r\n"
"\0"
END
3 TEXTINCLUDE DISCARDABLE
BEGIN
"\r\n"
"\0" END
#endif // APSTUDIO_INVOKED
#endif // Ресурсы для английского (США) языка

```

Следует помнить, что файл PIE.RC— это просто текстовый эквивалент описаний меню и диалоговых окон, созданных с помощью редактора ресурсов (см. предыдущую главу).

Файл PIE.C содержит основной текст программы. Несмотря на достаточно большой его размер, вы легко можете найти здесь знакомые блоки из файла SWP.C.

```

/*
* PIE.C
* Создание круговых диаграмм.
*/
#include <windows.h>
#include <string.h>
#include <math.h>
#include "pie.h"
#define radius      180
#define maxnurawedge 10
#define pi  3.14159265359
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM); BOOL CALLBACK
AboutDlgProc(HWND, UINT, WPARAM, LPARAM); BOOL CALLBACK
PieDlgProc(HWND, UINT, WPARAM, LPARAM);
char szProgName[] = "ProgName"; char szApplName[] = "PieMenu"; char
szCursorName[] = "PieCursor";
char szTString[80] = "(piechart title area)";
unsigned int iwedgesize[maxnumwedge] = {5,20,10,15};
long IColor[maxnumwedge] = {0x0L,0xFFL, '0xFF00L, 0xFFFFL,
0xFF0000L, 0xFF00FFL, 0xFFFF00L, 0xFFFFFFL, 0xS0S0L, 0x808080L};
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPreInst,
LPSTR lpszCmdLine, int nCmdShow) {
HWND hWnd;
MSG IpMsg;
WNDCLASS wcApp;
wcApp.lpszClassName = szProgName; wcApp.hInstance = hInst;
wcApp.lpfnWndProc = WndProc;
wcApp.hCursor = LoadCursor (hInst, szCursorName);
wcApp.hIcon = LoadIcon(hInst,szProgName); wcApp.lpszMenuName =
szApplName;
wcApp.hbrBackground = GetStockObject(WHITE_BRUSH); wcApp.style
= CS_HREDRAW | CS_VREDRAW; wcApp.cbClsExtra = 0; wcApp.cbWndExtra =
0; if (RegisterClass(SwcApp)) return 0;

```

```

hWnd = CreateWindow(szProgName, "PieChart Application",
WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
CW_USEDEFAULT, (HWND)NULL, (HMENU)NULL, (HANDLE)hInst, (LPSTR)NULL);
ShowWindow(hWnd, nCmdShow); UpdateWindow(hWnd); while
(GetMessage(&Msg, 0, 0, 0)) {
TranslateMessage(&Msg); DispatchMessage(&Msg); }
return (Msg.wParam); }
BOOL CALLBACK AboutDlgProc(HWND hDlg, UINT msg,
WPARAM wParam, LPARAM lParam) {
switch (msg) {
case WM_INITDIALOG:
break;
case WM_COMMAND: switch (wParam)
{
case IDOK:
EndDialog(hDlg, TRUE),
break; default:
return FALSE; }
break; default:
return FALSE; } return TRUE;
}
BOOL CALLBACK PieDlgProc(HWND hDlg, UINT msg,
WPARAM wParam, LPARAM lParam) {
switch (msg) {
case WM_INITDIALOG:
return FALSE; case WM_COMMAND: switch (wParam) {
case IDOK:
GetDlgItemText(hDlg, DMJTITLE, szTString, 80); iWedgesize[0] =
GetDlgItemInt(hDlg, DM_P1, NULL, 0) iWedgesize[1] =
GetDlgItemInt(hDlg, DM_P2, NULL, 0) iWedgesize[2] =
GetDlgItemInt(hDlg, DM_P3, NULL, 0) iWedgesize[3] =
GetDlgItemInt(hDlg, DM_P4, NULL, 0) iWedgesize[4] =
GetDlgItemInt(hDlg, DM_P5, NULL, 0) iWedgesize[5] =
GetDlgItemInt(hDlg, DM_P6, NULL, 0) iWedgesize[6] =
GetDlgItemInt(hDlg, DM_P7, NULL, 0) iWedgesize[7] =
GetDlgItemInt(hDlg, DM_P8, NULL, 0) iWedgesize[8] =
GetDlgItemInt(hDlg, DM_P9, NULL, 0) iWedgesize[9] =
GetDlgItemInt(hDlg, DM_P10, NULL, 0); EndDialog(hDlg, TRUE); break;
case IDCANCEL:
EndDialog(hDlg, FALSE); break;
default:
return FALSE; }
break; default:
return FALSE; } return TRUE;
}
LRESULT CALLBACK WndProc(HWND hWnd, UINT msg,
WPARAM wParam, LPARAM lParam) {
HDC hdc;
PAINTSTRUCT ps;
HBRUSH hBrush;
static HINSTANCE hInst1, hInst2;
static int xClientView, yClientView;
unsigned int iTotWedge[maxnumwedge+1]; int i, iNWedges;
iNWedges = 0;
for (i = 0; i < maxnumwedge; i++) {
if (iWedgesize[i] != 0) iNWedges++; } iTotWedge[0] = 0;

```

```

for(i = 0; i < iNWedges; i++)
iTotalwedge[i+1] = iTotalwedge[i] + iWedgesize[i];
switch (messg) (
case WM_SIZE:
xClientView = LOWORD(IParam); yClientView = HIWORD(IParam); break;
case WM_CREATE:
hInst1 = ((LPCREATESTRUCT) IParam)->hInstance; hInst2 =
((LPCREATESTRUCT) IParam)->hInstance; break; case WM_COMMAND:
switch (wParam) {
case IDM_ABOUT:
DialogBox(hInst1, "AboutDlgBox", hWnd, (DLGPROC) AboutDlgProc);
break; case IDM_INPUT:
DialogBox(hInst2, "PieDlgBox", hWnd, (DLGPROC) PieDlgProc);
InvalidateRect(hWnd, NULL, TRUE); UpdateWindow(hWnd); break; case
IDM_EXIT:
SendMessage(hWnd, WM_CLOSE, 0, 0);
break; default:
break; }
break; case WM_PAINT:
hdc = BeginPaint(hWnd, Sps);
SetMapMode(hdc, MM_ISOTROPIC);
SetWindowExtEx(hdc, 500, 500, NULL);
SetViewportExtEx(hdc, xClientView, -yClientView, NULL);
SetViewportOrgEx(hdc, xClientView/2, yClientView/2, NULL);
if(xClientView > 200)
TextOut(hdc, strlen (szTString) * (-8/2),
240, szTString, strlen (szTString));
for(i= 0; i < iNWedges; i++){ ' hBrush = CreateSolidBrush.(lCo,l.or
[i]) ; SelectObject(hdc, hBrush); Pie(hdc, -200, 200, 200, -
20:0,,... :
(int)(radius*cos(2*pi*iTotalWedge[i]/
iTotalWedgeJiNWedges)),
(int)(radius*sin(2*pi*iToialWedge[iJ/
ITotalWedge[iNWedges])),
(int)(radius*cos(2*pi*iTotalWedge[i+1]/
iTotalWedge tiNWedges)),
(int)(radius*sin(2*pi*iTotalWedge[i+1]/
iTotalWedge[iNWedges])); }
ValidateRect (hWnd, NULL);
EndPaint (hWnd, Sps); 1 break; case WM_DESTROY:
PostQuitMessage (0);
break; default:
return (DefWindowProc (hWnd, messg, wParam, IParam) ) ;
)
return(0);
)

```

Ниже мы несколько подробнее проанализируем содержимое приведенных файлов.

Файл PIE.H

Файл заголовков приложения содержит уникальные идентификаторы команд меню. Обратите внимание на идентификаторы с префиксом ом_, представляющие поля диалогового окна, в которые пользователь будет вводить значения.

Файл PIE.RC

Файл ресурсов приложения содержит команду подключения указателя мыши (piecursor), а также описания меню (piemenu) и двух диалоговых окон (aboutdlgbox и piedlgbox). На рис. 17.16 показано диалоговое окно **About**.

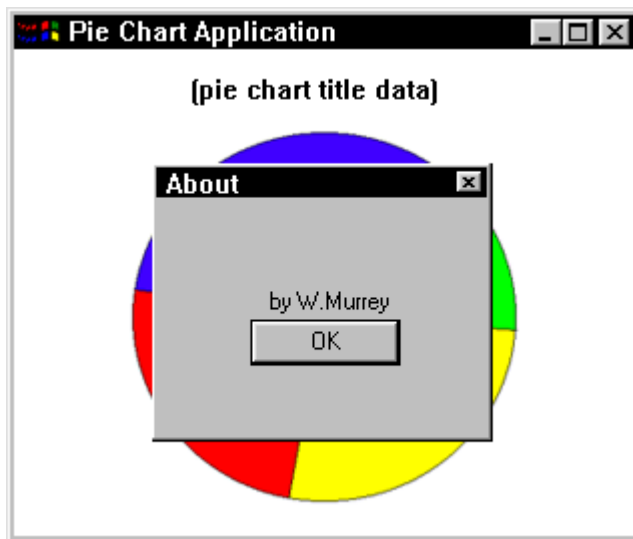


Рис. 17.16. Окно **About**

Процесс создания этих окон был рассмотрен в предыдущей главе. В описании диалогового окна для каждого элемента управления задаются его размеры и координаты на экране. Редактор ресурсов вычисляет эти значения автоматически. Впрочем, вы можете просто ввести данный файл в текстовом виде, а редактор скомпилирует его и воссоздаст описанные в нем ресурсы.

Файл PIE.C

Программа, записанная в файле PIE.C, создает круговую диаграмму, содержащую до десяти секторов. Если в меню выбрать команду **Input**, откроется диалоговое окно **PieChartData**, в котором пользователь может установить число секторов и задать их относительную ширину, а также ввести заголовок диаграммы. Эти данные передаются в программу после щелчка на кнопке **OK**. Информация, поступающая от диалоговых окон, обрабатывается в ветви caseIDOK посредством процедуры PieDlgProcO. Текст заголовка возвращается функцией GetDlgItemText(), а числовые значения — функцией GetDlgItemInt(). Последняя принимает четыре аргумента. Первые два — это дескриптор окна и идентификатор элемента управления. Третий аргумент является указателем на булеву переменную, устанавливаемую в TRUE при успешном завершении функции. В нашем случае для простоты его значение задано равным null. Четвертый аргумент определяет, следует ли вводимое значение интерпретировать как знаковое (аргумент не равен нулю) или беззнаковое (аргумент равен нулю). Введенные значения сохраняются в глобальном массиве iWedgesize[].

Основная работа выполняется в процедуре WndProc(), где обрабатываются пять сообщений: wm_size, wm_create, wm_command, wm_paint и wm_destroy. Сообщение wm_size посылается всякий раз, когда изменяются размеры окна приложения. Информация о размерах сохраняется в переменных xClientView и yClientView впоследствии, при обработке сообщения wm_paint, используется для настройки масштаба диаграммы.

В процессе обработки сообщения wm_create создаются два дескриптора приложения, hInst1 и hInst2, которые затем передаются функции DialogBox(), генерирующей экземпляры двух диалоговых окон.

При выборе какой-либо команды меню генерируется сообщение wm_command. Если это сообщение имеет подтип IDM_ABOUT, открывается окно **About**, а если подтип idm_input — окно ввода данных. При получении сообщения idm_exit приложение закрывает свое главное окно и завершает работу.

Код рисования непосредственно диаграммы содержится в блоке обработки сообщения wm_paint. По умолчанию в системе установлен режим отображения mm_text. В этом режиме

координаты объектов отсчитываются (в пикселях) начиная с верхнего левого угла окна, имеющего координаты 0,0. Вот почему в программе SINE.C вид изображения будет меняться всякий раз при изменении разрешения экрана. В программе PINE.C устанавливается режим отображения mm_isotropic.

```
SetMapMode(hdc, MM_ISOTROPIC); SetWindowExtEx(hdc, 500, 500, NULL);
SetViewportExtEx(hdc, xClientView, -yClientView, NULL); SetViewportOrgEx(hdc, xClientView/2,
yClientView/2, NULL);
```

Режимы отображения, существующие в Windows, перечислены в табл. 17.4.

Таблица 17.4. Режимы отображения	
Режим	Описание
mm_anisotropic	Одной логической единице соответствует произвольная физическая единица; масштабирование по осям x и y независимое
mm_hienglish	Одной логической единице соответствует 0,001 дюйма; ось y направлена снизу вверх
mm_himetric	Одной логической единице соответствует 0,01 мм; ось y направлена снизу вверх
mm_isotropic	Одной логической единице соответствует произвольная физическая единица; масштаб по осям x и y одинаков
mm_loenglish	Одной логической единице соответствует 0,01 дюйма; ось y направлена снизу вверх
mm_lometric	Одной логической единице соответствует 0,1 мм; ось y направлена снизу вверх
MM_TEXT	Одной логической единице соответствует один пиксель; ось y направлена сверху вниз (это режим по умолчанию)
MM_TWIPS	Одной логической единице соответствует 1/20 точки принтера (1/1440 дюйма); ось y направлена снизу вверх

Режим mm_isotropic позволяет программисту самостоятельно выбирать масштаб изображения по осям x и y. Функция SetWindowExtEx() задает размеры окна по горизонтали и вертикали (в нашем примере они равны 500). Эти логические размеры автоматически преобразуются системой в соответствии с физическими размерами экрана. В функции SetviewportExtEx() устанавливается размер области просмотра (в пикселях). Отрицательное значение координаты y означает, что ось y направлена снизу вверх. Система сравнивает размеры окна и области просмотра и вычисляет соотношение между логическими единицами и пикселями. Функция SetViewportOrgEx() устанавливает координаты точки начала области просмотра (в пикселях). В нашем примере это будет центр окна. После вызова этих функций все последующие координаты можно указывать в логических единицах.

Заголовок диаграммы выводится уже с учетом текущего режима отображения. Если размеры окна слишком малы, заголовок не показывается вообще.

```
if (xClientView > 200)
TextOutf(hdc, strlen(szTString)*(-8/2) , 240, szTString, . . ,1" strlen
(szTString) );
```

Прежде чем приступить к построению секторов диаграммы, давайте вернемся к началу процедуры WndProc и выясним, каким образом вычисляются относительные размеры секторов.

В следующем фрагменте определяется, из скольких секторов будет состоять круг:

```
iNWedges=0;
for(i=0; i < maxnumwedge; i++)
{ if (iWedgesize[i] != 0) iNWedges++;}
```

Приращение переменных iNWedges выполняется каждый раз, когда в массиве iWedgesize [i] обнаруживается ненулевое значение. Таким образом, по окончании цикла будет известно число секторов диаграммы.

Накопительные размеры секторов хранятся в массиве iTotWedge[] . Эти значения необходимы для вычисления границ между секторами. Например, если пользователь ввел в качестве размеров секторов значения 5, 10, 7 и 20, то в массиве iTotWedge[] .. будут представлены значения 0, 5, 15, 22 и 42. Вот как они получаются:

```
iTotWedge[0] = 0;
for(i= 0; i < iNWedges; i++)
```

```
iTotalWedge[i+1] = iTotalWedge [i]+ iWedgesize [i];
```

Элементы массива `iTotalWedge[]` используются для вычисления начального и конечного углов каждого сектора. Вспомните, что функция `Pie()` принимает девять параметров. Первый является дескриптором контекста устройства. Следующие четыре определяют координаты прямоугольника, в который вписывается эллипс. В нашем примере выбраны координаты -200,200 и 200,-200. Оставшиеся четыре параметра определяют координаты начальной и конечной точек сектора. Координата *x* вычисляется с помощью функции `cos()`, а координата *y* — с помощью функции `sin()`. Так, начальную координату *x* первого сектора можно определить как произведение радиуса окружности на косинус значения $2\pi \cdot iTotalWedge[0]$. Коэффициент 2π здесь необходим для преобразования градусов в радианы. Координаты конечной точки сектора определяются по тому же алгоритму, но для их вычисления используется следующий элемент массива `iTotalWedge[]` — `iTotalWedge[1]`. Чтобы все сектора вписались в круг, значение координаты делится на общий накопительный размер всех секторов — `iTotalWedge [iNWedges]`.

```
for(i = 0; i < iNWedges; i++) {
hBrush = CreateSolidBrush(lColor [i]) ;
SelectObject (hdc, hBrush);
Pie(hdc, -200, 200, 200, -200,
(int)(radius*cos (2*pi*iTotalWedge [i]/
iTotalWedge [iNWedges] )), (int)(radius*sin (2*pi*iTotalWedge [i]/
iTotalWedge [ iNWedges] )), (int)(radius*cos(2*pi*iTotalWedge[i+1] /
iTotalWedge [iNWedges] )),
(int)(radius*sin(2*pi*iTotalWedge[i+1] / ,
iTotalWedge [ iNWedges ] ));
}
```

Весь цикл повторяется столько раз, сколько указано в переменной `iNWedges`. Диаграмма, представленная на рис. 17.18, строится по умолчанию, а показанная на рис. 17.19 — на основании значений, введенных пользователем.

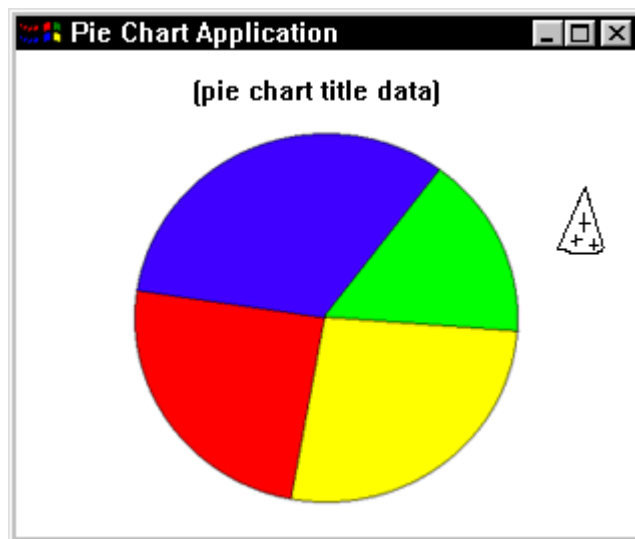


Рис. 17.18. Диаграмма, построенная по умолчанию

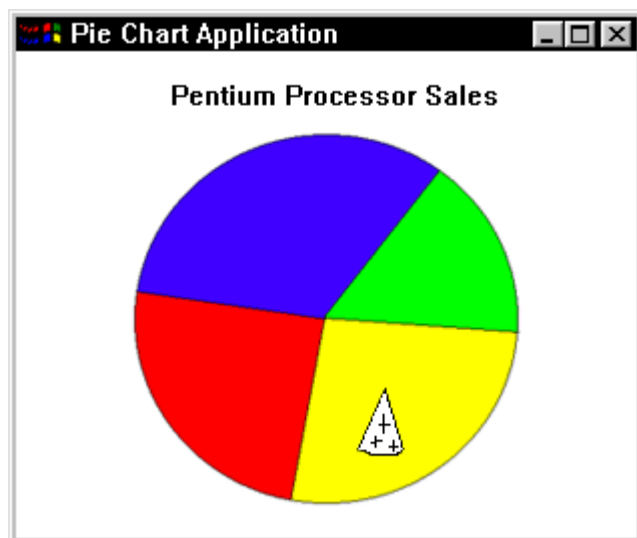


Рис. 17.19. Диаграмма, построенная с учетом введенных пользователем данных

Глава 18. Основы библиотеки MFC

- Зачем нужны библиотеки классов
- Принципы, лежащие в основе MFC
- Ключевые особенности MFC
- Все начинается с CObject
- Иерархия классов MFC
- Простейшее MFC -приложение
 - Файл SIMPLE.CPP
 - Файл AFXWIN.H
 - Создание класса, производного от CWinApp
 - Класс CFrameWnd
 - Реализация метода InitInstance()
 - Конструктор
 - Запуск программы

Из предыдущих глав вы узнали, что создание даже самых простых приложений Windowsc использованием стандартных API-функций является довольно сложной задачей, требующей много времени и внимания. Например, код простейшего приложения SWP.C(мы рассматривали его в предыдущей главе) на языке C занял почти две страницы текста. Большая часть кода нужна лишь для того, чтобы вывести на экран окно приложения и поддерживать его работу.

Компилятор MicrosoftVisualC++ предлагает мощную библиотеку классов (MicrosoftFoundationClasses— MFC), предназначенную для разработки 32-разрядных приложений, отвечающих всем стандартам и требованиям сегодняшнего дня. В MFC инкапсулированы все API-функции Windowsи дополнительно обеспечивается поддержка панелей инструментов, страниц свойств, технологии OLE, элементов управления ActiveXи многое другое. Кроме того, имеются функции управления базами данных, основанными на разных источниках данных, включая OAO и ODBC. Поддерживается и разработка приложений для Internet.

В настоящей главе обсуждаются преимущества использования библиотеки MFC при разработке приложений Windows, рассмотрена используемая терминология и описана базовая методика программирования. MFC будет применяться во всех примерах программ в оставшихся главах этой книги. О том, какую роль играет MFC в программировании для Windows, можно судить по тому факту, что на рассмотрение процедурного программирования отведена одна глава, тогда как все оставшиеся главы посвящены изучению объектно-ориентированного программирования с использованием MFC .

Зачем нужны библиотеки классов

MFC предоставляет программистам удобные классы объектов, в которых инкапсулированы все наиболее важные структуры и API-функции. Библиотеки классов, подобные MFC , обеспечивают гораздо больше возможностей и значительно проще в использовании, чем традиционные библиотеки функций языка C, о которых говорилось в двух предыдущих главах.

Ниже перечислены основные преимущества использования библиотек классов:

- устраняются конфликты, связанные с совпадением имен стандартных и программных функций, а также переменных;

- код и данные инкапсулируются в классах, и доступ к ним может быть ограничен;
- обеспечивается наследование кода;
- размер кода значительно сокращается.

Благодаря MFC код, требуемый для отображения окна приложения (мы рассматривали его в предыдущей главе), можно сократить примерно в три раза. При этом программист получает возможность уделять больше времени и внимания не разработке процедур взаимодействия приложения с Windows, а реализации тех задач, для решения которых и создается приложение.

Принципы, лежащие в основе MFC

Создатели MFC при разработке данного продукта придерживались строгих правил и стандартов. Основные принципы, на которых базируется MFC, перечислены ниже:

- возможность комбинировать обычные вызовы функций с использованием методов классов;
- баланс между производительностью и эффективностью базовых классов;
- преемственность в переходе от использования API-функций к библиотеке классов;
- простота переноса библиотеки классов на разные платформы: от Windows 3.1 к Windows 95/NT, затем к Windows 98 и т.д.;
- органичная взаимосвязь с традиционными средствами программирования на C++, позволяющая избежать чрезмерного усложнения программного кода.

Создатели библиотеки осознавали, что от совершенства ее программного кода зависит эффективность работы приложения, основанного на MFC. Четко соблюдались требования к размеру классов и скорости выполнения их методов. В результате по скорости работы классы MFC ничем не уступают библиотечным функциям языка C.

Одна из поставленных перед разработчиками MFC задач состояла в том, чтобы программисты, уже знакомые с API-функциями, не заучивали новый, далеко не малый набор имен функций и констант. Это требование строго соблюдалось при именовании членов классов. Данная особенность выгодно отличает MFC от других библиотек классов.

При создании MFC также учитывалась возможность работы в смешанном режиме. Другими словами, в одной программе могут применяться как библиотека MFC, так и API-функции. Некоторые функции, например SetCursor() и GetSystemMetrics(), должны вызываться напрямую, даже если в приложении используется MFC. Еще одно важное свойство, на которое обращали внимание разработчики Microsoft, — это возможность непосредственного использования базовых классов в программах. Существовавшие до MFC библиотеки классов были чересчур абстрактными. В Microsoft их называли "тяжелыми классами", поскольку основанные на них приложения отличались большим размером программного кода и недостаточно быстро выполнялись. Разработчикам MFC удалось найти золотую середину между разумным уровнем абстрактности и размером кода.

Ключевые особенности MFC

Ниже перечислены основные достоинства библиотеки MFC.

- Расширенная система обработки исключительных ситуаций, благодаря которой приложения менее чувствительны к ошибкам и сбоям. Ошибки типа "нехватка памяти" обрабатываются автоматически.
- Улучшенная система диагностики, позволяющая записывать в файл информацию об используемых объектах. Также сюда следует отнести возможность контроля за содержимым переменных-членов.

- Полная поддержка всех API-функций, элементов управления, сообщений, GDI, графических примитивов, меню и диалоговых окон.
- Возможность определить тип объекта во время выполнения программы. Это позволяет осуществлять динамическое управление переменными-членами в случае использования разных экземпляров класса.
- Отпала необходимость в использовании многочисленных громоздких структур switch/case, которые часто являются источниками ошибок в процедурном программировании. Все сообщения связываются с обработчиками внутри классов.
- Небольшой размер и быстрота выполнения программного кода классов. Как уже говорилось выше, по этим показателям MFC не уступает стандартным библиотечным функциям языка C.
- Поддержка COM (ComponentObjectModel— модель компонентных объектов).
- Использование тех же соглашений об именовании методов классов, которые применялись при подборе имен для API-функций Windows. Это существенно облегчает идентификацию действий, выполняемых классами.

Опытный программист сразу же оценит две наиболее важные особенности MFC : знакомые имена методов и привязка сообщений к обработчикам. Если вы вернетесь к программе PIE.C, рассмотренной в предыдущей главе, то убедитесь в обилии блоков switch/case. Увлечение такими конструкциями чревато возникновением трудно обнаруживаемых ошибок при выполнении программы. Эту проблему легко обойти с помощью MFC .

Профессиональный разработчик будет удовлетворен быстротой выполнения программного кода библиотечных классов. Интенсивное использование MFC теперь не влечет к тяжеловесности полученного приложения.

Все начинается с CObject

Большинство библиотек классов, в том числе и MFC , имеют какой-нибудь один общий родительский класс, от которого порождаются все остальные. В MFC таковым является CObject. Ниже показано описание этого класса, взятое из файла AFX.H. (Файлы заголовков библиотеки MFC хранятся в папке MFC /INCLUDE.)

```
////////// Класс CObject является базовым для классов MFC
class CObject
{
public:
// Модель объекта (проверка типа, выделение памяти, уничтожение)
virtual CRuntimeClass* GetRuntimeClass()
const/virtual ~CObject (); // необходим виртуальный деструктор
// Резервирование и удаление памяти
void* PASCAL operator new(size_t nSize);
void* PASCAL operator new(size_t, void* p) ;
void PASCAL operator delete(void* p) ;
#ifdef _DEBUG && ! defined (_AFX_NO_DEBUG_CRT) '?', '_
// позволяет отслеживать имя файла и номер строки
// (используется при отладке)
void* PASCAL operator new(size_t nSize, LPCSTR lpszFileName,
int nLine); #endif
// Конструктор копирования и оператор инициализации с присваиванием
// недоступны, поэтому в случае попытки использовать соответствующие
// конструкции будет выдано сообщение об ошибке компилятора,
// что позволит избежать непредсказуемости выполнения программы.
protected: CObject ();
private:
```

```

CObject(const CObjectS objectSrc);          //реализация отсутствует
void operator=(const CObjectS objectSrc); // реализация отсутствует
// Атрибуты
public:
BOOL IsSerializableOconst;
BOOL IsKindOf(const CRuntimeClass* pClass) const;
// Виртуальные методы
virtual void Serialize (CArchiveS ar) ;
// Поддержка диагностики
virtual void AssertValidO const;
virtual void Dump (CDumpContextS. dc) const;
// Реализация
public:
static const AFX_DATA CRuntimeClass classCObject;
#ifdef _AFXDLL
static CRuntimeClass* PASCAL _GetBaseClass();
#endif
};
}

```

В данном примере программный код для большей ясности слегка отредактирован, но в целом это тот же код, который вы найдете в указанном файле.

Обратите внимание, из каких компонентов состоит описание класса CObject. Как видите, здесь четко прослеживается выделение блоков открытых (public), закрытых (private) и защищенных (protected) членов класса. CObject также обеспечивает возможность динамического определения типа и сериализации. Вспомните, что возможность динамической проверки типа позволяет определять тип объекта во время выполнения программы. Сведения о состоянии объекта можно сохранить в файле на диске, благодаря чему реализуется концепция постоянства.

Все остальные классы MFC порождаются от CObject. Примером такого класса может служить CGdiObject (объявлен в файле AFXWIN.H), описание которого приведено ниже.

```

//////////
// Абстрактный класс CGdiObject для метода CDC: :SelectObject
class CGdiObject : public CObject {
DECLARE_DYNCREATE (CGdiObject)
public:
// Атрибуты
HGDIOBJ m_hObject; // должна быть первой переменной-членом
operator HGDIOBJ () const;
HGDIOBJ GetSafeHandleO const;
static CGdiObject* PASCAL FromHandle (HGDIOBJ hObject) ;
static void PASCAL DeleteTempMap () ;
BOOL Attach (HGDIOBJ hObject) ;
HGDIOBJ Detach () ;
// Конструкторы
CGdiObject() ; // должен создавать объект производного класса
BOOL DeleteObject() ;
// Методы
int GetObject (int nCount, LPVOID lpObject) const;
UINT GetObjectTypeO const;
BOOL CreateStockObject (int nIndex) ;
BOOL UnrealizeObjectO ;
BOOL operator- (const CGdiObjectS obj) const;
root. nfn=rator!= (const CGdiObjectS obi) const;
// Реализация

```

```
public:
virtual ~CGdiObject() ; #ifdef _DEBUG
virtual void Dump (CDumpContextS. dc) const;
virtual void AssertValid() const; #endif };
```

Класс CGdiObject и его методы позволяют создавать и использовать в приложениях такие графические объекты, как перья, кисти и шрифты. От CGdiObject порождаются некоторые другие классы, в частности CPen.

Библиотека MFC поставляется вместе с исходными текстами классов, что дает возможность программистам настраивать базовые классы в соответствии со своими потребностями. Впрочем, для начинающих программистов нет никакой необходимости заниматься редактированием MFC и даже знать, как реализован тот или иной класс.

Например, в традиционных процедурных программах вызов функции DeleteObject() имеет следующий синтаксис:

```
DeleteObject(hBRUSH); /* где hBRUSH- дескриптор кисти */
```

В MFC -приложениях того же результата можно достичь путем вызова функции-члена:

```
newbrush.DeleteObject (); // где newbrush- текущая кисть
```

Иерархия классов MFC

Ниже показан список классов библиотеки MFC , порожденных от CObject.

CObject

```
CException
CArchiveException CDaoException CDBException
CFileException
CInternetException
    CMemoryException
CNotSupportedException
COleDispatchException
COleException
CResourceException
CUserException
CFile
CMemFile
CSharedFile
CFileStreamFile
CMonikerFile
CAsyncMonikerFile
CDataPathProperty
CCachedDataPathProperty
CSocketFile
CStdioFile
CInternetFile
CGopherFile
CHttpFile
CRecentFileList CDC
CClientDC
CMetaFileDC
CPaintDC
CWindowDC
CDocState
CImageList
CGdiObject
CBitmap
CBrush
```

CFont
CPalette
CPen
CRgn CMenu
CCommandLineInfo
CDatabase
CRecordSet
CLongBinary
CDaoDatabase
CDaoQueryDef
CDaoRecordSet
CDaoTableDef
CDaoWorkspace
CSyncObject
CCriticalSection
CEvent
CMutex
CSemaphore CAsyncSocket
CSocket
CArray
CByteArray
CWordArray
CObArray
CPtrArray
CStringArray
CUIIntArray
CWordArray
CList
CObList
CPtrList
CStringList
CMap
CMapWordToOb
CMapWordToPtr
CMapPtrToPtr
CMapPtrToWord
CMapStringToOb
CMapStringToPtr
CMapStringToString
CInternetSession
CInternetconnection
CFtpConnection
CGopherConnection
CHttpConnection
CFileFind
CFtpFileFind
CGopherFileFind
CGopherLocator
CCmdTarget
CWinThread
CWinApp
COleControlModule
CDocTemplate
CMultiDocTemplate

CSingleDocTemplate
COleObjectFactory
COleTemplateServer
COleDataSource
COleDropSource
COleDropTarget
COleMessageFilter
CConnectionPoint
CDocument
COleDocument
COleLinkingDoc
COleServerDoc
CRichEditDoc CDocItem
COleClientItem
COleDocObjectItem
CRichEditCtrlItem
COleServerItem
CDocObjectServerItem
CDocObjectServer

CWnd

CFrameWnd
CMDIChildWnd
CMDIFrameWnd
CMiniFrameWnd
COleIPFrameWnd
CSplitterWnd
CControlBar
CDialogBar
COleResizeBar
CReBar
CStatusBar
CToolBar
CPropertySheet
CPropertySheetEx
CDialog
CCommonDialog
CColorDialog
CFileDialog
CFindReplaceDialog
CFontDialog
COleDialog
COleBusyDialog
COleChangeIconDialog
COleChangeSourceDialog
COleConvertDialog
COleInsertDialog
COleLinksDialog
COleUpdateDialog
COlePasteSpecialDialog
COlePropertiesDialog
COlePageSetupDialog
CPrintDialog
COlePropertyPage
CPropertyPage

CPropertyPageEx CView
CCtrlView
CEditView
CListView
CRichEditView
CTreeView
CScrollView
CFormView
CDaoRecordView
CHtmlView
COleDBRecordView
CRecordView
CAnimateCtrl
CButton
CBitmapButton
CComboBox
CComboBoxEx
CDateTimeCtrl
CEdit
CHeaderCtrl
CHotKeyCtrl
CIPRddressCtrl
CListBox
CCheckListBox
CDragListBox
CListCtrl
CMonthCalCtrl
COleControl
CProgressCtrl
CReBarCtrl
CRichEditCtrl
CScrollBar
CSliderCtrl
CSpinButtonCtrl
CStatic
CStatusBarCtrl
CTabCtrl
CToolbarCtrl
CToolTipCtrl
CTreeCtrl

В следующем списке перечислены классы, которые не порождены от CObject.

CHtmlStream
CHttpFilter
CHttpFilterContext
CHttpServer
CHttpServerContextf
CArchive
CDumpContext
CRuntimeClass
CPoint
CRect
CSize
CString

```

CTime
CTimeSpan
CCreateContext
CMemoryState
COleSafeArray
CPrintInfo
CCmdUI
COleCmdOI
CDaoFieldExchange
CDataExchange
CDBVariant
CFieldExchange
COleDataObject
COleDispatchDriver
CPropExchange
CRectTracker
CWaitCursor
CTypedPtrArray
CTypedPtrList
CTypedPtrMap
CFontHolder
CPictureHolder
COleCurrency
COleDateTime
COleDateTimeSpan
COleVariant
CMultiLock
CSingleLock

```

Эти два списка могут служить вам справочником при дальнейшем изучении возможностей MFC .

Простейшее MFC -приложение

Прежде чем приступить к созданию более сложных программ, обратимся к основе любого приложения — выводу на экран окна. В предыдущей главе мы реализовали эту возможность с помощью кода на языке C, текст которого, как вы помните, занял около двух страниц. Объем той же программы, написанной с использованием MFC , будет примерно в три раза меньшим.

Рассматриваемая ниже программа SIMPLE.CPP просто открывает на экране свое окно, помещая а строку заголовка определенный текст.

Файл SIMPLE.CPP

Чтобы создать простейшее Windows-приложение на базе MFC , введите в окне компилятора VisualC++ следующий код: .

```

//
//  simple.cpp
//  Пример приложения, написанного с использованием MFC .
//
#include <afxwin.h>
Class CTheApp : public CWinApp
(
public:
virtual BOOL InitInstance (); );
class CMainWnd : public CFrameWnd
(

```



```

public:
CMainWnd() (
Create(NULL, "Hello MFC World",
WSJDVERLAPPEDWINDOW, rectDefault, NULL, NULL), )
};
BOOL CTheApp: : InitlInstance 0
{
m_pMainWnd = new CMainWnd (); m_pMainWnd->ShowWindow (m_nCmdShow) ;
m_pMainWnd->UpdateWindow() ;
return TRUE; }
CTheApp TheApp;

```

В следующих параграфах мы детально рассмотрим назначение каждого блока программы.

Файл AFXWIN.H

Файл AFXWIN.H выполняет роль шлюза в библиотеку MFC . Через него подключаются все остальные файлы заголовков, включая WINDOWS. H. Использование файла AFXWIN.H упрощает создание предварительно скомпилированных файлов заголовков. Предварительная компиляция частей программы позволяет сократить время, затрачиваемое на повторное построение приложения.

Создание класса, производного от CWinApp

Приложение начинается с определения класса CTheApp, являющегося производным от CWinApp:

```

class CTheApp : public CWinApp
{
public:
virtual BOOL InitlInstance () ; };

```

Виртуальный метод InitlInstance() наследуется от CWinApp. Переопределяя этот метод, программист получает возможность управлять инициализацией приложения. В классе CWinApp применяются также открытые виртуальные функции ExitlInstance(), Run() и другие, но в большинстве приложений нет необходимости переопределять их.

Ниже приведено описание класса CWinApp, взятое из файла AFXWIN.H.

```

//////////
// CWinApp- базовый класс для всех приложений Windows
class CWinApp : public CWinThread {
DECLARE_DYNAMIC (CWinApp) public:
// Конструктор
CWinApp(LPCTSTR lpszAppName =• NULL); // имя приложения
// задается по умолчанию
// Атрибуты
// параметры запуска (не изменять)
HINSTANCE m_hlInstance;
HINSTANCE m_hPrev!Instance;
LPCTSTR m_lpCmdLine;
int m_nCmdShow;
// параметры выполнения (могут меняться в InitlInstance)
LPCTSTR m_pszAppName; // читаемое имя приложения
LPCTSTR m_pszRegistryKey; // используется для регистрации
CDocManager* m_pDocManager;
public: // устанавливаются в конструкторе
LPCTSTR m_pszExeName; // имя исполняемого файла (без пробелов)
LPCTSTR m_pszHelpFilePath; // определяется на основании пути к модулю

```

```

LPCTSTR m_pszProfileName; // определяется на основании имени
приложения
// Операции инициализации - должны выполняться в
InitInstance-protected:
void LoadStdProfileSettings(UINT nMaxMRU = _AFX_MRU_COUNT);
void EnableShellOpen ();
void SetDialogBkColor(COLORREF clrCtlBk = RGB(192,192,192),
COLORREF clrCtlText= RGB(0, 0, 0)); // установка фонового цвета
диалогового окна и окна сообщений
void SetRegistryKey(LPCTSTR lpszRegistryKey);
void SetRegistryKey(UINT nIDRegistryKey);
// позволяет хранить данные о приложении в реестре, а не в INI-файле
// (в качестве ключа обычно используется название компании)
BOOL Enable3dControls(); //для 3-мерных элементов управления
// используется файл CTL3D32.DLL ifndef _AFXDLL
BOOL EnableSdControlsStatic(); // для тех же целей статически
компилируется
// файл CTL3D.LIB endif
void RegisterShellFileTypes(BOOL bCompat=FALSE);
// вызывается после того, как будут зарегистрированы все шаблоны
// документа
void RegisterShellFileTypesCompat();
// требуется для совместимости с предыдущими версиями
void UnregisterShellFileTypes();
// Вспомогательные операции - обычно выполняются в InitInstance
public:
// Указатели мыши
HCURSOR LoadCursor(LPCTSTR lpszResourceName) const;
HCURSOR LoadCursor(UINT nIDResource) const;
HCDROR LoadStandardCursor(LPCTSTR lpszCursorName) const;
HCURSOR LoadOEMCursor(UINT nIDCursor) const;
// Значки
HICON LoadIcon(LPCTSTR lpszResourceName) const;
HICON LoadIcon(UINT nIDResource) const;
HICON LoadStandardIcon(LPCTSTR lpszIconName) const;
HICON LoadOEMIcon(UINT nIDIcon) const;
// могут переопределяться
virtual BOOL InitInstance();
virtual int ExitInstance(); // возвращает код завершения приложения
virtual int Run();
virtual BOOL OnIdle(LONG lCount);
virtual LRESULT ProcessWindowProcException(CException* e, const MSG*
pMsg);
public:
virtual CWinApp();
protected:
// { {AFX_MSG (CWinApp)
afx_msg void OnAppExit();
afx_msg void OnUpdateRecentFileMenu (CCmdUI* pCmdUI) ;
afx_msg BOOL OnOpenRecentFile (UINT nID) ;
//}}afx_msg declare_message_map ( ) };

```

Класс CWinApp отвечает за создание и работу цикла сообщений, который рассматривался в главе 5. Его использование позволяет избежать написания повторяющегося кода и тем самым сократить размер программы.

Класс CFrameWnd

Окно приложения, выводимое на экран и управляемое классом CMainWnd, создается на базе класса CFrameWnd:

```
class CMainWnd : public CFrameWnd
{
public:
    CMainWnd () (
        Create (NULL, "Hello MFC World",
            WS_OVERLAPPEDWINDOW, rectDefault, NULL, NULL);
    }
};
```

Конструктор класса CMainWnd вызывает метод Create(), предназначенный для установления начальных параметров окна. В данном примере задается стиль окна и строка заголовка. В главе 7 будет показано, что с помощью этой же функции можно задать строку меню и таблицу горячих клавиш.

Ниже представлено описание класса CFrameWnd, взятое из файла AFXWIN.H.

```
//////////
// CFrameWnd- базовый класс для масштабируемых окон
class CFrameWnd : public CWnd
{
    DECLARE_DYNCREATE(CFrameWnd)
    // Конструкторы
public:
    static AFX_DATA const CRect rectDefault;
    CFrameWnd();
    BOOL LoadAccelerTable(LPCTSTR lpszResourceName); BOOL Create(LPCTSTR
        lpszClassName,
        LPCTSTR lpszWindowName,
        DWORD dwStyle = WS_OVERLAPPEDWINDOW,
        const RECT& rect = rectDefault,
        CWnd* pParentWnd = NULL, // != NULL для всплывающих окон
        LPCTSTR lpszMenuName = NULL,
        DWORD dwExStyle = 0,
        CCreateContext* pContext = NULL);
    // динамическое создание - загружает рамку и связанные с окном ресурсы
    virtual BOOL LoadFrame(HINT nIDResource,
        DWORD dwDefaultStyle = WS_OVERLAPPEDWINDOW |
        FWS_ADDTOTITLE,
        CWnd* pParentWnd = NULL,
        CCreateContext* pContext = NULL);
    // специальная функция для создания области просмотра
    CWnd*. CreateView(CCreateContext* pContext,
        UINT nID = AFX_IDW_PANE_FIRST);
    // прикрепление панелей инструментов
    void EnableDocking(DWORD dwDockStyle);
    void DockControlBar(CControlBar* pBar, UINT nDockBarID = 0,
        LPCRECT lpRect = NULL);
    void FloatControlBar(CControlBar* pBar, CPoint point, DWORD dwStyle =
        CBS_ALIGN_TOP); CControlBar* GetControlBar(UINT nID);
    // Реализация
public:
    virtual ~CFrameWnd ();
    int m_nWindow; // номер окна - отображается как ":n"
```

```

HMENU m_hMenuDefault;          // ресурс меню
HACCEL m_hAccelTable;          // таблица горячих клавиш
DWORD m_dwPromptContext;
BOOL m_bHelpMode; // если TRUE, активен режим вызова справки
// по[Shift+F1]
CFrameWnd* m_pNextFrameWnd; // следующее окно в списке приложения
CRect m_rectBorder;
COleFrameHook* m_pNotifyHook;
CPtrList m_listControlBars; // массив панелей инструментов,
// связанных с данным окном int m_nShowDelay;
// обработчики оконных сообщений
afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnDestroy();
afx_msg void OnClose();
afx_msg void OnInitMenu(CMenu*);
afx_msg void OnInitMenuPopup(CMenu*, UINT, BOOL);
afx_msg void OnMenuSelect(OINT nItemID, UINT nFlags,
HMENU hSysMenu);
afx_msg LRESULT OnPopMessageString(WPARAM wParam, LPARAM lParam) ;
afx_msg LRESULT OnSetMessageString(WPARAM wParam, LPARAM lParam),
protected:
afx_msg LRESULT OnDDEInitiate(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnDDEExecute(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnDDETerminate(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnRegisteredMouseWheel(WPARAM wParam, LPARAM lParam);
DECLARE_MESSAGE_MAP ()
friend class CWinApp; );

```

Первый параметр метода Create() позволяет задать имя класса окна в соответствии с синтаксисом стандартной API-функции RegisterClass(). Обычно этот параметр не используется и равен null.

Реализация метода InitInstance()

В классе CTheApp переопределяется метод initInstance() базового класса

```

CWinApp:
BOOL CTheApp::InitInstance()
{
    m_pMainWnd=new CMainWndO;
    m_j>MainWnd->ShowWindow (m_nCmdShow) ;
    m_pMainWnd->UpdateWindow();
    return TRUE; }

```

Оператор new вызывает конструктор CMainWnd (), рассмотренный в предыдущем параграфе. Переменная-член m_pMainWnd(префикс ga_ указывает на переменную-члена класса) определяет положение окна на экране. Функция ShowWindow() выводит окно на экран. Параметр m_nCmdShow инициализируется конструктором класса CWinApp и определяет параметры отображения окна. Функция UpdateWindow() перерисовывает содержимое окна.

Конструктор

В последнем блоке программы вызывается конструктор класса CWinApp:

```
CTheApp TheApp;
```

Запуск программы

Рассмотренная нами программа очень проста. Приложение просто отображает свое окно, ничего не выводя в нем (рис. 18.1).



Рис. 18.1. Окно простейшей MFC -программы

Многократное использование одних и тех же базовых классов в различных приложениях — это основа проектирования программ на C++. Библиотеку MFC можно рассматривать как естественное и органичное расширение данного языка. Созданный нами программный код является основой для всех MFC -приложений, разрабатываемых в этой книге. В следующей главе мы создадим шаблон приложения, в котором осуществляется вывод информации в рабочую область окна.

Глава 19. Создание MFC -приложений

- Шаблон MFC -приложения
 - Файл MFC SWP.H
 - Файл MFC SWP.CPP
 - Запуск программы
- Рисование графических примитивов в рабочей области окна
 - Файл GDI.CPP
 - Запуск программы
- Построение ряда Фурье
 - Файл FOURIER.H
 - Файлы ресурсов
 - Файл FOURIER.CPP
 - Запуск программы
- Построение гистограмм
 - Файл BARCHART.H
 - Файлы ресурсов
 - Файл BARCHART.CPP
 - Запуск программы

Ранее, в главах 16 и 17, нами были рассмотрены ресурсы Windows, такие как меню, диалоговые окна, горячие клавиши. В предыдущей главе речь шла об основных принципах применения библиотеки MFC при создании приложений Windows. В настоящей главе мы попытаемся, воспользовавшись полученными знаниями о ресурсах и MFC , создать несколько полноценных 32-разрядных приложений.

Проанализированы будут четыре различных приложения, что поможет вам лучше понять, как на практике используются возможности, предоставляемые библиотекой MFC . Примеры подобраны таким образом, чтобы знания, полученные при изучении одной программы, закреплялись при работе со следующей. После ознакомления с четвертым приложением вы будете иметь опыт, необходимый для самостоятельной работы по созданию профессиональных MFC -приложений.

Шаблон MFC -приложения

В предыдущей главе вы познакомились с простейшей MFC -программой, которая отображала на экране свое окно. Ее код можно рассматривать как основу для любого другого приложения Windows, выводящего в окно текст или графику.

Первое из рассматриваемых в этой главе приложений, которое называется MFC SWP (MFC SimpleWindowsProgram), просто выводит строку текста в рабочую область окна. Прежде чем приступить к его изучению, обратимся к листингу программы. Собственно говоря, ниже показано два листинга: файл заголовков и программный файл.

Вспомните, что в предыдущей главе содержимое файла заголовков было включено в программный файл. Здесь в файле заголовков приводится информация о том, как классы приложения порождаются от базовых классов библиотеки MFC . Такой стиль рекомендован разработчиками Microsoft. Вот текст файла MFC SWP.H:

```

class CMainWnd : public CFrameWnd
{
public:
CMainWnd();
afx_rasg void OnPaint();
DECLARE_MESSAGE_MAP() ;
};
class CmfcswpApp : public CWinApp
{
public:
BOOL InitInstance(); } ;

```

Ниже показан текст файла MFC SWP.CPP:

```

//
// mfcswp.cpp
// Эту программу можно использовать как шаблон
// для разработки других MFC -приложений.
//
#include <afxwin.h>
#include "mfcswp.h"
CmfcswpApp theApp;
CMainWnd::CMainWnd() {
Create(NULL, "A MFC Windows Application",
WS_OVERLAPPEDWINDOW, rectDefault, NULL, NULL); }
void CMainWnd: :OnPaint () {
CPaintDC dc(this);
dc.TextOut(200, 200, "Using the MFC Library", 21); }
BEGIN_MESSAGE_MAP(CMainWnd, CFrameWnd)
ON_WM_PAINT() END_MESSAGE_MAP ()
BOOL CmfcswpApp::InitInstance() {
m_pMainWnd = new CMainWnd();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow() ;
return TRUE;
}

```

Файл MFC SWP.H

В MFC -приложениях обычно используются файлы заголовков двух типов. Файлы первого типа, наподобие рассматриваемых в этом параграфе, описывают порождение классов приложений от базовых классов MFC . Их имена совпадают с именами приложений и имеют расширение H. Файлы второго типа содержат идентификаторы элементов меню и диалоговых окон, а также описания различных констант. Для распознавания файлов этого типа мы будем в их именах добавлять букву R, что означает "resource" (ресурс). Такие файлы встретятся нам в этой главе в двух последних примерах программ.

Итак, в нашем файле заголовков содержится описание двух классов: CMainWnd, порождаемого от CWinApp, и CmfcswpApp, потомка CFrameWnd.

```

class CMainWnd : public CFrameWnd
public:
CMainWnd () ;
afx_msg void OnPaint();
DECLARE_MESSAGE_MAP () ;
);
class CmfcswpApp : public CWinApp
public:
BOOL InitInstance();
}

```

Класс CMainWnd содержит метод OnPaint() и макрос, связанный с обработкой сообщений. В случае обработчиков сообщений, например OnPaint(), вместо ключевого слова virtual используется afx_msg. Наличие метода OnPaint() позволяет изменять содержимое рабочей области окна приложения. Эта функция автоматически вызывается каждый раз, когда объекту CMainWnd передается сообщение WM_PAINT.

Макрос declare_message_map применяется практически во всех MFC -приложениях и сигнализирует о том, что в классе организуется собственная схема обработки сообщений. Разработчики Microsoft рекомендуют использовать схемы сообщений, а не просто вызовы виртуальных функций, так как благодаря этому удастся значительно сократить размер программного кода.

Файл MFC SWP.CPP

В основу нашего приложения положен код рассматривавшейся в главе 17 программы SIMPLE.CPP, однако между этими кодами имеется несколько существенных отличий. Они касаются, в частности, метода OnPaint(). Проанализируем показанный ниже фрагмент программы.

```
void CMainWnd::OnPaint()
{
    CPaintDC do(this);
    dc.TextOut(200,200, "Using the MFC Library", 21);
}
```

Сначала создается контекст устройства, в рамках которого может вызываться любая GDI-функция. Когда метод OnPaint() завершает свою работу, автоматически вызывается деструктор класса CPaintDC.

В данном приложении используется предельно короткая схема сообщений:

```
BEGIN_MESSAGE_MAP(CMainWnd, CFrameWnd)
ON_WM_PAINT, ()
END_MESSAGE_MAP() ...
```

В параметрах макроса begin_message_map указаны два класса: CMainWnd и CFrameWnd. Первому адресуются сообщения, а второй содержит стандартные обработчики, которые вызываются по умолчанию. Макрос on_wm_paint() управляет обработкой всех сообщений wm_paint и переадресовывает их только что рассмотренной функции OnPaint(). Макрос end_message_map завершает схему сообщений.

Существенное преимущество любой схемы сообщений состоит в том, что с ее помощью удастся избежать использования тяжеловесных и чреватых ошибками конструкций с характерными для процедурных приложений операторами switch/case.

Запуск программы

Создав файл проекта в компиляторе VisualC++, вы должны убедиться в том, что установили опцию использования библиотеки MFC, иначе во время компиляции программы получите серию непонятных сообщений об ошибках. Окно данного приложения показано на рис. 19.1.

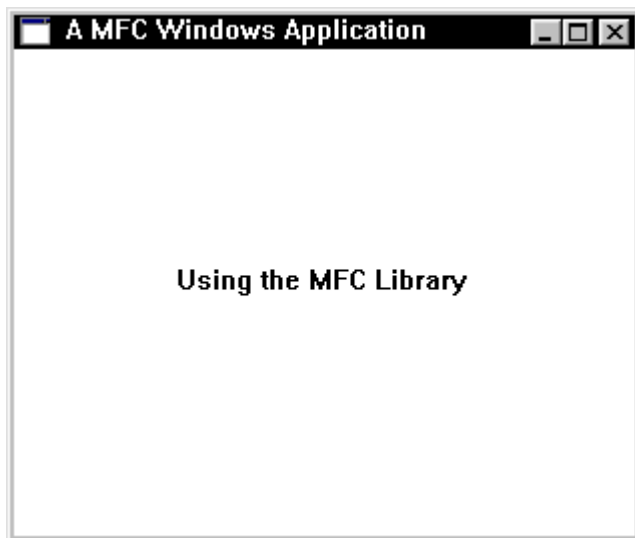


Рис. 19.1. Вывод текста в рабочую область окна

Если вы хотите поэкспериментировать с различными графическими примитивами, удалите из программы функцию `TextOut()` и вставьте вместо нее функцию `Rectangle()`, `Ellipse()` или, скажем, `LineTo()`. В следующем примере о применении этих функций мы поговорим более подробно.

Рисование графических примитивов в рабочей области окна

Вторая программа рисует в рабочей области окна различные графические фигуры. О том, как это происходит, речь шла в главе 17. Данное приложение также будет представлено двумя файлами: файлом заголовков `GDI.H` и программным файлом `GDI.CPP`. Файл заголовков, по сути, не отличается от используемого в предыдущем примере.

```
class CMainWnd : public CFrameWnd
{
public:
    CMainWnd ();
    afx_msg void OnPaint();
    DECLARE_MESSAGE_MAP() ;
};

class CgdiAApp : public CWinApp { public:
    BOOL InitInstance();
};
```

Теперь приведем текст программного файла:

```
//
//      gdi.cpp
//      Расширенный вариант приложения mfcswp.cpp,
//      ориентированный на создание графических примитивов.
//
#include <afxwin.h>
#include "gdi.h"
CgdiAApp theApp;
CMainWnd::CMainWnd() {
    Create(NULL, "Experimenting With GDI Primitives",
    WSJDVERLAPPEDWINDOW, rectDefault, NULL, NULL); >>
    void CMainWnd::OnPaint() {
        static DWORD dwColor[9] = {RGB(0, 0, 0),          // черный
        RGB(255,0, 0),          // красный
        RGB (0,255., 0),          // зеленый
```

```

RGB(0,0, 255),      // синий
RGB(255,255, 0),    // желтый
RGB(255,0, 255),    // пурпурный
RGB(0,255,255),     // голубой
RGB(127,127, 127),  // серый
RGB(255,255, 255) }; // белый
short   xcoord;
POINT   polylpts[4],polygppts[5] ;
CBrush  newbrush;
CBrush* oldbrush;
CPen    newpen;
CPen*oldpen;
CPaintDC dc(this);
// рисование толстой черной диагональной линии
newpen.CreatePen(PS_SOLID, 6, dwColor[0]);
oldpen = dc.Selectobject(Snewpen);
dc.MoveTo(0, 0) ;
dc.LineTo(640,430);
dc.TextOut(70,20,"<-diagonal line",15);
// удаление пера
dc.Selectobject(oldpen);
newpen.DeleteObject();
// рисование синей дуги
newpen.CreatePen(PS_DASH, 1, dwColor[3]);
oldpen = dc.Selectobject(Snewpen) ;
dc.ArcdOO,100, 200, 200, 15X), 175, 175,150),
dc.TextOut (80,180, "small -arc->", 11);
// удаление пера
dc.SelectObject(oldpen);
newpen.DeleteObject();
// рисование зеленого сегмента с толстым контуром
newpen.CreatePen(PS_SOLID, 8, dwColor[2]);
oldpen = dc.Selectobject(Snewpen);
dc.Chord(550, 20,630,80, 555, 25,625,70);
dc.TextOut(485,30,"chord->", 7) ;
// удаление пера
dc.Selectobject(oldpen);
newpen.DeleteObject();
// рисование эллипса и заливка его красным цветом
newpen.CreatePen(PS_SOLID, 1, dwColor[1]);
oldpen = dc.Selectobject(Snewpen);
newbrush.CreateSolidBrush(dwColor[1]);
oldbrush = dc.Selectobject(Snewbrush) ;
dc.Ellipse(180, 180, 285, 260);
dc.TextOut(210,215, "ellipse",7); // удаление кисти
dc.SelectObject(oldbrush) ; newbrush.DeleteObject(); // удаление пера
dc.SelectObject(oldpen) ; newpen.DeleteObject();
// рисование круга и заливка его синим цветом
newpen.CreatePen(PS_SOLID, 1, dwColor[3]);
oldpen = dc.SelectObject(Snewpen);
newbrush.CreateSolidBrush(dwColor[3]) ;
oldbrush = dc.SelectObject(Snewbrush);
dc.Ellipse(380,180, 570, 370);
dc.TextOut(450,265,"circle",6);

```

```

// удаление кисти
dc.SelectObject(oldbrush) ;
newbrush.DeleteObject();
// удаление пера
dc.SelectObject(oldpen);
newpen.DeleteObject() ;
// рисование черного сектора и заливка его зеленым цветом
newpen.CreatePen(PS_SOLID, 1, dwColor[0]);
oldpen = dc.SelectObject(Snewpen);
newbrush.CreateSolidBrush(dwColor[2]);
oldbrush = dc.SelectObject(Snewbrush);
dc.Pie(300,50,400, 150, 300, 50,300, 100);
dc.TextOut(350,80,"<-pie wedge", 11);
// удаление кисти
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
// удаление пера
dc.SelectObject(oldpen);
newpen.DeleteObject() ;
// рисование черного прямоугольника и заливка его серым цветом
newbrush.CreateSolidBrush(dwColor[7]) ;
oldbrush = dc.SelectObject(Snewbrush);
dc.Rectangle(50,300, 150, 400);
dc.TextOut(160,350,"<-rectangle", 11);
// удаление кисти
dc.SelectObject(oldbrush);
newbrush.DeleteObject() ;
// рисование черного закругленного прямоугольника
//изаливка его синим цветом
newbrush.CreateHatchBrush(HS_CROSS, dwColor[3]);
oldbrush = dc.SelectObject(Snewbrush) ;
dc.RoundRect(60, 310, 110,350, 20,20);
dc.TextOut(120,310, "<—rounded rectangle", 24);
// удаление кисти
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
// рисование нескольких зеленых точек
for(xcoord = 400; xcoord < 450; xcoord += 3)
dc.SetPixel(xcoord, 150, 01); dc.TextOut(455, 145, "<-pixels", 8);
// рисование толстой ломаной линии пурпурного цвета
newpen.CreatePen (PS_SOLID, 3, dwColor [5-] );
oldpen = dc.SelectObject(Snewpen);
polylpts[0].x = 10;
polylpts[0].y = 30;
polylpts[1].x = 10;
polylpts[1].y = 100; ;
polylpts[2].x = 50;
polylpts[2].y = 100;
polylpts[3].x = 10;
polylpts[3].y = 30;
dc.Polyline(polylpts, 4);
dc.TextOut(10,110, "polyline", 8);
// удаление пера
dc.SelectObject(oldpen);

```

```

newpen.DeleteObject();
// рисование голубого многоугольника
//изаливка его диагональной желтой штриховкой
newpen.CreatePen(PS_SOLID, 4, dwColor[6]);
oldpen = dc.SelectObject(Snewpen);
newbrush.CreateHatchBrush(HS_FDIAGONAL, dwColor[4]);
oldbrush = dc.SelectObject(Snewbrush);
polygpts[0].x = 40;
polygpts[0].y = 200;
polygpts[1].x = 100;
polygpts[1].y = 270;
polygpts[2].x = 80;
polygpts[2].y = 290;
polygpts[3].x = 20;
polygpts[3].y = 220;
polygpts[4].x = 40;
polygpts[4].y = 200;
dc.Polygon(polygpts, 5);
dc.TextOut(70,210,"<-polygon", 9) ;
// удаление кисти
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
// удаление пера
dc.SelectObject(oldpen);
newpen.DeleteObject(); } BEGIN MESSAGE MAP(CMainWnd, CFrameWnd)
ON_WM_PAINT() END_MESSAGE_MAP()
BOOL CgdiAApp::InitInstance() {
m_jpMainWnd = new CMainWnd();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
return TRUE;
}

```

Файл GDI.CPP

В функции OnPaint() создается массив dwColor, в котором хранятся девять RGB-значений цветов для используемых кистей и перьев.

```

static DWORD dwColor[9] = {RGB(0,0, 0),    //   черный
RGB(255, 0, 0), //   красный
RGB(0,255, 0),  //   зеленый
RGB(0,0, 255),  //   синий
RGB(255,255, 0), //   желтый
RGB(255, 0, 255), //   пурпурный
RGB(0,255,255), //   голубой
RGB(127,127, 127), //   серый
RGB(255,255, 255)}; //   белый

```

Классы CBrush и CPen позволяют создавать объекты кистей и перьев, которые могут использоваться любыми функциями класса cdcи его потомков (классов, связанных с контекстами устройств). Кисти могут иметь сплошную и штриховую заливку, а также заливку с применением растрового узора. Перья рисуются сплошной (ps_solid), штриховой (ps_dash), пунктирной (ps_dot), штрихпунктирной (ps_dashdot) и штрихточечной (ps_dashdotdot) линиями. Ниже показан синтаксис создания объектов кистей и перьев:

```

CBrush newbrush;
CBrush* oldbrush;
CPen newpen;

```

```
CPen* oldpen;
```

Поскольку для рисования различных графических примитивов применяются аналогичные алгоритмы, мы рассмотрим лишь два типичных примера. В первом на экран выводится толстая черная диагональная линия:

```
// рисование толстой черной диагональной линии
```

```
newpen. CreatePen(PS_SOLID, 6, dwColor[0]);
```

```
oldpen = dc.SelectObject (Snewpen) ;
```

```
dc.MoveTo(0, 0);
```

```
dc.LineTo(640, 430);
```

```
dc.TextOut (70, 20, "odiagonal line", 15);
```

```
// удаление пера
```

```
dc.SelectObject (oldpen) ;
```

```
newpen. DeleteObject() ;
```

Объект пера создается функцией CreatePen() „которая задает рисование черных сплошных линий толщиной в шесть логических единиц. Сразу после этого функция SelectObject() загружает созданный объект в контекст устройства и возвращает указатель на предыдущий объект пера. Функции MoveTo() и LineTo() формируют линию, которая рисуется выбранным пером. Наконец, функция Textout() выводит надпись рядом с нарисованной фигурой. В завершение снова вызывается функция Selectobject(), которая восстанавливает прежнее перо, а функция DeleteObject() удаляет ненужное перо.

Работа с кистями организуется аналогичным образом. В следующем фрагменте кода создается кисть с заливкой горизонтальными и вертикальными штрихами (hs_cross) синего цвета.

```
// рисование черного закругленного прямоугольника
```

```
//и заливка его синим цветом
```

```
newbrush.CreateHatchBrush(HS_CROSS, dwColor[3]);.
```

```
oldbrush = dc.Selectobject (Snewbrush);
```

```
dc.RoundRect(60, 310, 110, 350, 20,20);
```

```
dc.TextOut (120,310,"<—rounded rectangle", 24);
```

```
// удаление кисти
```

```
dc.Selectobject (oldbrush) ;
```

```
newbrush.DeleteObject() ;
```

Функция RoundRect() рисует закругленный прямоугольник с черным контуром, который заливается с помощью выбранной кисти.

Запуск программы

Окно аналогичной программы представлено на рис. 19.2.

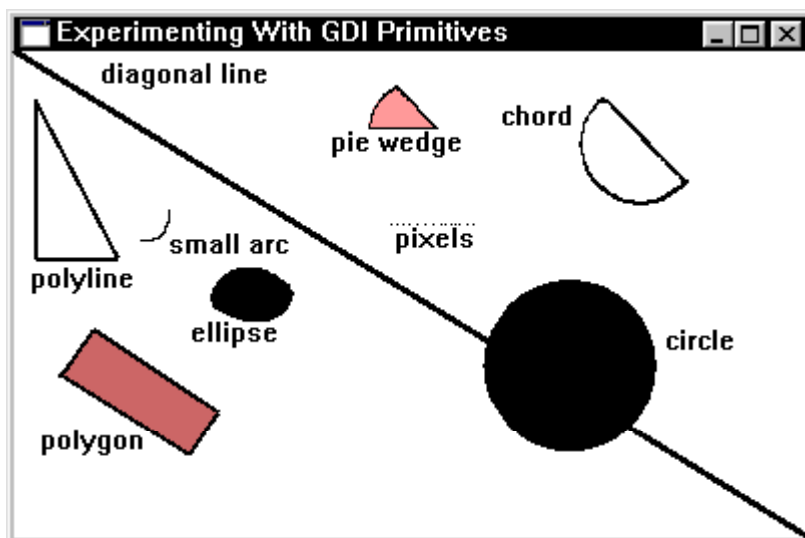


Рис. 19.2. Построение различных GDI объектов

Возможно, вы заметили, что в программе имеется один недостаток. Заключается он в том, что все координаты графических объектов, выводимых GDI-функциями, задаются в пикселях. А что произойдет, если изменится разрешение экрана, например вместо режима 800x600 будет установлен режим 1024x768? В этом случае выводимое изображение займет только верхнюю левую часть окна.

Во избежание этой проблемы приложение должно уметь определять характеристики экрана и соответствующим образом настраивать вывод изображения. Реализация этой возможности усложняет код программы. Тем не менее в следующих двух примерах будет показано, как можно масштабировать выводимые изображения в соответствии с текущим разрешением экрана.

Построение ряда Фурье

Следующая программа выводит в рабочей области окна графическое представление ряда Фурье. В данном приложении используются ресурсы двух видов: меню и диалоговые окна. По мере возрастания сложности программы увеличивается и число файлов, вовлеченных в проект. Данное приложение включает файл заголовков FOURIER.H, файл ресурсов FOURIERR.H (обратите внимание на дополнительное 'R' в конце имени файла), файл сценариев ресурсов FOURIER.RC и программный файл FOURIER.CPP.

Ниже показан текст файла FOURIER.H:

```
class CMainWnd : public CFrameWnd
{
public:
    CMainWnd();
    afx_msg void OnPaintO;
    afx_msg void OnSize(UINT, int, int);
    afx_msg int OnCreate(LPCREATESTRUCT cs);
    afx_msg void OnAboutO;
    afx_msg void OnFourierData() ;
    afx_msg void OnExit();
    DECLARE_MESSAGE_MAP()
};

class CTheApp : public CWinApp
{
public:
    virtual BOOL InitInstance();
};
```

```

class CFourierDataDialog : public CDialog
{
public:
CFourierDataDialog(CWnd* pParentWnd=NULL) : CDialog("FourierData",
pParentWnd)
{ }
virtual void OnOK(); };

```

Файл FOURIERR.H содержит идентификаторы элементов меню и диалоговых окон.

```

#define IDM_FOUR 100
#define IDM_ABOUT 110
#define IDM_EXIT 120
#define IDD_TERMS 200
#define IDD_TITLE 201

```

Файл FOURIER.RC включает описания меню и двух диалоговых окон. В приложении используются довольно простые диалоговые окна, а именно **About** и окно ввода данных.

```

#include "fourierr.h"
#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS #include "afxres.h"
////////////////////////////////////
tundef APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
// Ресурсы для английского (США) языка
#if !defined(AFX_RESOURCE_DLL) || defined (APX_TARG_ENU) . -tifdef
_WIN32 LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page (1252)
#endif
//_WIN32
////////////////////////////////////
//
// Меню//
FOURIERMENU MENU DISCARDABLE BEGIN
POPUP "Fourier Data" BEGIN
MENUITEM "Fourier Data...",IDM_FOUR MENUITEM "Fourier
About...",IDM_ABOUT MENUITEM "Exit",IDM_EXIT END END
////////////////////////////////////
//
// Диалоговые окна
//
ABOUTBOX DIALOG DISCARDABLE 14,22,200, 75
STYLE WS_POPUP | WS_CAPTION
CAPTION "About Box"
BEGIN
CTEXT "A Fourier Series Waveform", -1, 30, 5, 144, 8
CTEXT "A MFC Application", -1,30, 17,144,8
CTEXT "By William H. Murray and Chris H. Pappas", -1,28,28, 144, 8
CTEXT "(c)Copyright'1998",201, 68,38, 83, 8
DEFPUSHBUTTON "OK",IDOK, 84,55,32,14,WS_GROUP END
FOURIERDATA DIALOG DISCARDABLE 74,21,142, 70 STYLE WS_POPUP |
WS_CAPTION CAPTION "Fourier Data" BEGIN
LTEXT "Title: ", -1,6, 5, 28,8, NOT WS_GROUP

```

```

EDITTEXT IDDJTITLE, 33,1, 106,12
LTEXT "Number of terms: ", -1,6, 23,70/ 8, NOT WS_GROUP
EDITTEXT IDDJTERMS, 76,18, 32, 12
PUSHBUTTON "OK", IDOK, 25,52,24,14
PUSHBUTTON "Cancel", IDCANCEL, 89,53,28,14 END
tifdef APSTUDIO_INVOKED
////////////////////
1 TEXTINCLUDE DISCARDABLE BEGIN
"resource.hNO" END
2 TEXTINCLUDE DISCARDABLE
BEGIN
"#define APSTUDIO_HIDDEN_SYMBOLS\r\n\r\n"
"#include \"\"windows.h\"\"Nr\n"
"#undef APSTUDIO_HIDDEN_SYMBOLS\r\n"
"#include \"\"afxres.h\"\"Nr\n"
"NO" END
3 TEXTINCLUDE DISCARDABLE BEGIN
"Nr\n"
"NO"
END
#endif // APSTUDIO_INVOKED
#endif .11 Ресурсы для английского (США) языка

```

Текст файла FOURIER.CPP несколько сложнее, чем в предыдущем примере, поскольку программа должна поддерживать работу меню и двух диалоговых окон.

```

//
//  fourier.cpp
//  Построение ряда Фурье.
//
#include <afxwin.h>
#include <string.h>
#include <math.h>
#include "fourierR.h" // идентификаторы ресурсов
#include "fourier.h"
int m_cxClient, m_cyClient;
char mytitle[80] = "Title";
int nterms = 1;
CTheApp theApp;
CMainWnd::CMainWnd() {
Create(AfxRegisterWndClass(CS_HREDRAW I CS_VREDRAW,
LoadCursor(NULL, IDC_CROSS), (HBRUSH) (GetStockObject>{WLJUTE._BRUSH)
) NULL),
"Fourier Series Application with the MFC ", WS_OVERLAPPEDWINDOW,
rectDefault, NULL, "FourierMenu");
}
void CMainWnd::OnSize(UINT,int x, int y) {
m_cxClient = x;
m_cyClient = y; }
void CMainWnd::OnPaint() {
CPaintDC dc(this);
static DWORD dwColor[9]= {
RGB(0,0, 0), // черный
RGB (245,0, 0), // красный
RGB(0, 245, 0), // зеленый
RGB(0,0,245), // синий

```



```

RGB(245,245, 0),      // желтый
RGB(245,0, 245),      // пурпурный
RGB(0, 245, 245),     //голубой
RGB(127,127, 127),    // серый
RGB(245, 245, 245)); // белый
int i, j, Ititle, ang;
double y, yp;
CBrush newbrush;
CBrush* oldbrush;
CPen newpen;
CPen* oldpen;
// задание области просмотра
dc. SetMapMode (MM_ISOTROPIC) ;
dc.SetWindowExt (500,500);
dc. SetViewportExt (m_cxClient, -m_cyClient) ;
dc.SetViewportOrg(m_cxClient/20, m_cyClient/2) ;
ang = 0; yp = 0 . 0 ;
newpen.CreatePen(PS_SOLID, 2, RGB(0,0, 0) ) ;
oldpen = dc.SelectObject (Snewpen) ;
// рисование осей координат x и y
dc.MoveTo(0, 240);
dc.LineTo(0, -240);
dc.MoveTo(0, 0) ;
dc.LineTo(400, 0) ;
dc.MoveTo(0, 0) ;
// рисование ряда Фурье
for(i = 0; i <= 400; i++) {
for(j = 1; j <= nterms; j++) {
y= (150.0/{ (2.0*j)-1.0}) * sin(((j*2.0)-1.0)*0.015708*ang); yp = yp +
y;
}
dc.LineTo(i, (int)yp);
yp -= yp; ang'+t;
}
// заливка внутренних областей графика серым цветом
newbrush.CreateSolidBrush(dwColor [7]) ;
oldbrush = dc.SelectObject (Snewbrush) ;
dc.ExtFloodFill (150,10,dwColorf0,FLOODFILLBORDER) ;
dc.ExtFloodFill (300,-10, dwColor[0], FLOODFILLBORDER)
// вывод заголовка графика
Ititle= strlen(mytitle) ;
dc.TextOut (200-(Ititle*8/2) , 185, mytitle, Ititle);
// удаление кистей
dc. SelectObject (oldbrush) ;
newbrush.DeleteObject () ;
}
int CMainWnd::OnCreate(LPCREATESTRUCT) {
UpdateWindow();
return (0); }
void CMainWnd::OnAbout() {
CDialog about("AboutBox", this);
about.DoModal(); }
void CFourierDataDialog: :OnOK() {
GetDlgItemText (IDDJTITLE, mytitle, 80);

```

```

nterms = GetDlgItemInt(IDD_TERMS, NULL, 0) CDialog: :OnOK();
}
void CMainWnd::OnFourierData() {
CFourierDataDialog dlgFourierData(this); if (dlgFourierData.DoModal()
== IDOK) ( InvalidateRect(NULL, TRUE);
UpdateWindow(); } };
void CMainWnd::OnExit() {
DestroyWindowO ; }
BEGIN_MESSAGE_MAP (CMainWnd, CFrameWnd)
on_wm_paint() on_wm_size'o on_wm_create ( )
ON_COMMAND(IDM_ABODT, OnAbout) '! '!"6sf_%OMMAND(IDM_FOUR.,
OnFourierData)
O^COMMAND(IDM_EXIT, OnExit) END_MESSAGE_MAP ( )
BOOL CTheApp: : InitInstance () {
m_pMainWnd = new CMainWndO;
m_pMainWnd->ShowWindow(m_nCmdShow) ;
m_pMainWnd->UpdateWindow() ;
return TRUE;
}

```

Файл FOURIER.H

Класс CMainWnd теперь содержит несколько обработчиков сообщений: OnPaint(), OnSize(), OnCreate(), OnAbout(), OnFourierData() и OnExit():

```

afx_msg void OnPaint();
afx_msg void OnSize(UINT, int, int);
afx_msg int OnCreate(LPCREATESTRUCT cs);
afx_msg void OnAbout ( ) ;
afx_msg void OnFourierData ();
afx_msg void OnExit();

```

Функция OnPaint() вызывается автоматически, когда объекту CMainWnd поступает сообщение wm_paint от Windows или от самого приложения. Функция OnSize() вызывается в ответ на сообщение wm_size, которое генерируется при изменении размеров окна приложения. Информация о размерах необходима для динамического масштабирования содержимого окна. Функции OnCreate() передается структура, содержащая информацию о создаваемом окне: его размерах, стиле и других атрибутах. Функции OnAbout(), OnFourierData() и OnExit() определяются пользователем и вызываются при получении сообщений wm_command, генерируемых при выборе соответствующих команд меню.

Для работы с диалоговыми окнами в MFC существует класс CDialog. Его можно непосредственно использовать для вывода простейших диалоговых окон, например окна About. В случае более сложных окон потребуется создать собственный класс. В нашем примере используется диалоговое окно, в котором пользователь может ввести заголовок для графика и целочисленное значение, задающее число членов ряда Фурье. Класс CFourierDataDialog порождается от CDialog. Для ввода данных создается модальное окно, т.е. с приложением невозможно будет продолжать работу до тех пор, пока пользователь не введет данные и не закроет диалоговое окно. Объявление класса выглядит следующим образом:

```

class CFourierDataDialog : public CDialog
{
public:
CFourierDataDialog (CWnd* pParentWnd=NULL) : CDialog ("FourierData",
pParentWnd)
{ }
virtual void OnOK();
};

```

Классы диалоговых окон, порождаемые от CDialog, могут иметь собственную схему сообщений. Правда, если переопределяются только функции OnInitDialog(), OnOK() и OnCancel(), ее можно не создавать. В нашем простом примере конструктор

CFourierDataDialog() передает конструктору родительского класса имя диалогового окна, FourierData, и указатель на родительское окно.

Диалоговое окно возвращает введенные данные после щелчка пользователя на кнопке ОК. Для диалоговых окон, требующих предварительной инициализации, может быть переопределен метод OnInitDialog().

Файлы ресурсов

Файл FOURIERR.H содержит пять идентификаторов. Константы idm_four, idm_about и idm_exit служат идентификаторами команд меню, тогда как константы IDD_TERMS и IDD_TITLE используются для описания элементов диалогового окна. Файл FOURIER.RC содержит описания меню и диалоговых окон.

Файл FOURIER.CPP

Сложность программного кода рассматриваемого приложения возросла по сравнению с предыдущими примерами, поскольку в него были добавлены меню и два диалоговых окна. В следующих параграфах вы, в частности, узнаете, как:

- определить текущие размеры окна;
- нарисовать объект в окне и выполнить его заливку;
- выбрать новый указатель мыши;
- задать область просмотра и начало координат;
- установить цвет фона.

Создание окна

Одновременно с созданием окна выполняется регистрация класса окна с помощью функции AfxRegisterWndclass(). В качестве указателя мыши устанавливается перекрестие (idc_cross), для заливки фона выбирается белая кисть (white_brush), а значком приложения становится стандартный системный значок (последний параметр равен null).

```
CMainWnd::CMainWnd() (
Create(AfxRegisterWndclass(CS_HREDRAW | CS_VREDRAW,
LoadCursor(NULL, IDC_CROSS), (HBRUSH)(GetStockObject(WHITE_BRUSH)), NULL),
"Fourier Series Application with the MFC ", WS_JDVERLAPPEDWINDOW, rectDefault, NULL,
"FourierMenu"); 1
```

Обратите также внимание, что в функции Create() задается название меню приложения.

Определение текущих размеров окна

Сообщение WM_SIZE генерируется всякий раз, когда изменяются размеры окна приложения. В функцию OnSize() передаются текущие размеры окна, которые сохраняются в переменных m_cxClient и m_cyClient:

```
void CMainWnd::OnSize(UINT, int x, int y) {
m_cxClient = x;
m_cyClient = y;
}
```

Значения этих переменных используются для масштабирования содержимого окна в соответствии с его размерами.

Построение ряда Фурье

Чтобы избежать проблем с масштабированием выводимого изображения, о которых упоминалось выше, создается свободно масштабируемая поверхность рисования. Сначала с помощью функции SetMapMode() устанавливается режим отображения MM_ISOTROPIC:

```
dc.SetMapMode(MM_ISOTROPIC);
```

Далее логические размеры окна задаются равными по вертикали и горизонтали 500 единицам:

```
dc.SetWindowExt(500,500);
```

Это означает, что какими бы ни были действительные размеры окна, программа будет рассматривать его как квадрат со сторонами по 500 единиц. В следующей строке размеры области просмотра задаются равными текущим размерам рабочей области окна:

```
dc.SetViewportExt(ra_cxClient, -m_cyClient);
```

Это является гарантией того, что все содержимое окна будет видимым. Отрицательное значение у означает, что ось у увеличивается в направлении снизу вверх. В следующей строке точка начала области просмотра устанавливается на пересечении середины оси у и одной пятой от левого края оси х.

```
dc.SetViewportOrg(m_cxClient/20, m_cyClient/2);
```

Затем строятся оси координат х и у.

```
//рисование осей координат х и у
dc.MoveTo(0, 240);
dc.LineTo(0, -240);
dc.MoveTo(0, 0);
dc.LineTo(400, 0);
dc.MoveTo(0,0);
```

Показанный ниже фрагмент программы, ответственный за выведение на экран ряда Фурье, содержит два цикла for. В переменной цикла i хранится значение координаты х, а в переменной цикла j — значение текущей гармоник. Координата у каждой точки представляет собой сумму значений всех гармоник для заданного угла.

```
// рисование ряда Фурье
for(i= 0; i <= 400; i++) {
for (j = 1; j <= nterms; j++){
y= (150.0/((2.0*j)-1.0)) * sin(((j*2.0)-1.0)*0.015708*ang);
yp = yp + y; }
dc.LineTo (i, (int)yp);
yp -= yp;
ang++;
}
```

Для связи всех точек графика в единую кривую используется функция LineTo(). Внутренняя область графика будет закрашена серым цветом с помощью функции ExtFloodFill(). Эта функция требует указания координат произвольной точки внутри закрашиваемой области и цвета границы области. Установка параметра floodfillborder означает, что функция будет закрашивать всю область до границы, заданной указанным цветом.

```
// заливка внутренних областей графика серым цветом
newbrush.CreateSolidBrushfwdColor[7]) ;
oldbrush = dc.SelectObject(Snewbrush);
dc.ExtFloodFill (150,10, dwColor[0], FLOODFILLBORDER);
dc.ExtFloodFill (300,-10,dwColorf0,FLOODFILLBORDER);
```

Построение графика завершается выведением надписи и удалением из памяти созданного ранее объекта кисти:

```
// вывод заголовка графика ltitle = strlen(mytitle);
dc.TextOut(200-(ltitle*8/2), 185, mytitle, ltitle);
// удаление кистей
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
```

Диалоговое окно About

Окна типа **About** служат для выдачи пользователю информации о программе, ее авторах, авторских правах и т.д. Модальное диалоговое окно **About** появляется на экране при выборе в меню команды **Fourier About...**, в результате чего вызывается функция OnAbout():

```
void CMainWnd::OnAbout() {  
    CDialog about("AboutBox", this);  
    about.DoModal(); }  
};
```

Конструктор класса CDialog воспринимает текущее окно как родительский объект. С этой целью используется указатель this, который всегда указывает на текущий объект. Метод DoModal() выводит диалоговое окно **About** на экран. После щелчка на кнопке ОК Окно **About** удаляется с экрана и становится доступным окно приложения, которое при этом перерисовывается.

Окно ввода данных

Диалоговое окно, принимающее данные от пользователя, вызывается путем выбора в меню команды **FourierData....** В нем пользователь получает возможность ввести заголовок графика и задать количество членов ряда Фурье. После щелчка на кнопке **OK** окно удаляется с экрана и обновляется рабочая область окна приложения.

```
void CMainWnd::OnFourierData()  
{  
    CFourierDataDialog dlgFourierData(this);  
    if (dlgFourierData.DoModal() == IDOK)  
    {  
        InvalidateRect(NULL, TRUE); UpdateWindow();  
    } }  
};
```

В файле FOURIER.H класс CFourierDataDialog объявлен потомком класса CDialog. Обратите внимание, что к моменту завершения функции DoModal() введенные пользователем данные уже переданы в программу. Это происходит в методе OnOK (), вызываемом путем щелчка на кнопке ОК:

```
void CFourierDataDialog::OnOK() {  
    GetDlgItemText(IDD_TITLE, mytitle, 80);  
    nterms = GetDlgItemInt(IDD_TERMS, NULL, 0);  
    CDialog::OnOK();  
}
```

Функция GetDlgItemText() записывает строку заголовка графика (определяемую по идентификатору idTitle) в переменную mytitle. Аналогичным образом с помощью функции GetDlgItemInt() возвращается введенное пользователем количество членов ряда Фурье. Поле, в котором вводится это число, идентифицируется константой IDD_TERMS. Второй параметр функции GetDlgItemInt() позволяет проконтролировать успешность преобразования введенной строки в число, но в нашем приложении эта возможность не используется. Если третий параметр не равен нулю, функция должна проверять наличие знака минус и в случае необходимости возвращать знаковое число. Но поскольку в нашем случае значения должны быть только положительными, этот параметр установлен равным 0.

Вызов функции OnExit()

В меню также содержится команда **Exit**, при вызове которой с экрана удаляется окно приложения, для чего вызывается функция DestroyWindow():

```
void CMainWnd::OnExit() {  
    DestroyWindow(); }  
};
```

Схема сообщений

В макросе BEGIN_MESSAGE_MAP указываются два класса: CMainWnd и CFrameWnd. Принимающей стороной является класс CMainWnd, а в классе CFrameWnd содержатся базовые обработчики. Макрос on_hm_paint() управляет обработкой всех сообщений WM_PAINT, направляя их методу OnPaint(). Другой макрос, ON_WM_SIZE(), управляет сообщениями WM_SIZE и направляет их методу OnSize(). Макрос ON_WM_CREATE() связан с сообщениями WM_CREATE, которые передаются методу OnCreate(). И еще один макрос,

on_command(), поддерживает обработку сообщений, связанных с выбором команд меню. Здесь для каждой команды указывается свой обработчик.

```
BEGIN_MESSAGE_MAP(CMainWnd, CFrameWnd)
ON_WM_PAINT()
ON_WM_SIZE()
ON_WM_CREATE()
ON_COMMAND(IDM_ABOUT, OnAbout)
ON_COMMAND(IDM_FOUR, OnFourierData)
ON_COMMAND(IDM_EXIT, OnExit) END_MESSAGE_MAP()
```

Как уже было сказано, наличие схемы сообщений позволяет избежать использования в программе сложных и чреватых ошибками конструкций с операторами switch/case.

Запуск программы

После запуска программы в окне приложения будет построен заданный по умолчанию ряд Фурье, состоящий из одного члена, что соответствует синусоиде (рис. 19.6). На рис. 19.7 показан ряд Фурье.

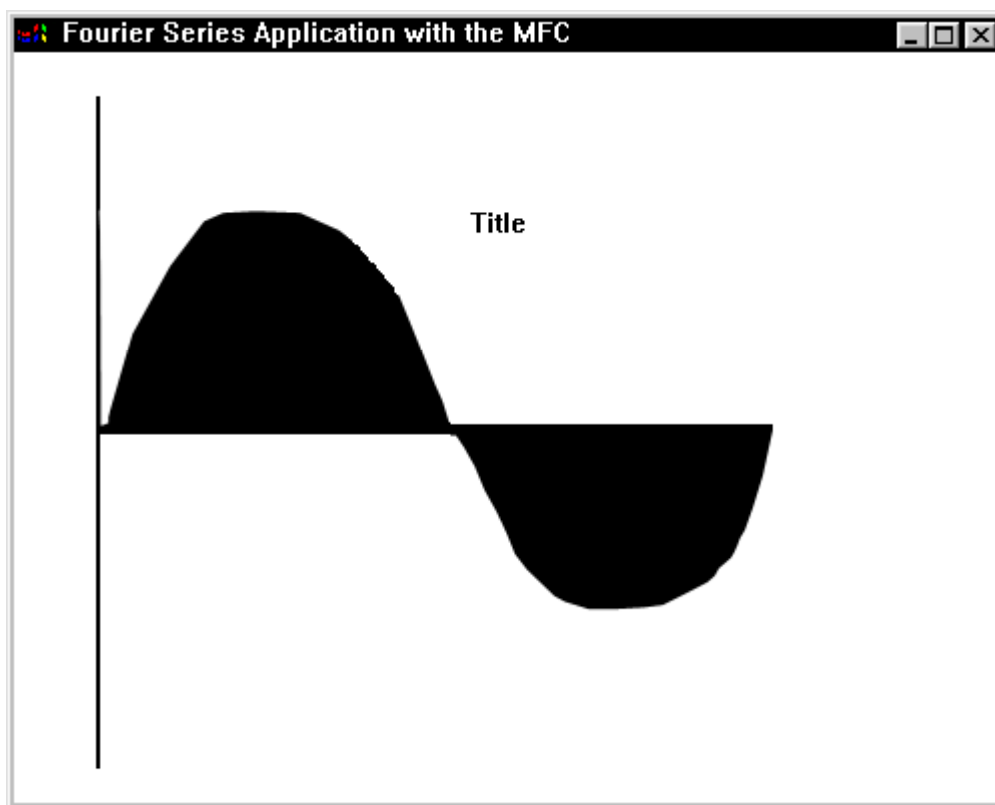


Рис. 19.6. График, выводимый по умолчанию

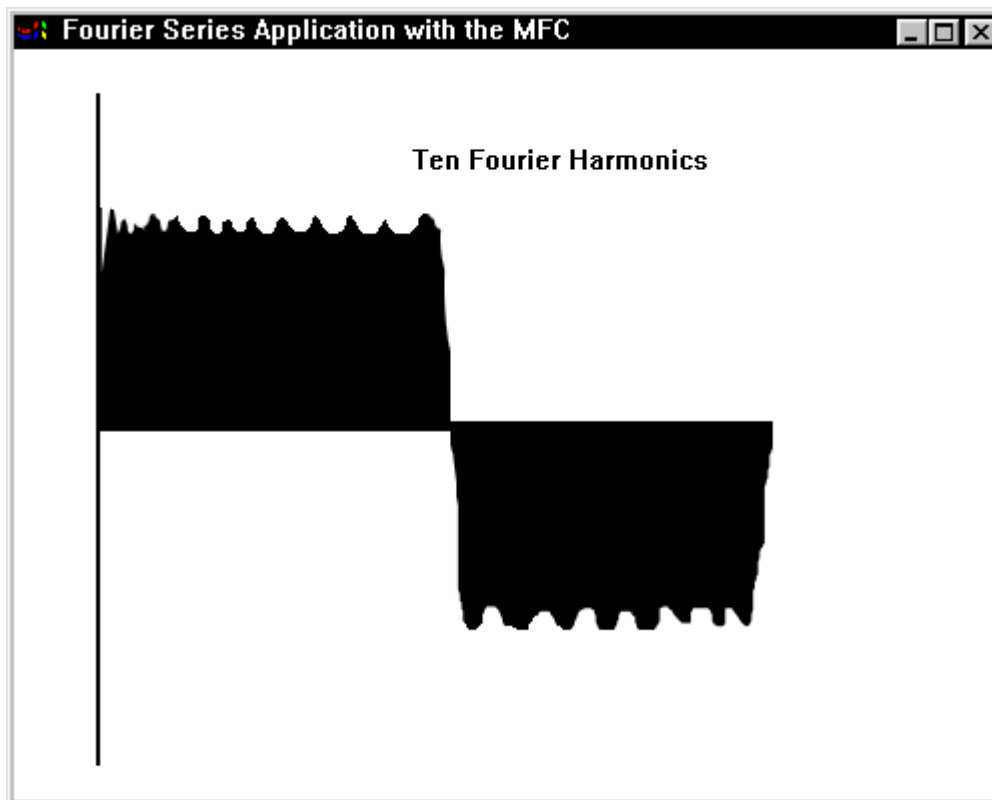


Рис. 19.7. Ряд Фурье из n членов

Построение гистограмм

Последний пример в этой главе представляет собой программу, которая на основании вводимых пользователем данных строит гистограммы. В этой программе также используются ресурсы, в том числе меню, диалоговое окно **About** и диалоговое окно для ввода данных.

В проект входят четыре файла: файл заголовков BARChart.H, файл ресурсов BARChart.R, файл сценариев ресурсов BARChart.RC и программный файл BARChart.CPP. Файл BARChart.H содержит описания классов CMainWnd, CTheApp и CBarDataDialog:

```
class CMainWnd : public CFrameWnd
{
public:
    CMainWnd () ;
    afx_msg void OnPaintO;
    afx_msg void OnSizeCUI, int, int);
    afx_msg int  OnCreate(LPCREATESTRUCT cs);
    afx_msg void OnAboutO;
    afx_msg void OnBarData ();
    afx_msg void OnExitO;
    DECLARE_MESSAGE_MAP ()
};

class CTheApp : public CWinApp
{
public:
    virtual BOOL InitInstance (); };

class CBarDataDialog : public CDialog
<
```

```

public:
CBarDataDialog (CWnd* pParentWnd=NULL)
: CDialog ("BarDlgBox", pParentWnd)
{ }
virtual void OnOK(); };

Файл BARCHARTR.Нсодержит идентификаторы команд меню и элементов
управления диалоговых окон:
#define IDM_ABOUT 10 #define IDM_INPUT 20
#define IDM_EXIT 30 #define DM_TITLE 300 #define DM_XLABEL 301
....."___
#define DM_YLABEL 302 ' #define DM_P1 303
#define DM_P2 304 #define DM_P3 305 Idefine DM_P4 306
#define DM_P5 307
#define DM_P6 308
#define DM_P7 309
#define DM_P8 310
f define DM_P9 311
#define DM_P10 312
Описания меню и двух диалоговых окно представлены в файле BARCHART.RC:
#include "resource. h"
# include "barchartr .h"
#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
#define APSTUDIO_HIDDEN_SYMBOLS
#include "windows.h"
#undef APSTUDIO_HIDDEN_SYMBOLS
#include "afxres.h"
////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
// Ресурсы для английского (США) языка
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG^ENGLISH, SUBLANG_ENGLISH_US tpragma code_page(1252)
#endif // WIN32
////////////////////////////////////
//
// Меню
//
BARMENU MENU DISCARDABLE
BEGIN
POPUP "Bar_Chart" BEGIN
MENUITEM "About Box...",IDM_ABOUT MENUITEM "Bar Values...",IDM_INPUT
MENUITEM "Exit", IDM_EXIT END END
////////////////////////////////////
//
// Диалоговые окна
//
ABOUTDLGBOX DIALOG DISCARDABLE 14,22, 200, 75
STYLE WS_POPUP I WS_CAPTION
CAPTION "About Box"
BEGIN
CTEXT "A Bar Chart Application", -1,30,5, 144,8
CTEXT "A Simple MFC Windows Application1;", -1,30,17,144, 8

```



```

CTEXT "By William H. Murray and Chris H. Pappas",
-1,28/ 28,144, 8
CTEXT "(c)Copyright 1998",-1, 68,38, 83, 8 DEFPUSHBUTTON "OK", IDOK,
84, 55, 32, 14, WS_GROUP
END
BARDLGBOX DIALOG DISCARDABLE 42,65526,223,209
STYLE WS_POPUP I WS_CAPTION
CAPTION "Bar Chart Data"
BEGIN
GROUPBOX "Bar Chart Title:",100, 5, 11,212,89,WSJTABSTOP
GROUPBOX "Bar Chart Heights", 101, 5, 105, 212, 90,WSJTABSTOP
LTEXT "Title:", -1,43,35,28,8, NOT WS_GROUP
EDITTEXT DM_TITLE, 75,30,137,12
LTEXT "x-axis label:",-1,15,55, 55,8, NOT WS_GROUP
EDITTEXT DM_XLABEL, 75,50,135, 12
LTEXT "y-axis label:",-1,15, 75, 60,8, NOT WS_GROUP
EDITTEXT.DM_YLABEL, 75,70,135,12
LTEXT "Bar #1 ", -1,45,125, 40, 8, NOT WS_GROUP
LTEXT "Bar #2 ", -1,45,140, 40, 8, NOT WS_GROUP
LTEXT "Bar #3 ", -1,45,155, 40, 8, NOT WS_GROUP
LTEXT "Bar #4 ", -1,45,170, 40, 8, NOT WS_GROUP
LTEXT "Bar #5 ", -1,45,185, 40, 8, NOT WS_GROUP
LTEXT "Bar #6 ", -1,130, 125> 40,8, NOT WS_GROUP
LTEXT "Bar #7 ", -1,130, 140, 40,8, NOT WS_GROUP
LTEXT "Bar #8 ", -1,130, 155, 40,8, NOT WS_GROUP
LTEXT "Bar #9 ", -1,130, 170,40,8, NOT WS_GROUP
LTEXT "Bar #10:",-1,130, 185, 45,8, NOT WS_GROUP
EDITTEXT DM_P1, 90,120, 30,12
EDITTEXT DM_P2, 90,135,30, 12
EDITTEXT DM_P3, 90,150, 30,12
EDITTEXT DM_P4, 90,165,30, 12
EDITTEXT DM_P5, 90,180, 30,12
EDITTEXT DM_P6, 180, 120, 30,12
EDITTEXT DM_P7, 180, 135, 30,12
EDITTEXT DM_P8, 180, 150, 30,12
EDITTEXT DM_P9, 180, 165,30,12
EDITTEXT DM_P10, 180, 180, 30,12
PUSHBUTTON "OK", IDOK, 54,195,24,14
END
PUSHBUTTON "Cancel", IDCANCEL, 124, 195,34,14
#ifdef APSTUDIO_INVOKED
////////////////////////////////////////
1 TEXTINCLUDE DISCARDABLE BEGIN
"resource.h\0" END
2 TEXTINCLUDE DISCARDABLE BEGIN
#define APSTUDIO_HIDDEN_SYMBOLS\r\n"
#include ""windows.h""\r\n"
#undef APSTUDIO_HIDDEN_SYMBOLS\r\n"
#include ""afxres.h""\r\n"
"\0" END
3 TEXTINCLUDE DISCARDABLE BEGIN
"\r\n"
"\0" END
#endif // APSTUDIO_INVOKED

```

```
#endif // Ресурсы для английского (США) языка
```

Ниже приведен текст файла BARCHART.CPP.

```
//
// barchart.cpp
// Построение гистограмм.
//
//
#include <afxwin.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include "barchart.h" // идентификаторы ресурсов
#include "barchart.h"
#define maxnumbar 10
char szTString[80] = "(barchart title area)";
char szXString[80] = "x-axis label";
char szYString[80] = "y-axis label";
int iBarSize[maxnumbar] = {20,10, 40, 50};
int m_cxClient, m_cyClient;
CTheApp theApp;
CMainWnd::CMainWnd() {
Create(AfxRegisterWndClass(CS_HREDRAW | CS_VREDRAW, LoadCursor(NULL,
IDC_CROSS),
(HBRDISH)'GetStockObject (WHITE_BRUSH) , NULL) , "Bar Chart Application
with the MFC ", WS_OVERLAPPEDWINDOW, rectDefault, NULL, "BarMenu");
}
void CMainWnd::OnSize(OINT, int x, int y) {
m_cxClient = x;
m_cyClient = y; }
void CMainWnd::OnPaint()
{
CPaintDC dc(this);
static DWORD dwColorf10[] = {
RGB(0,0, 0), // черный
RGB(245,0, 0), // красный
RGB(0,245,0), // зеленый
RGB(0,0, 245), // синий
RGB(245,245, 0), // желтый
RGB(245,0, 245), //пурпурный
RGB(0,245,245), // голубой
RGB(0,80,80), // голубовато-серый
RGB(80,80,80), // темно-серый
RGB(245, 245, 245)}; // белый
CFont newfont;
CFont* oldfont;
CBrush newbrush;
CBrush* oldbrush;
int i, iNBars, iBarWidth, iBarMax;
int ilenMaxLabel;
int x1,x2,y1,y2;
int iBarSizeScaled[maxnumbar] ; char sbuffer[10],*strptr;
iNBars = 0;
for(i=0;i < maxnumbar; i++){ if(iBarSize [i] != 0) iNBars++;
```

```

}
iBarWidth = 400/iNBars;
// Поиск столбца, имеющего максимальную высоту
iBarMax = iBarSize[0]; for(i= 0; i < iNBars; i++)
if(iBarMax < iBarSize[i])iBarMax = iBarSize [i];
// Преобразование максимального значения по y в строку
struptr = _itoa(iBarMax, sbuffer, 10); ilenMaxLabel = strlen(sbuffer);
// Масштабирование столбцов в массиве.
// Максимальная высота столбца – 270.
for(i= 0; i < iNBars; i++)
iBarSizeScaled[i] = iBarSize[i]* (270/iBarMax);
// Задание режима отображения
//и создание области просмотра
dc.SetMapMode(MM_ISOTROPIC) ;
dc.SetWindowExt(640,400);
dc.SetViewportExt(m_cxClient, m_cyClient);
dc.SetViewportOrg(0, 0) ;
// Выводтекставокно, еслидлянегохватаетместаif(m_cxClient > 200) ( '
newfont.CreateFont(12,12, 0, 0, FW_BOLD,
FALSE, FALSE, FALSE, OEM_CHARSET, OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, VARIABLE_PITCH | FF_ROMAN,
"Roman");
oldfont = dc.SelectObject(Snewfont); dc.TextOut((300-
(strlen(szTString)*10/2) ) ,
15,szTString, strlen(szTString));
dc.TextOut((300-(strlen(szXString)* 10/2)) ,
365,szXString, strlen(szXString));
dc.TextOut((90- ilenMaxLabel*12), 70,struptr, ilenMaxLabel); //
удалениеобъекташрифтаdc.SelectObject(oldfont); newfont.DeleteObject();
newfont.CreateFont(12,12,900,900,FW_BOLD,
FALSE, FALSE, .FALSE, OEM_CHARSET, OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, VARIABLE_PITCH | FF_ROMAN,
"Roman");
oldfont = dc.SelectObject(Snewfont) ;
dc.TextOut(50,200+ (strlen(szXString)*10/2), szYString,
strlen(szYString));
// удаление объекта шрифта
dc.SelectObject(oldfont) ;
newfont.DeleteObject() ; }
// Рисование осей координат
dc.MoveTo(99,49);
dc.LineTo(99,350);
dc.LineTo(500, 350);
dc.MoveTo(99,350);
// Начальные значения
x1 = 100;
y1 = 350;
x2 = x1 + iBarWidth;
// Рисование столбцов
for(i=0;i < iNBars; i++) {
newbrush.CreateSolidBrush(dwColor[i]),
oldbrush = dc.SelectObject(Snewbrush)
y2 = 350 - iBarSizeScaled[i];
dc.Rectangle(x1,y1,x2, y2);
x1 = x2;

```

```

x2 += IBarWidth;
// удаление кисти
dc.SelectObject(oldbrush);
newbrush.DeleteObject(); }
int CMainWnd::OnCreate(LPCREATESTRUCT) {
UpdateWindow();
return (0);
}
void CMainWnd::OnAbout() {
CDialog about("AboutDlgBox", this);
about.DoModal(); }
void CBarDataDialog::OnOK()
{
GetDlgItemText(DMJTITLE, szTString, 80);
GetDlgItemText(DM_XLABEL, szXString, 80);
GetDlgItemText(DM_YLABEL, szYString, 80);
iBarSize[0] = GetDlgItemInt(DM_P1, NULL, 0)
iBarSize[1]= GetDlgItemInt(DM_P2, NULL, 0)
iBarSize[2] = GetDlgItemInt(DM_P3, NULL, 0)
iBarSize[3] = GetDlgItemInt(DM_P4, NULL, 0)
iBarSize[4] = GetDlgItemInt(DM_P5, NULL, 0)
iBarSize[5] = GetDlgItemInt(DM_P6, NULL, 0)
iBarSize[6]= GetDlgItemInt(DM_P7, NULL, 0)
iBarSize[7] = GetDlgItemInt(DM_P8, NULL, 0)
iBarSize[8] = GetDlgItemInt(DM_P9, NULL, 0)
iBarSize[9]= GetDlgItemInt(DM_P10, NULL, 0)
CDialog: :OnOK();
}
void CMainWnd::OnBarData() {
CBarDataDialog dlgBarData(this); if(dlgBarData.DoModal() == IDOK)
InvalidateRect(NULL, TRUE); UpdateWindow(); } }
void CMainWnd::OnExit() {
DestroyWindow(); }
BEGIN_MESSAGE_MAP(CMainWnd, CFrameWnd) ON_WM_PAINT() ON_WM_SIZE()
ON_WM_CREATE ()
ON_COMMAND(IDM_ABOUT, OnAbout) ON_COMMAND(IDM_INPUT, OnBarData)
ON_COMMAND(IDM_EXIT, OnExit)
END_MESSAGE_MAP()
BOOL CTheApp::InitInstance() {
m_pMainWnd = new CMainWnd ();
m_pMainWnd->ShowWindow(m_nCmdShow);
m_pMainWnd->UpdateWindow();
return TRUE;
}

```

Файл BARChart.H

В данной программе используются многие блоки, встречавшиеся в предыдущем приложении. Обратите, например, внимание на однотипность описания класса

```

CMainWnd:
afx_msg void OnPaint();
afx_msg void OnSize(UINT, int, int);
afx_msg int OnCreate(LPCREATESTRUCT cs);
afxjmsg void OnAbout();
afx_msg void OnBarData();

```

```
afx_msg void OnExit();
```

Объявление класса CBarDataDialog аналогично объявлению класса CFourierDataDialog из предыдущего примера. В то же время окно ввода данных поддерживает теперь ввод большего числа значений, хотя при его разработке мы взяли за основу аналогичное диалоговое окно из предыдущей программы.

Файлы ресурсов

Два исходных файла, BARCHARTR.H и BARCHRT.RC, объединяются компилятором ресурсов Microsoft в единый файл ресурсов BARCHART.RES.

Файл заголовков BARCHARTR.H содержит идентификаторы трех команд меню: idm_about, idm_input и IDM_EXIT. Еще тринадцать идентификаторов относятся к элементам управления диалогового окна ввода данных. Три из них, dm_title, DM_XLABEL и dm_ylabel, связаны с заголовком и подписями к осям. Остальные десять констант, от DM_P1 до DM_P10, указывают на целочисленные значения, введенные для отдельных столбцов гистограммы.

Файл сценариев ресурсов BARCHART.RC содержит описания меню и двух диалоговых окон. Приложение содержит два диалоговых окна. Окно **About** практически идентично тому, что использовалось в предыдущем приложении, а окно ввода данных является более сложным.

Файл BARCHART.CPP

В этом параграфе мы сконцентрируем внимание в первую очередь на тех программных блоках, которые не были представлены в предыдущем примере, т.е. являются уникальными для данного приложения. Программа на основании введенных данных строит гистограмму. С помощью модального диалогового окна BarChartData пользователь может ввести заголовок гистограммы, подписи к осям x и y и до десяти значений отдельных столбцов. Программа автоматически масштабирует размеры столбцов в соответствии с размерами рабочей области окна и назначает каждому из них цвет из предварительно заданного списка.

В константе maxnumbar хранится информация о максимально допустимом числе столбцов гистограммы:

```
#define maxnumbar 10
```

В следующих массивах записаны заданные по умолчанию заголовки гистограммы, подписи к осям, а также размеры четырех столбцов:

```
char szTString[80] = "(barchart title area)";
char szXString[80] = "x-axis label";
char szYString[80] = "y-axis label";
int iBarSize[maxnumbar] = {20,10,40,50};
```

Размеры рабочей области окна приложения также сохраняются в глобальных переменных:

```
int i^i_cxClient, m_cyClient;
```

Наличие этих переменных позволяет масштабировать содержимое окна в соответствии с его текущими размерами.

Цвета столбцов гистограммы выбираются из массива dwColor в определенном порядке. Например, если гистограмма состоит из трех столбцов, то им будут назначены черный, красный и зеленый цвета.

Классы CFont и CBrush позволяют передавать объекты шрифтов и кистей любым функциям класса CDC (базовый класс для работы с контекстами устройств). Новые шрифты требуются для вывода заголовка гистограммы и подписей к осям. Как объявляются такие объекты, показано ниже:

```
CFont newfont;
CFont* oldfont;
CBrush newbrush;
CBrush* oldbrush;
```

Масштабирование столбцов

При построении гистограммы прежде всего необходимо установить, сколько в ней будет столбцов. Данные о размерах столбцов хранятся в глобальном массиве `iBarSize`. Для определения числа столбцов используется следующий цикл:

```
iNBars = 0;
for(i=0; i < maxnumbar; i++)
{
    if (iBarSize[i] != 0) iNBars++;
}
```

Значения записываются в этот массив при закрытии диалогового окна ввода данных, в методе `OnOK()`. Ширина столбцов зависит от их количества, так как ширина самой гистограммы постоянна. Для вычисления ширины столбцов используется следующая формула:

```
iBarWidth = 400/iNBars;
```

Высота столбцов регулируется с учетом максимального введенного значения. Сначала определяется, какова будет высота самого большого столбца:

```
// Поиск столбца с максимальной высотой
iBarMax = iBarSize[0]; for(i= 0; i< iNBars; i++)
if(iBarMax < iBarSize [i])iBarMax = iBarSize [i];
```

Максимальное значение будет выведено слева от оси *y*. Для преобразования числового значения в строку вызывается функция `__itoa()`:

```
// Преобразование максимального значения по оси y в строку
strptr = __itoa (iBarMax, sbuffer, 10); ilenMaxLabel= strlen (sbuffer)
;
```

Размеры остальных столбцов масштабируются в соответствии с максимальным значением:

```
// Масштабирование столбцов в массиве.
// Максимальная высота столбца — 270.
for(i=0; i < iNBars; i++)
iBarSizeScaled[i] = iBarSize[i] * (270/iBarMax);
```

Подготовка окна

Прежде чем выводить гистограмму, необходимо задать режим отображения, установить размеры окна и области просмотра, а также координаты точки начала области просмотра:

```
// Задание режима отображения
//исоздание области просмотра
dc.SetMapMode(MM_ISOTROPIC) ;
dc.SetWindowExt(640,400);
dc.SetViewportExt(m_cxClient, m_cyClient);
dc.SetViewportOrg(0, 0) ;
```

Благодаря этому коду изображение гистограммы при изменении размеров окна будет автоматически масштабироваться.

Вывод текста в окно

В предыдущем приложении мы выводили текстовую информацию в окно при использовании шрифта, установленного по умолчанию. При необходимости применить какой-нибудь специальный шрифт или изменить ориентацию текста можно воспользоваться функциями работы со шрифтами, основными из которых являются `CreateFont()` и `CreateFontIndirect()`. В нашем примере использовалась функция `CreateFont()`.

Что такое шрифт

Под шрифтом понимают набор печатных символов, имеющих одинаковые начертание и размер. Шрифт включает символы букв, знаков препинания и других вспомогательных знаков. В качестве примеров различных шрифтов можно привести Arial размером в 12 пунктов, TimesNewRoman в 12 пунктов, TimesNewRoman в 14 пунктов и т.д. Пункт — это наименьшая единица в типографской системе мер, равная 0,376 мм. 72 пункта составляют один дюйм.

Под гарнитурой шрифта понимают совокупность элементов, определяющих внешний вид его символов. Отличительными особенностями любой гарнитуры являются толщина линий и наличие засечек (маленьких линий по нижнему и верхнему краям символа, с помощью которых оформляется его основной контур, как, например, в букве 'М'). Выше уже отмечалось, что шрифт включает весь набор печатных символов определенной гарнитуры, размера и начертания (курсивное, полужирное и т.п.). Как правило, система содержит широкий набор стандартных шрифтов, которые могут использоваться всеми приложениями. Поэтому шрифты редко компилируют в качестве ресурсов в исполняемый файл.

Функция CreateFont()

Функция CreateFontОобъявлена в файле WINDOWS.H. Она выбирает из набора физических GDI-шрифтов тот шрифт, который наиболее точно соответствует характеристикам, указанным при ее вызове. Созданный логический шрифт может использоваться любыми устройствами. Функция CreateFontОимеет следующий синтаксис:

CreateFont(высота, ширина, ориентация, наклон, толщина, курсив, подчеркивание, зачеркивание, набор_символов, точность_вывода, точность_отсечения, качество, интервал_и_семейство, название)

Краткое описание параметров функции CreateFont() дано в табл. 19.1.

Таблица 19.1. Параметры функции CreateFont()	
Параметр	Описание
(int) высота	Высота символов в логических единицах
(int) ширина	Средняя ширина символов в логических единицах
(int) ориентация	Угол наклона строки (в десятых долях градуса) относительно горизонтальной оси; 0 — слева направо, 900 (90°) — по вертикали снизу вверх, 1800 (180°) — справа налево, 2700 (270°) — по вертикали сверху вниз
(int) наклон	Угол наклона символов (в десятых долях градуса) относительно горизонтальной оси; 0 — нормальное отображение, 900 — поворот на 90° против часовой стрелки, 1800 — перевернутое отображение, 2700 — поворот на 90° по часовой стрелке
(int) толщина	Толщина шрифта (от 0 до 1000); значение 400 (fw_normal) соответствует нормальному начертанию, 700 (fw_bold) — полужирному
(DWORD) курсив	Курсивное начертание, если параметр не равен нулю
(DWORD) подчеркивание	Подчеркивание символов, если параметр не равен нулю
(DWORD) зачеркивание	Зачеркивание символов, если параметр не равен нулю
(DWORD) набор_символов	Набор символов; возможные константы: ANSI_CHARSET— стандартные ANSI-символы, oem_charset — системно-зависимый набор, russian_charset— русские символы, hebrew_charset— иврит и т.д.
(DWORD) точность_вывода	Допустимая степень соответствия выбранного системой физического шрифта заданным установкам; некоторые из возможных констант: out_default_precis — стандартный режим подстановки, out_tt_precis— при наличии нескольких шрифтов с одинаковым названием выбирается контурный (TrueType), out_device_precis— при наличии нескольких шрифтов с одинаковым названием выбирается аппаратный
(DWORD) точность_отсечения	Способ отсечения символов, частично выходящих за границы области отсечения; возможные константы: clip_default_precis— стандартный режим, clip_character_precis— отсекается весь символ, clip_stroke_precis— отсекается часть символа (с точностью до штриха) и т.д.
(DWORD) качество	Требуемое качество выводимого шрифта; возможные константы: default_quality— вид шрифта не имеет значения, draft_quality— качество вывода играет минимальную роль, допускается масштабирование растровых шрифтов, PROOF_QUALITY— качество вывода важнее, чем соответствие логическим атрибутам шрифта, а масштабирование растровых шрифтов недопустимо — выбирается наиболее близкий по размеру шрифт
(DWORD) интервал_и_семейство	В двух младших битах задается межсимвольный интервал шрифта; возможные константы: default_pitch — интервал не известен или не имеет значения, fixed_pitch— фиксированный интервал, fixed_pitch— переменный интервал (пропорциональный шрифт). В четырех старших битах определяется семейство, к которому относится шрифт; возможные константы: ff_dontcare— семейство не известно или не имеет значения, ffroman— шрифт с засечками, с переменной шириной символов, FF_SWISS— шрифт без засечек, с переменной шириной символов, ff_modern— шрифт с постоянной шириной символов, с засечками или без таковых FF_SCRIPT— шрифт напоминающий рукописный

	ff_decorative— декоративный шрифт
(LPCTSTR) название	Строка, содержащая название гарнитуры шрифта

При первом вызове функции CreateFont() создается пропорциональный полужирный шрифт размером в 12 логических единиц. По этой команде Windows пытается найти среди установленных шрифтов такой, который максимально точно отвечает запрашиваемым параметрам. Этот шрифт используется для вывода заголовка гистограммы, подписи к горизонтальной оси и метки, обозначающей максимальный размер столбца.

Во второй раз функция CreateFont() вызывается следующим образом:

```
newfont.CreateFont(12,12,900,900,FW_BOLD,
FALSE, FALSE, FALSE,
OEM_CHARSET, OUT_DEFAULT_PRECIS,
CLIP_DEFAULT_PRECIS,
DEFAULT_QDALITY,
VARIABLE_PITCH | FF_ROMAN,
"Roman");
```

Как видите, изменены только параметры ориентации строки и наклона символов. Оба параметра измеряются в десятых долях градуса, поэтому наклону в 90 градусов соответствует значение 900. Этот шрифт используется для вывода подписи к вертикальной оси гистограммы. А вот как осуществляется вывод строки:

```
oldfont = dc.SelectObject(Snewfont);
dc.TextOut(50, 200+(strlen(szXString)*10/2), szYString,
strlen(szYString));
```

Рисование осей координат и столбцов

Координатные оси x и y выводятся на экран с помощью функций MoveTo() и LineTo():

```
// Рисование осей координат
dc.MoveTo(99,49);
dc.LineTo(99,350);
dc.LineTo(500,350);
dc.MoveTo(99,350);
```

Затем выполняются действия по подготовке к выводу столбцов гистограммы. Как видно из следующего фрагмента программы, первый столбец всегда начинается с точки 100,350, на что указывают значения переменных x1 и y1. Горизонтальная координата всех последующих столбцов вычисляется на основании координаты предыдущего столбца и ширины столбцов.

```
// Начальные значения
x1= 100;
y1 = 350;
x2 = x1 + iBarWidth;
```

Информация о высоте столбцов хранится в массиве iBarSizeScaled. Поскольку функция Rectangle() создает замкнутые фигуры, то полученный столбец можно заполнить цветом, воспользовавшись текущей кистью. Цвет кисти выбирается из массива dwColor, причем номер столбца соответствует номеру цвета в массиве.

```
// Рисование столбцов
for(i= 0; i < iNBars; i++)
{
newbrush.CreateSolidBrush(dwColor[ i ] );
oldbrush = dc.SelectObject(Snewbrush);
y2 = 350 - iBarSizeScaled[i];
dc.Rectangle(x1,y1,x2,y2);
x1 = x2;
x2 += iBarWidth;
// удаление кисти
dc.SelectObject(oldbrush);
```



```
newbrush.DeleteObject() ; }
```

После того как программа выведет один столбец, значения переменных x_1 и x_2 будут модифицированы таким образом, чтобы указывать на начальную точку следующего столбца. Цикл, `for` выполняется столько раз, сколько столбцов содержит гистограмма.

Запуск программы

После запуска программы на экране отображается гистограмма, заданная по умолчанию (рис. 19.11). Теперь можно создать свою гистограмму, вроде той, что показана на рис. 19.12. Для этого необходимо ввести заголовок и подписи к осям, а также значения для столбцов.

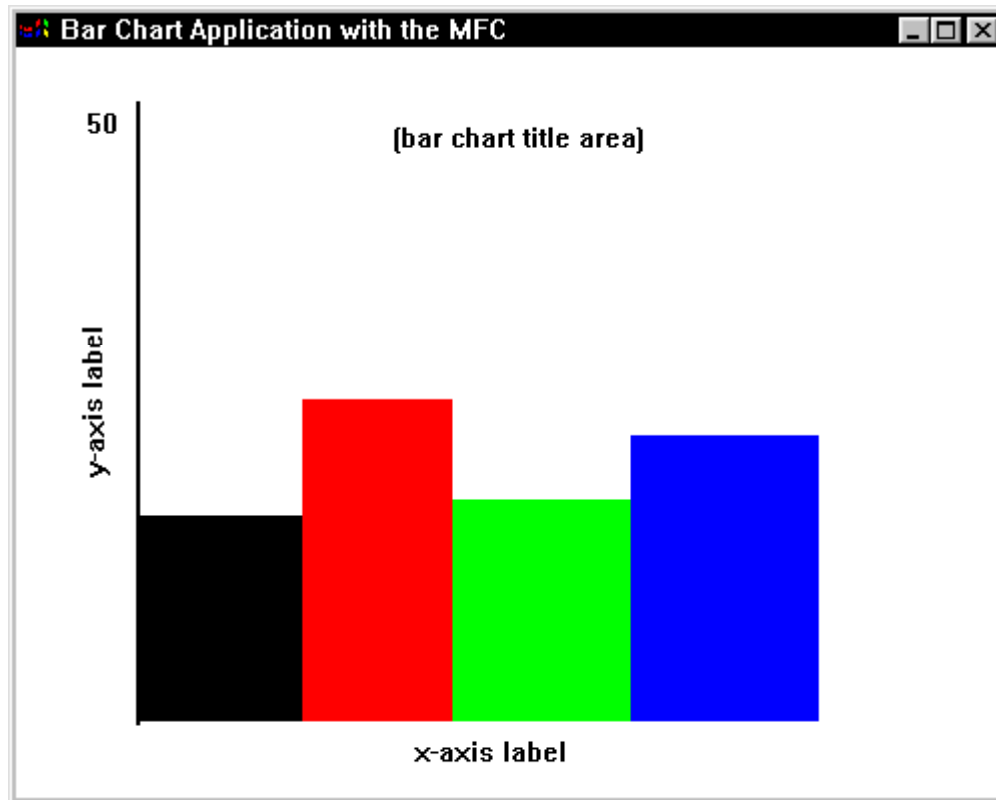


Рис. 19.11. Гистограмма, выводимая аналогичной программой по умолчанию

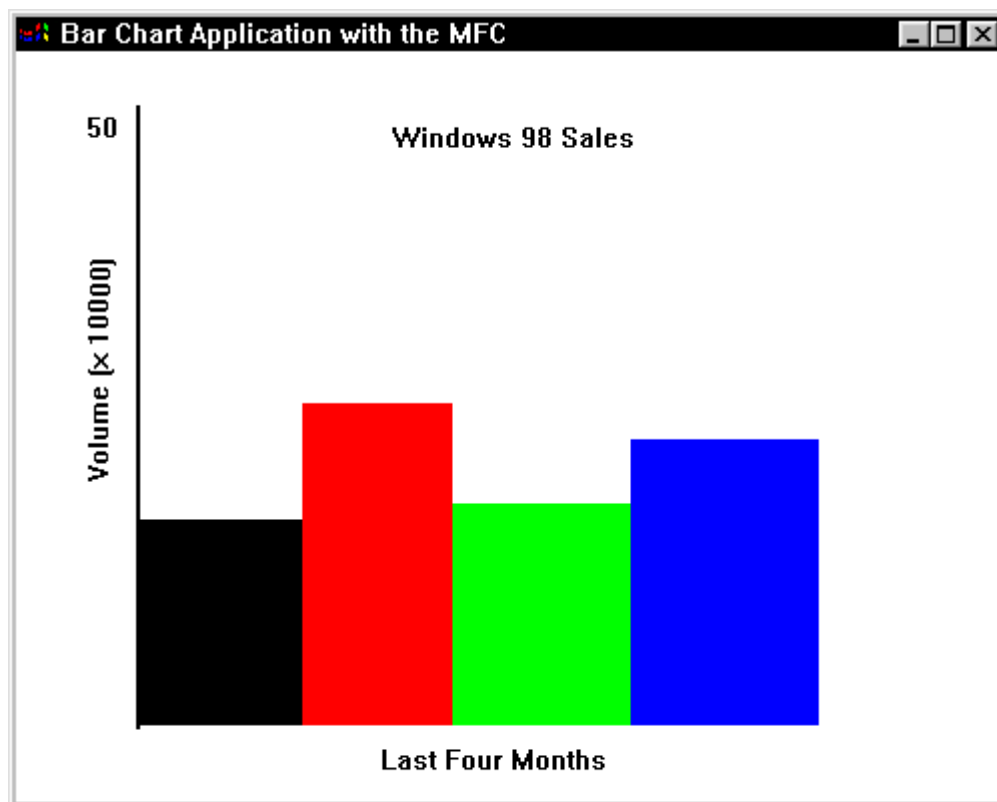


Рис. 19.12. Гистограмма, созданная на основании пользовательских данных

Глава 20. Мастера AppWizard и ClassWizard

- Программа Graph
 - Мастер приложений
 - Мастер классов
 - Построение приложения
 - Вывод данных в окно
- Текстовый редактор
 - Построение приложения

В предыдущих четырех главах вы узнали, как создаются приложения для 32-разрядных версий Windows95, 98 и NT с использованием методов как процедурного, так и объектно-ориентированного программирования с применением библиотеки MFC. Указанные главы были посвящены, главным образом, работе с программными шаблонами, благодаря которым программист может задействовать однотипные, часто повторяющиеся блоки в разных приложениях, не набирая их каждый раз по-новому. Но речь шла лишь об одном типе шаблонов — статических шаблонах, переносимых из одного приложения в другое вручную.

Однако часто этого бывает недостаточно. Конечно, если в новую программу необходимо внедрить лишь средства ввода/вывода, скажем создания, открытия и сохранения файлов, можно воспользоваться статическими шаблонами. Они же помогут вам и в тех случаях, когда в приложение нужно добавить стандартное меню с обычными командами — Cut, Copy, Paste и прочими. Но давайте расширим наши требования к программе. Предположим, нам необходимо создать MDI-приложение (MultipleDocumentInterface— многодокументный интерфейс), поддерживающее работу с несколькими документами, или добавить в программу поддержку технологии OLE. В этом случае одного статического шаблона будет недостаточно.

Специалисты Microsoft решили создать универсальный генератор динамических шаблонов, так называемый мастер AppWizard. Он тесно связан с библиотекой MFC и генерирует объектно-ориентированный код. Чтобы запустить мастер, необходимо в меню File активизировать команду New, а затем в раскрывшемся диалоговом окне выбрать из списка опцию MFC AppWizard. Мастер позволит вам самостоятельно определить функциональные возможности, которыми должна обладать программа. При этом у вас останется возможность модифицировать программу по своему усмотрению.

Близким "родственником" мастера AppWizard является другой мастер, ClassWizard. С его помощью можно добавить в приложение новый класс или отредактировать существующий. Часто мастер ClassWizard используют для редактирования программного кода, созданного перед этим с применением мастера AppWizard. Мастер классов запускается командой **ClassWizard** из меню View.

В данной главе подробно рассматривается использование обоих мастеров и даются практические советы по наиболее эффективному их применению. В качестве примеров будут разработаны два приложения. В первом случае мы сгенерируем каркас простого приложения, в окне которого выводится графическое изображение. Второе приложение должно быть более сложным: мы создадим упрощенный вариант текстового редактора, который будет поддерживать работу с несколькими документами, отображать панель инструментов, позволит вводить, редактировать и сохранять в файле текстовую информацию.

Программа Graph

В следующих параграфах мы пройдем через все этапы разработки базового приложения с помощью мастеров AppWizard и ClassWizard. Создание нового приложения происходит почти автоматически. Вам нужно только правильно установить соответствующие опции в окнах мастеров. Сейчас мы займемся проектированием нашего первого, пока что довольно простого приложения, названного Graph.

Мастер приложений

В окне компилятора Microsoft Visual C++ в меню File выберите команду New. Перед вами откроется диалоговое окно (рис. 20.1), с помощью которого можно начать новый проект, выбрав в списке элемент **MFC AppWizard (exe)**. Присвойте новому проекту имя Graph, задайте каталог проекта и щелкните на кнопке **OK**.

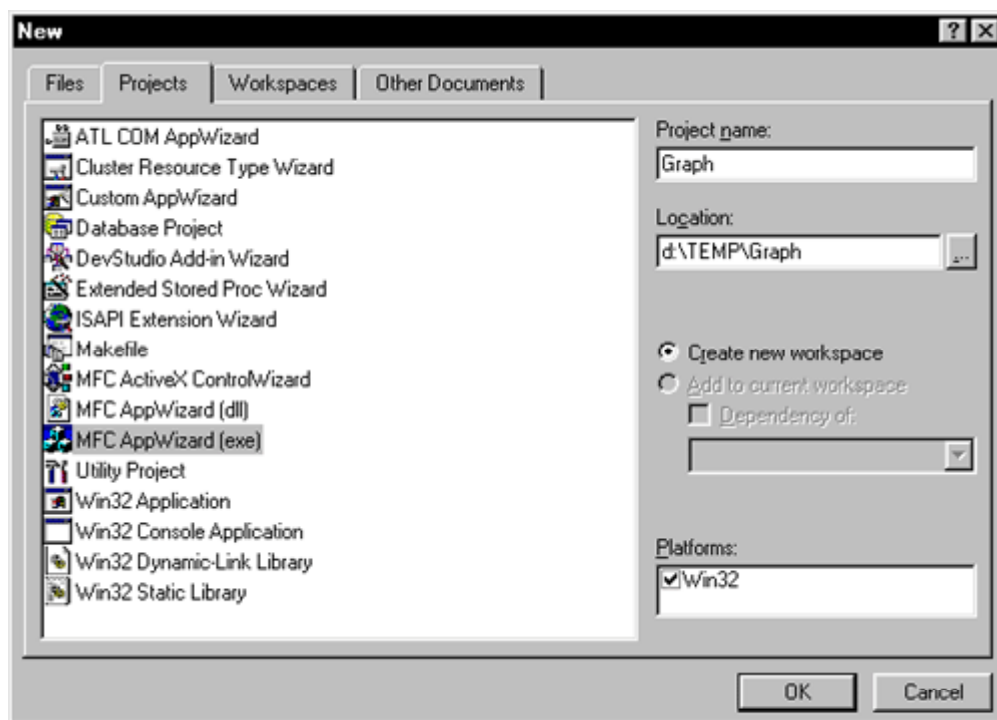


Рис. 20.1. Выбор типа проекта

Первое, что нужно будет сделать дальше, — это указать мастеру, должно ли приложение поддерживать работу с одним документом, с несколькими документами или же это будет диалоговое окно (рис. 20.2).

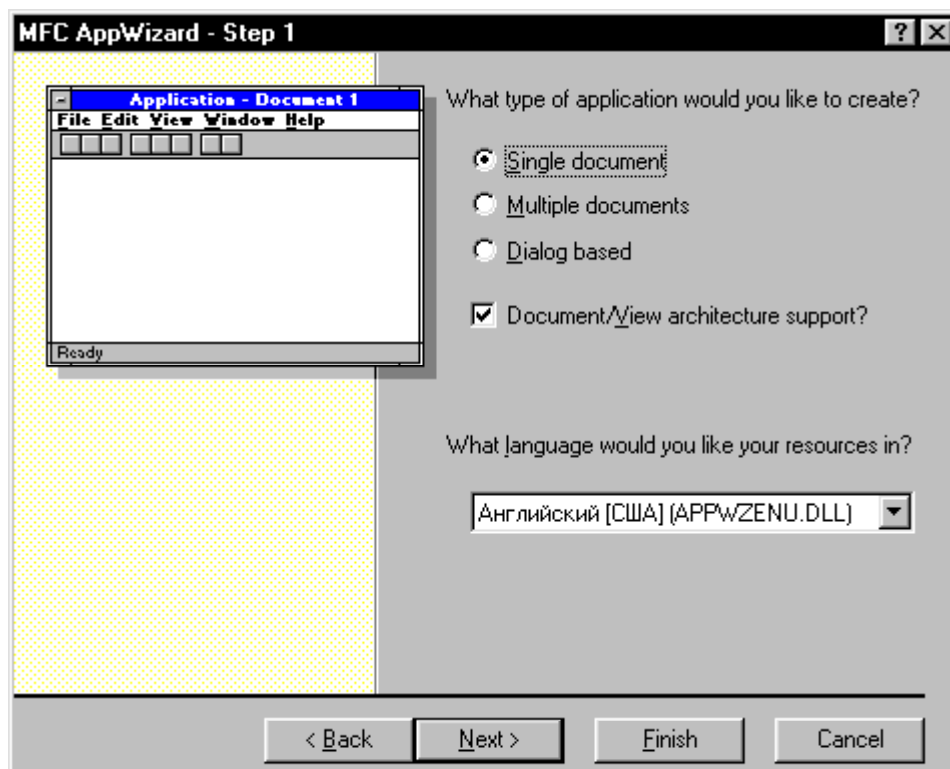


Рис.20.2. Шаг 1: выбор опции **Single Document**

SDI-приложение, поддерживающее работу с одним документом, создать проще всего, поскольку в этом случае не нужно учитывать взаимодействия между разными документами, одновременно открытыми одним приложением. Убедитесь, что установлена опция **Document/View architecture support?**. При этом можно принять язык ресурсов, предлагаемый по умолчанию. Чтобы перейти к следующему окну мастера, показанному на рис. 20.3, щелкните на кнопке **Next**.

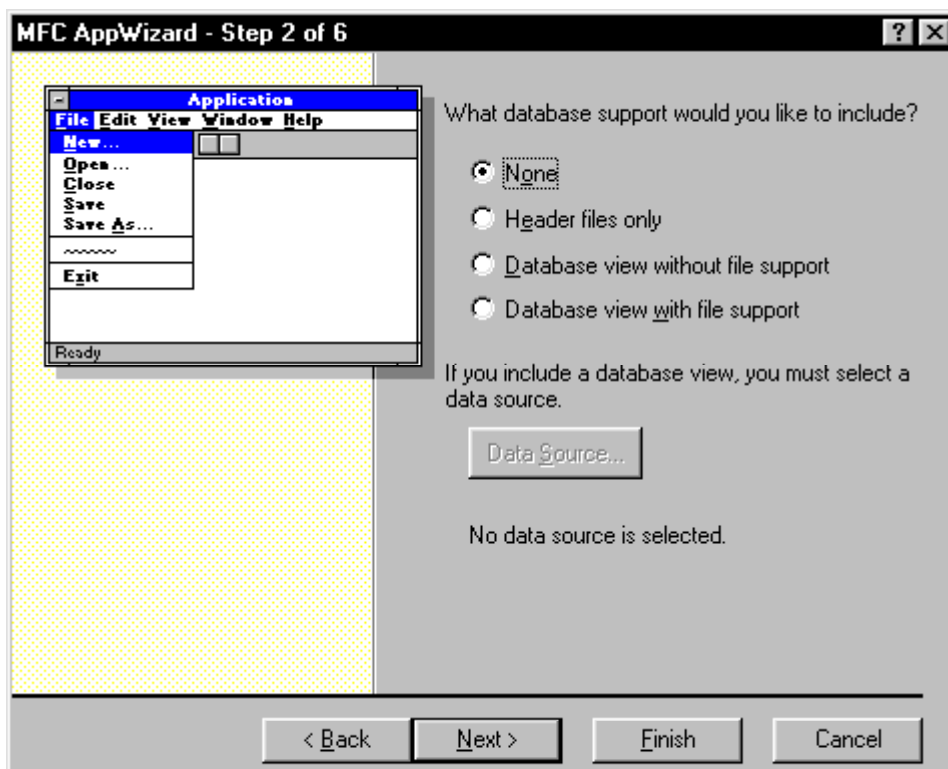


Рис. 20.3. Шаг 2: выбор опции **None** для отключения поддержки баз данных

Установки во втором окне мастера делаются только в том случае, если в приложении будет осуществляться работа с базами данных. В нашем примере следует оставить опцию **None**. Щелкните на кнопке Next, и вы перейдете к третьему окну (рис. 20.4).

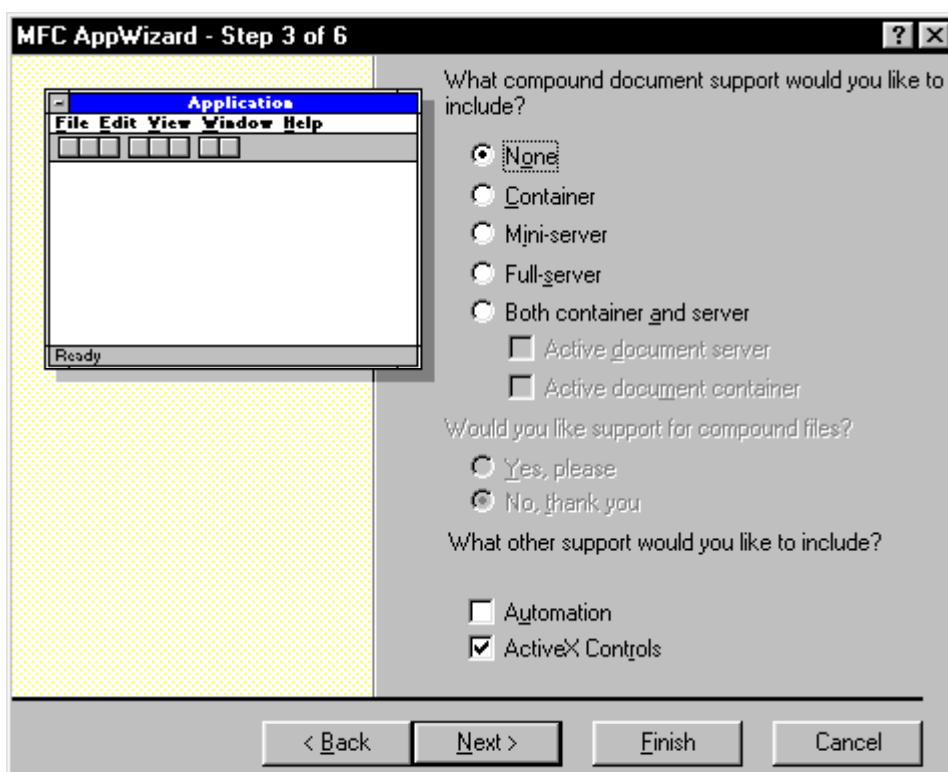


Рис. 20.4. Шаг 3: составные документы не поддерживаются

Теперь можно выбрать тип OLE-приложения (контейнер или сервер) а также задать другие опции, относящиеся к технологии OLE или ActiveX. Оставьте активной опцию **None** и щелкните на кнопке **Next**, чтобы перейти к следующему окну (рис. 20.5).

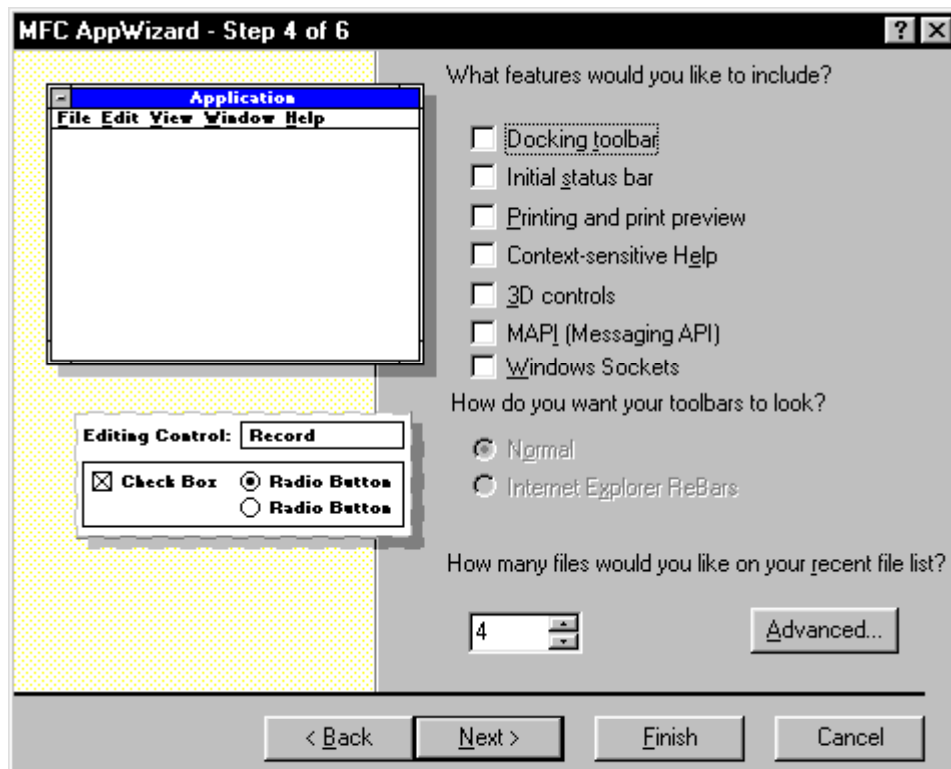


Рис. 20.5. Шаг 4: выбор дополнительных функциональных возможностей

Четвертое окно мастера позволяет добавлять в программу различные средства, определяющие интерфейс приложения (например, панель инструментов или строку состояния). Задавать какие-либо дополнительные возможности мы пока не будем, поэтому щелкните на кнопке **Next**, чтобы перейти к пятому окну мастера (рис. 20.6).

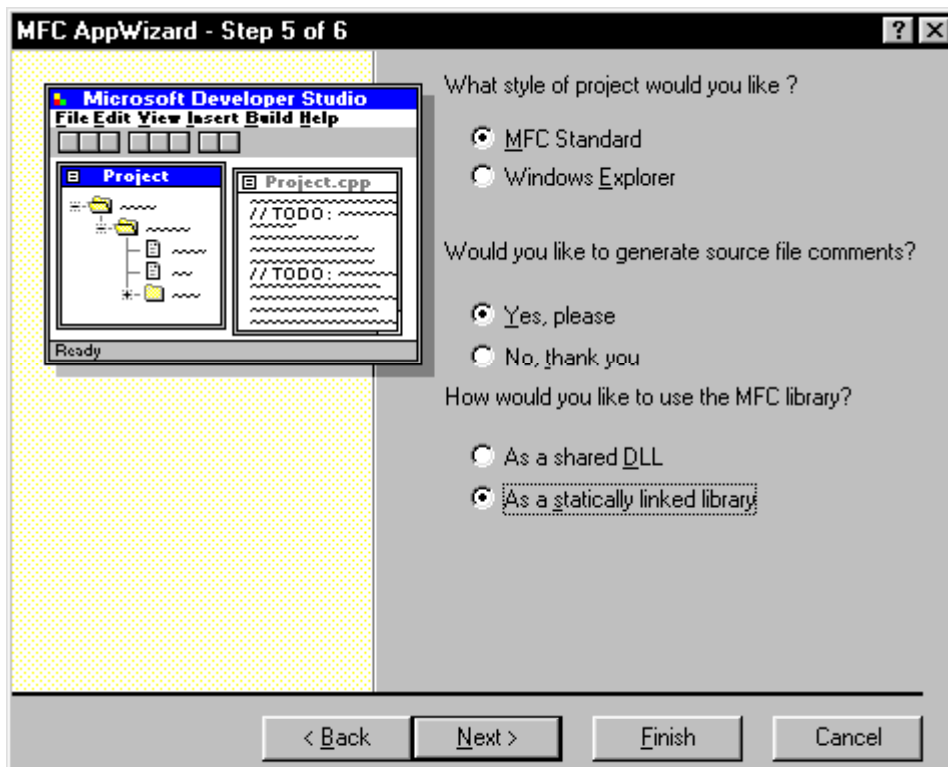


Рис. 20.6. Шаг 5: выбор стиля окна приложения, добавление комментариев и выбор способа компоновки библиотеки **MFC**

Установите опции так, как показано на рис. 20.6, а затем щелкните на кнопке Next, и вы перейдете к шестому окну мастера (рис. 20.7).

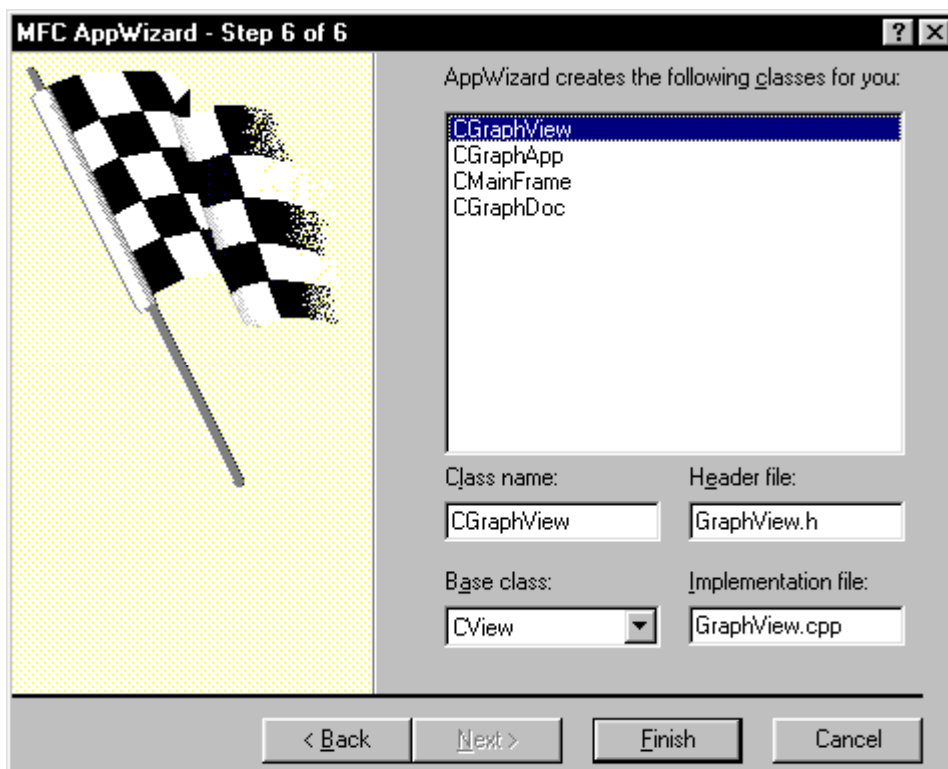


Рис. 20.7. Шаг 6: перечень генерируемых классов

В шестом окне мастера AppWizard отображается список классов, которые будут сгенерированы для нашего приложения, а именно CGraphApp, CMainFrame, CGraphDoc и CGraphView.

Для класса CGraphView, реализующего функции области просмотра, в списке **Baseclass** можно выбрать базовый класс (CEditView, CFormView, CHtmlView, CListView, CRichEditView, CScrollView, CTreeView или CView), от которого он будет порождаться. Класс CView, потомок CWnd, является родительским для всех перечисленных классов. Область просмотра служит буфером между документом и пользователем и представляет собой дочернее окно главного окна приложения. Она содержит графический образ документа, выводимого на экран или принтер, а также поддерживает возможность редактирования документа с помощью клавиатуры и мыши. Класс CFormView описывает область просмотра с полосами прокрутки, основанную на шаблоне диалогового окна и включающую элементы управления. Класс CEditView реализует функции текстового редактора и будет использован в следующем примере этой главы.

После щелчка на кнопке **Finish** будет выведено окно с итоговым отчетом, подготовленным мастером AppWizard (рис. 20.8).

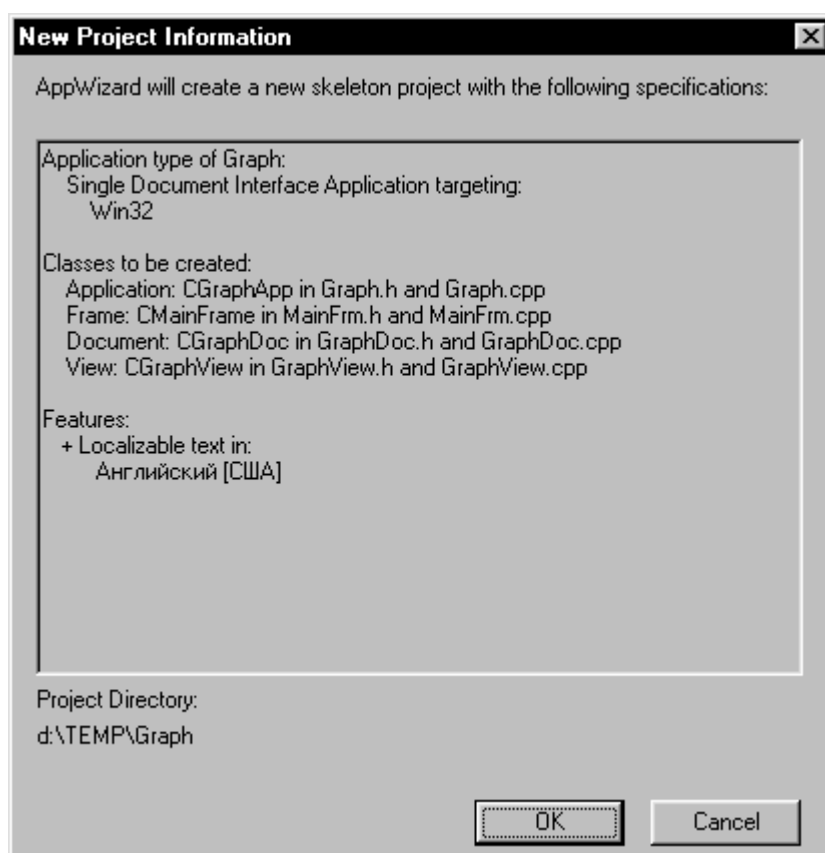


Рис. 20.8. Отчет, выдаваемый мастером AppWizard после установки всех опций

В отчете подытожены сделанные вами установки проекта. Если все правильно, нажмите кнопку **ОК**, после чего начнется создание кода программы. В процессе разработки проекта в указанную папку будут добавлены разные подпапки и файлы, необходимые для функционирования приложения.

Мастер классов

С помощью мастера ClassWizard для разрабатываемого приложения можно сгенерировать дополнительные программные блоки. В частности, вы можете создать подпрограммы обработки сообщений WM_PAINT, WM_MOUSEMOVE и т.д. Как уже упоминалось, для запуска мастера классов в меню View нужно выбрать команду **ClassWizard....** Добавив в программу

функцию OnPaint(), мы получим возможность обрабатывать сообщения `WM_PAINT`, связанные с перерисовкой рабочей области окна. Для этого в раскрывшемся окне, в списках **Classname** и **ObjectIDs**, необходимо выбрать класс `CGraphView` (рис. 20.9).

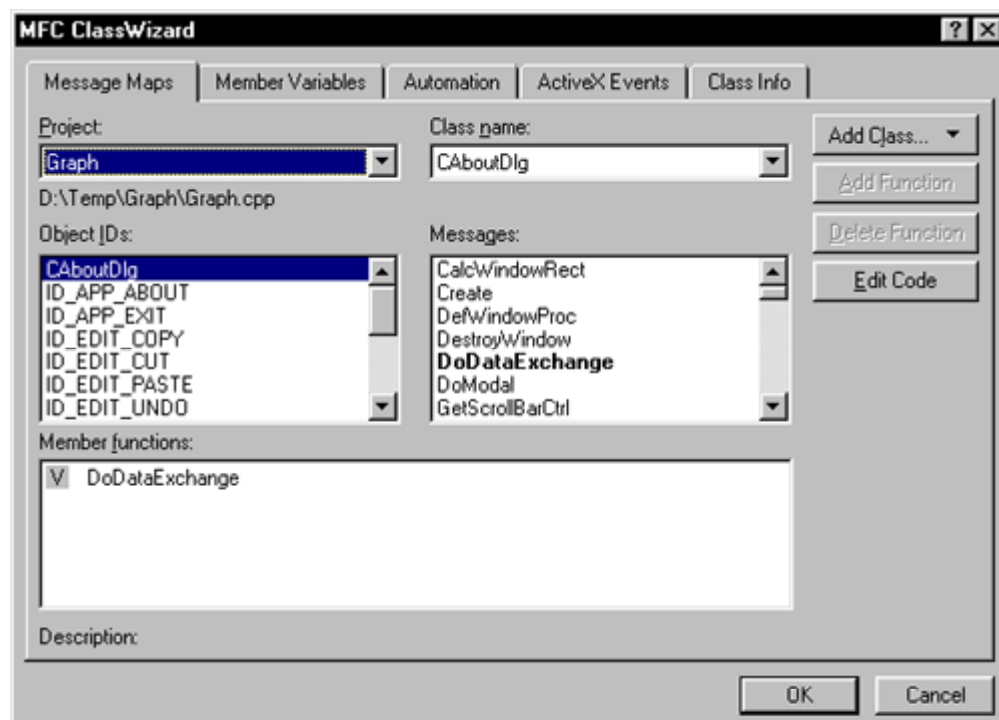


Рис. 20.9. Окно мастера классов, в котором выбран класс **CGraphView**

В списке **Messages** теперь отображается перечень сообщений, связанных с этим классом. Выполните двойной щелчок на сообщении `WM_PAINT`, и в списке **Memberfunctions** появится функция `OnPaint()` (рис. 20.10). После щелчка на кнопке **OK** данная функция будет добавлена в файл `GRAPHVIEW.CPP`.

Следующий этап — это компиляция и тестирование нового приложения.

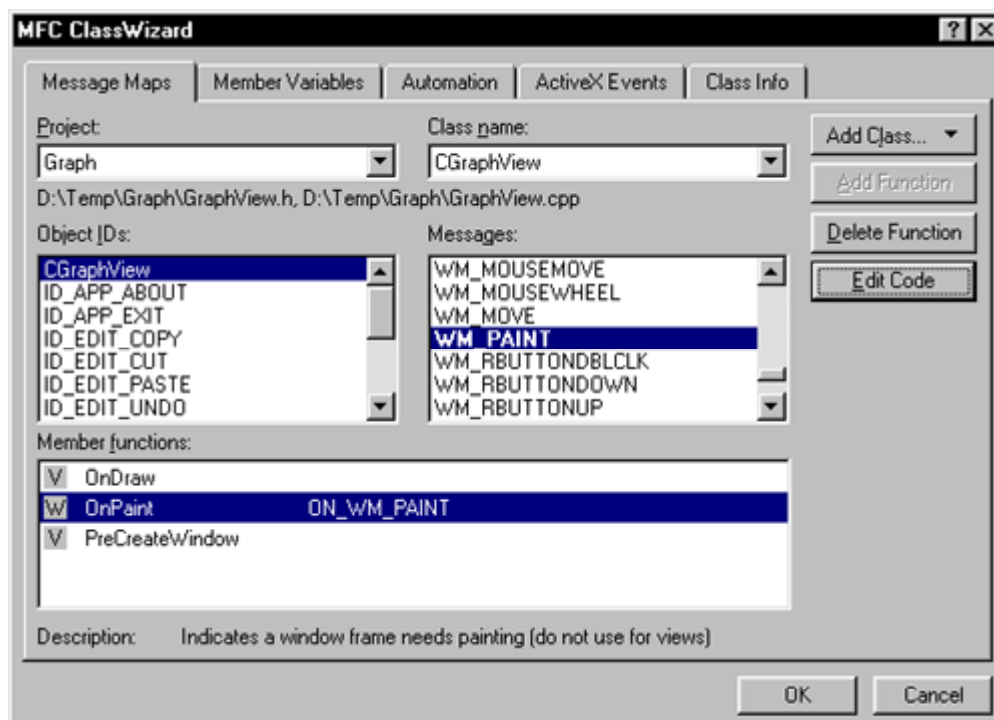


Рис. 20.10 В проект добавлена функция **OnPaint()**

Построение приложения

После того как все необходимые обработчики сообщений с помощью мастера классов будут добавлены в прогамму, вы можете приступить к построению приложения. Выберите в меню **Build** команду **RebuildAll**(рис. 20.11).

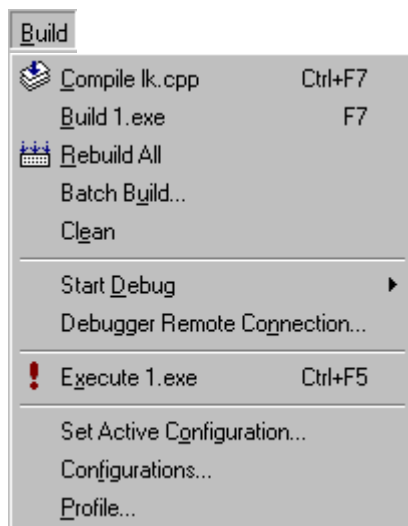


Рис. 20.11. Команда **Rebuild All** выполняет построение приложения

Во время построения приложения все данные об этом процессе будут выводиться на экран на вкладке **Build** окна **Output**(рис. 20.12).

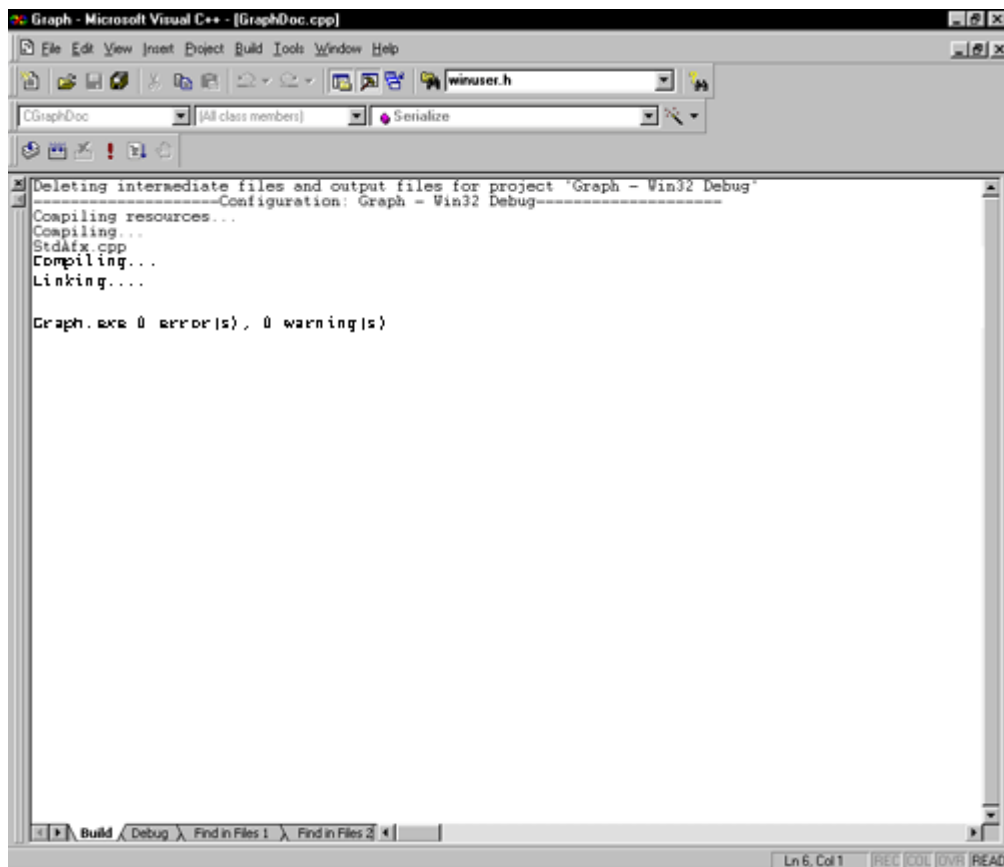


Рис. 20.12. Вывод информации о ходе компиляции

Компилятор сообщает о том, что четыре исходных файла — GRAPH.CPP, MAINFRM.CPP, GRAPHDOC.CPP и GRAPHVIEW.CPP- были успешно скомпилированы и скомпонованы. По правде говоря, это лишь вершина айсберга. Если вы просмотрите папку проекта, то найдете там еще около 30 различных файлов, сообщить о которых компилятор не посчитал нужным.

Исполняемый файл приложения по умолчанию будет записан в папку DEBUG.

Окно пусто, так как ни одна функция рисования в программу пока не добавлена. К тому же многие команды меню не будут работать. Дело в том, что код для обработки сообщений, связанных с выбором команд меню, вы должны ввести самостоятельно. Он не может быть сгенерирован автоматически. Но если вы хорошо усвоили предыдущий материал, то создание необходимого кода не станет для вас большой проблемой. В последнем примере этой главы будет показано, как написать соответствующие функции.

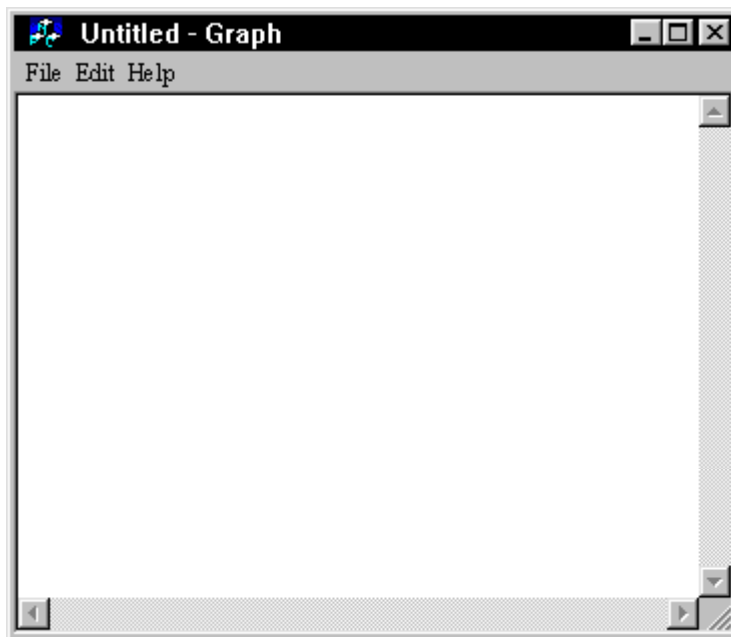


Рис. 20.13. Окно программы

Анализ программного кода

Мастер AppWizard автоматически создал четыре файла: Graph.cpp MainFrm.cpp GraphDoc.cpp и GraphView.cpp, каждый из них связан с соответствующим файлом заголовков. Файлы заголовков содержат описания классов, используемых в исходных файлах.

Файл Graph.cpp

Это основной файл и его код показан ниже:

```

////////////////////////////////////
// Единственный объект класса CGraphApp
CGraphApp theApp;
////////////////////////////////////
// инициализация класса CGraphApp
CGraphApp::InitInstance()
//Стандартная инициализация.
//Если вам не нужны используемые здесь возможности
// и вы хотите сократить размер исполняемого файла,
// удалите ненужные команды.
//Измените раздел реестра, где будут храниться
//параметры программы.
tRegistryKey(_T("LocalAppWizard-GeneratedApplications"));
adStdProfileSettings(); // Загрузка параметров из INI-файла,
//в том числе списка последних открытых файлов
//Регистрация шаблона документов приложения
SingleDocTemplate* pDocTemplate; 'OcTemplate = new CSingleDocTemplate
(
////////////////////////////////////
// CGraphApp
BEGIN_MESSAGE_MAP(CGraphApp, CWinApp) //{AFX_MSG_MAP(CGraphApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
// макросы схемы сообщений.

```

```

// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//}}AFX_MSG_MAP
// Стандартные операции с документами
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
END_MESSAGE_MAP()
//{{AFX_DATA_INIT(CAboutDlg) }}AFX DATA INIT
}
void CAboutDlg::DoDataExchange(CDataExchange* pDX) f
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CAboutDlg)
//}}AFX_DATA_MAP i
BEGIN_MESSAGE_MAP (CAboutDlg, CDialog)
// { (AFX_MSG_MAP (CAboutDlg)
//}}AFX_MSG_MAP END_MESSAGE_MAP ( )
// Функция, управляющая выводом окна About
void CGraphApp : : OnAppAbout ( )
{
CAboutDlg aboutDlg;
aboutDlg . DoModal ( ) ;
}
////////////////////////////////////
//Другие функции класса CGraphApp

```

Первая схема сообщений, содержащаяся в этом файле, принадлежит классу CGraphApp. В ней сообщения с идентификаторами id_app_about, id_file_new и ID_FILE_OPEN связываются с обработчиками OnAppAbout(), CWinApp::OnFileNew()и CWinApp::OnFileOpen(). В файле содержатся реализации конструктора класса CGraphApp, а также его методов initinstance() и OnAppAbout().

Диалоговое окно **About** принадлежит классу CAboutDlg, являющемуся потомком класса CDialog. У него имеется схема сообщений, конструктор и метод DoDataExchange ().

Файл MAINFRM.CPP

Файл MAINFRM.CPPсодержит реализацию класса CMainFrame, который порождается от класса CFrameWnd и управляет SDI-окном приложения.

```

// MainFrm.cpp: реализация класса CMainFrame//
#include "stdafx.h"
#include "Graph.h"
#include "MainFrm.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = _FILE
#endif
////////////////////////////////////
1 1 CMainFrame
IMPLEMENT_DYNCREATE (CMainFrame, CFrameWnd)
BEGIN_MESSAGE_MAP (CMainFrame, CFrameWnd)
// { (AFX_MSG_MAP (CMainFrame)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
// макросы схемы сообщений.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//}}AFX_MSG_MAP
END_MESSAGE_MAP ( )

```

```

////////////////////////////////////
1 1 Конструктор и деструктор класса CMainFrame
CMainFrame:: CMainFrame() {
// TODO: здесь добавьте код конструктора,
}
CMainFrame::~CMainFrame()
{
}
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT Ses) {
if(ICFrameWnd::PreCreateWindow(cs))
return FALSE;
// TODO: здесь можно модифицировать класс окна,
//      изменяя поля структуры cs.
return TRUE; }
////////////////////////////////////
// Диагностика класса CMainFrame
#ifdef _DEBUG
void CMainFrame: :AssertValid() const
{
CFrameWnd: :AssertValid() ;
void CMainFrame: : Dump (CDumpContext &dc) const {
CFrameWnd: : Dump (dc);
}
#endif // _DEBUG
////////////////////////////////////
// Обработчики сообщений класса CMainFrame

```

Внимательно просмотрев этот листинг, вы заметите, что конструктор и деструктор не содержат никакого кода, обработчики сообщений отсутствуют, а для функций AssertValid(), Dump() и PreCreateWindow() вызываются одноименные методы базового класса.

Файл GRAPHDOC.CPP

Файл GRAPHDOC.CPP содержит реализацию класса CGraphDoc, который управляет работой с документом, а также обеспечивает загрузку и сохранение данных документа.

```

// GraphDoc.cpp: реализация класса CGraphDoc//
#include "stdafx.h"
#include "Graph. h"
#include "GraphDoc. h"
#ifdef _DEBUG #define new DEBUGJSIEW
#undef THIS_FILE
static char THIS_FILE [ ] = _ FILE _ ;
#endif
////////////////////////////////////
// CGraphDoc
IMPLEMENT_DYNCREATE (CGraphDoc, CDocument)
BEGIN_MESSAGE_MAP (CGraphDoc, CDocument) //( AFX_MSGJXIAP (CGraphDoc)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
//      макросы схемы сообщений.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//}AFX_MSG_MAP
END_MESSAGE_MAP ( )
////////////////////////////////////
//Конструктор и деструктор класса CGraphDoc
CGraphDoc:: CGraphDoc() {

```

```

// TODO: здесь добавьте код конструктора
}
CGraphDoc::CGraphDoc ()
{
}
BOOL CGraphDoc::OnNewDocument()
if (ICDocument::OnNewDocument()) return FALSE;
: // TODO: здесь добавьте код повторной инициализации
// (специфика SDI-приложений).
return TRUE; )
////////////////////////////////////
// Сериализация класса CGraphDoc
void CGraphDoc: : Serialize (CArchive Sar) {
if (ar . IsStoring () ) {
// TODO: здесь добавьте код сохранения }
else{
// TODO: здесь добавьте код загрузки
} }
////////////////////////////////////
// Диагностика класса CGraphDoc
#ifdef _DEBUG
void CGraphDoc::AssertValid() const
{
CDocument::AssertValid(); }
void CGraphDoc: : Dump (CDumpContext Sdc) const f
CDocument :: Dump (dc); ) #endif //_DEBUG
////////////////////////////////////
// Другие функции класса CGraphDoc

```

Проанализировав этот листинг, вы вновь обнаружите, что схема сообщений, конструктор и деструктор класса пусты. Для остальных четырех функций вызываются их реализации из базового класса. Функция Serialized реализует концепцию постоянства документа. В следующем примере посредством этой функции будет организован файловый ввод/вывод данных документа.

Файл GRAPHVBEW.CPP

Файл GRAPHVIEW.CPP содержит реализацию класса CGraphview, который порождается от класса CView и отвечает за отображение документа.

```

// GraphView.cpp: реализация класса CGraphView//
#include "stdafx.h"
#include "Graph. h"
#include "GraphDoc.h"
#include "GraphView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS__FILE
static char THIS_FILE[] = __FILE__ ;
#endif
////////////////////////////////////
// CGraphview
IMPLEMENT_DYNCREATE (CGraphview, CView)
BEGIN_MESSAGE_MAP (CGraphview, CView)
// ( {AFX_MSG_MAP (CGraphview)
ON_WM_PAINT()
//}}AFX_MSG_MAP END_MESSAGE_MAP ( )

```



```

////////////////////////////////////
// Конструктор и деструктор класса CGraphview
CGraphview: : CGraphview() {
// TODO: здесь добавьте код конструктора
}
CGraphView::~~CGraphView()
{
}
BOOL CGraphView::PreCreateWindow(CREATESTRUCT Ses)
{
// TODO: здесь можно модифицировать класс окна,
// изменяя поля структуры cs.
return CView::PreCreateWindow(cs); }
////////////////////////////////////
// Отображение документа
void CGraphview::OnDraw(CDC* pDC)
{
CGraphDoc* pDoc = GetDocument(); ASSERT_VALID(pDoc);
// TODO: здесь добавьте код отображения }
////////////////////////////////////
// Диагностика класса CGraphView
#ifdef _DEBUG
void CGraphView::AssertValid() const
{
CView::AssertValid(); }
void CGraphview: : Dump (CDumpContext Sdc) const {
CView: : Dump (dc); }
CGraphDoc* CGraphview: : GetDocument () // отладочная версия (
ASSERT (m_pDocument->IsKindOf (RUNTIME_CLASS (CGraphDoc) ) ) ;
return (CGraphDoc*)m_pDocument; } #endif //_DEBUG
////////////////////////////////////
// Обработчик сообщений класса CGraphview
void CGraphview: :OnPaint () {
CPaintDCdc(this); // контекст устройства для рисования
// TODO: здесь добавьте собственный код обработки сообщения
// Не вызывайте метод CView: :OnPaint ()

```

Обычно схема сообщений остается пустой, но, если вы помните, мы использовали мастер классов для добавления обработчика `on_wm_paint`. Конструктор и деструктор остались пустыми.

В функции `OnDraw()` запрашивается указатель `pDoc` на данные, содержащиеся в документе. Для функций `AssertValid()` и `Dump()` вызываются их аналоги из базового класса.

Вывод данных в окно

На первом этапе разработки приложения `Graph` с помощью мастеров `App Wizard` и `Class Wizard` был создан интерфейс приложения, ориентированный на работу с одним документом. С помощью мастера классов в класс `CGraphview` был добавлен обработчик сообщений `WM_PAINT`. Код для функции `OnPaint()` мы возьмем из программы `GDI.CPP`, рассматривавшейся в предыдущей главе.

```

// Обработчик сообщений класса CGraphView
void CGraphView::OnPaint()
{
static DWORD dwColor[9]={
RGB(0,0, 0), // черный
RGB(255,0, 0), // красный

```

```

RGB(0,255,0),      // зеленый
RGB(0,0, 255),     // синий
RGB(255, 255, 0),  // желтый
RGB(255,0, 255),   // пурпурный
RGB(0,255, 255),   // голубой
RGB (127,127,127), // серый
RGB (255,255, 255)}; //.белый
int xcoord; POINT polypts[4], polygpts[5];
CBrush newbrush; CBrush* oldbrush;
CPen newpen; CPen* oldpen;
CPaintDCdc(this); // контекст устройства для рисования
// рисование эллипса и заливка его красным цветом
newpen.CreatePen(PS_SOLID, 1, dwColor[1]);
oldpen = dc.SelectObject(Snewpen);
newbrush.CreateSolidBrush(dwColor[1]) ;
oldbrush = dc.SelectObject(Snewbrush);
dc.Ellipse(275,300, 200, 250);
dc.TextOut(220,265,"ellipse",7);
dc.SelectObject(oldbrush) ;
newbrush.DeleteObject() ;
dc.SelectObject(oldpen) ;
newpen.DeleteObject();
// рисование круга и заливка его синим цветом
newpen.CreatePen(PS_SOLID, 1, dwColor[3]);
oldpen = dc.SelectObject(snewpen) ;
newbrush.CreateSolidBrush(dwColor[3]) ;
oldbrush = dc.SelectObject(Snewbrush);
dc.Ellipse(375,75, 525, 225);
dc.TextOut(435,190,"circle",6) ;
dc.SelectObject(oldbrush) ;
newbrush.DeleteObject();
dc.SelectObject(oldpen);
newpen.DeleteObject();
// рисование нескольких зеленых точек
for(xcoord = 400; xcoord < 450; xcoord += 5) dc.SetPixel(xcoord, 350,
OL);
dc.TextOut(460,345, "<-pixels",9);
// рисование толстой черной диагональной линии
newpen.CreatePen(PS_SOLID, 6, dwColor[0]);
oldpen = dc.SelectObject (Snewpen);
dc.MoveTo(20, 20); dc.LineTo(100, 100);
dc.TextOut(60,20,"<-diagonal line",16);
dc.SelectObject(oldpen); newpen.DeleteObject();
// рисование синей дуги
newpen.CreatePen(PS_DASH, 1, dwColor[3]);
oldpen = dc.SelectObject(Snewpen);
dc.Arc(25,125, 175, 225, 175, 225, 100, 125);
dc.TextOut(50,150,"small arc ->",12);
dc.SelectObject(oldpen);
newpen.DeleteObject();
// рисование зеленого сегмента с толстым контуром
newpen.CreatePen(PS_SOLID, 8, dwColor[2]);
oldpen = dc.SelectObject(Snewpen);
dc.Chord(125, 125, 275, 225, 275, 225, 200, 125);

```

```

dc.TextOut(280,150, "<-chord", 8);
dc.SelectObject(oldpen);
newpen.DeleteObject();
// рисование черного сектора и заливка его зеленым цветом
newpen.CreatePen(PS_SOLID, 1, dwColor[0]);
oldpen = dc.SelectObject(Snewpen);
newbrush.CreateSolidBrush(dwColor[2]);
oldbrush = dc.SelectObject(Snewbrush);
dc.Pie(200,0, 300, 100, 200, 50, 250, 100);
dc.TextOut (260,80, "<-pie wedge", 12);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
dc.SelectObject(oldpen);
newpen.DeleteObject();
// рисование черного прямоугольника и заливка его серым цветом
newbrush.CreateSolidBrush(dwColor[7]);
oldbrush = dc.SelectObject(Snewbrush);
dc.Rectangle(25,300, 150, 375);
dc.TextOut(50,325,"rectangle", 9);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
// рисование черного закругленного прямоугольника
// и заливка его синим цветом
newbrush.CreateHatchBrush(HS_CROSS, dwColor[3]);
oldbrush = dc.SelectObject(Snewbrush);
dc.RoundRect(350, 250, 400, 290,20, 20);
dc.TextOut(410,270, "<-rounded rectangle", 20);
dc.SelectObject(oldbrush);
newbrush.DeleteObject();
// рисование толстой ломаной линии пурпурного цвета
newpen.CreatePen(PS_SOLID, 3, dwColor[5]);
oldpen=dc.SelectObject(Snewpen);
polylpts[0].x = 10;
polylpts[0].y = 30;
polylpts[1].x = 10;
polylpts[1].y = 100;
polylpts[2].x = 50;
polylpts[2].y = 100;
polylpts[3].x = 10;
polylpts[3].y = 30;
dc.Polyline(polylpts, 4);
dc.TextOut(10,110, "polyline",8) ;
dc.SelectObject(oldpen) ;
newpen.DeleteObject() ;
// рисование голубого многоугольника
//изаливка его диагональной желтой штриховкой
newpen.CreatePen(PS_SOLID, 4, dwColor[6]);
oldpen = dc.SelectObject(Snewpen);
newbrush.CreateHatchBrush(HS_FDIAGONAL, dwColor[4]);
oldbrush = dc.SelectObject(Snewbrush); }
polygpts[0].x = 40;
polygpts[0].y = 200;
polygpts[1].x = 100;
polygpts[1].y = 270;

```

```

polygpts[2^.x = 80;
polygpts[2].y = 290;
polygpts[3].x = 20;
polygpts[3].y = 220;
polygpts[4].x = 40;
polygpts[4].y = 200;
dc.Polygon(polygpts, 5);
dc.TextOut(80,230, "<-polygon", 10);
dc.SelectObject(oldbrush) ;
newbrush.DeleteObject() ;
dc.SelectObject(oldpen);
newpen.DeleteObject() ;
// Не вызывайте метод CView::OnPaint() }

```

После внесения изменений скомпилируйте и запустите новую версию приложения. Мастер AppWizard сгенерировал базовый код программы и строку меню с подменю **File**, **Edit** и **Help**. При выборе в меню **Help** команды **About** на экран открывается диалоговое окно **About**. Остальные команды меню пока не работают. Почему? Потому что базовая программа не содержит функций, отвечающих за обработку сообщений, которые поступают при выборе команд меню.

В следующем примере мы воспользуемся мастером AppWizard для разработки простого текстового редактора, который значительно функциональнее рассмотренного только что приложения.

Текстовый редактор

Теперь займемся разработкой приложения, которое позволит нам вводить и редактировать текст. Выберите в окне компилятора Microsoft Visual C++ команду New из меню File, а в открывшемся диалоговом окне New — опцию MFC AppWizard (exe), чтобы запустить мастер AppWizard. Имя нового проекта — Editor (рис. 20.16).

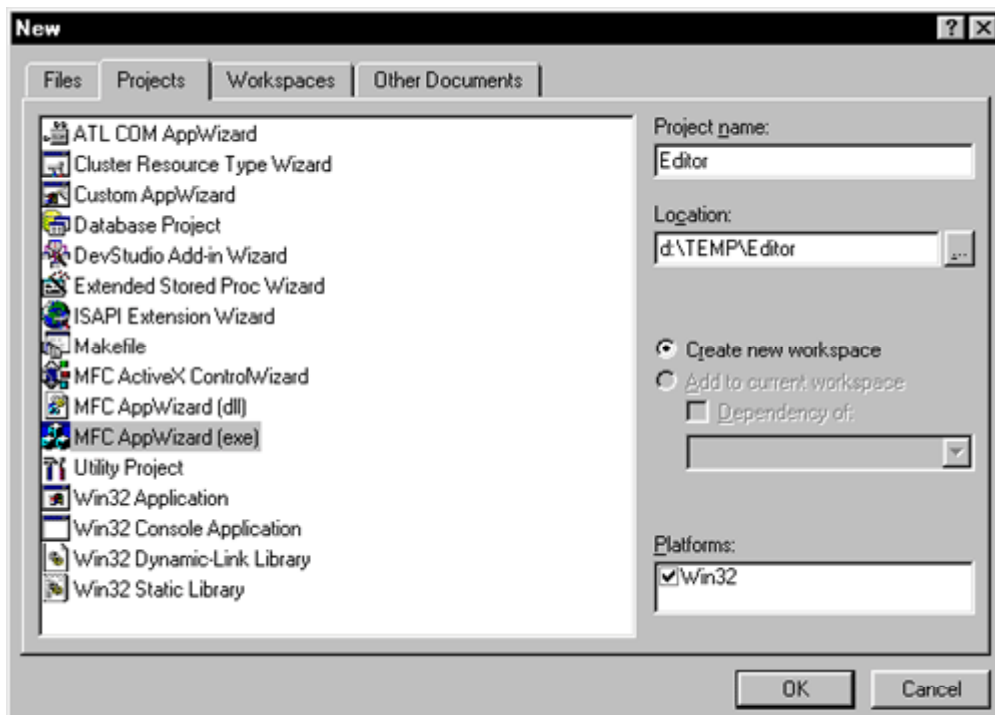


Рис. 20.16. Выбор типа проекта

Мастер приложений автоматически создаст папку для нового приложения с тем именем, которое вы задали. На экране откроется первое, уже знакомое вам окно мастера. Теперь установите опцию Multiple documents, отвечающую за создание приложения, которое будет поддерживать работу с несколькими документами (рис. 20.17).

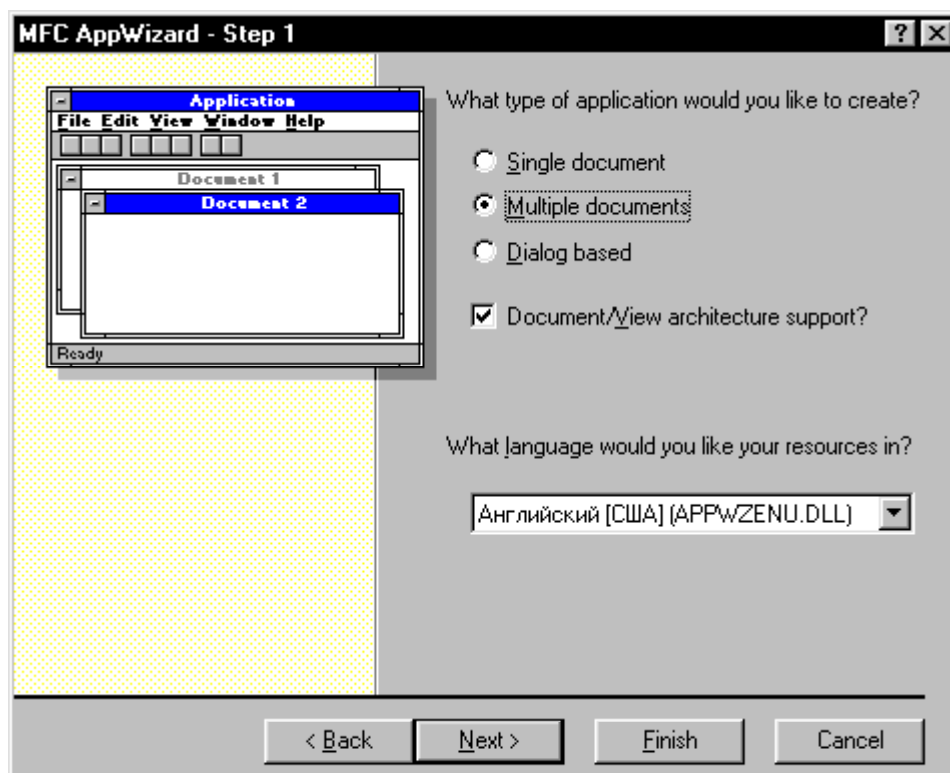


Рис. 20.17. Шаг 1: Выбор многодокументного интерфейса

Как следует из рис. 20.18 и 20.19, в создаваемом приложении не предусмотрено использование баз данных и составных документов.

На следующем этапе (шаг 4) установите опции, позволяющие добавить в окно приложения панель инструментов, строку состояния, трехмерные элементы управления и включить возможность вывода документов на печать (рис. 20.20).

Далее устанавливаются опции, подавляющие включение комментариев в код программы и задающие статический режим компоновки библиотеки MFC (рис. 20.21).

Теперь (шаг 6) мастер должен вывести список из пяти классов, которые будут им созданы, а именно: CEditorApp, CMainFrame, CChildFrame, CEditorDoc и CEditorView (рис. 20.22).

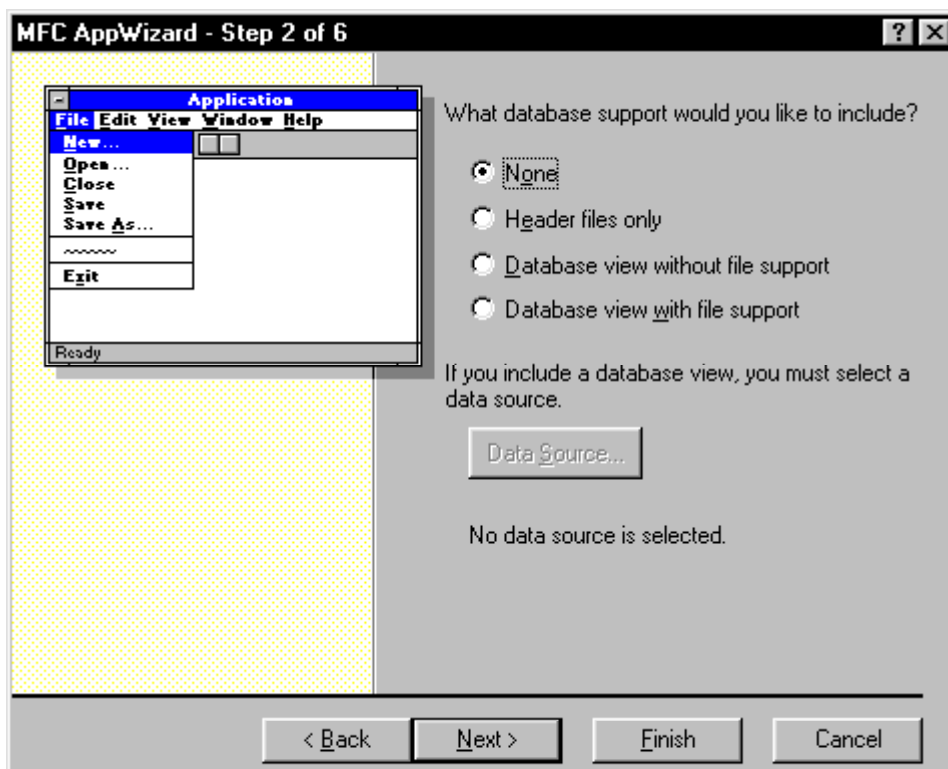


Рис.20.18. Шаг 2: базы данных не используются

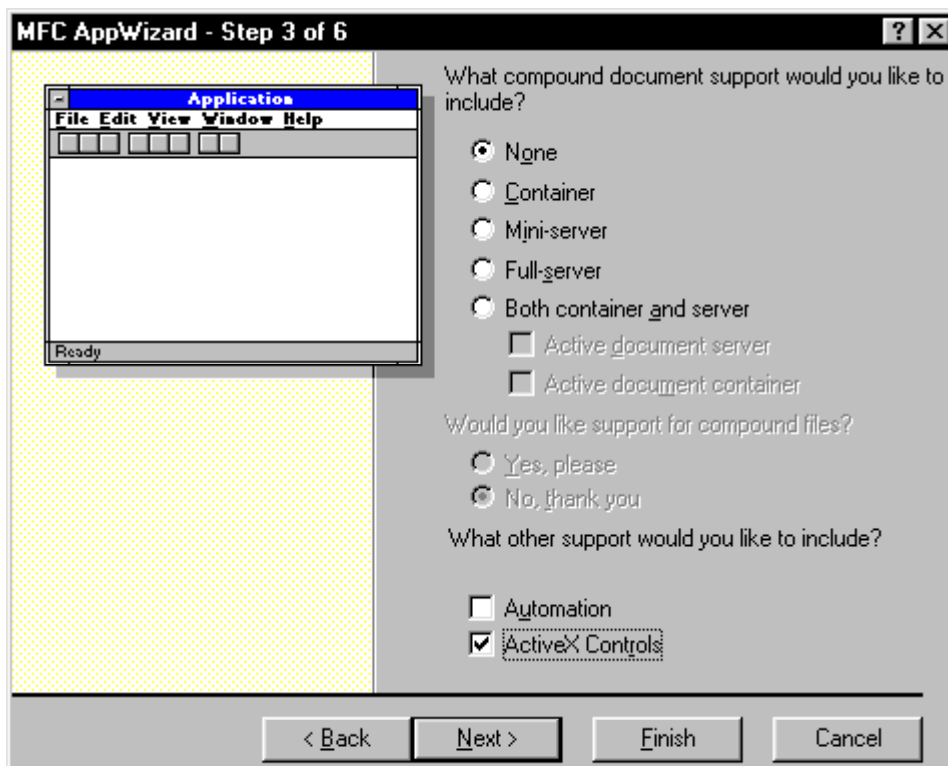


Рис. 20.19. Шаг 3: работа с составными документами не поддерживается, но используются элементы **ActiveX**

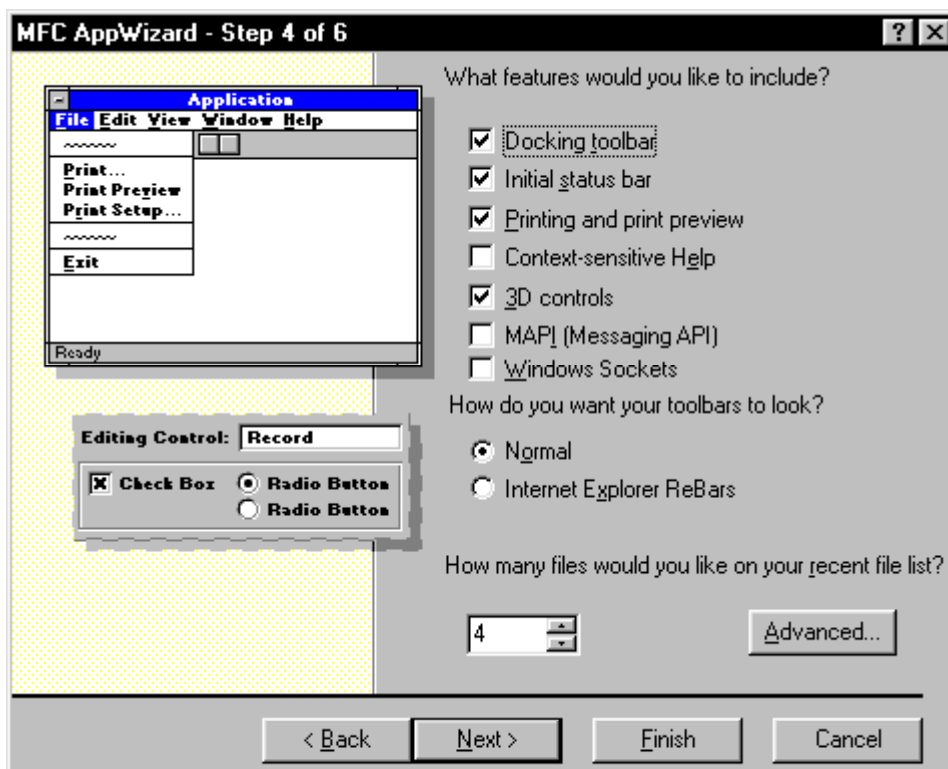


Рис. 20.20. Шаг 4: в окне приложения теперь можно использовать панель инструментов, строку состояния, трехмерные элементы управления, а также выводить документы на печать

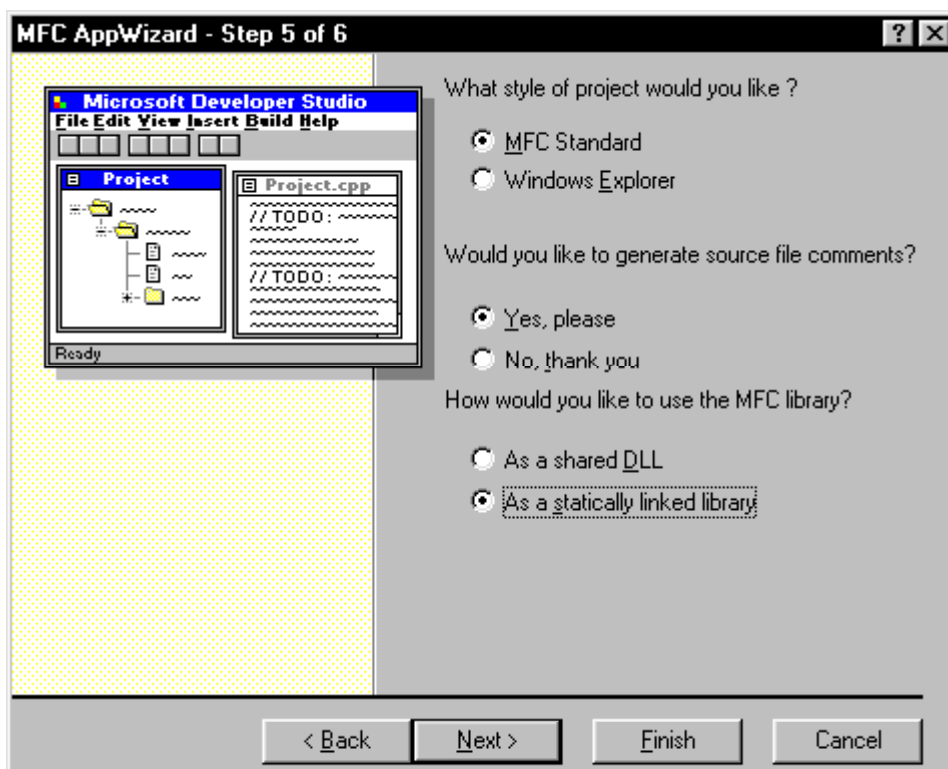


Рис.20.21. Шаг 5: отменено добавление комментариев в код программы и задан статический режим компоновки библиотеки MFC

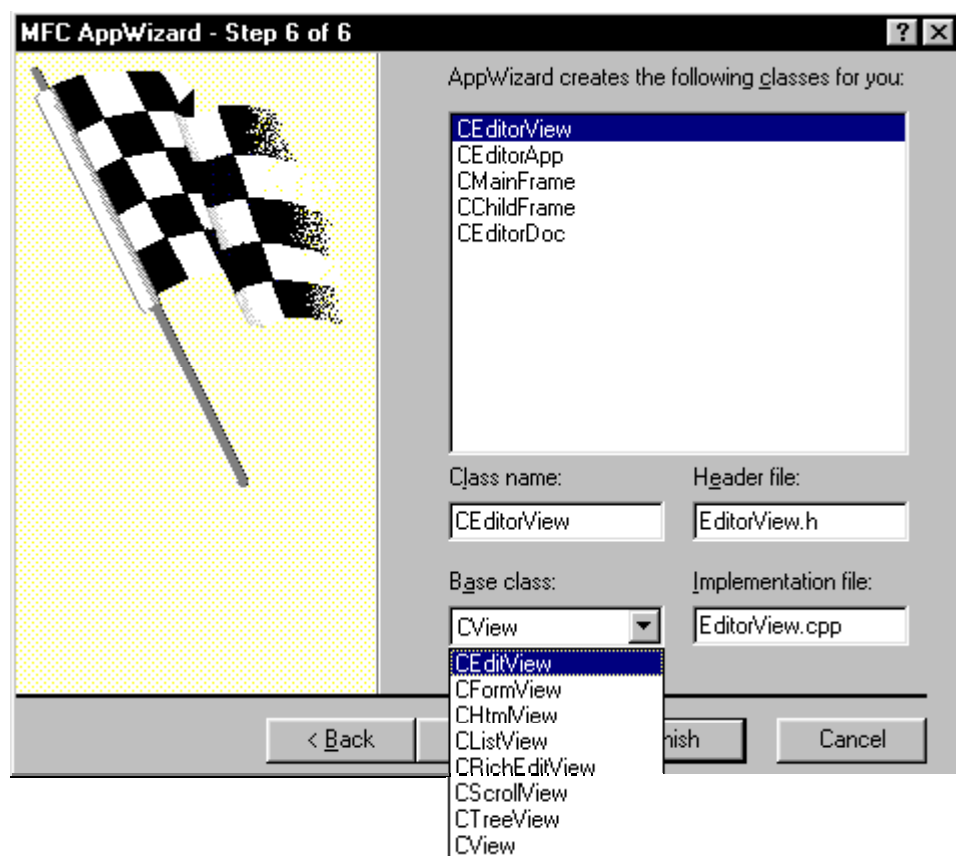


Рис. 20.22. Шаг 6: для работы приложения **Editor** необходимо сгенерировать пять классов

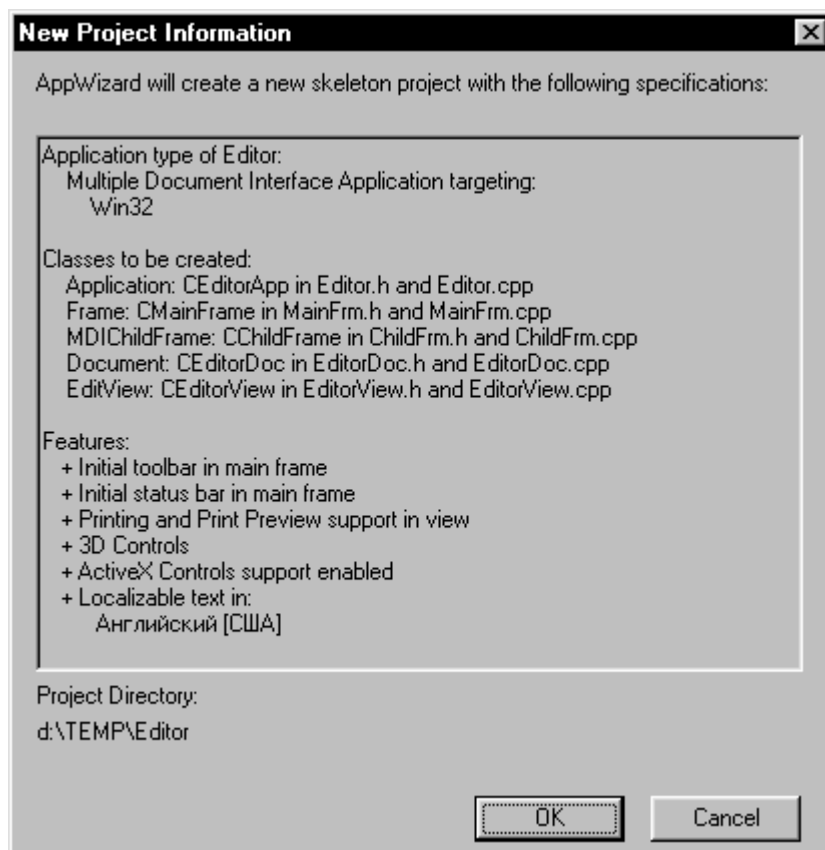


Рис. 20.23. Отчет об установках, сделанных в окнах мастера приложений

Если выбрать класс CEditorView, в списке **Baseclass** будет предложен перечень классов, которые можно использовать в качестве базового. в их числе: CEditview, CFormView, CHtmlView, CListView, CRichEditView, CScrollView, CTreeView и CView. В нашем примере класс CEditorView порождается от CEditview и используется в качестве родительского для пользовательских классов областей просмотра. Базовый класс CEditview инкапсулирует основные функции текстового редактора.

Далее необходимо щелкнуть на кнопке **Finish**, в результате чего будет выведен отчет о работе, выполняемой мастером AppWizard(рис. 20.23).

Щелкните на кнопке **OK**, чтобы запустить процесс создания программного кода. На рис. 20.24 показан список файлов проекта, созданных мастером автоматически. Все эти файлы сохраняются в папке, которая была выбрана для проекта в окне New.

Класс CEditview реализует базовые функции текстового редактора. Созданный шаблон программы позволит вывести документ на печать, осуществить поиск и замену текста, вырезать, скопировать, вставить и удалить текстовые блоки, а также выполнить отмену предыдущей команды. Этот класс по умолчанию обрабатывает сообщения идентификаторами ID_FILE_PRINT, id_edit_cut, ID_EDIT_COPY, ID_EDIT_PASTE, ID_EDIT_CLEAR, ID_EDIT_UNDO, ID_EDIT_SELECT_ALL, ID_EDIT_FIND, ID_EDIT_REPLACE и ID_EDIT_REPEAT.

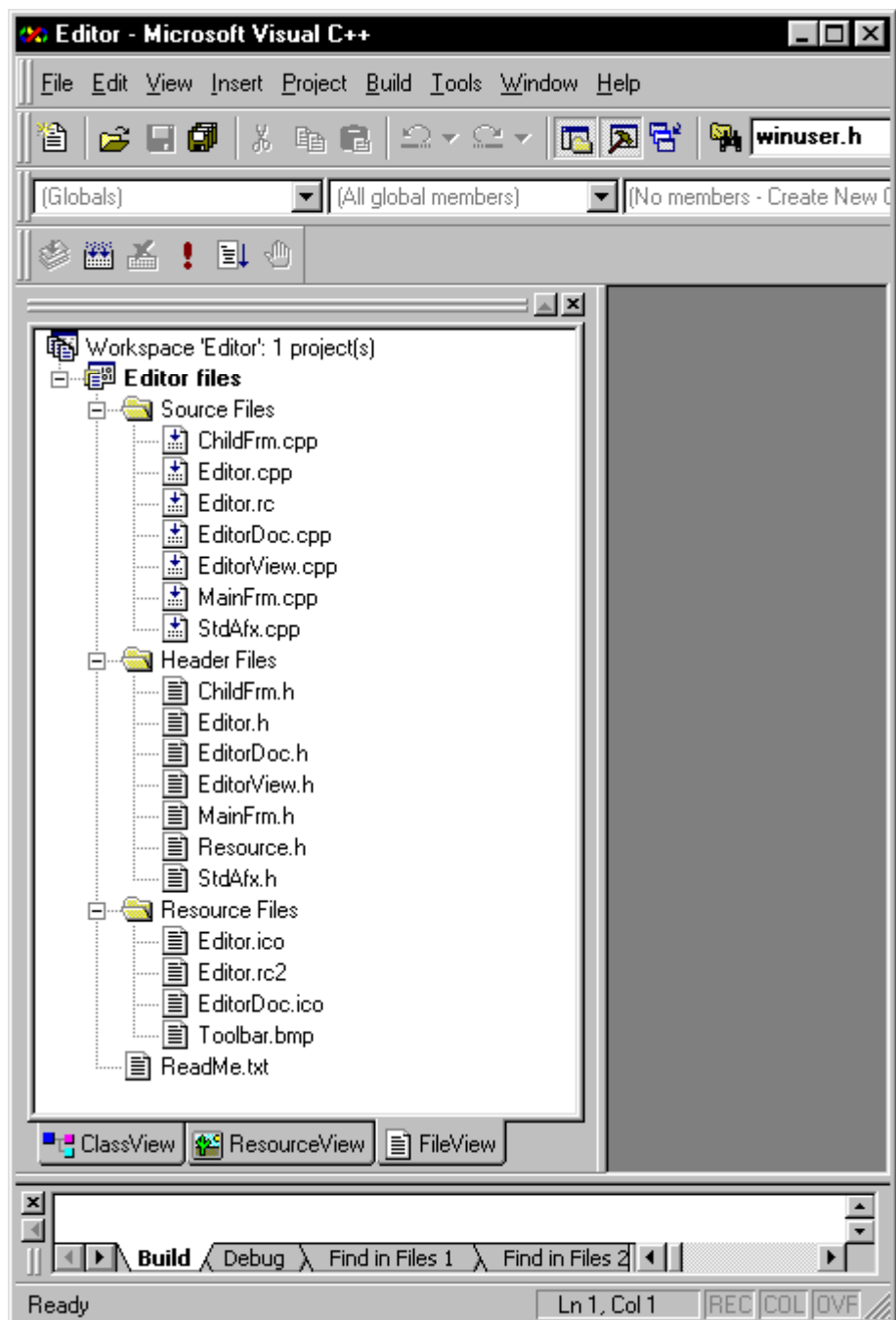


Рис. 20.24. На вкладке **FileView** выведен перечень файлов, которые были автоматически сгенерированы мастером **AppWizard**

Построение приложения

Теперь приложение можно скомпилировать и запустить. Исполняемый файл будет помещен в папку DEBUG. Наше приложение дает возможность открывать и редактировать существующие текстовые файлы, создавать новые файлы и сохранять их на диске.

Давайте проанализируем код программы, сгенерированный мастером AppWizard, и рассмотрим некоторые новые блоки.

Анализ программного кода

Мастер приложений создал для проекта Editor пять исходных файлов: EDITOR.CPP, MAINFRM.CPP, EDITORDOC.CPP, CHILDFRM.CPP и EDITOR-VIEW.CPP. Каждому из них соответствует свой файл заголовков: EDITOR.H, MAINFRM.H, EDITORDOC.H, CHILDFRM.H и EDITORVIEW.H. Файлы заголовков содержат описания классов, используемых в исходных файлах.

Файл EDITOR.CPP

Файл EDITOR.CPP является основным файлом приложения. Он содержит реализацию класса CEditorApp.

```
// Editor. cpp: определяет работу приложения.
#include "stdafx.h"
#include "Editor. h"
#include "MainFrm.h"
#include "ChildFrm.h"
#include "EditorDoc.h"
#include "EditorView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__ ;
#endif
////////////////////
// CEditorApp
BEGIN_MESSAGE_MAP(CEditorApp, CWinApp)
// { (AFX_MSG_MAP (CEditorApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
//}AFX_MSG_MAP
// Стандартные операции с документами
ON_COMMAND(ID_FILE_NEW, CWinApp : :OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp: :OnFileOpen)
// Стандартная команда задания -установок принтера
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp: :OnFilePrintSetup)
END_MESSAGE_MAP ( )
////////////////////
// Конструктор класса CEditorApp
CEditorApp: : CEditorApp ( )
}
////////////////////
//Единственный объект класса CEditorApp
CEditorApp theApp;
////////////////////
// Инициализация класса CEditorApp
BOOL CEditorApp: : InitInstance ( )
{
    COLORREF clrCtlBk, clrCtlText;
    // Цвет фона диалоговых окон задается синим,
    // а цвет текста – белым
    SetDialogBkColor(clrCtlBk = RGB (0,0, 255),
    clrCtlText = RGB(255, 255, 255));
    AfxEnableControlContainer();
    // Стандартная инициализация
#ifdef _AFXDLL
    EnableSdControls();
    // эта функция вызывается при
    // динамической компоновке MFC
```

```

#else
EnableSdControlsStatic(); // эта функция вызывается при
// статической компоновке MFC
#endif
// Измените раздел реестра, где будут храниться
// параметры программы.
SetRegistryKey(_T ("LocalAppWizard-Generated Applications"));
LoadStdProfileSettings();
// загрузка параметров из INI-файла
// Регистрация шаблонов документов
CMultiDocTemplate* pDocTemplate; pDocTemplate = new CMultiDocTemplate(
IDR_EDITORTYPE,
RUNTIME_CLASS(CEditorDoc),
RUNTIME_CLASS(CChildFrame), // пользовательское
// дочернее MDI-окно
RUNTIME_CLASS(CEditorView)); AddDocTemplate(pDocTemplate);
// Создание основного MDI-окна
CMainFrame* pMainFrame = new CMainFrame;
if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
return FALSE; m_pMainWnd = pMainFrame;
// Анализ командной строки на предмет поиска системных команд,
// DDE-команд или команд открытия файлов.
CCommandLineInfo cmdlInfo;
ParseCommandLine(cmdlInfo);
// Обработка команд, указанных в командной строке
if(!ProcessShellCommand(cmdlInfo))
return FALSE;
pMainFrame->ShowWindow(m_nCmdShow); pMainFrame->UpdateWindow();
return TRUE;
}
////////////////////////////////////
// Класс CAboutDlg, управляющий окном About
class CAboutDlg : public CDialog
{
public:
CAboutDlg () ;
// Данные диалогового окна//
//{{AFX_DATA(CAboutDlg) enum { IDD = IDD_ABOUTBOX }; }}AFX_DATA
// Виртуальные функции, сгенерированные мастером ClassWizard
// { {AFX_VIRTUAL (CAboutDlg)
protected:
virtual void DoDataExchange(CDataExchange* pDX);
//{{AFX_VIRTUAL
// Реализацияprotected:
//{{AFX_MSG(CAboutDlg)
// Обработчики сообщений отсутствуют
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};
CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD) {
//{{AFX_DATA_INIT (CAboutDlg)
//}}AFX_DATA_INIT }
void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{

```

```

CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CAboutDlg)
/ / } } AFX_DATA_MAP (
BEGIN_MESSAGE_MAP (CAboutDlg, CDialog)
// ( {AFX_MSG_MAP (CAboutDlg)
// Обработчики сообщений отсутствуют
//}}AFX_MSG_MAPEND_MESSAGE_MAP( )
// Функция, управляющая выводом окна About
void CEditorApp : : OnAppAbout ( )
{
CAboutDlg aboutDlg;
aboutDlg . DoModal ( ) ;
}
////////////////////////////////////
// Другие функции класса CEditorApp

```

Первая схема сообщений принадлежит классу CEditorApp. В ней сообщения с идентификаторами id_app_about, id_file_new, id_file_open и id_file_PRINT_SETUP связываются соответственно с обработчиками OnAppAbout(), CWinApp: :OnFileNew(), CWinApp: :OnFileOpen() и CWinApp: :OnFilePrintSetup(). В этом файле реализуются конструктор класса CEditorApp, а также его методы InitInstance() и OnAppAbout().

Единственное изменение, внесенное нами в сгенерированный текст программы, — это переустановка цвета фона и цвета текста всех диалоговых окон, создаваемых программой.

Данное приложение, в отличие от рассматриваемого в предыдущем примере, позволяет работать сразу с несколькими документами, на что указывает следующий блок программы:

```

// Регистрация шаблонов документов
CMultiDocTemplate* pDocTemplate;
pDocTemplate = new CMultiDocTemplate (
IDR_EDITORTYPE,
RUNTIME_CLASS (CEditorDoc) ,
RUNTIME_CLASS (CChildFrame) , // пользовательское
// дочернее MDI-окно
RUNTIME_CLASS (CEditorView) ) ; AddDocTemplate (pDocTemplate) ;

```

Диалоговое окно **About** принадлежит классу CAboutDlg, являющемуся, как и в предыдущем примере, потомком класса CDialog. У него имеется схема сообщений, конструктор и метод DoDataExchange().

Файл MAINFRM.CPP

Файл MAINFRM.CPP содержит реализацию класса CMainFrame, который порождается от класса CFrameWnd и управляет всеми дочерними MDI-окнами.

```

// MainFrm.cpp: реализация класса CMainFrame
//
#include "stdafx.h" #include "Editor, h"
#include "MainFrm.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__ ;
#endif
////////////////////////////////////
// CMainFrame
IMPLEMENT_DYNAMIC (CMainFrame, CMDIFrameWnd)
BEGIN_MESSAGE_MAP (CMainFrame, CMDIFrameWnd)
// { (AFX_MSG_MAP (CMainFrame)

```

```

ON_WM_CREATE ( )
//}}AFX_MSG_MAP END_MESSAGE_MAP ( )
static UINT indicators[] =
{
    ID_SEPARATOR,          // поля строки состояния
    ID_INDICATOR_CAPS , ID_INDICATOR_NUM , ID_INDICATOR_SCROLL ,
};
////////////////////////////////////
// Конструктор и деструктор класса CMainFrame
CMainFrame::CMainFrame()
{
}
CMainFrame::~CMainFrame ()
{
}
1
int CMainFrame::OnCreate (LPCREATESTRUCT lpCreateStruct) {
if (CMDIFrameWnd::OnCreate (lpCreateStruct) == -1) return -1;
if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_JFLYBY |
CBRS_SIZE_DYNAMIC) || ! m_wndToolBar . LoadToolBar { IDR_MAINFRAME} )
<
    TRACED ("Failed to create toolbarXn") ;
    return-1;          //не удалось создать панель инструментов
}
if (!m_wndStatusBar.Create(this) ||
!m_wndStatusBar.SetIndicators(indicators,
sizeof(indicators)/sizeof(UINT))) {
    TRACED("Failed to create status bar\n"); return -1;          //не удалось
создать строку состояния
m_wndToolBar . EnableDocking (CBRS_ALIGN_ANY) ;
EnableDocking(CBRS_ALIGN_ANY) ; DockControlBar (Sm wndToolBar) ;
return 0; }
BOOL CMainFrame::PreCreateWindow (CREATESTRUCT cs) (
if(! CMDIFrameWnd::PreCreateWindow (cs)) return FALSE;
return TRUE;
}
////////////////////////////////////
// Диагностика класса CMainFrame
#ifdef __DEBUG
void CMainFrame::AssertValid() const
{
    CMDIFrameWnd::AssertValid();
}
void CMainFrame::Dump(CDumpContext &dc) const
{
    CMDIFrameWnd::Dump(dc); }
#endif // __DEBUG
////////////////////////////////////
// Обработчики сообщений класса CMainFrame

```

Заметьте, что в схему сообщений добавлен обработчик сообщений `wm_create`. Обратите также внимание на этот небольшой фрагмент:

```

static UINT indicators[] =
ID_SEPARATOR,          // поля строки состояния
ID_INDICATOR_CAPS,
ID_INDICATOR_NUM,

```

ID_INDICATOR_SCRL,

Как вы помните, в одном из окон мастера AppWizard была задана возможность добавления в окно приложения строки состояния. В массиве indicators перечислены идентификаторы различных полей строки состояния, которые служат индикаторами нажатия клавиш [CapsLock], [NumLock] и [ScrollLock].

Файл EDITORDOC.CPP

Файл EDITORDOC.CPP содержит реализацию класса CEditorDoc, который управляет работой с конкретным документом, а также обеспечивает загрузку и сохранение данных документа.

```
// EditorDoc.cpp: реализация класса CEditorDoc
#include "stdafx.h"
#include "Editor.h"
#include "EditorDoc.h"
#ifdef _DEBUG Idefine new DEBDG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__ ;
#endif
////////////////////////////////////
//CEditorDoc
IMPLEMENT_DYNCREATE (CEditorDoc, CDocument)
BEGIN_MESSAGE_MAP (CEditorDoc, CDocument)
// { (AFX_MSG_MAP (CEditorDoc)
//) } AFX_MSG_MAP END_MESSAGE_MAP ( )
////////////////////////////////////
// Конструктор и деструктор класса CEditorDoc
CEditorDoc: : CEditorDoc ()
{
}
CEditorDoc::~CEditorDoc ()
{
}
BOOL CEditorDoc: : OnNewDocument () {
if ( ! CDocument: : OnNewDocument ( ) ) return FALSE;
return TRUE;
}
////////////////////////////////////
// Сериализация класса CEditorDoc
void CEditorDoc::Serialize(CArchive &ar) {
((CEditView*)m_viewList.GetHead())->SerializeRaw(ar);
}
////////////////////////////////////
// Диагностика класса CEditorDoc
#ifdef _DEBUG
void CEditorDoc: :AssertValid()const {
CDocument: :AssertValid() ; }
void CEditorDoc: : Dump (CDumpContext &dc) const {
CDocument: : Dump (dc); } #endif //_DEBOG
////////////////////////////////////
// Другие функции класса CEditorDoc
```

Строка функции Serialize(), выделенная полужирным шрифтом, поддерживает работу команд-меню FUE, обеспечивающих создание, открытие и сохранение файлов.

Файл EDITORV1EW.CPP

Файл EDITORVIEW.CPP содержит реализацию класса CEditorView, который порождается от класса CEditView и управляет отображением документа.

```
// EditorView.cpp: реализация класса CEditorView//
# include "stdafx.h"
#include . "Editor .h"
#include "EditorDoc.h" #include "EditorView.h"
#ifdef _DEBUG
#define new DEBOG_NEW
#undef THIS_FILE
static char THIS_FILE[] = _FILE_ ;
#endif
////////////////////////////////////
// CEditorView
IMPLEMENT_DYNCREATE (CEditorView, CView)
BEGIN_MESSAGE_MAP (CEditorView, CView) //{AFX_MSG_MAP (CEditorView)
//}AFX_MSG_MAP // Стандартные команды печати
ON_COMMAND(ID_FILE_PRINT, CView: : OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView: :OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView: :OnFilePrintPreview)
END_MESSAGE_MAP ( )
////////////////////////////////////
// Конструктор и деструктор класса CEditorView
CEditorView::CEditorView()
{ }
CEditorView::~CEditorView()
{
}
BOOL CEditorView: :PreCreateWindow(CREATESTRUCT Ses) {
BOOL bPreCreated = CEditView: :PreCreateWindow (cs);
cs.style &= ~ (ES_AUTOHSCROLL I WS_HSCROLL) ; // разрешен перенос слов
return bPreCreated;
}
////////////////////////////////////
// Отображение документа
void CEditorView::OnDraw(CDC* pDC)
{
CEditorDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc) ; }
////////////////////////////////////
// Печать документа
BOOL CEditorView: :OnPreparePrinting (CPrintInfo* pInfo) {
// стандартные действия по подготовке к печати
return CEditView: :OnPreparePrinting (pInfo);
}
void CEditorView: :OnBeginPrinting (CDC* pDC, CPrintInfo* pInfo) {
CEditView: :OnBeginPrinting (pDC, pInfo); }
void CEditorView: :OnEndPrinting (CDC* pDC, CPrintInfo* pInfo) {
CEditView: :OnEndPrinting (pDC, pInfo);
}
////////////////////////////////////
// Диагностика класса CEditorView
#ifdef _DEBUG
void CEditorView: :AssertValid() const
{

```



```

CView::AssertValid() ; }
void CEditorView: : Dump (CDumpContext &dc) const
{
CView::Dump(dc); }
CEditorDoc* CEditorView::GetDocument() // отладочная версия{
ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CEditorDoc)));
return (CEditorDoc*)m_pDocument; }
#endif // _DEBUG
////////////////////////////////////
//Обработчики сообщений класса CEditorView
void CEditorView::OnRButtonDown(DINT nFlags, CPoint point)
{
char szTimeStr[20];
CTime tm = CTime::GetCurrentTime();
sprintf(szTimeStr, "It's now %02d:%02d:%02d", tm>. GetHour () , tm.
GetMinute () , tm.GetSecondO );
MessageBox(szTimeStr, "Is it time to quit yet?", MB_OK);
CEditView::OnRButtonDown(nFlags, point); }

```

Анализируя схему сообщений, вы заметите, что в ней содержится макрос `on_wm_rbuttondown`, который был добавлен мастером `ClassWizard`, и обработчики

Сообщений `ID_FILE_PRINT`, `ID_FILE_PRINT_DIRECT`, `ID_FILE_PREVIEW`, используемые совместно с классом `CEditorView`. Конструктор и деструктор класса остались пустыми. Печать документов реализуется с помощью функций `onPreparePrinting()`, `OnBeginPrinting()` и `OnEndPrinting()`.

В конце листинга помещен текст функции `OnRButtonDown()`, осуществляющей обработку сообщений `wm_rbuttondown`. Процесс добавления этой функции с помощью мастера `ClassWizard` проиллюстрирован на рис. 20.25.

Выполните двойной щелчок на элементе `OnRButtonDown` в списке `Memberfunctions`, чтобы перейти непосредственно к разделу файла `EDITORVIEW.CPP`, отвечающему за обработку сообщений. На рис. 20.26 показано место, куда был вставлен новый программный блок.

Теперь, после компиляции приложения, в результате щелчка правой кнопкой мыши в окне текстового редактора отобразится небольшое окошко, содержащее текущее время (рис. 20.27).

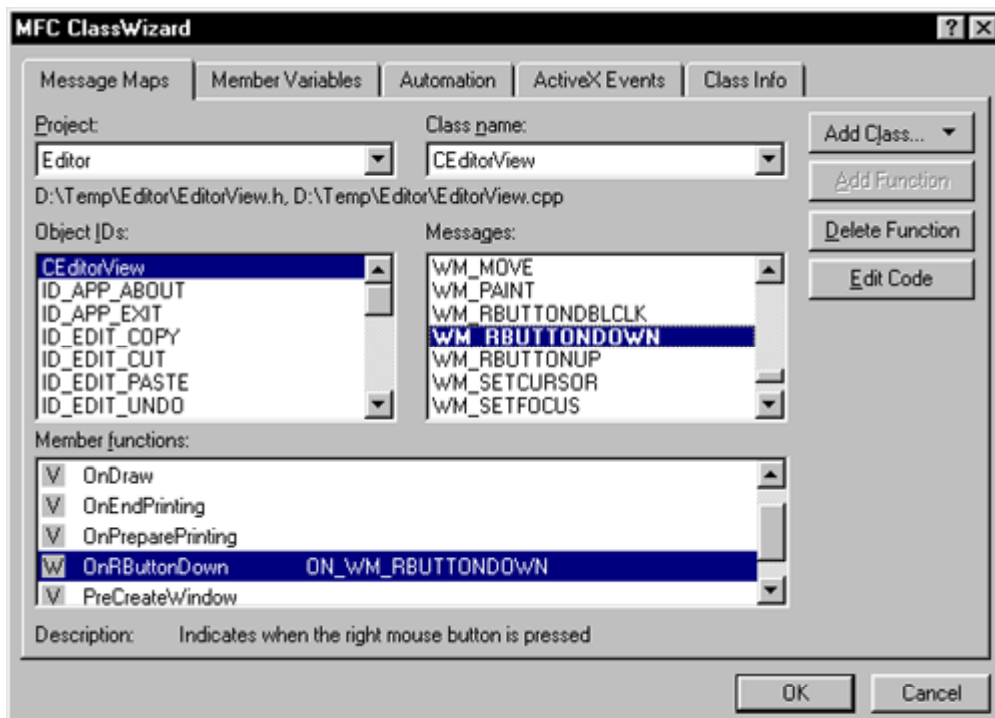


Рис. 20.25. В приложение добавляется обработчик сообщений, связанных с нажатием правой кнопки мыши

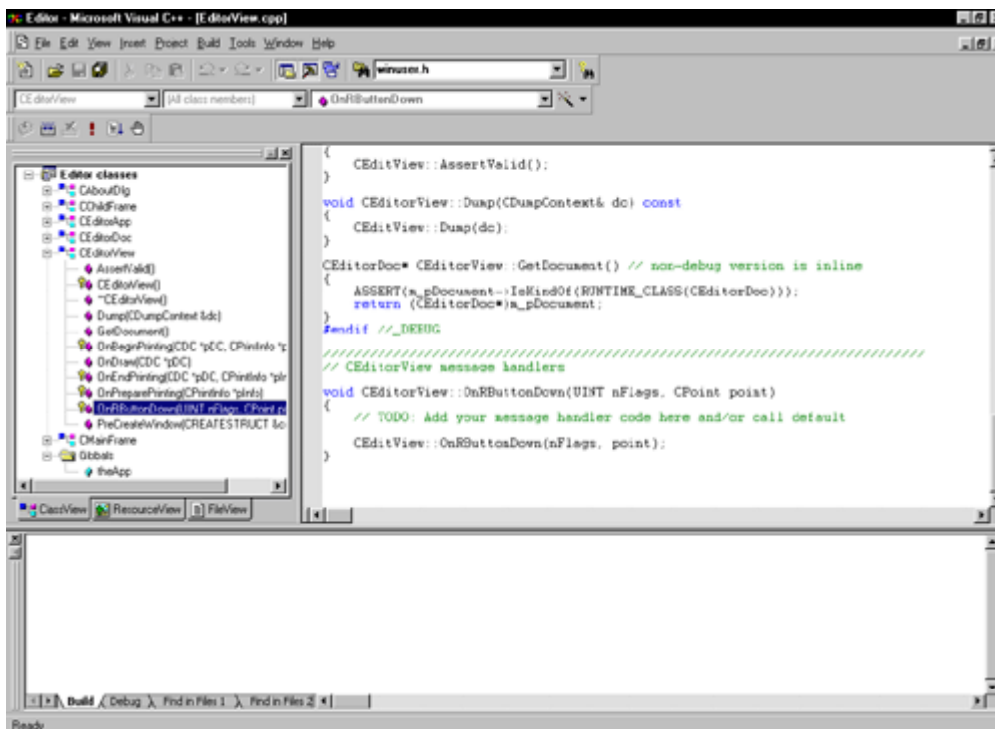


Рис. 20.26. Добавление функции **OnRButtonDown**

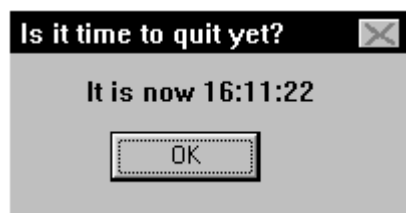


Рис. 20.27. Диалоговое окно, отображающееся при нажатии правой кнопки мыши.

Глава 21. Введение в OLE

- Основные концепции
 - Объекты
 - Структурированные файлы
 - Унифицированная передача данных
 - Внедрение
 - Связывание
- Создание OLE -контейнера
 - Работа с мастером приложений
 - Анализ программного кода
- Проверка работы контейнера

В настоящей главе вы познакомитесь с основными концепциями технологии OLE(ObjectLinkingandEmbedding— связывание и внедрение объектов), которую можно определить как объектно-ориентированный протокол совместного доступа к данным и программному коду из разных процессов и даже из разных компьютеров в пределах локальной сети. OLE позволяет программистам создавать приложения для работы с составными документами, представляющими собой динамические связанные структуры, отдельные части которых могут разрабатываться в различных программах. Составной документ обычно включает в себя главный документ и ряд внедренных или связанных объектов.

Разработку OLE-контейнеров и серверов проще всего вести в среде VisualC++ при помощи специальных мастеров, а также библиотеки MFC . Делать такого рода работу вручную бессмысленно, так как придется писать тысячи строк кода, большая часть которого будет повторяться от программы к программе.

В этой главе мы поговорим о создании OLE-приложений с помощью мастера AppWizard, с которым вы уже познакомились в предыдущей главе. Благодаря этому мастеру программист избавляется от необходимости вводить однотипные программные блоки — они будут добавляться автоматически. Мастер AppWizard позволяет также не заботиться о деталях реализации многочисленных концепций технологии OLE. Вы убедитесь, что применение возможностей OLE-становится достаточно простой задачей.

Основные концепции

Объекты

Процедурные приложения для Windows базируются главным образом на использовании стандартных API-функций. Из-за этого иногда бывает трудно определить язык реализации программы (C или C++), поскольку вся она может состоять только из вызовов стандартных функций!

В главах 18 и 19 мы постепенно перешли от процедурных подходов в программировании к объектно-ориентированной методике. Этот переход стал возможным благодаря использованию библиотеки MFC . Технология OLE открывает новые возможности для объектно-ориентированного программирования.

В основе OLE лежит модель компонентных объектов — COM (ComponentObjectModel), представляющая собой двоичный стандарт, который предназначен для организации взаимодействия между двумя не связанными приложениями. Подобное взаимодействие организуется посредством интерфейсов, которые должны реализовываться объектами. Объекты, подчиняющиеся правилам COM, называются COM-объектами.

Каждый COM-объект имеет уникальный идентификатор класса (CLSID) и создается с помощью функций, содержащихся в специальной фабрике классов (classfactory), связанной с данным

идентификатором. При создании Объекта приложение получает указатель на базовый интерфейс IUnknown данного объекта, и в дальнейшем все функции объекта вызываются через этот указатель. Такая схема позволяет создавать объекты независимо от языка программирования, на котором написано приложение. В обязанности библиотек OLE входит также передача параметров вызова функций и возвращаемых значений через границы процессов.

Структурированные файлы

Данные составных документов записываются на диск в виде структурированных файлов, в которых используются специальные объекты — потоки (stream) и хранилища (storage). Потоки напоминают обычные файлы, а хранилища аналогичны папкам. Структурированные файлы выполняют роль оболочки, скрывающей реальное размещение данных на диске и облегчающей пользователям процесс манипулирования документами.

Унифицированная передача данных

Унифицированная передача данных реализуется посредством объектов данных, инкапсулирующих сами данные. Наличие указателя на объект облегчает подключение к источнику данных различных клиентов. Объект данных, в свою очередь, осуществляет полный контроль за обменом данными между приложениями. Поэтому, с точки зрения программистов, обмен данными, осуществляемый методом drag-and-drop, ничем не отличается от передачи через буфер обмена.

Внедрение

В составных документах часто хранится информация из разных источников, не связанных между собой. Например, текстовый документ Microsoft Word может содержать электронную таблицу Excel и точечный рисунок из программы Paint.

До появления OLE диаграммы и точечные рисунки можно было копировать в текстовый документ лишь через буфер обмена. Как только объект помещался в новый документ, он терял все связи с приложением, в котором был создан. Объект превращался в статическое, "мертвое" изображение. Если со временем возникала необходимость внести в такой объект изменения, пользователю приходилось возвращаться к исходному приложению, вносить изменения в оригинал, копировать новый объект и замещать им старый.

В описываемом случае Word является контейнером, а Excel и Paint — серверами. Иными словами, контейнер содержит объекты, созданные в других приложениях, а сервер — это приложение, являющееся источником самих объектов. В качестве примера мы рассмотрим внедрение в документ Word объекта Paint.

Запустите программу Microsoft Word. Типичное окно Word с введенным текстом показано на рис. 21.1.

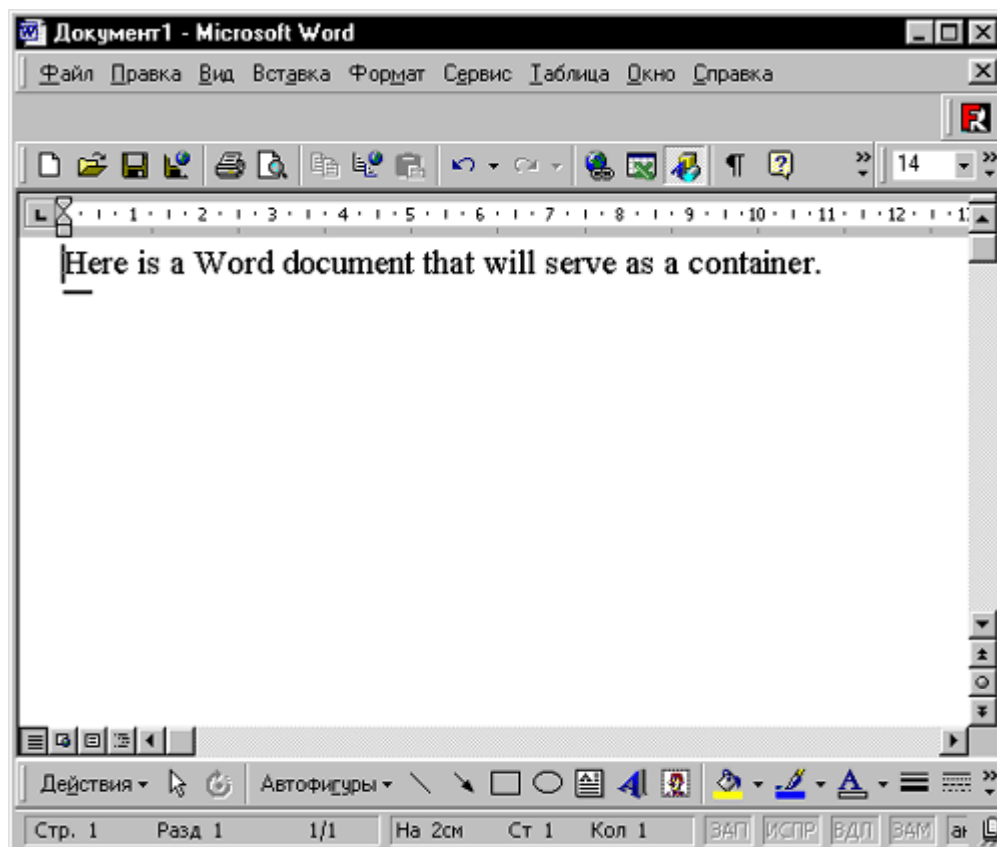


Рис. 21.1. Microsoft Word является приложением-контейнером

Далее в меню **Insert** выберите команду **Object....** Перед вами откроется диалоговое окно вставки объекта (рис. 21.2).

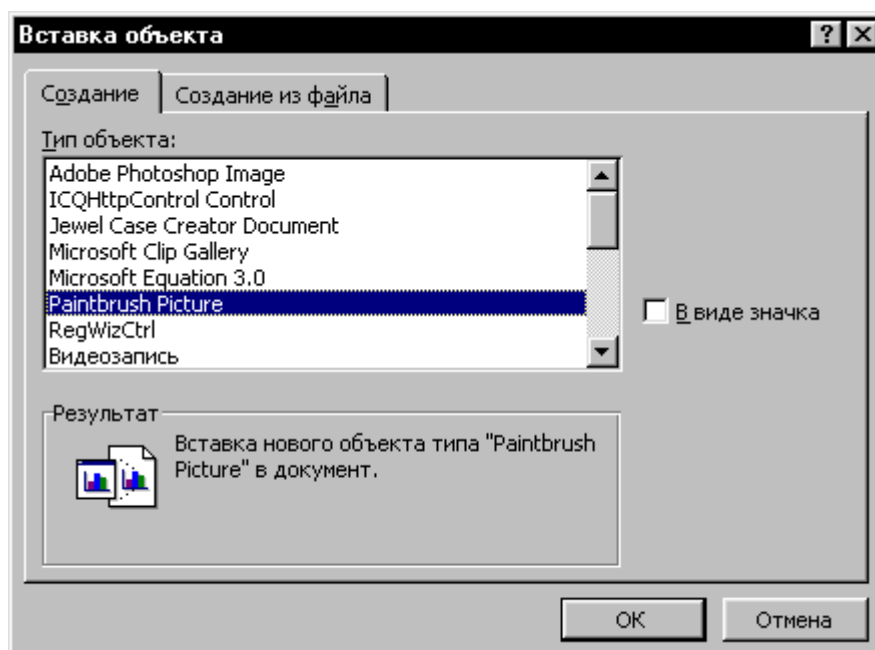


Рис. 21.2. Окно вставки позволяет выбрать тип внедряемого объекта

В списке типов объектов выделите элемент **PaintbrushPicture**. После щелчка на кнопке **OK** будет автоматически запущена программа Paint, область рисования которой разместится

поверх документа Word, а панели инструментов и меню будут интегрированы в окно Word(рис. 21.3). При этом сам Wordавтоматически перейдет в режим разметки страниц.

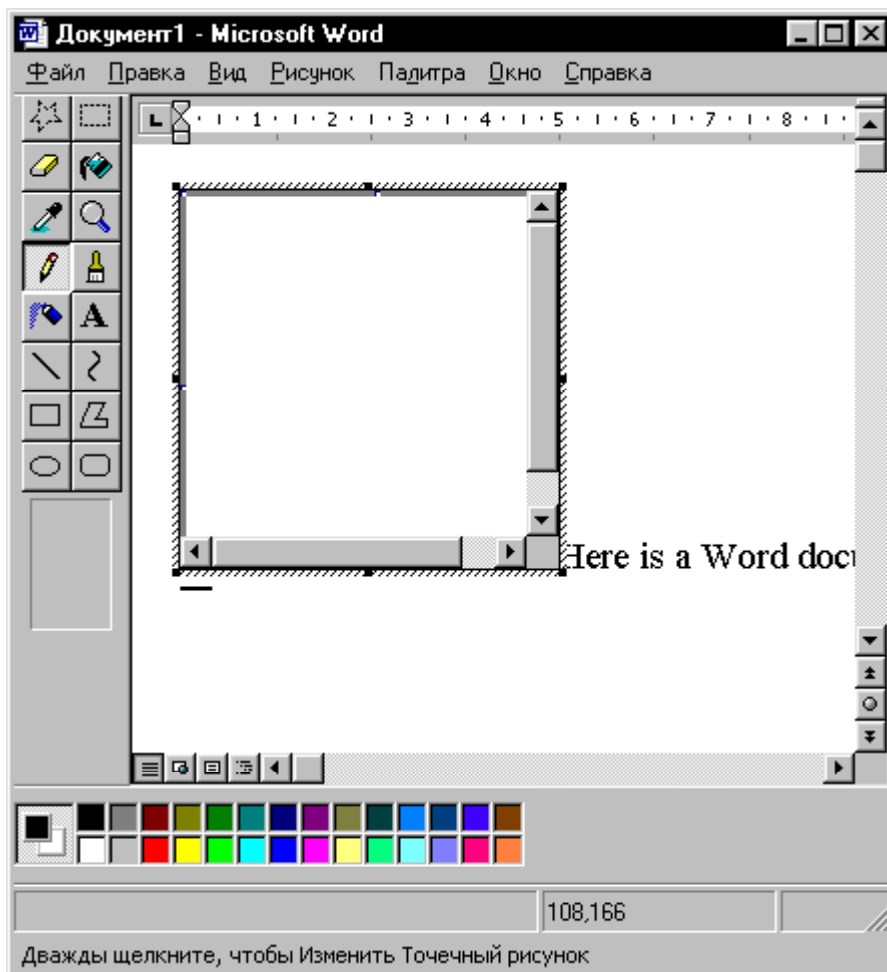


Рис. 21.3. В качестве внедряемого объекта выбран рисунок из приложения Paint

Теперь, используя все средства и возможности программы Paint, можно нарисовать изображение, которое вы хотите добавить в текстовый документ (рис. 21.4). После завершения работы над рисунком выберите в меню **File** команду **Save**.

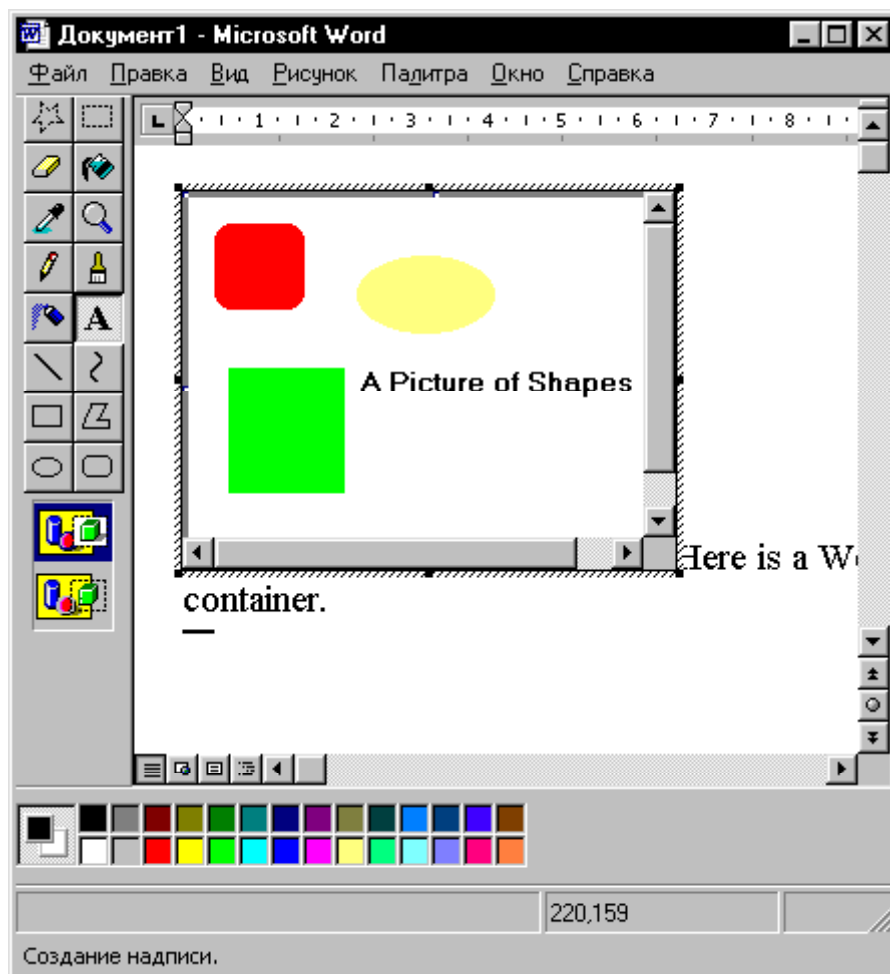


Рис. 21.4. Рисунок, внедряемый в текстовый документ, создается с помощью программы Paint

Окончив работу над рисунком, щелкните мышью на текстовом документе за пределами области рисования, чтобы' возвратиться в Wordи закрыть редактор Paint. На рис. 21.5 показан документ Word, содержащий внедренный объект.

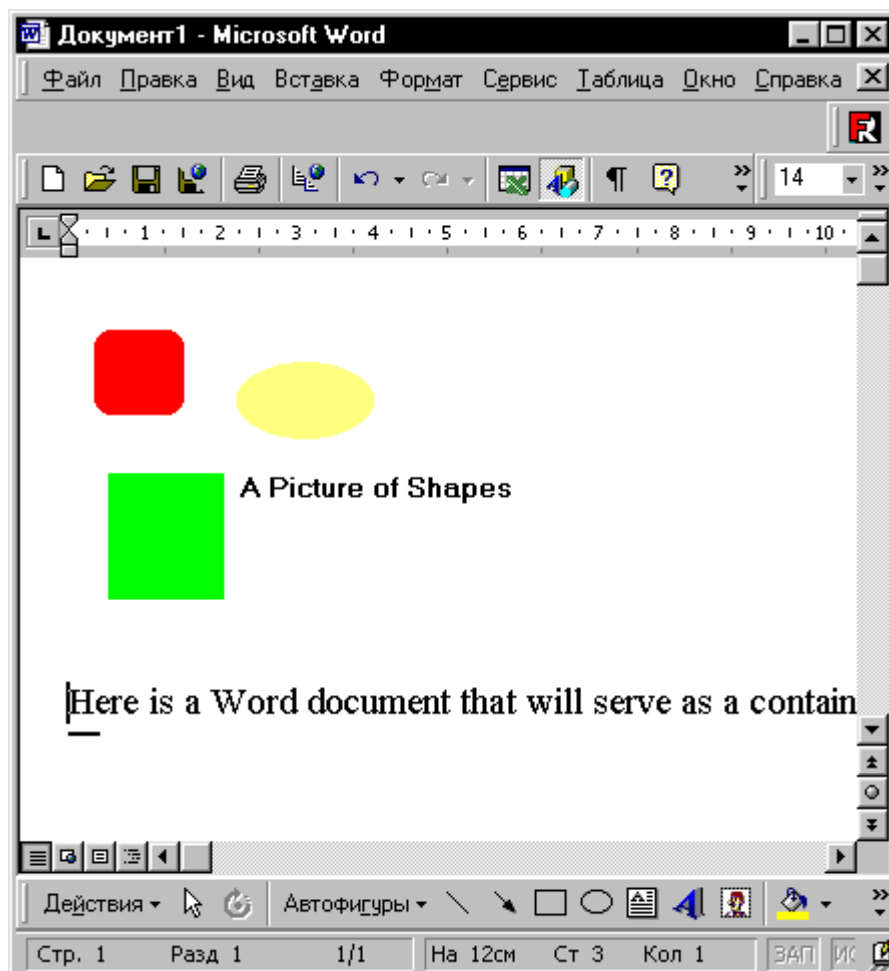


Рис. 21.5. Текстовый документ Word с внедренным рисунком

А теперь самое интересное. Предположим, вы решили, что внедренный объект требует изменений. Выполните на нем двойной щелчок. При этом запустится исходное приложение, в которое будет загружен объект для редактирования (рис. 21.6).

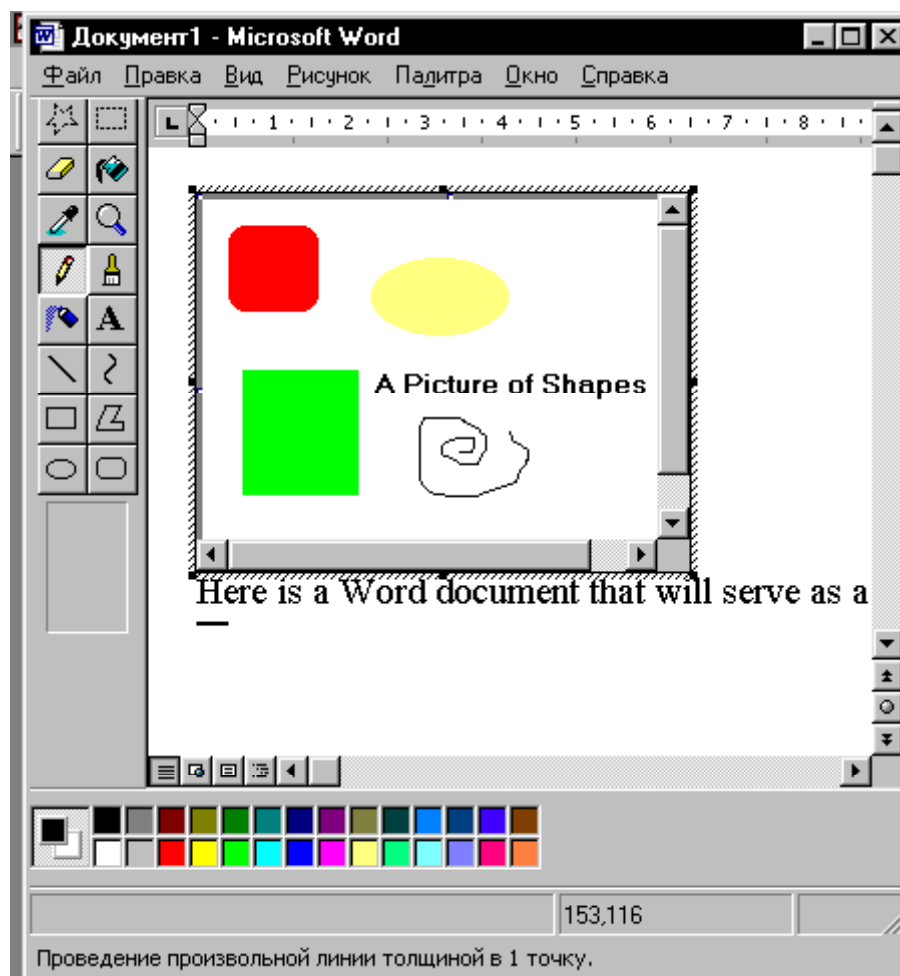


Рис. 21.6. При необходимости отредактировать внедренный объект выполните на нем двойной щелчок

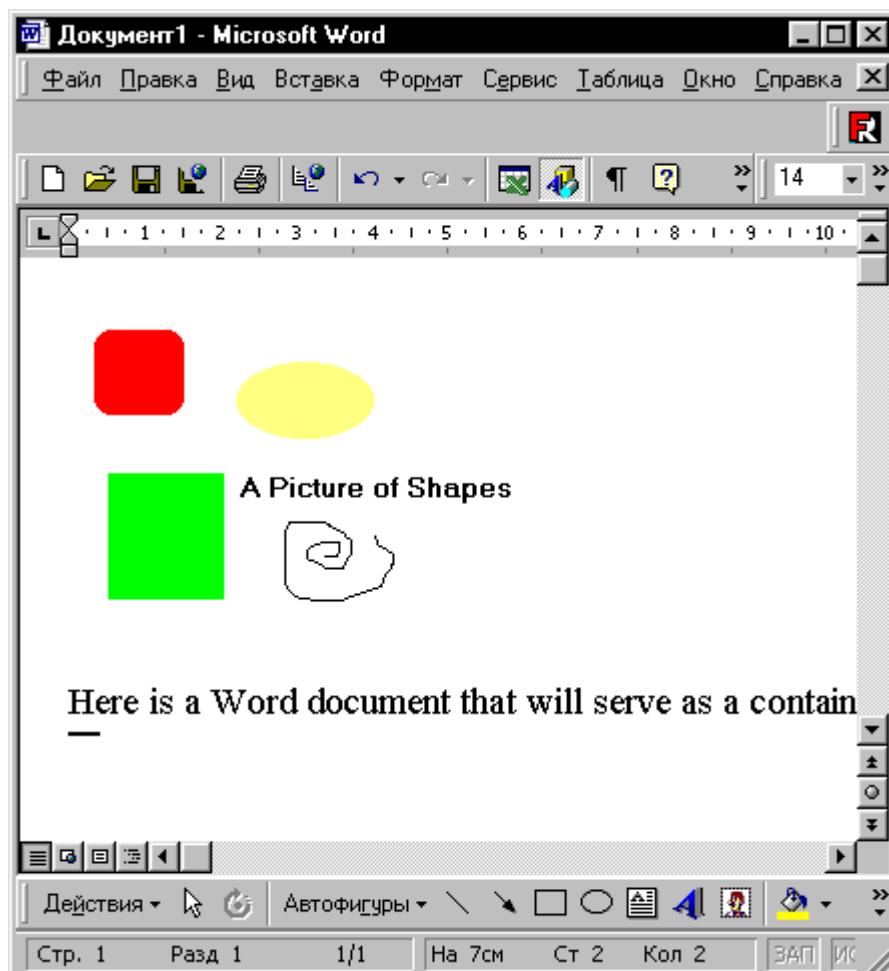


Рис. 21.7. Отредактированный объект

Связывание

Технология OLE поддерживает также динамическое связывание объектов, созданных в разных приложениях. При наличии связи приложения могут одновременно использовать одни и те же объекты. Раньше связывание было довольно неустойчивым механизмом, поскольку связи легко разрывались при перемещении файлов на диске. В настоящий момент в OLE используются псевдонимы (monikers), позволяющие успешно решать многие из существовавших прежде проблем. Псевдонимом называется специальный COM-объект, в котором хранится имя связанного объекта и информация о его местоположении. Именно псевдонимы выполняют задачу поиска объектов, освобождая от этого приложение-контейнер.

Создание OLE-контейнера

Приложение-контейнер Cnt, созданием которого мы сейчас займемся, напоминает программу Graphcs SDI-интерфейсом, созданную нами в предыдущей главе. В данном приложении будут использованы два важных OLE-класса: COleClientItem и COleDocument. Класс COleDocument управляет списком объектов класса COleClientItem. Класс COleClientItem, в свою очередь, управляет внедренными или связанными объектами и поддерживает взаимодействие между контейнером и сервером.

Обратите внимание на тот факт, что код приложения полностью создан мастером AppWizard. Полученный шаблон впоследствии можно расширить дополнительными средствами, написав соответствующие фрагменты самостоятельно или воспользовавшись мастером ClassWizard.

Работа с мастером приложений

Работа с мастером AppWizard достаточно подробно рассматривалась в предыдущей главе. Поэтому сейчас мы сконцентрируем внимание лишь на наиболее важных моментах создания приложения Cnt.

- Чтобы приступить к созданию нового проекта, выберите в окне компилятора Microsoft Visual C++ в меню **File** команду **New**.
- В окне **New** выберите элемент **MFC AppWizard(exe)**, в результате чего будет запущен мастер приложений, работа с которым осуществляется в шесть этапов.
 1. В первом окне установите опцию **Singledocument**.
 2. Во втором окне не задавайте поддержку баз данных.
 3. В третьем окне установите опцию **Container**, указывающую на то, что приложение будет OLE-контейнером. В этом же окне следует включить поддержку элементов управления ActiveX.
 4. В следующем, четвертом, окне необходимо оставить все опции, заданные по умолчанию.
 5. В пятом окне установите опцию **MFC Standard**, опцию включения комментариев в программу и опцию статической компоновки библиотеки MFC.
 6. Наконец, в шестом окне, просмотрите список классов, которые будут созданы автоматически, и щелкните на кнопке **Finish**.
- Мастер приложений отобразит окно с отчетом о сделанных установках. Если все правильно, щелкните на кнопке **OK**, с тем чтобы запустить процесс генерации кода нового приложения.

Теперь осталось только построить исполняемый файл приложения, выбрав для этого в меню **Build** команду **Rebuild All**. В результате в папку **DEBUG** будет добавлен файл **CMT.EXE**.

Анализ программного кода

Приложение включает пять основных исходных файлов, сгенерированных мастером AppWizard: **CNT.CPP**, **MAINFRM.CPP**, **CNTDOC.CPP**, **CNTVIEW.CPP** и **CNTRITEM.CPP**.

Файл **CNT.CPP**

Текст, содержащийся в файле **CNT.CPP**, почти идентичен тексту файлов **GRAPH.CPP** и **EDITOR.CPP** приложений **Графи Editor**, описанных в предыдущей главе. Поэтому рекомендуем вам вернуться к той главе и еще раз прочитать пояснения к указанным файлам.

Листинг файла **CNT.CPP** показан ниже.

```
// Cnt.cpp: определяет работу приложения.
//
#include "stdafx.h"
#include "Cnt.h"
#include "MainFrm.h" #include "CntDoc.h" #include "CntView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = _FILE_;
#endif
////////////////////////////////////
// CCntApp
BEGIN_MESSAGE_MAP {CCntApp, CWinApp} //{AFX_MSG_MAP (CCntApp)
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
// макросы схемы сообщений.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//}}AFX_MSG_MAP
// Стандартные операции с документами
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
```

```

// Стандартная команда задания установок принтера
ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
////////////////////////////////////
// Конструктор класса CCntApp
CCntApp::CCntApp() {
// TODO: здесь добавьте код конструктора.
// Все наиболее важные команды инициализации
// разместите в методе InitInstance.
}////////////////////////////////////
// Единственный объект класса CCntAppCCntApptheApp;
////////////////////////////////////
// Инициализация класса CCntApp
BOOLCCntApp::InitInstance() {
// Инициализация библиотек OLE
if(!AfxOleInit() )
{
AfxMessageBox(IDP_OLE_INIT_FAILED); return FALSE; }
AfxEnableControlContainer ();
// Стандартная инициализация.
// Если вам не нужны используемые здесь возможности
// и вы хотите сократить размер исполняемого файла,
// удалите ненужные команды.
#ifdef _AFXDLL
Enable3dControls();
// эта функция вызывается при
// динамической компоновке MFC
#else
Enable3dControlsStatic();
// эта функция вызывается при
// статической компоновке MFC
#endif
// Измените раздел реестра, где будут храниться
// параметры программы.
SetRegistryKey(_T("Local AppWizard-Generated Applications"));
LoadStdProfileSettings();
// загрузка параметров из INI-файла
// Регистрация шаблонов документов приложения
CSingleDocTemplate* pDocTemplate; pDocTemplate = new
CSingleDocTemplate (
IDR_MAINFRAME,
RUNTIME_CLASS (CCntDoc) ,
RUNTIME_CLASS (CMainFrame) , // основное SDI-окно>>
RUNTIME_CLASS (CCntView) ) ;
pDocTemplate->SetContainerInfo(IDR_CNTR_INPLACE) ;
AddDocTemplate (pDocTemplate) ;
// Анализ командной строки на предмет поиска системных команд, // DDE-
команд или команд открытия файлов. CCommandLineInfo cmdInfo;
ParseCommandLine (cmdInfo) ;
// Обработка команд, указанных в командной строке
if ( ! ProcessShellCommand (cmdInfo) ) return FALSE;
// Отображение окна приложениям
_pMainWnd->ShowWindow (SW_SHOW) ; m_pMainWnd->UpdateWindow() ;
return TRUE;

```

```

}
////////////////////////////////////
// Класс CAboutDlg, управляющий окном About
class CAboutDlg : public CDialog { public:
CAboutDlg ( ) ;
// Данные диалогового окна
//{{AFX_DATA (CAboutDlg) enum { IDD = IDD_ABOUTBOX }; }}AFX_DATA
// Виртуальные функции, сгенерированные мастером ClassWizard
//{{AFX_VIRTUAL (CAboutDlg)
protected:
virtual void DoDataExchange (CDataExchange* pDX) ;
//}}AFX_VIRTUAL
// Реализация
protected:
// { { AFX_MSG (CAboutDlg)
// Обработчики сообщений отсутствуют
//}}AFXMSG
DECLARE_MESSAGE_MAP()
};
CAboutDlg: : CAboutDlg ( ) : CDialog (CAboutDlg: : IDD) {
//{{AFX_DATA_INIT (CAboutDlg)
//}}AFX_DATA_INIT
}
void CAboutDlg::DoDataExchange(CDataExchange* pDX) {
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP (CAboutDlg)
/ / } } AFX_DATA_MAP }
BEGIN_MESSAGE_MAP(CAboutDlg, CDialog) //{{AFX_MSG_MAP (CAboutDlg)
// Обработчики сообщений отсутствуют //}}AFX_MSG_MAPEND_MESSAGE_MAP()
// Функция, управляющая выводом окна About
void CCntApp::OnAppAbout()
{
CAboutDlg aboutDlg;
aboutDlg.DoModal(); }
////////////////////////////////////
// Другие функции класса CCntApp

```

Данный листинг содержит один фрагмент, заслуживающий особого внимания. В OLE используется концепция непосредственного редактирования (in-placeediting). Это означает, что после двойного щелчка на объекте, внедренном в документ контейнера, такой как наш, строка меню и панели инструментов соответствующего OLE-сервера замещают меню и панели инструментов контейнера. Например, если в документ приложения Cnt будет внедрена электронная таблица Excel, то после двойного щелчка на ней строка меню и панели инструментов программы Excel появятся в окне программы Cnt.

Смена меню происходит автоматически и обрабатывается библиотекой MFC с помощью средств OLE. Этот процесс становится возможным благодаря тому, что в приложение добавляется два ресурса меню: idr_mainframeи idr_cntr_inplace (имя последнего идентификатора уникально для данного приложения). По умолчанию отображается меню IDR_MAINFRAME. Но когда внедренный объект активизируется для непосредственного редактирования, загружается меню idr_cntr_in-PLACE.

Файл MAINFRM.CPP

Опять-таки, текст файла MAINFRM.CPP, по сути, идентичен тексту одноименного файла приложения Editor из предыдущей главы. И мы рекомендуем вам еще раз вернуться к той главе и прочесть комментарии к нему. Но для полноты изложения текст файла все же приведем.

```

// MainFrm.cpp: реализация класса CMainFrame
//
#include "stdafx.h"
#include "Cnt.h"
#include "MainFrm.h"  ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
Static char THIS_FILE[] = _FILE_;
#endif

////////////////////////////////////
// CMainFrame
IMPLEMENT_DYNCREATE (CMainFrame, CFrameWnd)
i BEGIN_MESSAGE_MAP (CMainFrame, CFrameWnd)
//{{AFX_MSG_MAP (CMainFrame)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
//      макросы схемы сообщений.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//}}AFX_MSG_MAPEND_MESSAGE_MAP ( )
static UINT indicators[] = {
ID_SEPARATOR,          // поля строки состояния
ID_INDICATOR_^CAPS,
ID_INDICATOR_NUM,
ID_INDICATOR_SCRL,
};
////////////////////////////////////
// Конструктор и деструктор класса CMainFrame
CMainFrame::CMainFrame() {
// TODO: здесь добавьте код конструктора.
}

CMainFrame::~CMainFrame() {
int CMainFrame::OnCreate (LPCREATESTRUCT IpCreateStruct) {
if (CFrameWnd::OnCreate (IpCreateStruct) == -1) return -1;
if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD |
WS_VISIBLE | CBRS_TOP | CBRS_GRIPPER | CBRSJTOOLTIPS | CBRS_FLYBY |
CBRS_SIZE_DYNAMIC) | !m_wndToolBar.LoadToolBar(IDR_MAINFRAME) ) {
TRACEO ("Failed to create toolbar\n") ;
return -1;  // создать панель инструментов не удалось
}
if (!m_wndStatusBar.Create(this) ||
!m_wndStatusBar.SetIndicators(indicators,
sizeof(indicators)/sizeof(UINT))) (
TRACEO ("Failed to create status bar\n"); return -1;  // создать
строку состояния не удалось}
// TODO: удалите следующие три строки, если вы не хотите,
//      чтобы панель инструментов была перемещаемой.
m_wndToolBar . EnableDocking (CBRS_ALIGN_ANY) ;
EnableDocking(CBRS_ALIGN_ANY) ;
DockControlBar (Sm_wndToolBar) ;
return 0;
}

BOOL CMainFrame: : PreCreateWindow (CREATESTRUCT Ses) {
if(ICFrameWnd: : PreCreateWindow (cs)) return FALSE;
// TODO: здесь можно модифицировать класс окна, изменяя поля структуры
cs.

```

```

return TRUE; }
////////////////////////////////////
// Диагностика класса CMainFrame
#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
CFrameWnd::AssertValid();
}
void CMainFrame::Dump(CDumpContext Sdc) const
{
CFrameWnd::Dump(dc); }
#endif // _DEBUG
////////////////////////////////////
// Обработчики сообщений класса CMainFrame

```

Файл CNTDOC.CPP

Файл CNTDOC.CPP содержит ряд дополнительных фрагментов, с которыми ранее мы не встречались.

```

// CntDoc.cpp: реализация класса CCntDoc//
#include "stdafx.h" #include "Cnt.h"
# include "CntDoc.h"
#include "CntItem.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__ ;
#endif
////////////////////////////////////
// CCntDoc
IMPLEMENT_DYNCREATE (CCntDoc, CDocument)
BEGIN_MESSAGE_MAP (CCntDoc, CDocument) //{AFX_MSG_MAP (CCntDoc)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
// макросы схемы сообщений.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//}AFX_MSG_MAP
// Используется стандартная реализация OLE-контейнера
ON_UPDATE_COMMAND_UI ( ID_EDIT_PASTE,
CDocument::OnUpdatePasteMenu) ON_UPDATE_COMMAND_UI (
ID_EDIT_PASTE_LINK ,
CDocument::OnUpdatePasteLinkMenu) ON_UPDATE_COMMAND_UI (
ID_OLE_EDIT_CONVERT ,
CDocument::OnUpdateObjectVerbMenu) ON_COMMAND
(ID_OLE_EDIT_CONVERT,
CDocument::OnEditConvert) ON_UPDATE_COMMAND_UI (ID_OLE_EDIT_LINKS
,
CDocument::OnUpdateEditLinksMenu)
ON_COMMAND (ID_OLE_EDIT_LINKS ,
CDocument::OnEditLinks)
ON_UPDATE_COMMAND_UI_RANGE ( ID_OLE_VERB_FIRST ,
ID_OLE_VERB_LAST,
CDocument::OnUpdateObjectVerbMenu) END_MESSAGE_MAP ()
////////////////////////////////////
// Конструктор и деструктор класса CCntDoc
CCntDoc::CCntDoc() {

```



```

* // включается поддержка составных файлов EnableCompoundFile ( ) ;
// TODO: здесь добавьте код конструктора.
}
CCntDoc::CCntDoc()
{
}
BOOL CCntDoc::OnNewDocument( )
{
if ( ! COleDocument : : OnNewDocument ( ) ) return FALSE;
// TODO: здесь добавьте код повторной инициализации
// (специфика SDI-приложений)
return TRUE;
}
/////////////////////////////////////////////////////////////////
// Сериализация класса CCntDoc
void CCntDoc::Serialize(CArchive sar)          {
if (ar. IsStoringO )
{
// TODO: здесь добавьте код сохранения.
}
else.
// TODO: здесь добавьте код загрузки.
}
// Вызов функции базового класса COleDocument обеспечивает
// сериализацию объектов COleClientItem, содержащихся
// в документе контейнера.
COleDocument : : Serialize (ar ) ;
}
/////////////////////////////////////////////////////////////////
// Диагностика класса CCntDoc
#ifdef _DEBUG
void CCntDoc::AssertValid() const
{
COleDocument::AssertValid(); }
void CCntDoc::Dump(CDumpContext &dc) const {
COleDocument::Dump(dc) ; } #endif // _DEBUG
/////////////////////////////////////////////////////////////////
// Другие функции класса CCntDoc

```

Наиболее важное отличие кода рассматриваемого файла от уже знакомых нам состоит в расширении схемы сообщений. В программу добавлены обработчики сообщений, реализующие работу стандартного контейнера.

Обратите внимание на функцию EnableCompoundFile(), вызываемую в конструкторе. Эта функция осуществляет поддержку составных файлов, что позволяет сохранять документы с вложенными объектами в специальном структурированном формате.

Файл CNTVIEW.CPP

Файл CNTVIEW.CPP также имеет ряд существенных особенностей по сравнению

с соответствующим файлом приложения Graph.

```

// CntView.cpp : реализация класса CCntView
//
#include "stdafx.h"
#include "Cnt.h"

```

```

#include "CntDoc.h"
#include "CntCtrlItem.h"
#include "CntView.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = _FILE_;
#endif
////////////////////////////////////
// CCntView
IMPLEMENT_DYNCREATE(CCntView, CView)
BEGIN_MESSAGE_MAP(CCntView, CView) //{AFX_MSG_MAP(CCntView)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
//      макросы схемы сообщений.
//ПЕРЕДАКТИРУЙТЕ то, что здесь находится. ON_WM_DESTROY()
ON_WM_SETFOCUS() ON_WM_SIZE()
ON_COMMAND(ID_OLE_INSERT_NEW, OnInsertObject)
ON_COMMAND(ID_CANCEL_EDIT_CNTR, OnCancelEditCntr) //{AFX_MSG_MAP //
Стандартныекомандыпечати
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP ()
////////////////////////////////////
// Конструктор и деструктор класса CCntView
CCntView::CCntView() {
m_pSelection = NULL;
// TODO: здесь добавьте код конструктора.
}
CCntView::~CCntView()
{
}
BOOL CCntView::PreCreateWindow(CREATESTRUCT Ses) {
// TODO: здесь можно модифицировать класс окна,
//      изменяя поля структуры сз.
return CView::PreCreateWindow(cs); }
////////////////////////////////////
// Отображение документа
void CCntView::OnDraw(CDC* pDC) {
CCntDoc* pDoc = GetDocument () ;
ASSERT_VALID(pDoc);
// TODO: здесь добавьте код для отображения собственных данных.
// TODO: должны отображаться все OLE-объекты,
//      содержащиеся в документе.
// Выделенный элемент может быть нарисован в произвольном месте.
// Этот код следует удалить в том случае, если вы вводите
// собственный код рисования. Указанные ниже координаты
// в точности соответствуют координатам, возвращаемым
// объектом CCntCtrlItem, что создает эффект непосредственного
// редактирования.
// TODO: удалите следующий код, если реализуете собственный // код
рисования.
if (m_pSelection == NULL)
{
POSITION pos = pDoc->GetStartPosition() ;

```

```

m_pSelection = (CCntCntrlItem*) pDoc->XSetNextClientItem(pos); > if
(m_pSelection != NULL)
m_pSelection->Draw(pDC, CRect(10, 10, 210, 210));
}
void CCntView::OnInitialUpdate() {
CView::OnInitialUpdate();
// TODO: удалите следующий код, если реализуете собственный
// код инициализации.
m_pSelection = NULL;
// инициализация переменной, содержащей
// указатель на выделенный объект
}
////////////////////////////////////
// Печать документа
BOOL CCntView::OnPreparePrinting (CPrintInfo* pInfo) {
// стандартные действия по подготовке к печати
return DoPreparePrinting (pInfo) ;
}
void CCntView::OnBeginPrinting (CDC* /*pDC*/, CPrintInfo* /*pInfo*/) {
// TODO: добавьте код дополнительной инициализации перед печатью. }
void CCntView::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/) ""
{
// TODO: добавьте код очистки после печати.
}
void CCntView::OnDestroy()
{
// Деактивизировать объект при удалении; это важно,
// если используется режим разделения окна просмотра.
CView::OnDestroy(); COleClientItem* pActiveItem = GetDocument()->
GetInPlaceActiveItem(this); if (pActiveItem != NULL && pActiveItem->
GetActiveView() == this) {
pActiveItem->Deactivate0; ASSERT(GetDocument()->
GetInPlaceActiveItem(this) == NULL);
}
}
////////////////////////////////////
// Поддержка OLE-клиентов
BOOL CCntView::IsSelected(const COleClientItem* pItem) const
{
// Представленная ниже реализация подходит, если выделенными
// являются только объекты CCntCntrlItem. В противном случае
// данный код следует заменить.
// TODO: реализуйте функцию, которая проверяет тип выделенного
// объекта.
return pItem == m_pSelection; }
void CCntView::OnInsertObject() {
// Вызов стандартного диалогового окна InsertObject
// для получения информации о новом объекте CCntCntrlItem.
COleInsertDialog dlg;
if (dlg.DoModal() != IDOK) return;
BeginWaitCursor() ;
CCntCntrlItem* pItem = NULL;
TRY
{

```

```

// Создаем новый объект, связанный с этим документом.
CCntDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);
pltem = new CCntCntrlItem(pDoc);
ASSERT_VALID(pltem);
// Инициализация объекта на основании данных,
// полученных из диалогового окна.
if (!dig.CreateItem(pltem))
AfxThrowMemoryException();
// подойдет исключение любого типа
ASSERT_VALID(pltem);
// Если объект был выбран из списка классов, а не загружен
// из файла, запускаем сервер для редактирования объекта,
if (dlg.GetSelectionType() ==
COleInsertDialog::createNewItem) pltem->DoVerb(OLEIVERB_SHOW, this);
ASSERT_VALID(pltem);
// Последний введенный объект выделяется.
// TODO: введите код, соответствующий требованиям вашего приложения.
m_pSelection = pltem;
// Указатель устанавливается на
// последний введенный объект
pDoc->UpdateAllViews (NULL) ;
CATCH(CException, e) if (pltem != NULL)
ASSERT_VALID(pltem) ; pltem->Delete() ;
AfxMessageBox(IDP_FAILED_TO_CREATE); END_CATCH
EndWaitCursor(); >
// Следующий обработчик позволяет с помощью клавиатуры
// прерывать сеанс непосредственного редактирования.
// Иницируется это контейнером, а не сервером,
void CCntView::OnCancelEditCntrl()
// Редактируемый объект закрывается.
COleClientItem* pActiveItem= GetDocument()->
GetInPlaceActiveItem(this); if (pActiveItem != NULL)
pActiveItem->Close();
ASSERT(GetDocument()->GetInPlaceActiveItem(this) == NULL);
}
// Обработчики OnSetFocus и OnSize требуются контейнеру
// в случае непосредственного редактирования объекта.
void CCntView::OnSetFocus (CWnd* pOldWnd)
{
COleClientItem* pActiveItem = GetDocument ( ) ->
GetInPlaceActiveItem(this) ; if (pActiveItem != NULL &&
pActiveItem->GetItemState ( ) ==
COleClientItem::activeUIState) {
// Фокус необходимо установить на объект, если он находится
// в той же области просмотра.
CWnd* pWnd = pActiveItem->GetInPlaceWindow ( );
if (pWnd != NULL)
{
pWnd->SetFocus ( ) ; // метод SetFocus базового класса не вызывается
return;
}
}
CView::OnSetFocus(pOldWnd); }
void CCntView::OnSize (UINT nType, int ex, int cy)

```

```

{
    . '
CView::OnSize (nType, ex, cy) ;
COleClientItem* pActiveItem = GetDocument ()->
GetInPlaceActiveItem(this) ; if (pActiveItem != NULL)
pActiveItem->SetItemRects () ;
}
////////////////////////////////////
// Диагностика класса CCntView
#ifdef _DEBUG
void CCntView::AssertValid() const
{
CView::AssertValid(); }
void CCntView::Dump(CDumpContext Sdc) const
CView::Dump(dc); }
CCntDoc* CCntView::GetDocument() // отладочная версия
ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CCntDoc))); return
(CCntDoc*)m_pDocument;
#endif // _DEBUG
////////////////////////////////////
// Обработчики сообщений класса CCntView

```

Чтобы приложение могло отображать внедренные объекты, были внесены изменения в функции OnDraw(). Обратите внимание на такой фрагмент:

```

if (m_pSelection !=NULL)
m_pSelection->Draw(pDC, CRectdO, 10,210, 210));
}

```

По умолчанию объект размещается в окне приложения в заранее заданной области, которая определяется конструктором CRect(). Указанные координаты можно изменить вручную.

В файл были также добавлены функции OnInitialUpdate(), IsSelectedO, OnInsertObject(), OnCancelEditCtr(), OnSetFocusOИ OnSize(). В частности, функция OnInsertObject() вызывает стандартное диалоговое окно класса COleInsertDialog, предназначенное для вставки объектов.

Файл CNTRITEM.CPP

Файл CNTRITEM.CPP, листинг которого приведен ниже, содержит реализацию класса CCntCntrlItem.

```

// CntrlItem.cpp: реализация класса CCntCntrlItem
//
#include "stdafx.h"
#include "Cnt.h"
#include "CntDoc.h"
#include "CntView.h"
#include "CntrlItem.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__ ;
#endif
////////////////////////////////////
// CCntCntrlItem
IMPLEMENT_SERIAL (CCntCntrlItem, COleClientItem, 0)
CCntCntrlItem::CCntCntrlItem (CCntDoc* pContainer)
: COleClientItem (pContainer) {
// TODO: здесь добавьте код конструктора.
CCntCntrlItem::~~CCntCntrlItem () {

```

```

// TODO: здесь добавьте код очистки.
}
void CCntCntrlItem::OnChange(OLE_NOTIFICATION nCode, DWORD dwParam)
ASSERT_VALID(this); ColeClientItem::OnChange(nCode, dwParam);
// Редактируемому объекту посылается уведомление OnChange,
// свидетельствующее об изменении его состояния или внешнего вида.
// TODO: обозначьте рабочую область объекта как недействительную
// путем вызова функции UpdateAllViews.
GetDocument()->UpdateAllViews(NULL);
BOOL CCntCntrlItem::OnChangeItemPosition(const CRect SrectPos)
ASSERT_VALID(this);
// В процессе непосредственного редактирования данная функция
// вызывается сервером для изменения позиции окна редактирования.
// По умолчанию вызывается метод базового класса, который,
// в свою очередь, задействует функцию ColeClientItem::SetItemRects
// для перемещения объекта в новую позицию.
if (!ColeClientItem::OnChangeItemPosition(rectPos)) return FALSE;
// TODO: обновите всю кэшированную информацию,
// связанную с размерами объекта.
return TRUE; )
void CCntCntrlItem::OnGetItemPosition(CRect SrPosition)
ASSERT_VALID(this);
// В процессе непосредственного редактирования данная функция
// позволяет определить координаты области, занимаемой объектом.
// По умолчанию заданы фиксированные координаты.
// TODO: запишите правильные координаты (в пикселях)
// в переменную rPosition.
rPosition.SetRect(10, 10, 210, 210);
}
void CCntCntrlItem: :OnActivate() {
// Допускается только один сеанс редактирования в окне приложения.
CCntView* pView = GetActiveView ( ) ;
ASSERT_VALID(pView) ;
ColeClientItem* pltem = GetDocument ( ) ->
GetInPlaceActiveItem(pView) ; if (pltem != NULL ss-pltem != this)
pltem->Close ( ) ;
ColeClientItem: : OnAct ivate ( ) ;
}
void CCntCntrlItem: :OnDeactivateUI (BOOL bUndoable) {
ColeClientItem: :OnDeactivateUI (bUndoable) ;
// Скрывает объект, если он был . активизирован обычным способом
DWORDdwMisc = 0;
m_lpObject->GetMiscStatus (GetDrawAspect ( ) , SdwMisc) ;
if (dwMisc & OLEMISC_INSIDEOUT)
DoVerb (OLEIVERB_HIDE, NULL) ;
}
void CCntCntrlItem: : Serialize (CArchive Sar) {
ASSERT_VALID(this) ;
// Вызов функции базового класса для считывания данных,
// связанных с объектом ColeClientItem. При этом
// инициализируется указатель m_j>Document, возвращаемый
// функцией CCntCntrlItem: :GetDocument, поэтому функцию базового
// класса
// лучше вызывать вначале.
ColeClientItem: : Serialize (ar);

```

```

// Теперь сериализуем данные, специфичные для объекта CCntCntrlItem.
if (ar.IsStoringO )
{
// TODO: здесь добавьте код сохранения. }
else{
// TODO: здесь добавьте код загрузки.
}
}
}
////////////////////////////////////
//Диагностика класса CCntCntrlItem
#ifdef _DEBUG
void CCntCntrlItem: :AssertValid () const
COleClientItem: :AssertValid0;
}
void CCntCntrlItem: :Dump(CDumpContext' &dc) const
COleClientItem: :Dump(dc);
#endif

```

Основное назначение этого файла состоит в отслеживании координат и размеров области, занимаемой внедренным объектом. Обратите внимание на фрагмент, выделенный полужирным шрифтом. Именно здесь задаются координаты, о которых говорилось при рассмотрении файла CNTVIEW.CPP.

Проверка работы контейнера

А теперь давайте проверим, как функционирует разработанное нами приложение-контейнер. Вспомните, что в данном случае мы имеем дело лишь с шаблоном приложения. Никакие дополнительные функциональные возможности в него не добавлялись, тем не менее такая программа является вполне работоспособной.

Окно нашей программы показано на рис. 21.8. В нее можно внедрять объекты, созданные в любом другом приложении Windows, поддерживающем технологию OLE. В качестве примера мы внедрим в программу электронную таблицу Excel. Выберите в меню **Edit** команду **InsertNewObject...**, в результате чего откроется стандартное диалоговое окно вставки объекта. В этом окне выделите элемент, соответствующий электронной таблице Excel(рис. 21.9)

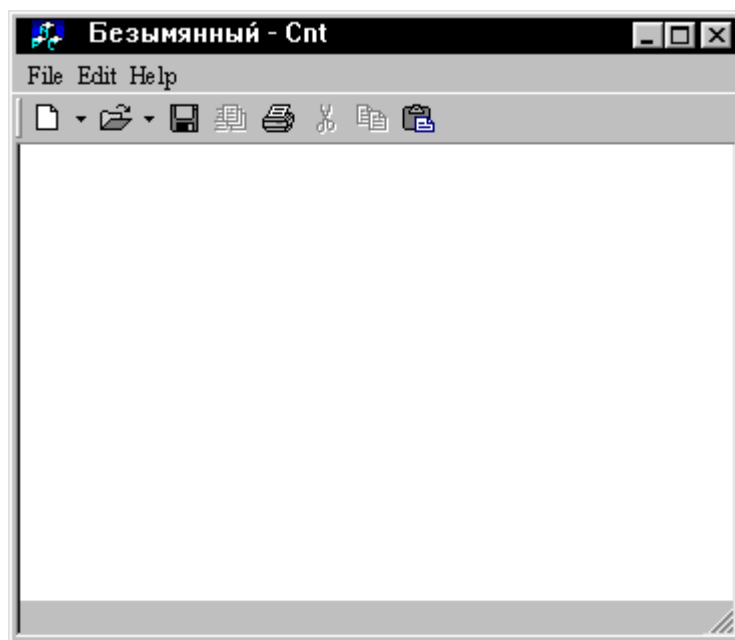


Рис.21.8. Окно приложения Cnt

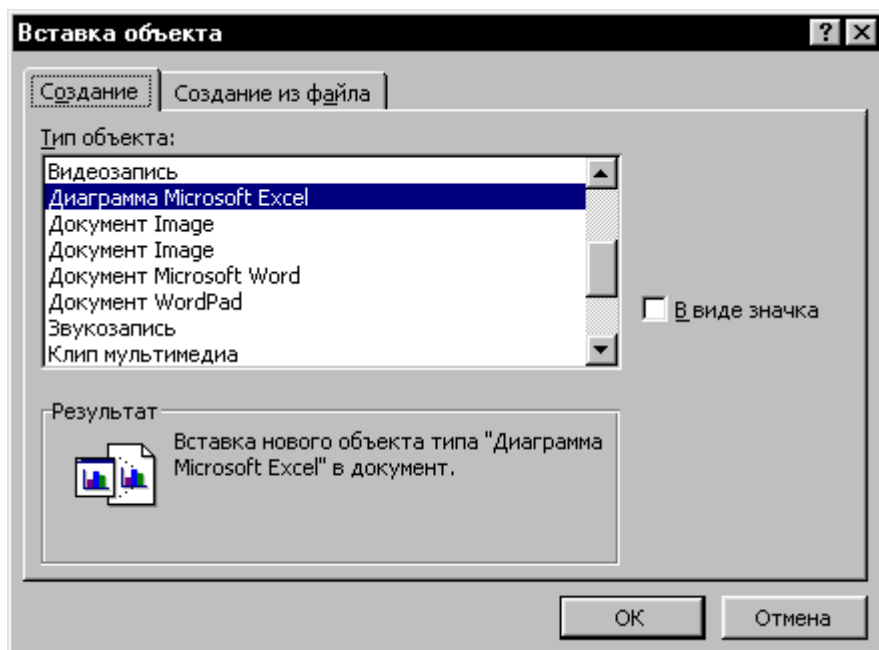


Рис. 21.9. Выбор типа внедряемого объекта

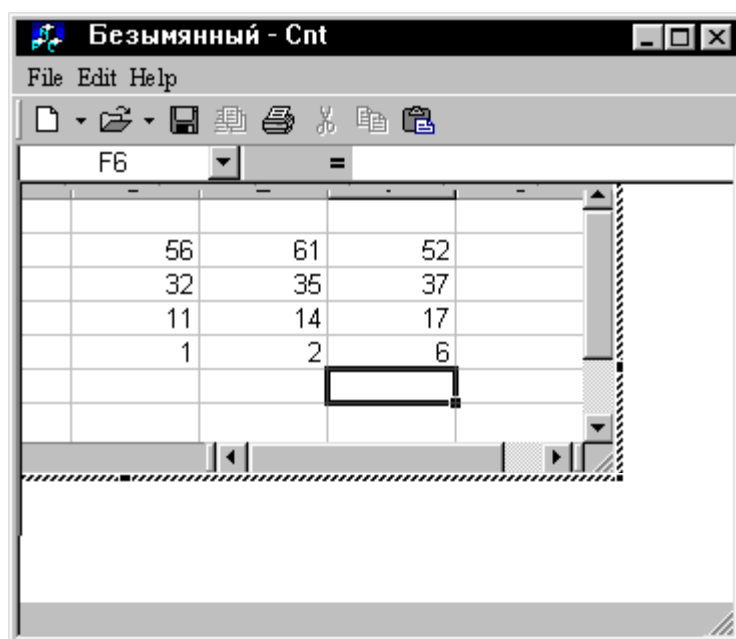


Рис. 21.10. В документ приложения Cnt внедряется электронная таблица Excel

На рис 21.10 показан результат внедрения в документ электронной таблицы Excel со значениями, введенными пользователем.

Еще раз напомним, что все функциональные возможности программы были заданы автоматически. Мы не изменили ни единой строчки в тексте, сгенерированном мастером AppWizard. Итак, вы убедились, что с помощью данного мастера можно легко создавать OLE-контейнеры. Теперь попробуйте самостоятельно создать простое приложение-сервер, а затем внедрите его объект в контейнер Cnt.

Таким образом технология OLE действительно является непростым предметом для изучения, но, безусловно, достойным вашего внимания. Как вы могли убедиться, с помощью мастера

приложений и библиотеки MFC совсем не трудно создать простое приложение, поддерживающее технологию OLE. Собственно говоря, существенная помощь в создании OLE-приложений — это основное достоинство мастера AppWizard.

В следующей главе мы поговорим о таких важных средствах создания современных приложений, как элементы управления ActiveX.

Глава 22. Основы создания элементов управления ActiveX

- Основные концепции
 - Критерии разработки элементов управления
 - Класс COleControl
 - Контейнеры
- Создание простого элемента управления ActiveX с использованием MFC
 - Создание ActiveX-шаблона
 - Код, сгенерированный мастером
- Модификация шаблона
 - Изменение формы, размеров и цвета элемента управления TDCtrl
 - Реакция на события мыши
- Тестирование элемента управления TDCtrl

Вы уже знакомы с разнообразными элементами управления среды Windows, такими как переключатели, флажки, списки и т.д. Но многие разработчики предпочитают наряду с ними использовать и собственные элементы управления, известные как элементы управления ActiveX. Впервые они появились в языке VisualBasic и изначально носили название VBX-элементов (таковым было расширение соответствующих файлов). По сути, эти элементы управления являлись небольшими библиотеками динамической компоновки (DLL), только имевшими расширение VBX. Возможности современных 32-разрядных элементов управления значительно расширены, а для их файлов теперь применяется расширение OCX.

Многие программисты использовали VisualBasic для разработки собственных элементов управления, а затем включали их в приложения, написанные на C/C++. Стало очевидным, что языки C/C++ неплохо было бы снабдить своим средством разработки элементов управления. Однако в то время, когда компания Microsoft подошла к решению данной проблемы, началась смена поколений операционной системы — переход от 16-разрядной Windows3.1 к 32-разрядным Windows95 и NT. Но оказалось, что аппаратно-зависимые 16-разрядные элементы управления VBX не могли служить так же хорошо на новых 32-разрядных платформах. Специалисты Microsoft решили, что целесообразнее будет заняться не расширением спецификации VBX, а разработкой для 32-разрядных платформ принципиально новой архитектуры элементов управления. Для их файлов было выбрано расширение OCX, а сами элементы получили название элементов управления ActiveX.

Хорошей новостью для программистов, работающих с C++, стало включение в среду MicrosoftVisualC++ специального мастера, предназначенного для построения элементов управления ActiveX. Мастер создает код на основе библиотеки MFC. Элемент управления в процессе разработки можно протестировать с помощью утилиты ActiveXControlTestContainer. Готовый элемент можно внедрить в любое приложение, поддерживающее технологию OLE, скажем в MicrosoftWord или Excel.

Основные концепции

Элементы управления ActiveX часто используются совместно с обычными элементами управления, такими как переключатели, кнопки и флажки; все они могут одновременно содержаться в диалоговом окне. Однако реализация элементов управления ActiveX на порядок сложнее.

Вся ответственность за функционирование элемента управления ложится на его разработчика. При этом необходимо учитывать два аспекта. С одной стороны, на стадии разработки следует написать, отладить и скомпилировать весь код, управляющий выводом элемента управления и реализующий все его свойства и методы. В результате будет создана небольшая динамическая библиотека с расширением OCX. С другой стороны, приложение, использующее элемент управления ActiveX, должно будет взаимодействовать с ним, вызывая его методы,

запрашивая его данные и т.д. Разработчик должен определить интерфейс этого взаимодействия таким образом, чтобы элемент управления был полностью независимым от использующего его приложения. О таких объектах говорят, что они обеспечивают повторное вхождение. Вспомните, что элемент управления ActiveX представляет собой отдельную библиотеку динамической компоновки, не связанную ни с каким приложением. Повторное вхождение достигается за счет создания экземпляров библиотеки в каждом приложении, содержащем внедренный элемент управления. Взаимодействие между приложением и элементом осуществляется только посредством сообщений.

Критерии разработки элементов управления

Чтобы создаваемый элемент управления ActiveX получился как можно более привлекательным и функциональным, необходимо определиться по ряду моментов.

- Во-первых, следует решить, как он должен выглядеть на экране. Требуется определенный талант, для того чтобы совместить в одном элементе управления высокую производительность и привлекательный внешний вид.
- Во-вторых, нужно учесть возможность изменения свойств элемента управления посредством интерфейса автоматизации. У элемента управления должны быть страницы свойств, позволяющие пользователям в процессе выполнения программы менять его свойства.
- Наконец, должна быть организована поддержка постоянства свойств элемента управления.

Класс COleControl

Элементы управления ActiveX порождаются от MFC -класса COleControl. В приведенном ниже листинге содержится часть файла AFXCTL.H с сокращенным описанием данного класса. Мы не станем подробно объяснять назначение каждой части файла, а лишь наглядно проиллюстрируем наиболее важные моменты.

```
// Это класс библиотеки MFC .
// Copyright (C)1992-1998Microsoft Corporation
// All rights reserved.
//
// Данный исходный код служит дополнением к справочному
// руководству и сопутствующей электронной документации по MFC .
// Обращайтесь к ним для получения более полной информации.
//
// AFXCTL.H поддержка элементов управления OLE
//
// Базовые события
#define EVENT_STOCK_CLICK() \
{afxEventStock, DISPID_CLICK, _T("Click"),VTS_NONE },
#define EVENT_STOCK_DBLCLICK() \
{ afxEventStock, DISPID_DBLCLICK, _T("Dblick"), VTS_NONE },
#define EVENT_STOCK_KEYDOWN() \
{ afxEventStock, DISPID_KEYDOWN, _T("KeyDown"), VTS_PI2 VTS_I2 },
#define EVENT_STOCK_KEYPRESS() \
{ afxEventStock, DISPID_KEYPRESS, _T("KeyPress"), VTS_PI2 },
#define EVENT_STOCK_KEYUP() \
{ afxEventStock, DISPID_KEYUP, JTC'KeyUp" ) , VTS_PI2 VTS_I2 },
#define EVENT_STOCK_MOUSEDOWN() \
( afxEventStock, DISPID_MOUSEDOWN, _T("MouseDown"), \ VTS_I2 VTS_I2
VTS_XPOS_PIXELS VTS_YPOS_PIXELS },
#define EVENT_STOCK_MOUSEMOVE() \
{ afxEventStock, DISPID_MOUSEMOVE, _T("MouseMove"), \
VTS_I2 VTS_I2 vts_xpos_pixels vts_ypos_pixels },
```

```

#define EVENT_STOCK_MOOSEUP() \
{ afxEvtStock, DISPID_MOUSEUP, _T("MouseUp"), \ VTS_I2 VTS_I2  
VTS_XPOS_PIXELS VTS_YPOS_PIXELS },
#define EVENT_STOCK_ERROREVENTO \{ afxEvtStock, DISPID_ERROREVENT,  
_T("Error"), \
VTS_I2 VTS_PBSTR VTS_SCODE VTS_BSTR VTS_BSTR VTS_I4 VTS_PBOOL },
#define EVENT_STOCK_READYSTATECHANGE () \
{ afxEvtStock, DISPID_READYSTATECHANGE, _T("ReadyStateChange"), \
VTS_I4 },
////////////////////////////////////
// Базовые свойства
#define DISP_PROPERTY_STOCK(theClass, szExternalName, dispid,  
pfnGet, pfnSet, vtPropType) \
{ _T(szExternalName), dispid, NULL, vtPropType, \ (AFX_PMSG)(void  
(theClass::*)(void)JSfnGet, \ (AFX_PMSG)(void  
(theClass::*)(void)SfnSet, 0, afxDispStock ), \
#define DISP_STOCKPROP_APPEARANCE() \
DISP_PROPERTY_STOCK(ColeControl, "Appearance", DISPID_APPEARANCE, \
ColeControl::GetAppearance, ColeControl::SetAppearance, VT_I2)
#define DISP_STOCKPROP_BACKCOLOR() \
DISP_PROPERTY_STOCK(ColeControl, "BackColor", DISPID_BACKCOLOR, \
ColeControl::GetBackColor, ColeControl::SetBackColor, VT_COLOR)
#define DISP_STOCKPROP_BORDERSTYLE() \
DISP_PROPERTY_STOCK(ColeControl, "BorderStyle", DISPID_BORDERSTYLE, \
ColeControl::GetBorderStyle, ColeControl::SetBorderStyle, VT_I2)
#define DISP_STOCKPROP_CAPTION() \
DISP_PROPERTY_STOCK(ColeControl, "Caption", DISPID_CAPTION, \
ColeControl::GetText, ColeControl::SetText, VT_BSTR)
#define DISP_STOCKPROP_ENABLED() \
DISP_PROPERTY_STOCK(ColeControl, "Enabled", DISPID_ENABLED, \
ColeControl::GetEnabled, ColeControl::SetEnabled, VT_BOOL)
#define DISP_STOCKPROP_FONT() \
DISP_PROPERTY_STOCK (ColeControl, "Font", DISPID_FONT, \
ColeControl::GetFont, ColeControl::SetFont, VT_FONT)
#define DISP_STOCKPROP_FORECOLOR() \
DISP_PROPERTY_STOCK(ColeControl, "ForeColor", DISPID_FORECOLOR, \
ColeControl::GetForeColor, ColeControl::SetForeColor, VT_COLOR)
#define DISP_STOCKPROP_HWND() \
DISP_PROPERTY_STOCK(ColeControl, "hWnd", DISPID_HWND, \
ColeControl::GetHwnd, SetNotSupported, VT_HANDLE)
#define DISP_STOCKPROP_TEXT() \
DISP_PROPERTY_STOCK(ColeControl, "Text", DISPID_TEXT, \
ColeControl::GetText, ColeControl::SetText, VT_BSTR)
#define DISP_STOCKPROP_READYSTATE() \
DISP_PROPERTY_STOCK(ColeControl, "ReadyState", DISPID_READYSTATE, \
ColeControl::GetReadyState, SetNotSupported, VT_I4)
////////////////////////////////////
// Базовые методы
#define DISP_FUNCTION_STOCK(theClass, szExternalName, dispid,  
pfnMember, vtRetVal, vtsParams) \
( _T(szExternalName), dispid, vtsParams, vtRetVal, \ (AFX_PMSG)(void  
(theClass::*)(void)SfnMember, (AFX_PMSG)0, 0, \ afxDispStock }, \
#define DISP_STOCKFDNC_REFRESH() \
DISP_FUNCTION_STOCK(ColeControl, "Refresh", DISPID_REFRESH, \
ColeControl::Refresh, VT_EMPTY, VTS_NONE)
#define DISP_STOCKFUNC_DOCLICK() \

```

```

DISP_FUNCTION_STOCK(ColeControl, "DoClick", DISPID_DOCLICK, \
ColeControl::DoClick, VT.EMPTY, VTS_NONE)
////////////////////////////////////
// ColeControl - базовый класс элемента управления.
// реализованного на C++ с использованием MFC
class COleControl: public CWnd
// Базовые методы
void Refresh () ; void DoClick () ;
// Базовые свойства
short GetAppearance();
void SetAppearance(short);
OLE_COLOR GetBackColorO ;
void SetBackColor(OLE_COLOR);
short GetBorderStyle();
void SetBorderStyle(short);
BOOL GetEnabledO ;
void SetEnabled(BOOL);
CFontHolderS InternalGetFont();
LPFONTDISP GetFontO;
void SetFont(LPFONTDISP);
OLE_COLOR GetForeColor();
void SetForeColor(OLE_COLOR);
OLE_HANDLE GetHwnd();
const CStringS InternalGetText () ;
BSTR GetText () ;
void SetText(LPCTSTR);
long GetReadyState();
void InternalSetReadyState(long INewReadyState) ,
// Внешние свойства
short AmbientAppearance();
OLE_COLOR AmbientBackColor();
CString AmbientDisplayNarae();
LPFONTDISP AmbientFont();
OLE_COLOR AmbientForeColor();
LCID AmbientLocaleIDO;
CString AmbientScaleUnits(> short AmbientTextAlign());
BOOL AmbientUserModeO;
BOOL AmbientUIDeadO;
BOOL AmbientshowGrabHandles();
BOOL AmbientShowHatching();
// Генерация событий
void AFX_CDECL FireEvent(DISPID dispid, BYTE* pbParams,
// Функции генерации базовых событий
void FireKeyDown(USHORT* pnChar, short nShiftState);
void FireKeyUp(USHORT* pnChar, short nShiftState);
void FireKeyPress(USHORT* pnChar); void FireMouseDown(short nButton,
short nShiftState,
OLE_XPOS_PIXELS x, OLE_YPOS_PIXELS y) ;
void FireMouseDp(short nButton, short nShiftState,
OLE_XPOS_PIXELS X, OLE_YPOS_PIXELS y);
void FireMouseMove(short nButton, short nShiftState,
OLE_XPOS_PIXELS x, OLE_YPOS_PIXELS y);
void FireClickO; void FireDbClickO ;
void FireError(SCODE scode, LPCTSTR IpszDescription,

```

```

UINT nHelpID. = 0) ;
void FireReadyStateChange();
// Базовые события
void KeyDown(USHORT* pnChar);
void KeyUp(USHORT* pnChar);
void ButtonDown(USHORT iButton, UINT nFlags, CPoint point);
void ButtonUp(USHORT iButton, UINT nFlags, CPoint point);
void ButtonDbLClk(USHORT iButton, UINT nFlags, CPoint point);
// Базовые свойства
OLE_COLOR m_clrBackColor;    // Цвет фона
OLE_COLOR m_clrForeColor;    // Цвет переднего плана
CString m_strText; // Текст/Надпись
CFontHolder m_font;          // Шрифт
HFONT m_hFontPrev; // Предыдущий шрифт
short m_sAppearance; . // Внешний вид
short m_sBorderStyle; // Стил ь рамки
BOOL m_bEnabled; // Доступность
long m_lReadyState; // Состояние готовности
// Схема сообщений
protected:
//{{AFX_MSG(COLEControl)
afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags),
afx_msg void OnKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags);
afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
afx_msg void OnMouseMove(UINT nFlags, CPoint point);
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
afx_msg void OnLButtonDbLClk(UINT nFlags, CPoint point);
afx_msg void OnMButtonDown(UINT nFlags, CPoint point);
afx_msg void OnMButtonUp(UINT nFlags, CPoint point);
afx_msg void OnMButtonDbLClk(UINT nFlags, CPoint point);
afx_msg void OnRButtonDown(UINT nFlags, CPoint point);
afx_msg void OnRButtonUp(UINT nFlags, CPoint point);
afx_msg void OnRButtonDbLClk(UINT nFlags, CPoint point);
afx_msg void OnInitMenuPopup(CMenu*, UINT, BOOL);
afx_msg void OnMenuSelect(UINT nItemID, UINT nFlags, HMENU hSysMenu),
afx_msg LRESULT OnSetMessageString(WPARAM wParam, LPARAM lParam);
afx_msg void OnEnterIdle(UINT nWhy, CWnd* pWho);
afx_msg void OnCancelMode();
afx_msg void OnPaint(CDC* pDC);
afx_msg BOOL OnEraseBkgnd(CDC* pDC) ;
afx_msg void OnSysKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
afx_msg void OnSysKeyUp(UINT nChar, UINT nRepCnt, UINT nFlags);
afx_msg int OnMouseActivate(CWnd* pDesktopWnd, UINT nHitTest,
UINT message);
afx_msg LRESULT OnSetText(WPARAM wParam, LPARAM lParam);
afx_msg BOOL OnNcCreate(LPCREATESTRUCT lpCreateStruct);
afx_msg void OnDestroy(); afx_msg void OnKillFocus(CWnd* pNewWnd);
afx_msg void OnSetFocus(CWnd* pOldWnd) ;
afx_msg void OnNcPaint();
afx_msg void OnNcCalcSize(BOOL bCalcValidRects,
NCCALCSIZE_PARAMS* lpncsp); afx_msg void OnNcHitTest(CPoint point);

```

```

afx_msg void OnNcLButtonDown(UINT nHitTest, CPoint point); afx_msg
BOOL OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message); afx_msg
UINT OnGetDlgCode();
afx_msg int OnCreate(LPCREATESTRUCT IpCreateStruct); afx_msg void
OnSize(UINT nType, int ex, int cy) ; afx_msg void OnMove(int x, int
y);
afxjmsg void OnShowWindow(BOOL bShow, UINT nStatus); //))AFX_MSG
afx_msg LRESULT OnOcrCtlColorBtn(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorDlg(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorEdit(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorListBox(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorMsgBox(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnOcmCtlColorScrollBar(WPARAM wParam, LPARAM lParam) ,
afx_msg LRESULT OnOcmCtlColorStatic(WPARAM wParam, LPARAM lParam);
DECLARE_MESSAGE_MAP()
// вспомогательные обработчики
void OnButtonUp(USHORT nButton, UINT nFlags, CPoint point);
void OnButtonDown(USHORT nButton, UINT nFlags, CPoint point);
void OnButtonDblClk(USHORT nButton, UINT nFlags, CPoint point);
// Схемы интерфейсов public:
// IPersistStorage
BEGIN_INTERFACE_PART(PersistStorage, IPersistStorage)
INIT_INTERFACE_PART(ColeControl, PersistStorage)
STDMETHOD(GetClassID) (LPCLSID);
STDMETHOD(IsDirty) ();
STDMETHOD(InitNew) (LPSTORAGE);
STDMETHOD(Load) (LPSTORAGE);
STDMETHOD(Save) (LPSTORAGE, BOOL);
STDMETHOD(SaveCompleted) (LPSTORAGE);
STDMETHOD(HandsOffStorage) ();
END_INTERFACE_PART_STATIC(PersistStorage)
// IOleInPlaceObject
BEGIN_INTERFACE_PART(OleInPlaceObject, IOleInPlaceObjectWindowless)
INIT_INTERFACE_PART(ColeControl, OleInPlaceObject)
STDMETHOD(GetWindow) (HWND*);
STDMETHOD(ContextSensitiveHelp) (BOOL);
STDMETHOD(InPlaceDeactivate) ();
STDMETHOD(OIDeactivate) ();
STDMETHOD(SetObjectRects) (LPCRECT, LPCRECT);
STDMETHOD(ReactivateAndUndo) ();
STDMETHOD(OnWindowMessage) (UINT msg, WPARAM wParam, LPARAM lParam,
    LRESULT* pResult);
STDMETHOD(GetDropTarget) (IDropTarget **ppDropTarget);
END_INTERFACE_PART(OleInPlaceObject)
// IOleInPlaceActiveObject
BEGIN_INTERFACE_PART(OleInPlaceActiveObject, IOleInPlaceActiveObject)
INIT_INTERFACE_PART(ColeControl, OleInPlaceActiveObject)
STDMETHOD(GetWindow) (HWND*);
STDMETHOD(ContextSensitiveHelp) (BOOL);
STDMETHOD(TranslateAccelerator) (LPMSG);
STDMETHOD(OnFrameWindowActivate) (BOOL);
STDMETHOD(OnDocWindowActivate) (BOOL);
STDMETHOD(ResizeBorder) (LPCRECT, LPOLEINPLACEOIWIND, BOOL);
STDMETHOD(EnableModeless) (BOOL);
END_INTERFACE_PART(OleInPlaceActiveObject)

```

}

События

События генерируются элементом управления в ответ на выполняемые над ним действия, такие как нажатие определенной комбинации клавиш или щелчок мышью. Примерами событий могут служить KeyUp и KeyDown.

Класс элемента управления ActiveX, порождаемый от класса GOleControl, может иметь собственную специальную схему сообщений, посредством которой события посылаются приложению, содержащему элемент управления. Такое приложение называется контейнером. Дополнительную информацию о событии можно передать с помощью аргументов.

Именно путем передачи событий организуется взаимодействие между элементом управления ActiveX и контейнером. Различаются базовые и пользовательские события. События первого типа могут генерироваться любым элементом управления и по умолчанию обрабатываются классом CQleControl. События второго типа посылаются контейнеру при выполнении над элементом управления действий, указанных разработчиком элемента.

Методы и свойства

Элемент управления ActiveX должен обладать рядом методов и свойств, позволяющих контейнеру в интерактивном режиме менять его, элемента управления, настройки. Методами называются функции элемента управления, которые могут вызываться контейнером. Под свойствами понимают цвет, шрифт, надпись и другие атрибуты внешнего вида элемента управления. Методы и свойства элемента управления определяются в процессе его разработки с помощью мастера ClassWizard. Базовые методы присущи всем элементам управления и по умолчанию реализуются классом COleControl.

Доступ к методам и свойствам элемента управления возможен на этапе раннего связывания. При этом методы рассматриваются как обычные функции-члены класса, а свойства представляются парами функций getxxx и setxxx. Доступ к элементу управления на этапе позднего связывания регулируется интерфейсом IDispatch. Контейнер автоматически определяет тип связывания, установленный пользователем. Интерфейс IProvideClassInfo содержит функцию GetClassInfo(), которая возвращает объект ITypeInfo с информацией об элементе управления, взятой из его библиотеки типов.

Элементы управления ActiveX часто содержат расширенные наборы свойств, методов и событий, которые используются только контейнером.

Постоянство

Элементы управления позволяют сохранять свои данные в потоках с помощью интерфейса IPersistStream, а в хранилищах — с помощью интерфейса IPersistStorage. Первый необходим для записи данных в простой последовательный поток, а второй реализует концепцию структурированного хранения составных документов, когда каждый внедренный или связанный объект имеет собственную область хранения в общем хранилище данных документа.

Благодаря механизму постоянства элемент управления ActiveX при необходимости может записывать значения своих свойств в файл или поток, а затем считывать их. Значения свойств элемента управления могут сохраняться самим контейнером и считываться, скажем, при создании нового экземпляра элемента управления.

Контейнеры

Контейнеры, обеспечивающие возможность непосредственного редактирования внедренных объектов, обладают необходимым интерфейсом для использования элементов управления ActiveX. В дополнение к этому контейнеры должны поддерживать обработку событий и внешние свойства.

Элемент управления ActiveX выступает в роли переводчика сообщений. Он должен уметь преобразовывать сообщения, получаемые от пользователя, в события, понятные контейнеру. Для каждого события должна быть определена точка входа в контейнере, связанная с обслуживающей процедурой.

Свойства контейнера, применимые ко всем внедренным элементам управления, называются внешними (ambient). К таковым, в частности, относятся заданные по умолчанию цвета и шрифты.

Создание простого элемента управления ActiveX с использованием MFC

В данном параграфе мы шаг за шагом пройдем процесс создания шаблона элемента управления ActiveX с помощью мастера MFC ActiveXControlWizard. Затем мы изменим шаблон таким образом, чтобы получить элемент управления, отвечающий нашим требованиям.

Создание ActiveX-шаблона

В окне компилятора Microsoft Visual C++ выберите в меню File команду New и в открывшемся диалоговом окне установите опцию MFC ActiveX ControlWizard (рис. 22.1). Назовем наш проект TDCtrl.

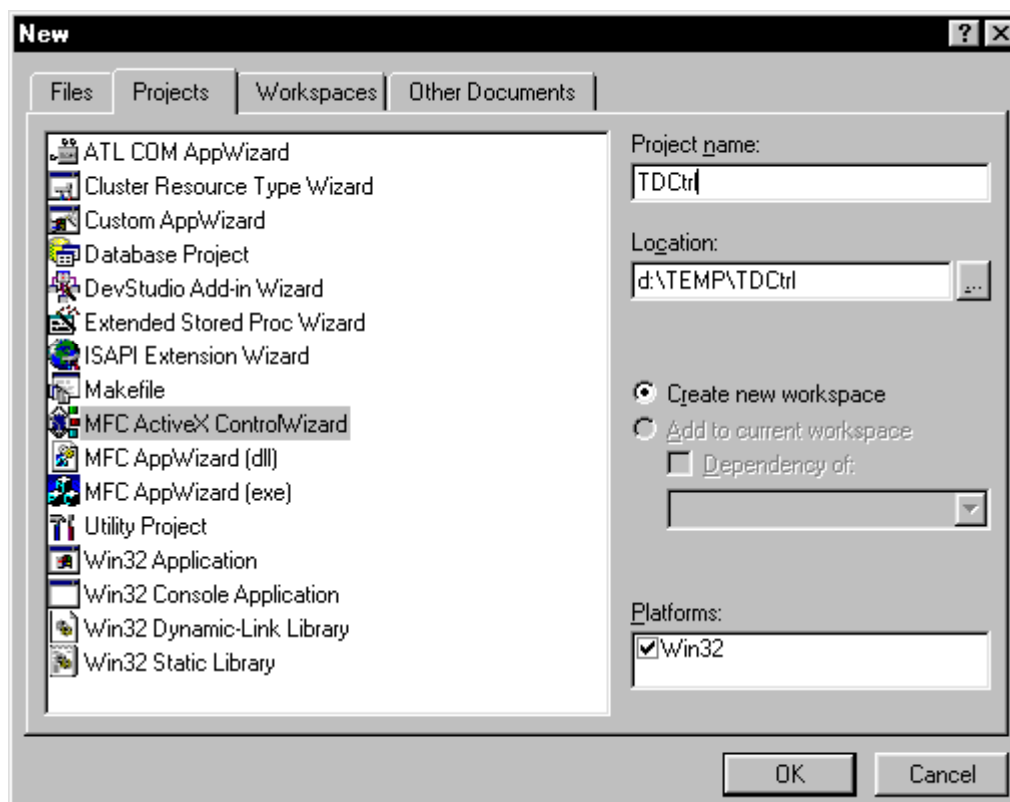


Рис. 22.1. Выбор типа проекта

Мастер создает элемент управления в два этапа. Сначала задаются установки всего проекта, такие как число элементов управления, наличие лицензии на выполнение, добавление комментариев в программный код и создание файлов справки (рис. 22.2). Не меняйте опции, заданные по умолчанию.

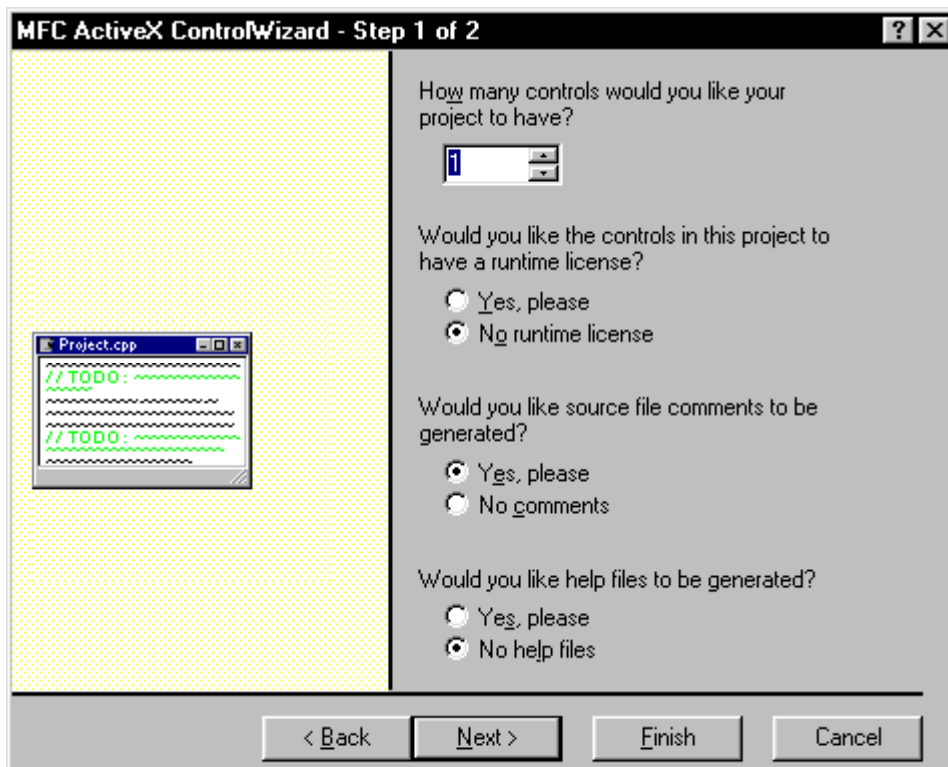


Рис. 22.2. Шаг 1: принимаем опции, заданные по умолчанию

На втором этапе задаются параметры каждого элемента управления, включаемого в проект. Установите в этом окне опцию Available in "Insert Objects" dialog, чтобы название элемента управления появлялось в окне вставки объектов (рис. 22.3.)

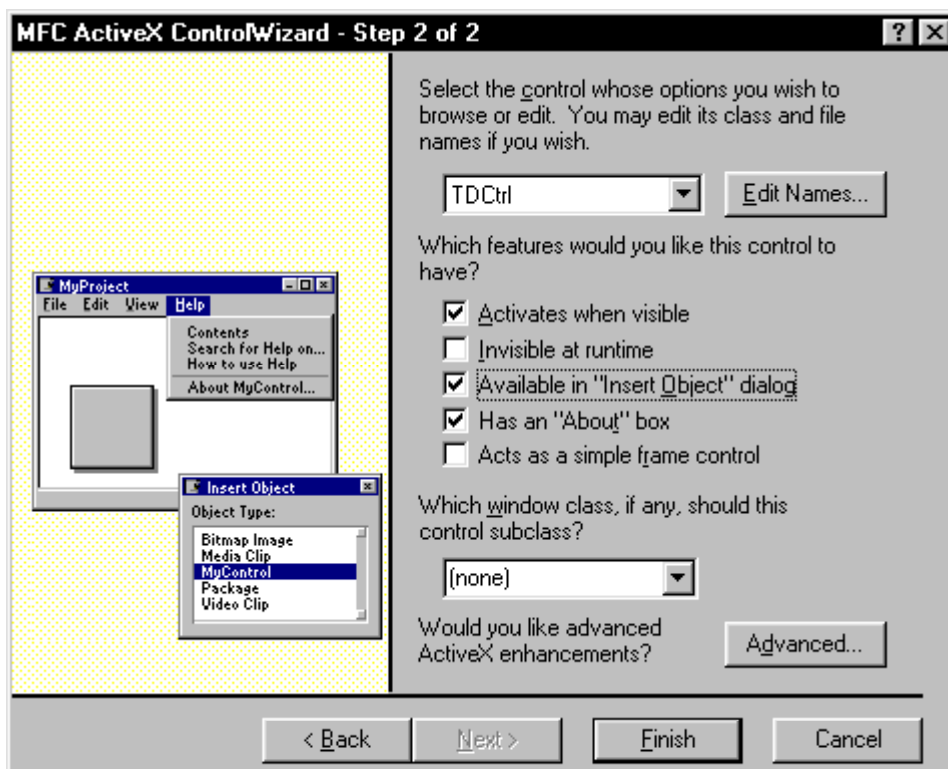


Рис. 22.3. Шаг 2: параметры элемента управления TDCtrl

Далее на экране появится отчет с перечнем установок проекта (рис 22.4)

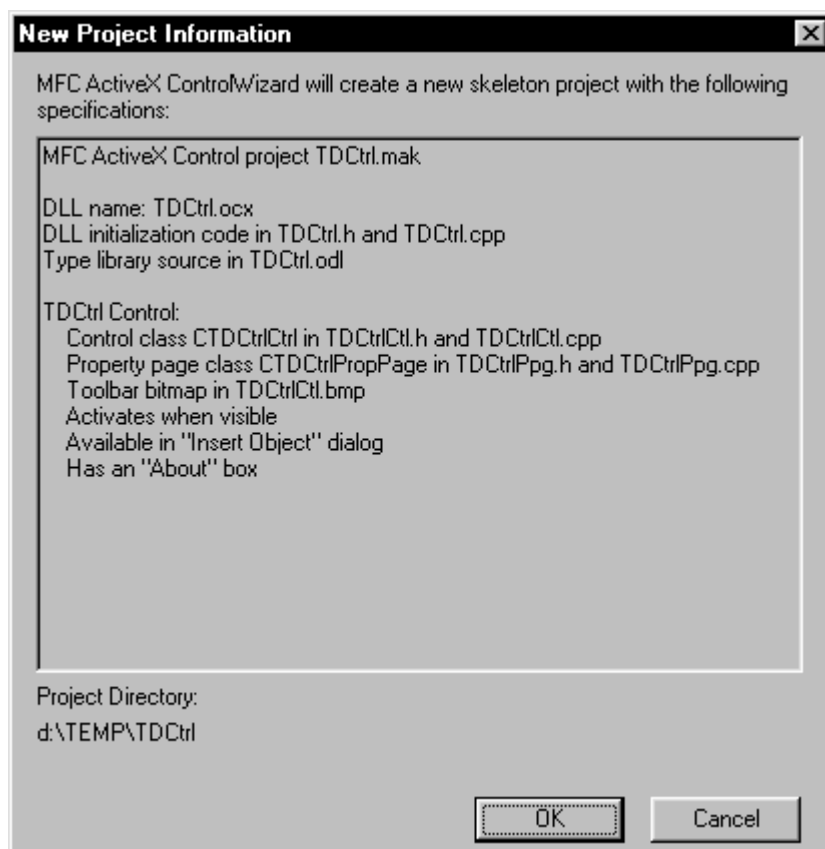


Рис. 22.4. Отчет о параметрах создаваемого элемента управления

После щелчка на кнопке **OK** мастер автоматически сгенерирует код шаблона элемента управления. Список его классов и глобальных атрибутов представлен на вкладке ClassView(рис 22.5)

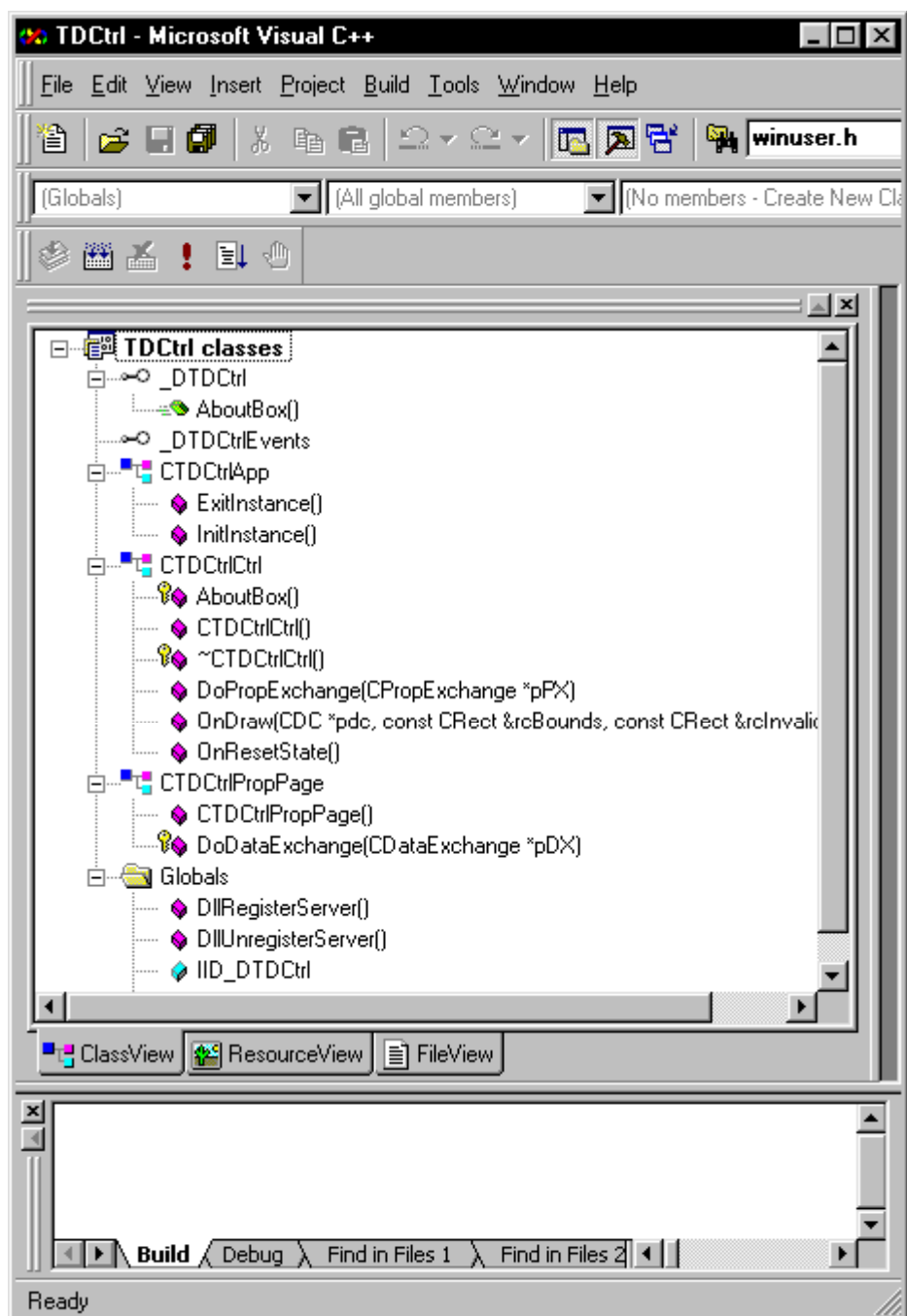


Рис. 22.5. Список классов и глобальных атрибутов элемента управления TD Ctrl

Для того чтобы скомпилировать проект, выберите в меню **Build** команду **Build** или **Rebuild All**. После завершения этого процесса в папке DEBUG появится файл TDCTRL.OCX.

Элементы управления ActiveX представляют собой небольшие библиотеки динамической компоновки, которые можно тестировать в соответствующих контейнерах. Компания Microsoft предоставляет для этих целей специальную утилиту ActiveXControlTestContainer; запускаемую из меню **Tools**(рис. 22.6).

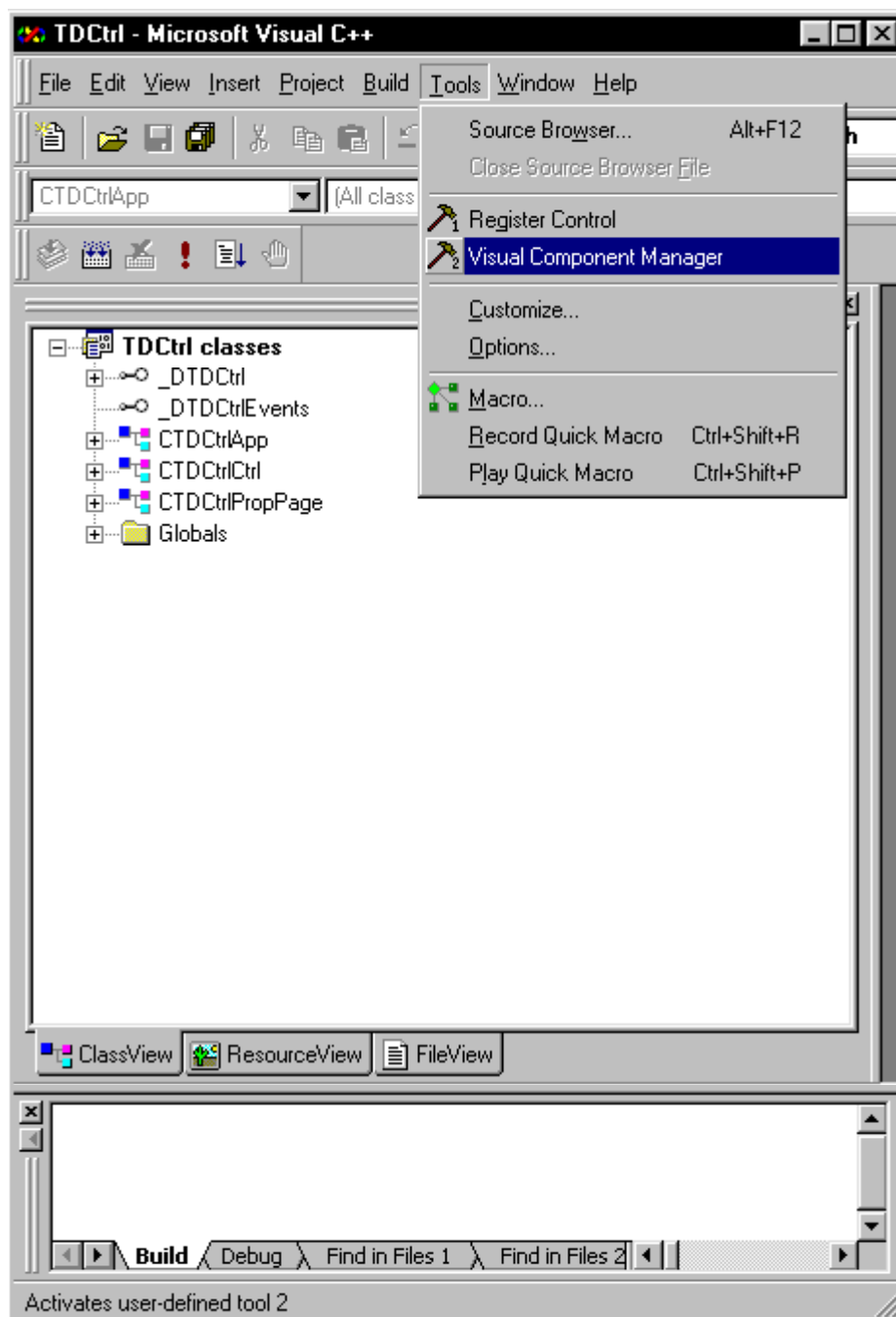


Рис. 22.6. Местонахождение инструментов ActiveX

В окне контейнера выберите в меню **Edit** команду **InsertNewControl**, и перед вами откроется диалоговое окно **InsertControl**, в котором будет представлен список всех зарегистрированных элементов управления. Теперь щелкните на кнопке **OK**, чтобы внедрить выбранный элемент управления в окно контейнера.

Новый элемент управления представляет собой эллипс, окруженный рамкой. Никакой функциональной нагрузки он пока не несет. Чтобы данный элемент управления заработал в соответствии с нашими требованиями, в сгенерированный мастером код следует добавить некоторые программные блоки.

Но прежде чем приступить к редактированию кода, посмотрим, что именно сгенерировал мастер. Особое внимание нужно уделить тем частям программы, которые в дальнейшем будут подвергнуты модификации.

Код, сгенерированный мастером

Для большинства элементов управления ActiveX мастер ControlWizard создает четыре программных файла. В нашем проекте это файлы STDAFX.CPP, TDCTRL.CPP, TDCTRLCTL.CPP и TDCTRLPPG.CPP. Каждому из них соответствует отдельный файл заголовков.

Файл STDAFX.CPP предназначен для включения в программу всех файлов заголовков, необходимых для работы элемента управления ActiveX. Файл TDCTRL.CPP отвечает за реализацию класса CTDCtrlApp и регистрацию DLL-модуля.

В следующих параграфах мы проанализируем назначение файлов TDCTRLCTL.CPP и TDCTRLPPG.CPP.

Файл TDCTRLCTL.CPP

Файл TDCTRLCTL.CPP содержит реализацию собственно класса элемента управления ActiveX. В нашем примере это класс CTDCtrlCtrl. Именно данный файл больше других требует изменений со стороны программиста.

```
// TDCtrlCtrl.cpp: Реализация класса CTDCtrlCtrl элемента управления.
#include "stdafx.h"
#include "TDCtrl.h"
#include "TDCtrlCtl.h"
#include "TDCtrlPpg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = _FILE_;
#endif
IMPLEMENT_DYNCREATE(CTDCtrlCtrl, COleControl)
////////////////////////////////////
// Схема сообщений
BEGIN_MESSAGE_MAP(CTDCtrlCtrl, COleControl)
//{{AFX_MSG_MAP(CTDCtrlCtrl)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
// макросы схемы-сообщений.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//}}AFX_MSG_MAP
ON_OLEVERB (AFX_IDS_VERB_EDIT, OnEdit)
ON_OLEVERB (AFX_IDS_VERB_PROPERTIES, OnProperties) END_MESSAGE_MAP ()
////////////////////////////////////
// Схема диспетчеризации
BEGIN_DISPATCH_MAP(CTDCtrlCtrl, COleControl)
// { {AFX_DISPATCH_MAP (CTDCtrlCtrl)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
// макросы схемы диспетчеризации.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//}}AFX_DISPATCH_MAP
DISP_FUNCTION_ID (CTDCtrlCtrl, "AboutBox", DISPID_ABOUTBOX,
AboutBox, VT_EMPTY, VTSJTONE) END_DISPATCH_MAP ( )
////////////////////////////////////
// Схемасобытий
BEGIN_EVENT_MAP (CTDCtrlCtrl, COleControl)
// { (AFX_EVENT_MAP (CTDCtrlCtrl)
```

```

// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
//      макросы схемы событий.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
// }AFX_EVENT_MAP END_EVENT_MAP ( )
////////////////////////////////////
// Страницы свойств
// TODO: Сюда можно добавлять новые страницы свойств.
//      Не забудьте увеличить значение счетчика!
BEGIN_PROPPAGEIDS (CTDCtrlCtrl, 1)
PROPPAGEID (CTDCtrlPropPage : : quid) END_PROPPAGEIDS (CTDCtrlCtrl)
////////////////////////////////////
// Инициализация фабрики класса и формирование GUID
IMPLEMENT_OLECREATE_EX (CTDCtrlCtrl, "TDCTRL . TDCtrlCtrl .1",
Oxc0377506, Oxb276, Ox11dl, Oxba, Oxe9, 0, Oxa0, Oxc9, Ox8c, Ox4,
Oxbe)
////////////////////////////////////
// Идентификатор библиотеки типов и ее версия IMPLEMENT_OLETYPELIB
(CTDCtrlCtrl, _tlid, _wVerMajor, _wVerMinor)
////////////////////////////////////
// Идентификаторы интерфейсов
const IID BASED_CODE IID_DTDCtrl =
{ Oxc0377504, Oxb276, Ox11dl, { Oxba, Oxe9, 0, Oxa0, Oxc9,
Ox8c, Ox4, Oxbe } } ;
const IID BASED_CODE IID_DTDCtrlEvents =
{ Oxc0377505, Oxb276, Ox11dl, { Oxba, Oxe9, 0, Oxa0, Oxc9,
Ox8c, Ox4, Oxe } } ;
////////////////////////////////////
// Информация о типе элемента управления
static const DWORD BASED_CODE _dwTDCtrl!01eMisc =
OLEMISC_ACTIVATEWHENVISIBLE I OLEMISC_SETCLIENTSITEFIRST I
OLEMISC_INSIDEOUT I OLEMISC_CANTLINKINSIDE I
OLEMISC_RECOMPOSEONRESIZE;
IMPLEMENT_OLECTLTYPE (CTDCtrlCtrl, IDSJTDCtrl, _dwTDCtrl!01eMisc)
////////////////////////////////////
CTDCtrlCtrl: :CTDCtrlCtrlFactory: :UpdateRegistry
// Добавляет и удаляет записи системного реестра
// для класса CTDCtrlCtrl
BOOL CTDCtrlCtrl: : CTDCtrlCtrlFactory : :UpdateRegistry (BOOL
bReglster)
{
// TODO: Проверьте, соответствует ли элемент управления
// правилам изолированной потоковой модели.
// За информацией обратитесь к разделу документации
// MFC TechNote64.
// Если элемент управления не поддерживает эту модель,
// замените шестой параметр константой afxReglinsertable.
if (bRegister)
return AfxOleRegisterControlClass (
AfxGetInstanceHandle () ,
m_clsid,
m_lpszProgID,
IDSJTDCtrl,
IDBJTDCtrl,
afxReglinsertable I afxRegApartmentThreading,
_dwTDCtrl!01eMisc,

```

```

_tlid, _wVerMajor, _wVerMinor) ; else
return AfxOleUnregisterClass(m_clsid, m_lpszProgID); }
////////////////////////////////////
// CTDCtrlCtrl::CTDCtrlCtrl - Конструктор
CTDCtrlCtrl::CTDCtrlCtrl() {
InitializellDs(SIID_DTDCtrl, SIID_DTDCtrlEvents);
// TODO: В этом месте инициализируйте данные
//      текущего экземпляра элемента управления.
}
////////////////////////////////////
// CTDCtrlCtrl::~CTDCtrlCtrl - Деструктор
CTDCtrlCtrl: : ~CTDCtrlCtrl( )
{
// TODO: В этом месте удалите данные
//      текущего экземпляра элемента управления.
}
////////////////////////////////////
I/ CTDCtrlCtrl::OnDraw- Функция отображения
void CTDCtrlCtrl::OnDraw(CDC* pdc, const CRect SrcBounds,
const CRect SrcInvalid) {
// TODO: Замените следующие операторы
//      собственными операторами отображения.
pdc->FillRect(rcBounds,
CBrush::FromHandle((HBRUSH)GetStockObject(WHITE_BRUSH)))
pdc->Ellipse(rcBounds); }
////////////////////////////////////
// CTDCtrlCtrl::DoPropExchange - Поддержка постоянства элемента
void CTDCtrlCtrl::DoPropExchange(CPropExchange* pPX) f
ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
ColeControl::DoPropExchange(pPX) ;
// TODO: Поместите здесь вызовы PX_-функций для каждого
//      постоянного пользовательского свойства.
}
////////////////////////////////////
// CTDCtrlCtrl::OnResetState- Восстанавливает первоначальное
//      состояние элемента управления
void CTDCtrlCtrl::OnResetState() {
ColeControl::OnResetState(); // Сброс значений, установленных
// в методе DoPropExchange()
// TODO: Восстановите здесь все остальные переменные состояния
//      элемента управления. )
////////////////////////////////////
// CTDCtrlCtrl::AboutBox- Отображает окно About
void CTDCtrlCtrl::AboutBox() {
CDialog dlgAbout(IDD_ABOUTBOX_TDCTRL) ;
dlgAbout.DoModal();
////////////////////////////////////
// Обработчики сообщений класса CTDCtrlCtrl

```

Схемы сообщений, диспетчеризации и событий создаются и модифицируются различными мастерами автоматически. В большинстве случаев эти фрагменты не редактируются.

Механизм автоматизации заключается в вызове методов и свойств элементов управления из разных приложений. Эти запросы распределяются посредством схемы диспетчеризации. Схема событий помогает обрабатывать события элемента управления.

Функция OnDraw() скоро станет объектом нашего особого внимания, поскольку это именно тот блок программы, который отвечает за вывод графического образа элемента управления. По умолчанию в созданном мастером элементе управления ActiveX вызывается функция Ellipse(), выводящая на экран эллипс.

```

////////////////////////////////////
// CTDCtrlCtrl::OnDraw— Функция отображения
void CTDCtrlCtrl::OnDraw (CDC* pdc, const CRect SrcBounds, const
CRect SrcInvalid) {
// TODO: Замените следующие операторы
// собственными операторами отображения.
pdc->FillRect (rcBounds,
CBrush::FromHandle( (HBRUSH) GetStockObject (WHITE_BRUSH) ) );
pdc->Ellipse (rcBounds) ; }

```

Мастер ControlWizard также создает для разрабатываемого проекта диалоговое окно **About**. Содержимое окна можно изменить таким образом, чтобы в нем отражалась специфическая информация о проекте. Описание окна вы найдете в файле TDCTRL.RC, а выводится оно на экран посредством следующей функции:

```

////////////////////////////////////
// CTDCtrlCtrl::AboutBox— Отображает окно About
void CTDCtrlCtrl::AboutBox () {
CDialog dlgAbout (IDD_ABOUTBOX_TDCTRL) ;
dlgAbout . DoModal ( ) ;
}

```

Файл TDCTRLPPG.CPP

Этот файл содержит реализацию класса CTDCtrlPropPage, являющегося потомком класса COlePropertyPage.

```

// TDCtrlPpg.cpp: реализация класса CTDCtrlPropPage страницы свойств.
#include "stdafx.h"
#include "TDCtrl.h"
#include "TDCtrlPpg.h"
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = _FILE_;
#endif
IMPLEMENT_DYNCREATE(CTDCtrlPropPage, COlePropertyPage)
////////////////////////////////////
// Схема сообщений
BEGIN_MESSAGE_MAP(CTDCtrlPropPage, COlePropertyPage)
//{{AFX_MSG_MAP(CTDCtrlPropPage)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять и удалять здесь
// макросы схемы сообщений.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//}}AFX_MSG_MAP END_MESSAGE_MAP ()
////////////////////////////////////
// Инициализация фабрики классов и формирование GUID
IMPLEMENT_OLECREATE_EX(CTDCtrlPropPage,
"TDCTRL.TDCtrlPropPage.1", 0xc0377507,
0xb276, 0x11d1, 0xba, 0xe9, 0, 0xa0,
0xc9, 0x8c, 0xc4, '0x6e')
////////////////////////////////////
// CTDCtrlPropPage::CTDCtrlPropPageFactory::UpdateRegistry —
// Добавляет и удаляет записи системного реестра
// для класса CTDCtrlPropPage

```

```

BOOL CTDCtrlPropPage::CTDCtrlPropPageFactory::UpdateRegistry
(BOOL bRegister) {
if (bRegister)
return AfxOleRegisterPropertyPageClass(AfxGetInstanceHandle(),
m_clsid, IDS_TDCTRL_PPG) else
return AfxOleUnregisterClass(m_clsid, NULL); }
////////////////////////////////////
// CTDCtrlPropPage::CTDCtrlPropPage - Конструктор
CTDCtrlPropPage::CTDCtrlPropPage() :
COlePropertyPage(IDD, IDS_TDCTRL_PPG_CAPTION) !
//({AFX_DATA_INIT(CTDCtrlPropPage)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять здесь
// операторы инициализации переменных-членов.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//})AFXDATAINIT
}
////////////////////////////////////
//CTDCtrlPropPage::DoDataExchange- Осуществляет обмен данными
// между страницей и связанными с ней свойствами
void CTDCtrlPropPage::DoDataExchange(CDataExchange* pDX) { '
//({AFX_DATA_MAP(CTDCtrlPropPage)
// ПРИМЕЧАНИЕ: мастер классов будет добавлять здесь
// вызовы DDP-, DDX- и DDV-функций.
// НЕ РЕДАКТИРУЙТЕ то, что здесь находится.
//})AFX_DATA_MAP
DDP_PostProcessing(pDX); }
////////////////////////////////////
// Обработчики сообщений класса CTDCtrlPropPage

```

Функция AfxOleRegisterPropertyPageClass() предназначена для регистрации класса страницы свойств в системном реестре. Благодаря этому страница свойств может использоваться другими контейнерами, которые поддерживают внедрение элементов управления ActiveX. Регистрационная запись с указанием имени страницы свойств и ее местоположения в системе обновляется всякий раз при вызове данной функции.

Обратите внимание, что в конструкторе базового класса COlePropertyPage, от которого порожден класс CTDCtrlPropPage, можно указать идентификатор шаблона диалогового окна, положенного в основу страницы свойств, а также идентификатор строки заголовка.

Функция DoDataExchange() обычно используется для обмена данными между диалоговым окном и программой, а также для проверки этих данных. В нашем случае задача указанной функции состоит в присваивании значений, вводимых пользователем на странице свойств, соответствующим свойствам элемента управления.

Модификация шаблона

Элемент управления ActiveX, автоматически созданный мастером ControlWizard, можно изменить с помощью мастера ClassWizard. Мы внесем следующие изменения в имеющийся шаблон:

- элемент управления будет представлен на экране не эллипсом, а прямоугольником;
- прямоугольный элемент управления будет закрашен желтым цветом;
- элемент управления будет реагировать на события мыши, выводя внутри прямоугольника текущее системное время и дату.

Сделать все это можно путем внесения изменений в файлы TDCtrlCtl.CPP и TDCtrlCtl.H.

Изменение формы, размеров и цвета элемента управления TDCtrl

В окне компилятора VisualC++ в меню **View** выберите команду **ClassWizard**, чтобы запустить мастер ClassWizard, и выполните действия, перечисленные далее.

1. В диалоговом окне **MFC ClassWizard** перейдите на вкладку **Automation**.
2. В списке Classname выберите класс CTDCtrlCtrl.
3. С помощью кнопки **AddProperty** откройте одноименное диалоговое окно (рис. 22.9).

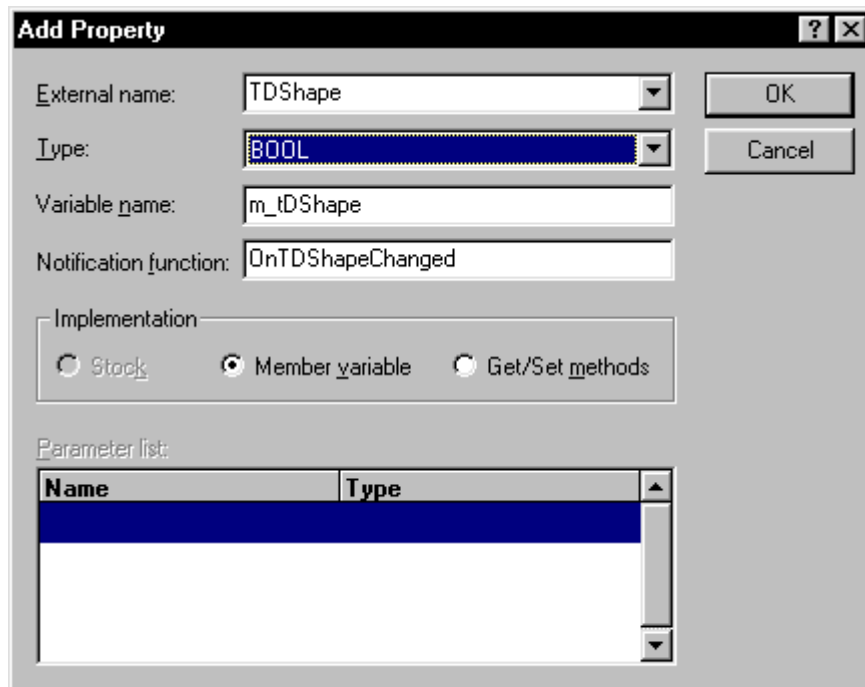


Рис. 22.9. Добавление свойства TDShape в окне AddProperty

4. В поле **Externalname** введите имя свойства TDShape.
5. В группе опций **Implementation** (тип реализации) выберите переключатель **Member variable** (значение свойства будет храниться в переменной-члене).
6. В списке **Type** (тип свойства) выберите тип данных BOOL. Обратите внимание, что в ходе выполнения предыдущих установок в поле **Notificationfunction** (функция обработки изменений) автоматически появилась запись OnTDShapeChanged, а для переменной-члена (поле **Variable name**) было подобрано имя m_tDShape, также автоматически.
7. Щелкните на кнопке **OK**, чтобы принять заданные установки и вернуться к окну **MFC ClassWizard**.
8. Еще раз щелкните на кнопке **AddProperty**, и перед вами повторно откроется диалоговое окно **AddProperty**.
9. В этот раз в списке **Externalname** выберите свойство BackColor.
10. В группе опций **Implementation** выберите переключатель **Stock** (базовое свойство).
11. Щелкните на кнопке **OK**, с тем чтобы вернуться к диалоговому окну **MFC ClassWizard** (рис. 22.10).

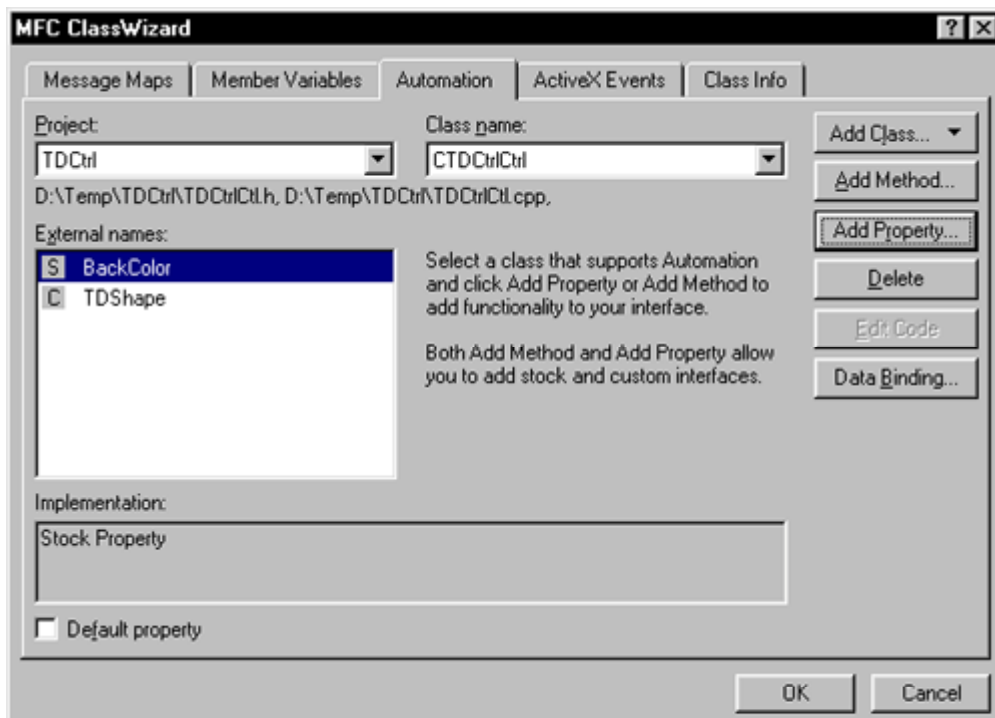


Рис. 22.10. Мастер ClassWizard добавляет в класс TDCtrlCtrl новые свойства - BackColor и TDShape

12. Щелкните на кнопке OK, и мастер ClassWizard внесет изменения в код элемента управления.

Мастер классов добавит в файл TDCTRLCTL.Нобъявление функции OnTDShapeChanged () и переменной m_tDShape, а в файл TDCTRLCTL.CPP добавит стандартную реализацию указанной функции и поместит в схему диспетчеризации соответствующие макросы.

Все перечисленные изменения автоматически вносились мастером классов. Теперь наступает наш черед производить изменения.

Возвращаемся к файлу TDCTRLCTL.CPP

В следующем листинге показаны изменения, вносимые в файл TDCTRLCTL.CPP (выделены полужирным шрифтом).

```

////////////////////////////////////
// CTDCtrlCtrl::OnDraw- Функция отображения
void CTDCtrlCtrl::OnDraw(CDC* pdc, const CRect SrcBounds,
const CRect srcInvalid) {
    CBrush* pOldBrush;
    CBrush NewBrush;
    CPen* pOldPen;
    CPen NewPen;
    pdc->FillRect(rcBounds, CBrush::FromHandle((HBRUSH)
GetStockObject(WHITE_BRUSH)));
    NewPen.CreatePen(PS_SOLID, 3, RGB(0, 0, 0));
    pOldPen = (CPen*)pdc->SelectObject(SNewPen),
    // Создание желтой кисти
    NewBrush.CreateSolidBrush(RGB(255, 255, 0)); pOldBrush = (CBrush*)pdc-
>SelectObject(SNewBrush);
    // Рисование прямоугольника с заливкой
    pdc->Rectangle(rcBounds);

```

```

pdc->SelectObject(pOldPen);
pdc->SelectObject(pOldBrush);
}

```

Действие функции `Rectangle()` приводит к тому, что элемент управления принимает вид прямоугольника, закрашенного в желтый цвет.

Реакция на события мыши

В данном параграфе будет показано, как заставить элемент управления `TDCtrl` реагировать на действия, осуществляемые с помощью мыши. Мы сделаем так, что элемент управления после щелчка на нем указателем мыши изменит свой цвет на светло-серый и в нем отобразятся системное время и дата.

Ниже перечислены действия, которые необходимо выполнить для осуществления намеченного плана.

1. В диалоговом окне **MFC ClassWizard** перейдите на вкладку **Automation**.
2. В списке **Classname** выберите класс `CTDCtrlCtrl`.
3. С помощью кнопки **AddProperty** откройте диалоговое окно **AddProperty**.
4. В поле **External names** введите `HitTDCtrl`.
5. В группе опций **Implementation** выберите переключатель **Member variable**.
6. В списке **Type** выделите тип данных `ole^color` и очистите поле **Notification function**.
7. Щелкните на кнопке **OK**, чтобы принять сделанные установки и вернуться к диалоговому окну **MFC ClassWizard** (рис. 22.11).
8. Перейдите на вкладку **MessageMaps**.
9. В списке **Classname** выберите класс `CTDCtrlCtrl`.
10. В списке **Messages** выберите сообщение `wm_lbuttondown`.
11. Щелкните на кнопке **AddFunction**.
12. Повторите действия, указанные в пунктах 10 и 11, по отношению к сообщению `wm_lbuttonup`. Теперь ваше диалоговое окно должно выглядеть так, как показано на рис. 22.12.

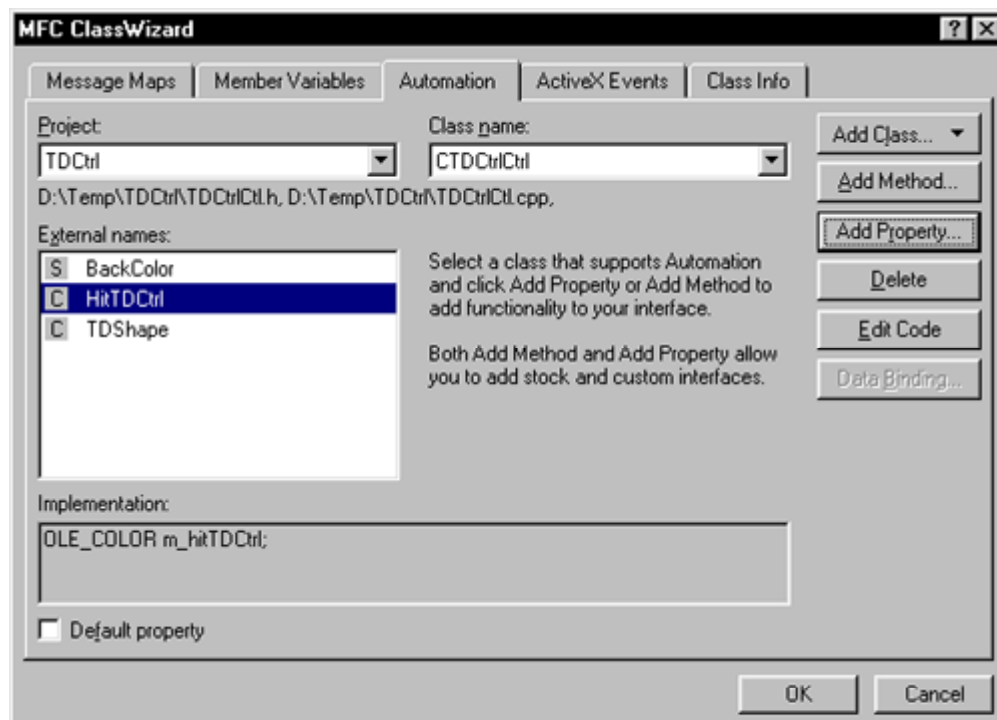


Рис. 22.11. Свойство `HitTDCtrl` позволит элементу управления реагировать на события мыши

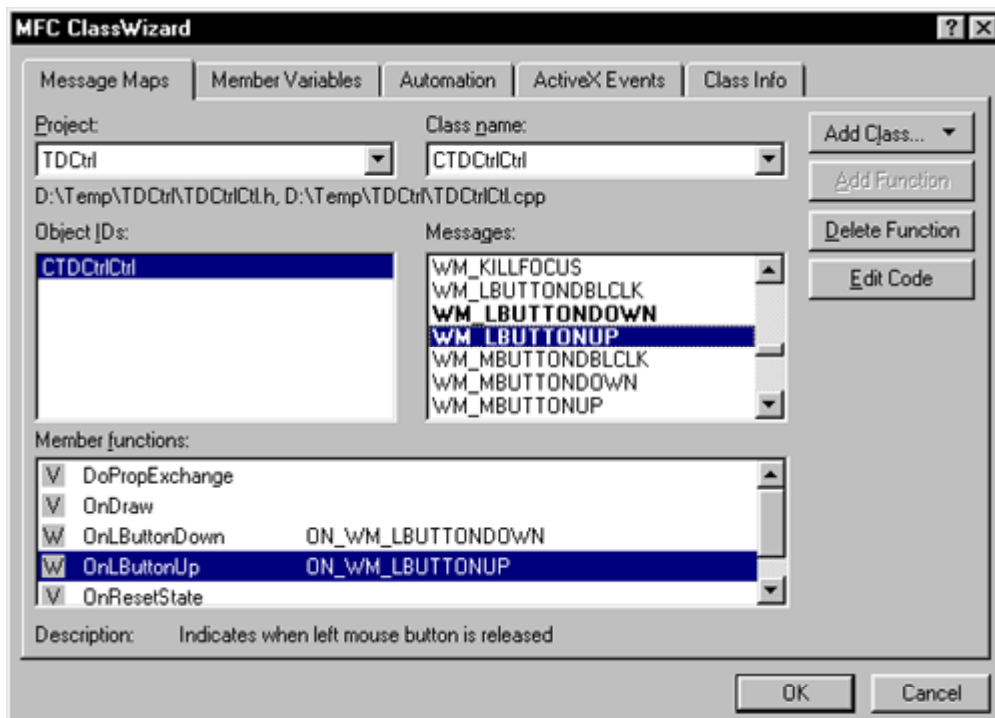


Рис. 22.12. Добавление обработчиков событий мыши

13. Щелкните на кнопке ОК, и мастер ClassWizard внесет изменения в код элемента управления.

Мастер классов автоматически добавит в файл TDCTRLCTL.H объявление переменной `m_hitTDCtrl`, а в файл TDCTRLCTL.CPP — стандартные реализации обработчиков `OnLButtonDown()` и `OnLButtonUp()`.

Теперь нужно самостоятельно добавить код, определяющий реакцию элемента управления на соответствующие события.

Файл TDCTRLCTL.H

В данный файл заголовков, сразу после объявления деструктора, следует поместить прототип новой функции `HitTDCtrl()`, предназначенной для изменения цвета элемента управления в ответ на щелчок мышью.

```
// Реализация
protected:
~CTDCtrlCtrl();
void HitTDCtrl(CDC* pdc); // изменение цвета
```

Далее нужно написать код этой функции.

Возвращаемся к файлу TDCTRLCTL.CPP

При щелчке левой кнопкой мыши на элементе управления должен измениться цвет его прямоугольной области. Частично это событие обрабатывается функцией `DoPropExchange()`. Ниже показана исправленная версия данной функции в Любав-ленной строкой, которая выделена полужирным шрифтом:

```
////////////////////////////////////
// CTDCtrlCtrl::DoPropExchange— Поддержка постоянства элемента
void CTDCtrlCtrl::DoPropExchange(CPropExchange* pPX) {
ExchangeVersion(pPX, MAKELONG(_wVerMinor, _wVerMajor));
COleControl::DoPropExchange(pPX);
}
```

```
// TODO: Поместите здесь вызовы PX_-функций для каждого
//      постоянного пользовательского свойства. ;
// Цвет элемента управления меняется на светло-серый
PX_Color(pPX, _T("HitTDCtrl"), m_hitTDCtrl, RGB(200, 200, 200)); }
```

Эта функция отвечает за инициализацию переменной m_hitTDCtrl, которой присваивается значение, соответствующее светло-серому цвету. Функция px_color () осуществляет обмен данными между переменной m_hitTDCtrl и свойством HitTDCtrl.

Мастер классов создал стандартные реализации функций OnLButtonDown() и OnLButtonUp(). Ниже показано, какие изменения вносятся в эти функции.

Функция OnLButtonDown() получает контекст устройства для рисования и вызывает функцию HitTDCtrl(), осуществляющую вывод текущих значений времени и даты. После отпускания левой кнопки мыши функция OnLButtonUp() вызывает функцию InvalidateControl(), что является сигналом системе послать элементу управления сообщение wm_paint.

```
void CTDCtrlCtrl::OnLButtonDown(DINT nFlags, CPoint point) {
// TODO: Здесь добавьте собственный код обработчика
CDC* pdc;
// Отображение даты к времени
pdc = GetDC() ;
HitTDCtrl(pdc); ReleaseDC(pdc);
ColeControl::OnLButtonDown(nFlags, point); }
void CTDCtrlCtrl::OnLButtonUp(OINT nFlags, CPoint point)
{
// TODO: Здесь добавьте собственный код обработчика
InvalidateControl();
ColeControl::OnLButtonUp(nFlags, point); }
Ниже показан код функции HitTDCtrl(), который следует добавить в конец
файла TDCTRLCTL.CPP:
void CTDCtrlCtrl::HitTDCtrl(CDC* pdc)
{
CBrush* pOldBrush;
CBrush hitBrush (TranslateColor(m_hitTDCtrl));
CRect re;
TEXTMETRIC tm;
struct tm *date_time;
time_t timer;
// Устанавливается прозрачный фон pdc->SetBkMode(TRANSPARENT);
GetClientRect(re);
pOldBrush = pdc->SelectObject(ShitBrush);
// Рисование прямоугольника с заливкой pdc->Rectangle(re);
// Получение даты и времени time(stimer);
date_time = localtime(stimer);
const CStringS strtime = asctime(date_time);
// Получение информации о шрифте и вывод надписи
pdc->GetTextMetrics (J.tm); pdc->SetTextAlign(TA_CENTER | TA_TOP);
pdc->ExtTextOut((re.left + re.right) /2,
(re.top + re.bottom - tm.tmHeight) /2,
ETO_CLIPPED, re, strtime,
strtime.GetLengthO - 1, NULL);
pdc->SelectObject(pOldBrush);
}
```

Цвет кисти определяется на основании переменной m_hitTDCtrl, которая была модифицирована в функции DoPropExchange(). Заливке подвергается вся область элемента

управления. Значения даты и времени запрашиваются с применением стандартных функций языка C — `time()` и `localtime()`.

Тестирование элемента управления TDCtrl

Для тестирования окончательной версии элемента управления TDCtrl воспользуемся программой Microsoft Word. Вызовите окно вставки объектов и выберите из списка элемент управления TDCtrl. После вставки в документ элемент управления можно масштабировать (рис. 22.13).

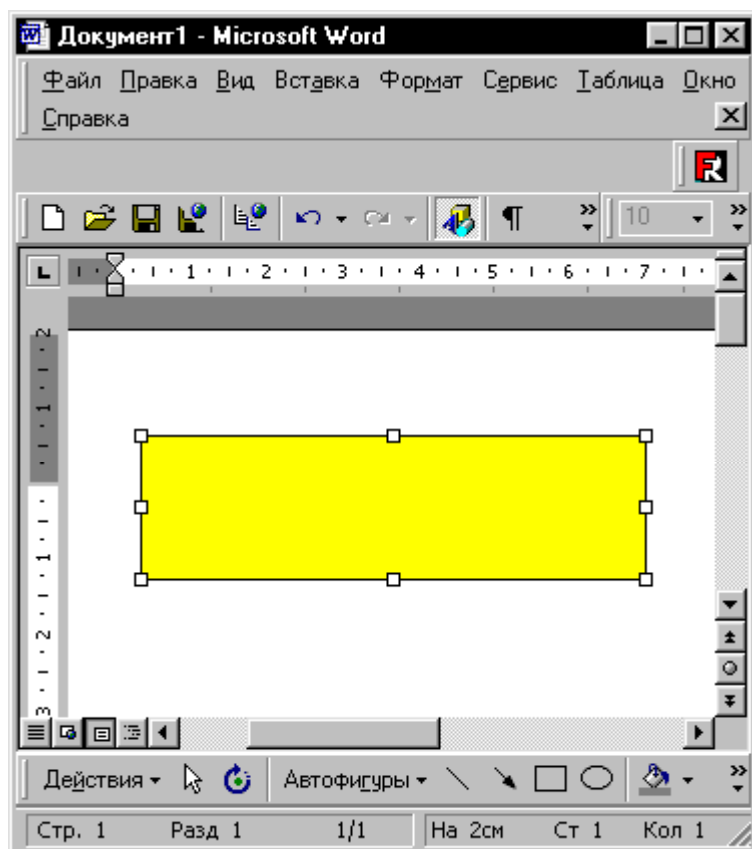


Рис. 22.13. Измененные размеры элемента управления TDCtrl

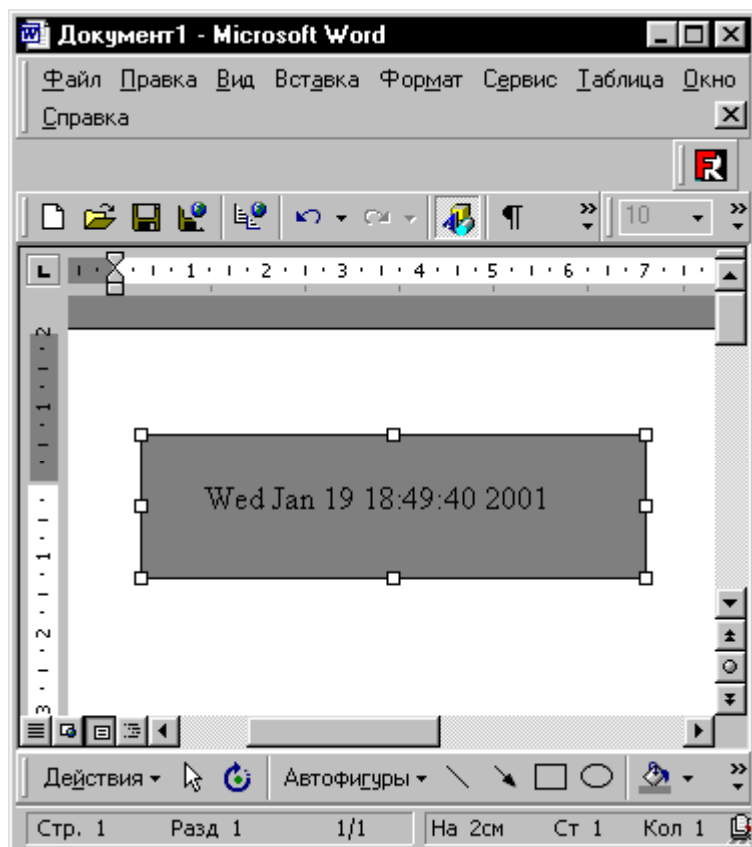


Рис. 22.14. После щелчка мыши на элементе управления в нем отображаются время и дата

Теперь наведите указатель мыши на элемент управления и щелкните левой кнопкой мыши, в результате чего желтый цвет прямоугольной области будет заменен серым и в ней отобразятся текущие значения времени и даты (рис. 22.14).

Глава 23. COM и ATL

- Создание ATL-проекта Polygon
 - Модификация шаблона
 - Тестирование элемента управления ATL на Web-странице

В предыдущих двух главах говорилось о том, что технологии OLE и ActiveX основаны на модели компонентных объектов — COM. Как вы помните, COM определяет механизмы предоставления объектом своих свойств и методов, а также принципы межадачного взаимодействия объектов и контейнеров. В этой главе вы познакомитесь с библиотекой активных шаблонов — ATL (ActiveTemplateLibrary), с помощью которой можно легко создавать COM-объекты, элементы управления ActiveX и многое другое. ATL также содержит встроенные средства поддержки большинства базовых интерфейсов COM.

Мы создадим простое ATL-приложение, в котором будут совмещены возможности сгенерированного шаблонного проекта Polygon и элемента управления ActiveX, разработанного нами в предыдущей главе. Затем мы встроим полученный COM-объект в HTML-документ, который можно просматривать с помощью браузера Internet Explorer.

Создание ATL-проекта Polygon

ATL-проекты создаются с помощью мастера ATLCOMAppWizard. Делается это следующим образом.

1. В окне компилятора VisualC++ в меню File выберите команду New и в открывшемся диалоговом окне перейдите на вкладку Projects.
2. Выберите элемент ATLCOMAppWizard.
3. В качестве имени проекта введите Polygon (рис. 23.1).

После щелчка на кнопке OK откроется окно мастера ATLCOMAppWizard (рис. 23.2).

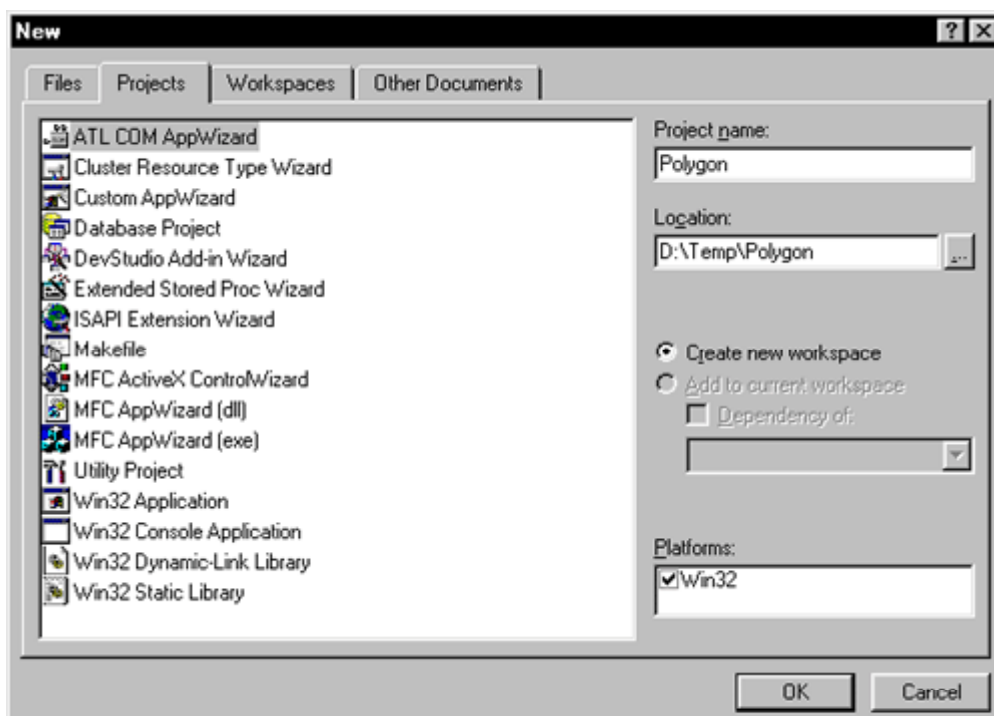


Рис. 23.1. Выбор типа проекта

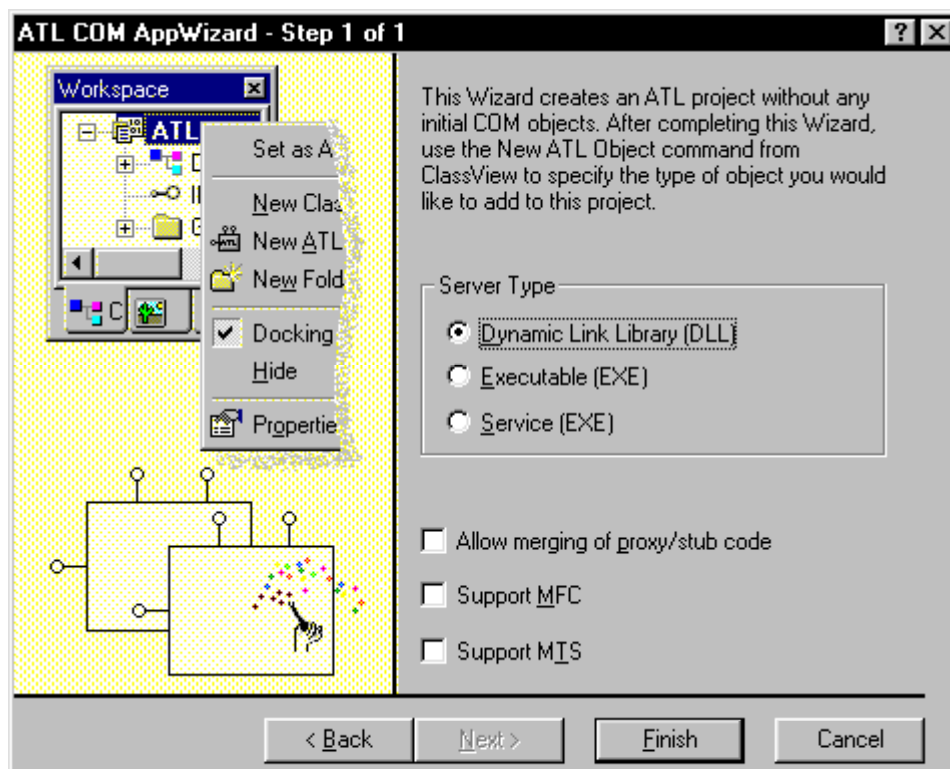


Рис. 23.2. Окно мастера ATL-проекта

В группе опций **ServerType** установите переключатель **DynamicLinkLibrary (DLL)** и щелкните на кнопке **Finish**. Откроется окно **NewProjectInformation** с итоговой информацией о создаваемом ATL-проекте (рис. 23.3).

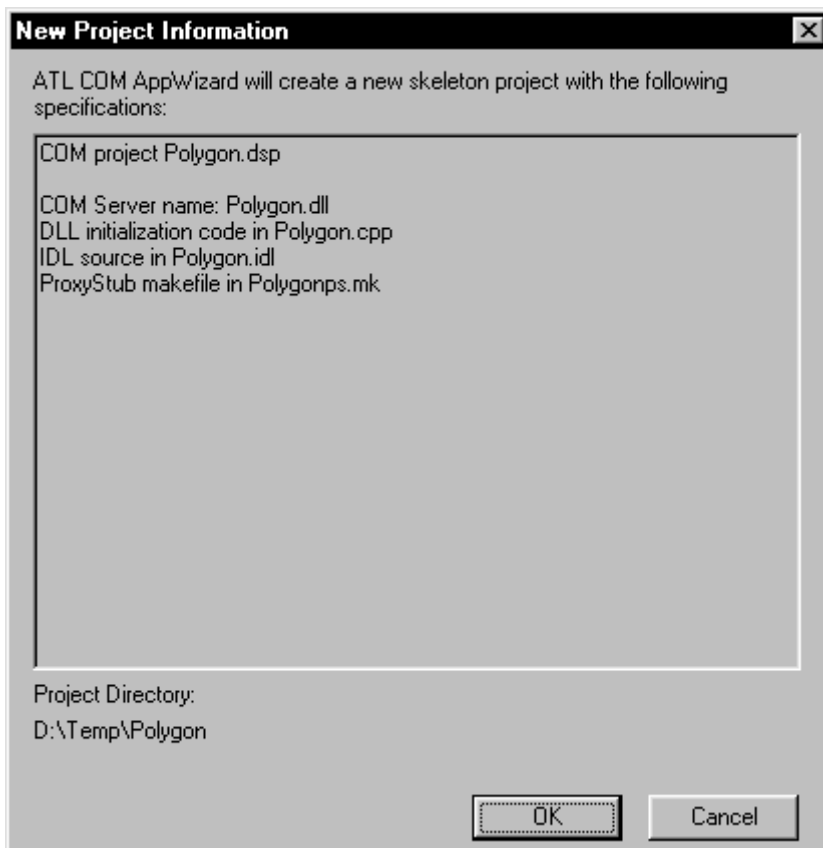


Рис. 23.3. Диалоговое окно **New Project Information**

Щелкните на кнопке **OK**, чтобы запустить процесс генерирования кода программы.

Пока что мастер создал лишь оболочку проекта, в которую необходимо добавить один или несколько элементов управления. Это можно сделать с помощью мастера ATL-объектов, который вызывается по команде **NewATLObject...** из меню **Insert**(рис. 23.4).

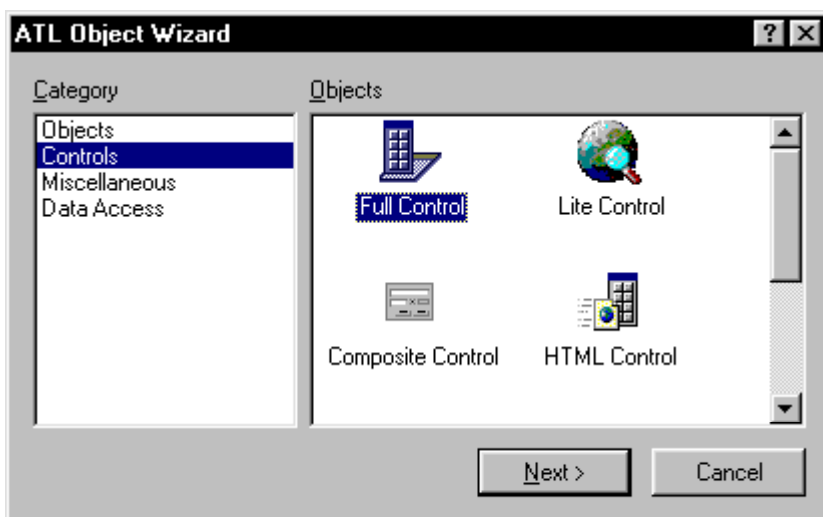


Рис. 23.4. Мастер **ATL Object Wizard** позволяет добавлять в ATL-проект различные элементы управления

В окне мастера в списке **Category** выберите элемент **Controls**, затем в списке **Objects**— элемент **FullControl** и щелкните на кнопке **Next**. В открывшемся диалоговом окне вам представится возможность задать различные параметры конфигурации создаваемого

элемента управления. Сначала на вкладке **Names** в поле **ShortName** введите имя объекта — **PolyCtl**. Все остальные поля будут автоматически инициализированы самим мастером (рис. 23.5).

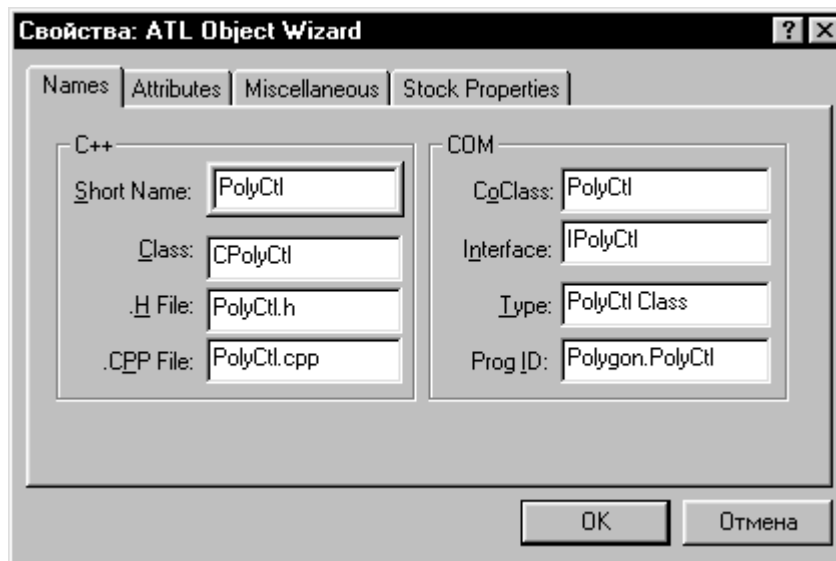


Рис. 23.5. В окне свойств можно установить характеристики элемента управления

Теперь перейдите на вкладку **Attributes** и установите опции **Support ISupportErrorInfo** и **Support Connection Points** (рис. 23.6).



Рис. 23.6. На вкладке **Attributes** устанавливаются служебные опции

Далее активизируйте вкладку **StockProperties** и выберите свойство **FillColor**, которое будет поддерживаться элементом управления (рис. 23.7).

Щелкните на кнопке **OK**, после чего мастер добавит в проект код элемента управления.

Процесс завершен, и теперь в меню **Build** можно выбрать команду **Build** или **RebuildAll** для компиляции проекта. По окончании компиляции задайте в меню **Tools** команду **ActiveXControlTestContainer**, чтобы протестировать созданный элемент управления.

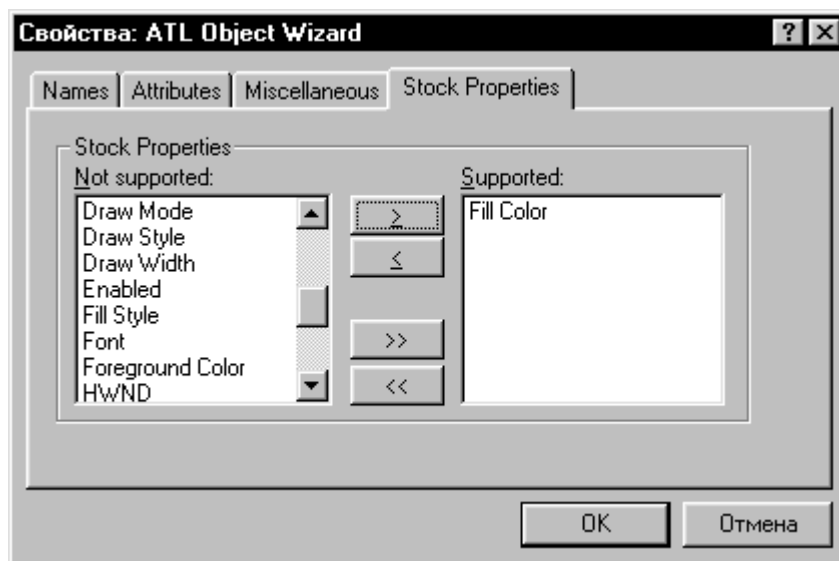


Рис. 23.7. На вкладке **Stock Properties** выбираются базовые свойства, поддерживаемые элементом управления

В окне контейнера в меню **Edit** выберите команду **InsertNewControl**, в открывшемся диалоговом окне **InsertControl** найдите класс **PolyCtl**, после чего нажмите кнопку **OK**.

Этот элемент управления не будет выполнять никаких функций до тех пор, пока мы не свяжем с ним различные свойства и события.

Модификация шаблона

Далее необходимо внести некоторые изменения в сгенерированный мастером **ATLCOMAppWizard** шаблон проекта **Polygon**.

Добавление свойства

Свойства элемента управления реализуются посредством интерфейса **IPolyCtl**. Чтобы добавить новое свойство, в окне компилятора **VisualC++** перейдите на вкладку **ClassView**, выберите интерфейс **IPolyCtl** и щелкните правой кнопкой мыши, после чего в открывшемся контекстном меню выберите команду **AddProperty....** На экране появится диалоговое окно **AddPropertytoInterface** (рис. 23.9).

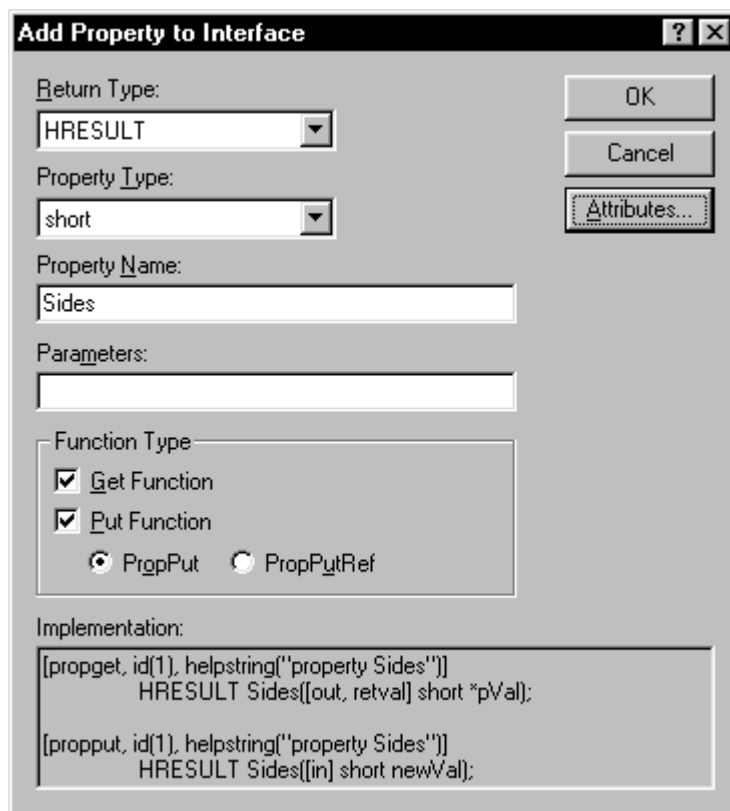


Рис. 23.9. В окне **Add to Property to Interface** можно добавить к элементу управления новое свойство

Из списка **PropertyType** выберите тип данных **short**, затем в поле **PropertyName** введите имя свойства **Sides** и щелкните на кнопке **OK**. Компилятор MIDL (программа, которая создает файлы с расширением IDL) сгенерирует методы `get_Sides()` и `put_sides()`, управляющие получением и установкой значения данного свойства. Прототипы этих функций добавляются в файл **POLYCTL.H**, а их базовые реализации — в файл **POLYCTL.CPP**. Мы к ним еще вернемся.

Реализация точек подключения

Для нашего проекта также необходим интерфейс точек подключения (**connectionpoints**), формирующий канал передачи сообщений между элементом управления и его контейнером. COM-объект может иметь несколько точек подключения и, кроме того, должен реализовывать интерфейс контейнера точек подключения — **iConnectionPointContainer**.

Прежде всего нам нужно добавить к интерфейсу диспетчеризации событий **_IPolyCtlEvents** два метода — `clickin()` и `ClickOut()`, которые сигнализируют о том, что щелчок мышью выполнен соответственно внутри и вне многоугольника, рисуемого в элементе управления. Перейдите на вкладку **ClassView**, щелкните правой кнопкой мыши на элементе **_IPolyCtlEvents** и выберите из контекстного меню команду **AddMethod**. В открывшемся диалоговом окне **AddMethodtoInterface** в списке **ReturnType** укажите тип возвращаемого значения **void**, в поле **MethodName** введите имя метода `Clickin`, а в поле **Parameters** задайте описание входных параметров `x` и `y`, как показано на рис. 23.10. Повторите эту процедуру для метода `ClickOut`.

Строки с описанием данных методов будут добавлены в файл **POLYGON.IDL** — библиотеку типов элемента управления. Прежде чем переходить к следующему шагу, скомпилируйте библиотеку типов, щелкнув на имени указанного файла на вкладке **FileView** и выбрав в контекстном меню команду **CompilePOLYGON.IDL**.

Далее на вкладке **ClassView** щелкните правой кнопкой мыши на элементе **CPolyCtl** и выберите из контекстного меню команду **ImplementConnectionPoint**. В открывшемся одноименном диалоговом окне (рис. 23.11) должна содержаться единственная вкладка **POLYGONLib** (это название библиотеки типов элемента управления). В списке **Interfaces** перечисляются интерфейсы диспетчеризации, описанные в данной библиотеке. В нашем случае такой

интерфейс один: `_IPolyCtlEvents`. В поле **Filename** указано имя файла (`POLYGONCP.H`), в который будет помещен специальный код, управляющий транспортировкой сообщений между элементом управления и контейнером. Поставьте метку напротив интерфейса `_IPolyCtlEvents` и нажмите кнопку **OK**

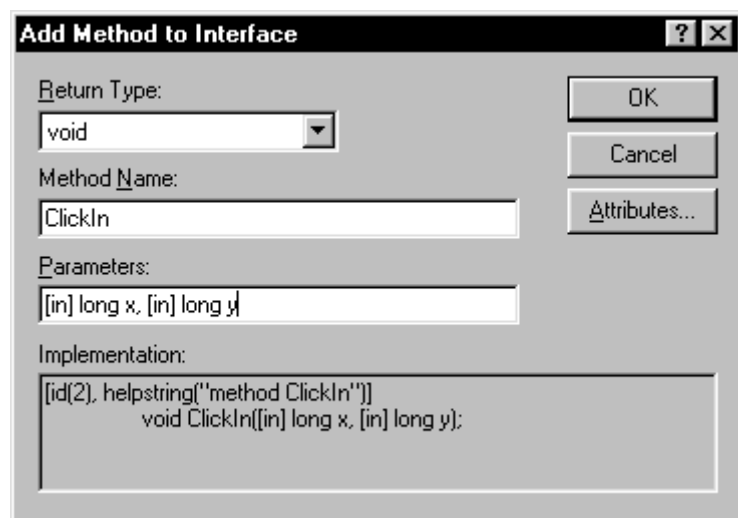


Рис. 23.10. Окно **Add Method to Interface**

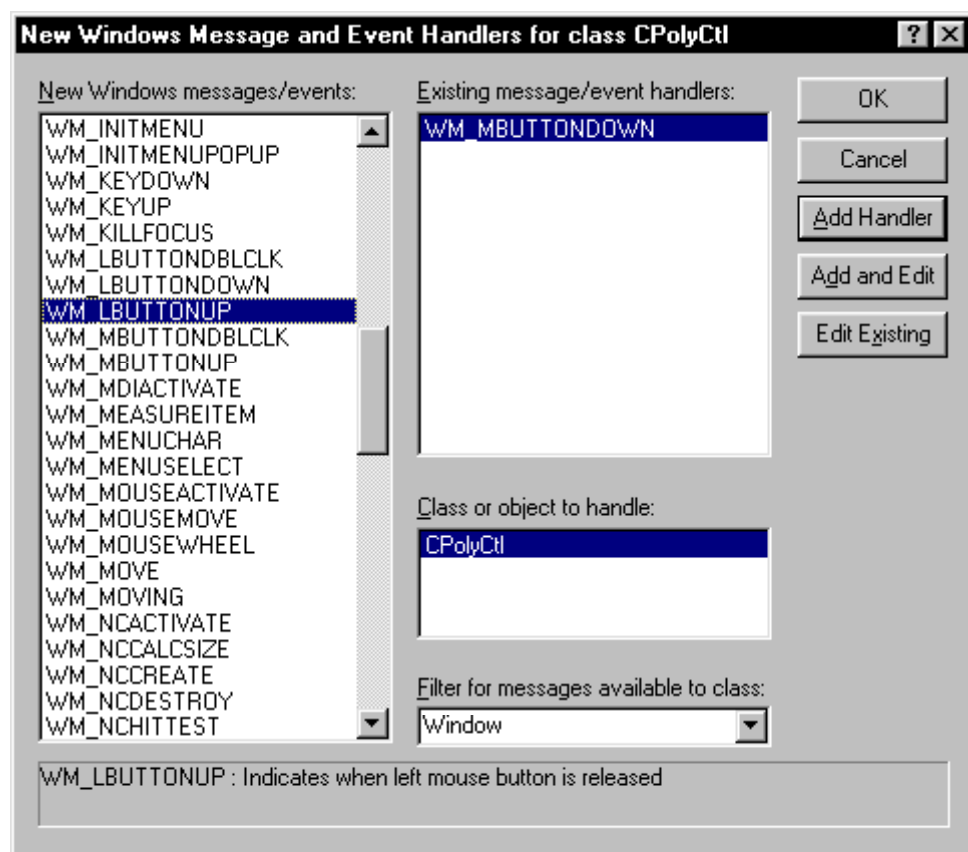


Рис. 23.11. Окно **Implement Connection Point**

В результате выполненных действий будет создан файл `POLYGONCP.H`, содержащий реализацию класса `CProxy_IPolyCtlEvents`, потомка класса `IConnectionPointImpl`, и двух его методов, `Fire_ClickIn()` и `Fire_ClickOut()`, управляющих вызовом упоминавшихся ранее методов

ClickIn() и ClickOut()). В файле POLYCTL.H в список предков класса CPolyCtl будет добавлен класс CProxy_IPolyCtlEvents, а в схему COM-интерфейсов будет помещена запись

```
COM_INTERFACE_ENTRY_IMPL(IConnectionPointContainer)
```

Кроме того, будет обновлена схема точек подключения класса CPolyCtl:

```
BEGIN_CONNECTION_POINT_MAP(CPolyCtl)
CONNECTION_POINT_ENTRY(IID_IPropertyNotifySink)
CONNECTION_POINT_ENTRY(DIID_I_PolyCtlEvents)
END_CONNECTION_POINT_MAP()
```

Добавление обработчика сообщения

Обработчик сообщения `wm_lbuttondown` необходим для определения момента, когда пользователь щелкает левой кнопкой мыши на элементе управления. Перейдите на вкладку **ClassView**, щелкните правой кнопкой мыши на элементе CPolyCtl и выберите из контекстного меню команду **AddWindowsMessageHandler**. В открывшемся диалоговом окне в левом списке выберите сообщение `wm_lbuttondown`, щелкните на кнопке **AddHandler** и нажмите кнопку **OK**.

В результате в файл POLYCTL.H будет добавлена стандартная реализация метода `OnLButtonDown()`, а в схеме сообщений класса CPolyCtl появится новая запись:

```
BEGIN_MSG_MAP(CPolyCtl)
CHAIN_MSG_MAP(CComControl<CPolyCtl>)
DEFAULT_REFLECTION_HANDLER()
MESSAGE_HANDLER(WM_LBUTTONDOWN, OnLButtonDown)
END_MSG_MAP()
```

Внесение изменений в файл POLYCTL.H

Следующим шагом будет внесение некоторых изменений в файл POLYCTL.H, текст которого показан ниже. Дополнения и изменения выделены полужирным шрифтом.

```
// PolyCtl.h: Объявление класса CPolyCtl
#ifndef _POLYCTL_H_
#define _POLYCTL_H_
#include <math.h>
#include "resource.h" // основные константы
#include <atlctl.h>
#include "PolygonCP.h"
////////////////////////////////////
ll CPolyCtl
class ATL_NO_VTABLE CPolyCtl :
public CComObjectRootEx<CComSingleThreadModel>,
public CStockPropImpKCPolyCtl, IPolyCtl, SIID_IPolyCtl,
SLIBID_POLYGONLib>,
public CComControl<CPolyCtl>,
public IPersistStreamInitImpl<CPolyCtl>,
public IOleControlImpl<CPolyCtl>,
public IOleObjectImpl<CPolyCtl>,
public IOleInPlaceActiveObjectImpl<CPolyCtl>,
public IViewObjectExImpKCPolyCtl>,
public IOleInPlaceObjectWindowlessImpl<CPolyCtl>,
public ISupportErrorInfo,
public IConnectionPointContainerImpl<CPolyCtl>,
public IPersistStorageImpl<CPolyCtl>,
public ISpecifyPropertyPagesImpl<CPolyCtl>,
public IQuickActivateImpl<CPolyCtl>,
public IDataObjectImpl<CPolyCtl>,
```

```

public IProvideClassInfo2Impl<SCLSID_PolyCtl, SDIID_IPolyCtlEvents,
&LIBID_POLYGONLib>,
public IPropertyNotifySinkCP<CPolyCtl>,
public CComCoClass<CPolyCtl, SCLSID_PolyCtl>
public CProxy_IPolyCtlEvents< CPolyCtl > { public:
CPolyCtl ()
{
m_nSides =6;          // по умолчанию строится шестиугольник
m_clrFillColor = RGB(0xFF, 0xFF, 0); // желтый цвет заливки
}
DECLARE_REGISTRY_RESOURCEID(IDR_POLYCTL) DECLARE_PROTECT_FINAL
CONSTRUCT))
BEGIN_COM_MAP(CPolyCtl)
COM_INTERFACE_ENTRY(IPolyCtl)
COM_INTERFACE_ENTRY(IDispatch)
COM_INTERFACE_ENTRY(IViewObjectEx)
COM_INTERFACE_ENTRY(IViewObject2)
COM_INTERFACE_ENTRY(IViewObject)
COM_INTERFACE_ENTRY(IDlelnPlaceObjectWindowless)
COM_INTERFACE_ENTRY(IoleInPlaceObject)
COM_INTERFACE_ENTRY2(IQleWindow, IQlelnPlaceObjectWindowless)
COM_INTERFACE_ENTRY(IoleInPlaceActiveObject)
COM_INTERFACE_ENTRY(IoleControl)
COM_INTERFACE_ENTRY(IQleObject)
COM_INTERFACE_ENTRY(IPersistStreamInit)
COM_INTERFACE_ENTRY2(IPersist, IPersistStreamInit)
COM_INTERFACE_ENTRY(ISupportErrorInfo)
COM_INTERFACE_ENTRY(IConnectionPointContainer)
COM_INTERFACE_ENTRY(ISpecifyPropertyPages)
COM_INTERFACE_ENTRY(IQuickActivate)
COM_INTERFACE_ENTRY(IPersistStorage)
COM_INTERFACE_ENTRY(IDataObject)
COM_INTERFACE_ENTRY(IProvideClassInfo)
COM_INTERFACE_ENTRY(IProvideClassInfo2)
COM_INTERFACE_ENTRY_IMPL(IConnectionPointContainer)      ^
END_COM_MAP()
BEGIN_PROP_MAP(CPolyCtl)
PROP_DATA_ENTRY("_cx",m_sizeExtent.cx, VT_UI4)
PROP_DATA_ENTRY("_Gy",m_sizeExtent.cy, VT_UI4)
PROP_ENTRY("FillColor",DISPID_FILLCOLOR, CLSID_StockColorPage)
END_PROP_MAP()
BEGIN_CONNECTION_POINT_MAP(CPolyCtl)
CONNECTION_POINT_ENTRY(IID_IPropertyNotifySink)
CONNECTION_POINT_ENTRY(DIID_IPolyCtlEvents)
END_CONNECTION_POINT_MAP ()
BEGIN_MSG_MAP(CPolyCtl)
CHAIN_MSG_MAP(CComControl<CPolyCtl>)
DEFAULT_REFLECTION_HANDLER() END_MSG_MAP ()
// ISupportErrorInfo
STDMETHOD(InterfaceSupportsErrorInfo) (REFIID riid)
{
static const IID* arr[]=
{
&IID_IPolyCtl,

```

```

{
for (int i = 0; i < sizeof(arr)/sizeof(arr[0]); i++)
{
if (InlineIsEqualGDID(*arr[i], riid))
return S_OK; } return S_FALSE;
}.
// IViewObjectEx
DECLARE_VIEW_STATUS(VIEWSTATUS_SOLIDBKGND | VIEWSTATUS_OPAQUE)
// IPolyCtl public:
STDMETHOD(get_Sides) ([out,retval] short *pVal);
STDMETHOD(put_Sides) ([in] short newVal);
HRESULT OnDraw(ATL_DRAWINFOS di);
LRESULT OnLButtonDown(UINT uMsg,
WPARAM wParam, LPARAM lParam, BOOLS bHandled)
OLE_COLOR m_clrFillColor;
short m_nSides; POINT m_arrPoint[10];
};
#endif // _POLYCTL_H_

```

В конструкторе класса CPolyCtl устанавливается, что по умолчанию число сторон многоугольника, рисуемого в элементе управления, равно шести (минимальное — 3, максимальное — 10), а для заливки используется желтый цвет. Для вычисления координат вершин многоугольника потребуются тригонометрические функции, поэтому в проект включается библиотека MATH.H.

Стандартные реализации функций OnDraw() и OnLButtonDown() были удалены из файла, так как мы значительно расширим их и включим в файл POLYCTL.CPP.

Реализация методов элемента управления

Ниже приведен текст файла POLYCTL.CPP. Все дополнения и изменения выделены полужирным шрифтом.

```

// PolyCtl.cpp: Реализация класса CPolyCtl
#include "stdafx.h"
#include "Polygon.h"
#include "PolyCtl.h"
#include <time.h>
#include <string.h>
////////////////////////////////////
// CPolyCtl
HRESULT CPolyCtl: : OnDraw (ATL_DRAWINFO& di)
{
struct tm *date_time;
time_t timer;
static TEXTMETRIC tin;
RECTS rc = *(RECT*)di.prcBounds; HDC hdc = di.hdcDraw;
COLORREF colFore;
HBRUSH hOldBrush, hBrush;
HPEN hOldPen, hPen;
// Приведение переменной m_clrFillColor к типу COLORREF
OleTranslateColor(m_clrFillColor, NULL, colFore);
// Выбор пера и кисти для рисования окружности
hPen = (HPEN)GetStockObject(BLACK_PEN);
hOldPen = (HPEN)SelectObject (hdc, hPen);
hBrush = (HBRUSH)GetStockObject(WHITE_BRUSH);
hOldBrush = (HBRUSH)SelectObject(hdc, hBrush);

```

```

const double pi = 3.14159265358979; POINT ptCenter;
double dblRadiusx = (rc.right - rc.left) / 2;
double dblRadiusy = (rc.bottom - rc.top) / 2;
double dblAngle = 3 * pi / 2;
double dblDiff = 2 * pi / m_nSides;
ptCenter.x = (rc.left + rc.right) / 2;
ptCenter.y = (rc.top + rc.bottom) / 2;
// Вычисление координат вершин
for (int i = 0; i < m_nSides; i++) {
m_arrPoint[i].x = (long) \
(dblRadiusx*cos(dblAngle)+ptCenter.x+0.5);
m_arrPoint[i].y = (long) \
(dblRadiusy*sin (dblAngle) -1-ptCenter.y+0.5) ;
dblAngle += dblDiff; } Ellipse(hdc, rc.left, rc.top, rc.right,
rc.bottom);
// Создание и выбор кисти для заливки многоугольника
hBrush = CreateSolidBrush(colFore);
SelectObject(hdc, hBrush);
Polygon(hdc, Sm_arrPoint[0], m_nSides);
// Вывод даты и времени
time(Stimer);
date_time = localtime(<<timer);
const char* strtime;
strtime = asctime(date_time);
SetBkMode(hdc, TRANSPARENT);
SetTextAlign(hdc, TA_CENTER | TA_TOP); ExtTextOut(hdc, (rc.left +
rc.right)/2,
(rc.top + rc.bottom - tan.tmHeight)/2,
ETO_CLIPPED, Src, strtime, strlen (strtime) -1,NULL);
// Восстановление старых пера и кисти
SelectObject(hdc, hOldPen) ;
SelectObject(hdc, hOldBrush) ;
DeleteObject (hBrush) ;
return S_OK;
LRESULT CPolyCtl : OnLButtonDown (UINT uMsg, WPJWMM wParam,
LPARAM lParam, BOOLS bHandled)
{
HRGNhRgn;
WORDxPos= LOWORD(lParam) ; // положение указателя по горизонтали
WORDyPos= HIWORD(lParam) ; // положение указателя по вертикали
// Создание многоугольника по списку координат вершин
hRgn = CreatePolygonRgn(sm_arrPoint[0] , m_nSides, WINDING);
// Если точка щелчка попадает внутрь многоугольника,
// то генерируется событие ClickIn,
// в противном случае - событие ClickOut
if (PtInRegion(hRgn, xPos, yPos) )
Fire_ClickIn (xPos , yPos) ; else
Fire_ClickOut (xPos , yPos) ;
// Освобождение дескриптора
DeleteObject (hRgn) ;
return 0 ;
}
STDMETHODIMP CPolyCtl::get_Sides(short *pVal) (
*pVal = m_nSides;

```

```

return S_OK; }
STDMETHODIMP CPolyCtl::put_Sides(short newVal)
{
    if (newVal > 2 && newVal < 11) {
        m_nSides = newVal;
        FiifeViewChangeO ;
        return S_OK; } else
        return Error(_T("Must have between 3 and 10 sides"));
    }
}

```

Отображаемые дата и время будут обновляться каждый раз после выполнения щелчка на элементе управления. Метод `put_Sides()` изменен таким образом, чтобы вызывать функцию `FireViewChange()`, которая, в свою очередь, вызывает функцию `invalidateRect()`, обозначающую область перерисовки элемента управления как недействительную. Если этого не сделать, изображение элемента управления не будет обновляться после щелчка на нем мышью.

Добавление страницы свойств

С помощью того же мастера ATL-объектов можно добавить в элемент управления страницу свойств. Для этого выберите в меню `Insert` команду `New ATL Object...` (рис. 23.13).

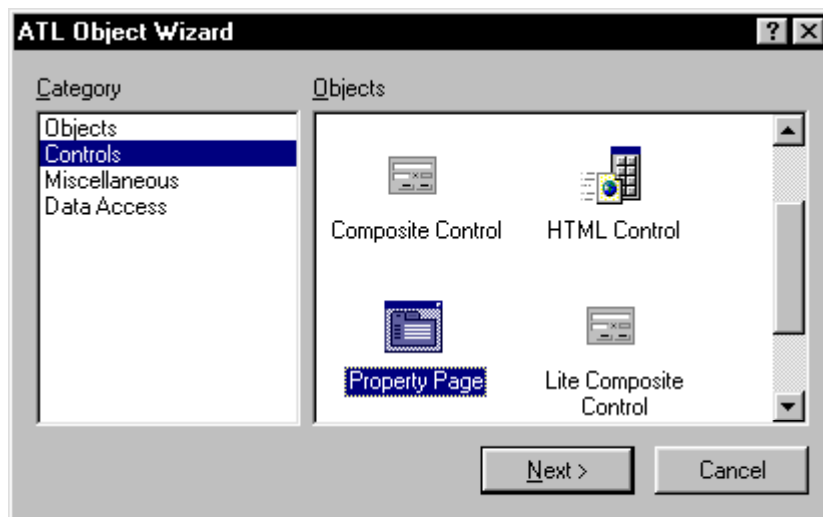


Рис. 23.13. С помощью мастера ATL Object Wizard можно добавить в проект страницу свойств

Выделите категорию `Controls`, в ней — элемент `PropertyPage` и щелкните на кнопке `Next`. В следующем окне мастера можно установить параметры, определяющие работу страницы свойств. На вкладке `Names` поле `ShortName` введите имя объекта — `PolyProp`. Все остальные поля будут заполнены мастером автоматически (рис. 23.14).

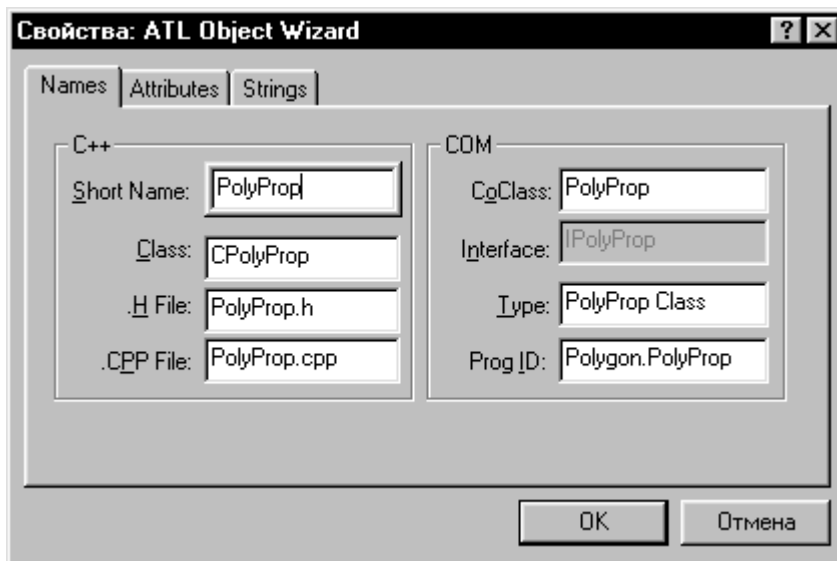


Рис. 23.14. Задание параметров создаваемой страницы свойств

Теперь перейдите на вкладку **Strings** и заполните поля **Title**(текст ярлычка вкладки) и **DocString**(строка описания), а поле **Helpfile** (ассоциированный файл справки) очистите (рис. 23.15).



Рис. 23.15. На вкладке Strings следует заполнить поля Title и Doc String

После щелчка на кнопке **OK** будут созданы новые файлы POLYPROP.H, POLYPROP.CPP и POLYPROP.RGS. Далее необходимо изменить внешний вид страницы свойств. Откройте с помощью вкладки **ResourceView** диалоговое окно с идентификатором IDD_POBYPROP, поменяйте существующую надпись на **Sides**: и добавьте текстовое поле с идентификатором IDC_SIDES(рис. 23.16).



Рис. 23.16. Изменение внешнего вида страницы свойств

Теперь нужно добавить обработчик, управляющий изменением значения поля `idc_sides`. Для этого выполните щелчок правой кнопкой мыши на поле и выберите в контекстном меню команду **Events**. В появившемся окне выберите в списке **Class or object to handle** элемент `IDC_SIDES`, а затем в списке **New Windows Messages/Events**— сообщение `EN_CHANGE`. Щелкните на кнопке **Add Handler**, после чего в открывшемся окне **Add Member Function** будет предложено имя обработчика (рис. 23.17).

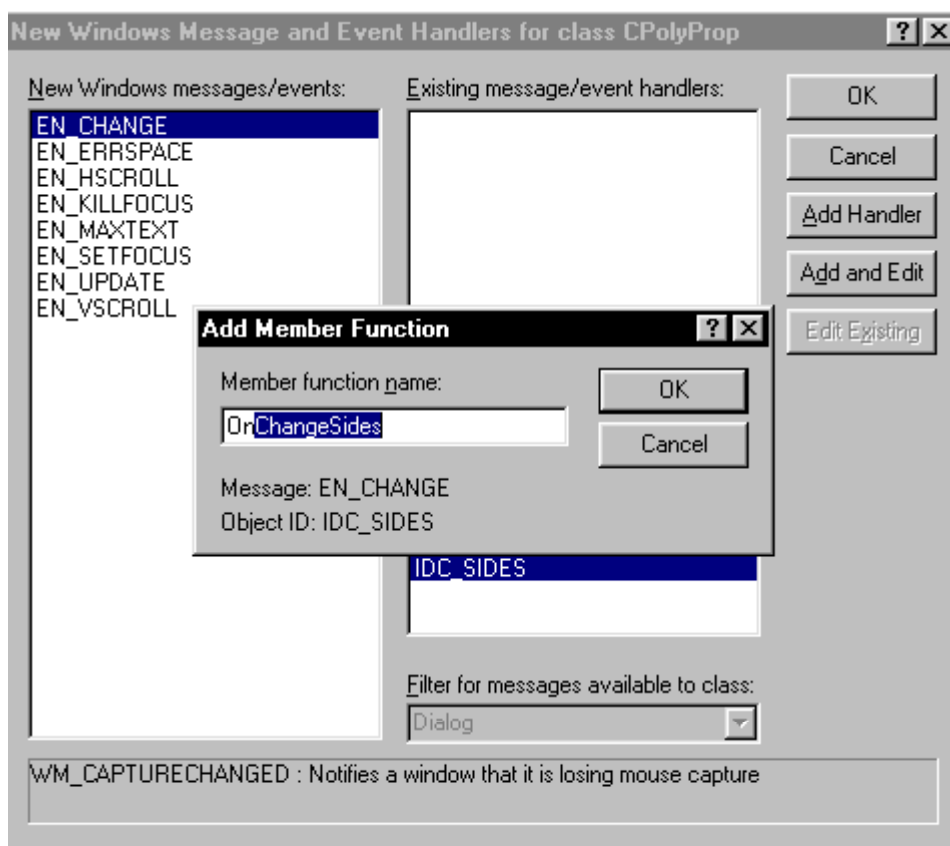


Рис. 23.17. Добавление обработчика сообщений

Примите предлагаемое имя `OnChangeSides` и закройте оба окна. В результате в файл `POLYPROP.H` будет добавлена стандартная реализация данного обработчика, а в схеме сообщений класса `CPolyProp` появится новая запись:

```
BEGIN_MSG_MAP(CPolyProp)
CHAIN_MSG_MAP(IPropertyPageImpKCPolyProp)
COMMAND_HANDLER(IDC_SIDES, EN_CHANGE, OnChangeSides)
END_MSG_MAP()
```

Следующим шагом будет внесение изменений в файл POLYPROP.H:

```
// PolyProp.h: Объявление класса CPolyProp
#ifndef _POLYPROP_H_
#define _POLYPROP_H_
#include "resource.h"          // основные константы
#include "Polygon.h"
EXTERN_C const CLSID CLSID_PolyProp;
////////////////////////////////////
II CPolyProp
class ATL_NO_VTABLE CPolyProp :
public CComObjectRootEx<CComSingleThreadModel>,
public CComCoClass<CPolyProp, &CLSID_PolyProp>,
public IPropertyPageImpl<CPolyProp>,
public CDialogImpl<CPolyProp> { public:
CPolyProp ( ) {
m_dwTitleID = IDSJTITLEPolyProp; m_dwHelpFileID =
IDS_HELPFILEPolyProp; m_dwDocStringID = IDS_DOCSTRINGPolyProp;
}
enura (IDD = IDD_POLYPROP);
DECLARE_REGISTRY_RESOURCEID ( IDR_POLYPROP)
DECLARE_PROTECT_FINAL_CONSTRUCT ( )
BEGIN_COM_MAP (CPolyProp)
-
COM_INTERFACE_ENTRY ( I PropertyPage ) END_COM_MAP ( )
BEGIN_MSG_MAP (CPolyProp)
CHAIN_MSG_MAP ( IPropertyPageImpKCPolyProp>)
COMMAND_HANDLER(IDC_SIDES, EN_CHANGE, OnChangeSides)
END_MSG_MAP ( )
STDMETHOD (Apply) (void) {
OSes_CONVERSION ;
ATLTRACE ( _T ( "CPolyProp: :Apply\n" ) ) ;
for (UINT i = 0; i < m_nObjects; i++)
{
CComQIPtr<IPolyCtl , SIID_IPolyCtl> pPoly (m_ppOnk[i]) ;
short nSides = ( short) GetDlgItemInt (IDC_SIDES) ;
if FAILED (pPoly->put_Sides (nSides) ) {
CComPtr<IErrorInfo o> pError ; CComBSTR      strError;
GetErrorInfo(0, SpError) ;
pError->GetDescription (fistrError) ;
MessageBox(OLE2T(strError) , _T ("Error"),
MB_ICONEXCLAMATION) ;
return E_FAIL; } } m_bDirty = FALSE;
return S_OK; }
LRESULT OnChangeSides(WORD wNotifyCode, WORD wID,
HWND hWndCtl, BOOL& bHandled) {
SetDirty(TRUE);
return 0;
}
};
#endif // _POLYPROP_H_
```

Страница свойств может быть вызвана сразу несколькими клиентами. Для обслуживания всех клиентов в функции Apply() запускается цикл и в нем вызывается метод put_Sides() для каждого клиента, данные которого были введены в текстовое поле.

Страница свойств добавляется в проект с помощью единственной строки в файле POLYCTL.H:

```
BEGIN_PROP_MAP(CPolyCtl)
PROP_DATA_ENTRY("_c.x", m_sizeExtent.ex, VT_UI4)
PROP_DATA_ENTRY("_cy", m_sizeExtent.cy, VT_UI4)
PROPJ3NTRY("FillColor", DISPID_FILLCOLOR, CLSID_StockColorPage)
PRQP_ENTRY("Sides", 1, CLSID_PolyProp) END_PROP_MAP 0
```

Теперь можно приступить к тестированию элемента управления на Web-странице.

Тестирование элемента управления ATL на Web-странице

Мастер ATL-объектов создает исходный элемент управления вместе с тестовым HTML-файлом, который находится в папке проекта. Он называется POLYCTL.HTM и может быть открыт в браузере Microsoft Internet Explorer. С помощью данного файла можно протестировать созданный нами элемент управления. Но прежде в этот файл следует внести изменения, выделенные ниже полужирным шрифтом:

```
<HTML>
<HEAD>
<TITLE>ATL 3.0 test page for object PolyCtl</TITLE>
</HEAD>
<BODY>
<OBJECT ID="PolyCtl"
CLASSID="CLSID:4CBBC676-507F-11D0-B98B-000000000000"> </OBJECT>
<SCRIPT LM><GOAGE="VBScript"> <! —
Sub PolyCtl_ClickIn(x, y)
PolyCtl.Sides = PolyCtl.Sides + 1
End Sub
Sub PolyCtl_ClickOut(x, y)
PolyCtl.Sides = PolyCtl.Sides - 1
End Sub —>
</SCRIPT>
</BODY>
</HTML>
```

Теперь запустите Internet Explorer и откройте в нем файл POLYCTL.HTM. Начальное содержимое Web-страницы показано на рис. 23.18.

Выполните несколько щелчков мышью внутри и вне многоугольника. Как вы убедитесь, число вершин станет автоматически увеличиваться >> уменьшаться, если только не будут достигнуты граничные значения 2 и 11 — в этом случае выдается сообщение об ошибке **Must have between 3 and 10 sides**. На рис. 23.19 показано, как изменится внешний вид элемента управления после двух щелчков внутри многоугольника.

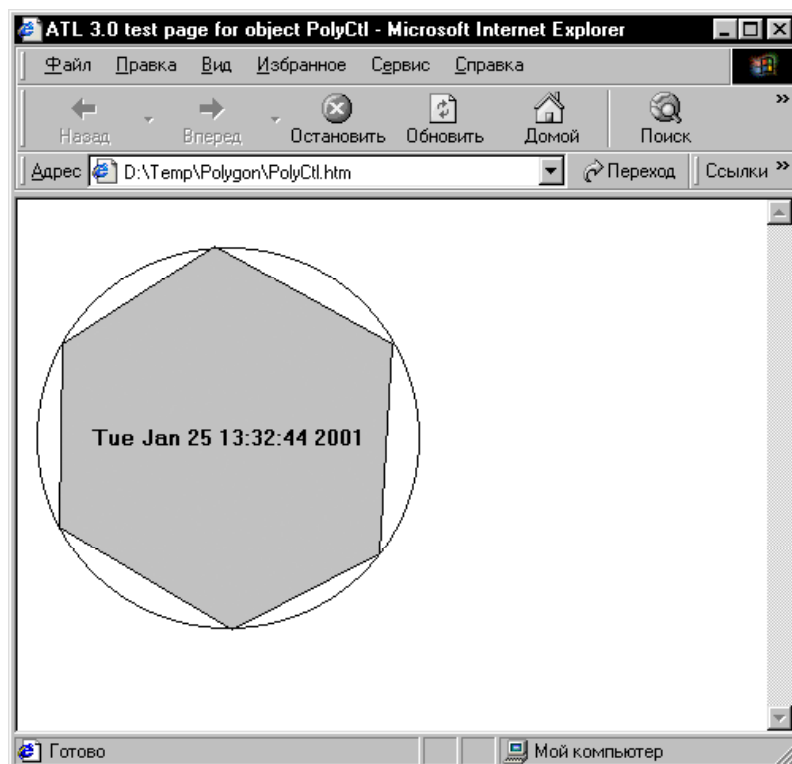


Рис. 23.18. Исходный вид созданного вами элемента управления ATL

Примечание

Убедитесь, что в свойствах Internet Explorer установлен низкий уровень безопасности. По умолчанию задается средний уровень, но в этом случае выполнение сценариев неподписанных элементов управления ActiveX запрещено.

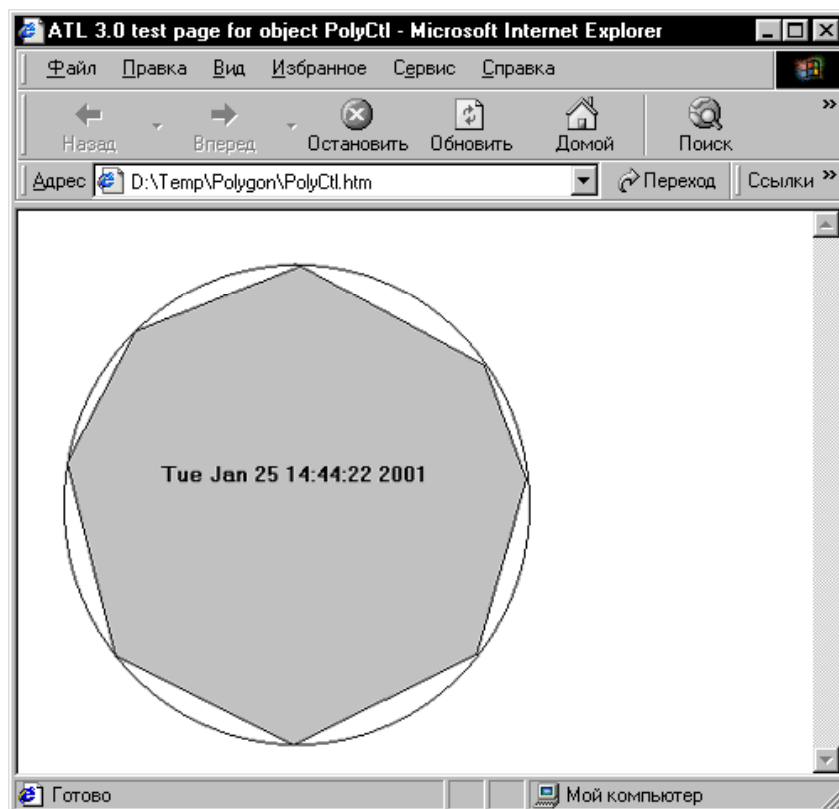


Рис. 23.19. Вид элемента управления после двух последовательных щелчков на нем