ИЗДАТЕЛЬСТВО
БИНОМ

Приглашаем к сотрудничеству авторов и научных редакторов в области электроники и программирования
103473, Москва, а/я 133 Телефоны (095) 973-9062, 973-9063

CLUB Computer
Просто и ясно о Borland C++

Bruneau BABET

Lean & Mean Borland C++

Ilflrady

Бруно БАБЭ

Просто и ясно о Borland C++

Издание третье дополненное

Перевод В. Тимофеева

БИНОМ
Москва 1996

Бруно Бабэ
Просто и ясно о Borland C++: Пер. с англ. — М.: БИНОМ. — 416 с.: ил.
ISBN 1-56686-134-9 ISBN 5-7503-0098-6
Книга представляет собой пособие по программированию на языке C++, ориентированное на использование компилятора Borland C++ 4.x. Особое внимание уделяется нововведениям в ANSI C++, таким, как шаблоны или управление исключениями. Сжатые формальные описания языковых конструкций сопровождаются подробными примерами кода. В книге дается также вводная

информация по среде Borland C++, помогающая пользователю быстро освоиться с компилятором. В конце книги имеется раздел, отражающий изменения в последнем продукте серии Borland C++ 5.0. Книга предназначена как для опытных программистов, переходящих на объектно-ориентированную технологию, так и для начинающих программистов, осваивающих среду Borland C++.

ISBN 1-56686-134-9 ©Издание на русском языке. БИНОМ, 1996

© Художник серии

ISBN 5-7503-0098-6 Н. Лозинская, 1996

Производственно-техническое издание

Бруно Бабэ

ПРОСТО И ЯСНО О BORLAND C++

Тимофеев В.В. Главы 11, 12 На обложке компьютерная графика «Астрель»

Подписано в печать 18.03.96. Формат 70x108 $^{1}/_{16}$,. Печ. л. 26

Бумага типографская. Печать офсетная

Тираж 12 000 *экз.* Заказ 1558

Издательство «БИНОМ», 1996 г.

Москва, Новослободская, 50/1, стр. 1а.

Лицензия на издательскую деятельность № 063367 от 20 мая 1994 г.

Отпечатано с готовых диапозитивов

в полиграфической фирме «Красный пролетарий»

103473, Москва, Краснопролетарская, 16

Содержание

Об авторе

Бруно Бабэ работает в Borland International более трех лет. До того, как присоединиться к группе разработчиков проекта ObjectWindows, он был старшим техническим консультантом при группе поддержки Borland C++.

Благодарности

Спасибо всем вам, работающим в группе поддержки Borland C++, за созданную вами обстановку, в которой идеи текут непрерывным потоком; особенно ценным для меня было участие Чарли Калверта. Спасибо вам, Нэн Боресссон из Borland International и Сунтар Висувалингам из Brady Publishing; вы очень помогли мне. И самое большое спасибо — моей жене Одетте и нашей маленькой семье: Калипсо (шведская гончая), Шамони (Лабрадор-чау-лайка) и Шоколаду (помесь овчарки).

Бруно Бабэ

*****

Издательство "Бином" выражает благодарность Borland A/O за предоставленную возможность использовать предварительную версию компилятора Borland C++ 4.5 при подготовке к изданию этой книги.

Введение

Эта книга о том, как пользоваться пакетом Borland C++, причем основной упор в ней сделан на особенностях реализации языка C++ в компиляторе Борланда. Материал книги, как таковой, составлен из сжатых описаний языковых конструкций, сопровождаемых отрывками кода, которые иллюстрируют их использование. Кроме того, по ходу изложения в книге даются разъяснения и подсказки, в основном относительно идиом C++, позволяющих наиболее эффективно работать с языком.

Например, в Главе 5 программист-новичок найдет краткое описание оператора new языка C++, являющегося гибким инструментом для выделения динамической памяти. Несколько примеров показывают способы его применения. Раздел, в котором описано, как перегрузить этот оператор, чтобы следить за распределением памяти, вероятно, привлечет внимание более искушенных пользователей. Там же можно найти подсказку относительно синтаксиса, позволяющего вызвать конструктор для уже конструированного объекта — такого рода технические приемы обычно редко бывают описаны в документации.

Как организована эта книга

Начальные главы книги посвящены основным чертам среды Borland C++ и фундаментальным понятиям языков C и C++. Последующие охватывают более сложный материал, включая некоторые недавние добавления к C++:

- Шаблоны в C++
- Обработка исключительных ситуаций в C++
- Информация о типе времени исполнения
- Новые операторы приведения типа

Каждую главу можно читать независимо, хотя примеры могут использовать материал, описанный в книге ранее. Таким образом, лучше всего читать ее, пропуская только то, что вам и так хорошо знакомо.

Глава 1
Borland C++: основы

Компоненты пакета Borland C++ предназначены, прежде всего, для создания и сопровождения исполняемых модулей и библиотек. Он также содержит целый ряд утилит, которые можно применять и отдельно — например, для того, чтобы исследовать структуру файла или нарисовать пиктограмму. Цель данной главы — обеспечить необходимое знакомство с инструментарием Borland C++*.

Первый раздел покажет вам, как создавать исполняемые модули или библиотеки, используя либо интегрированную среду Borland C++, либо инструментальные средства с командной строкой. Следующий раздел содержит информацию, которая облегчит обновление уже существующего кода для Borland C++.

Язык C/C++ меняется, и мы обсуждаем отличия библиотек пакета от тех, которые имелись в ранних версиях компилятора. Этот материал будет полезен и в том случае, если вы планируете переносить код, написанный для других компиляторов C или C++.

Глава не содержит какого-либо учебного материала собственно о языках C или C++. Тем не менее, как новичкам, так и опытным программистам, которым среда Borland C++ пока незнакома, будет полезно иметь ясное представление о том, из каких шагов складывается создание прикладных программ и библиотек при помощи этой системы.

* Более подробные сведения о составе пакета, имеющихся библиотеках и об отличиях в версиях 4.0 и 4.5 приведены в 11 главе. — *Прим. перев.*

Использование инструментария

Сердцевину пакета Borland C++ составляют главные инструменты: компиляторы, библиотекари и редакторы связей. Именно они обрабатывают ваш исходный код, объектные модули и библиотеки, и генерируют конечный продукт — прикладную программу.

Роль компилятора, библиотекаря и редактора связей

Программы на языках C и C++ составляются из исходных файлов (обычно с расширениями .C и .CPP, соответственно для C и C++) и заголовочных файлов (с расширением .H). Рис.1.1 иллюстрирует последовательность шагов и функции компилятора, библиотекаря и редактора связей (компоновщика) в процессе построения простой прикладной программы.

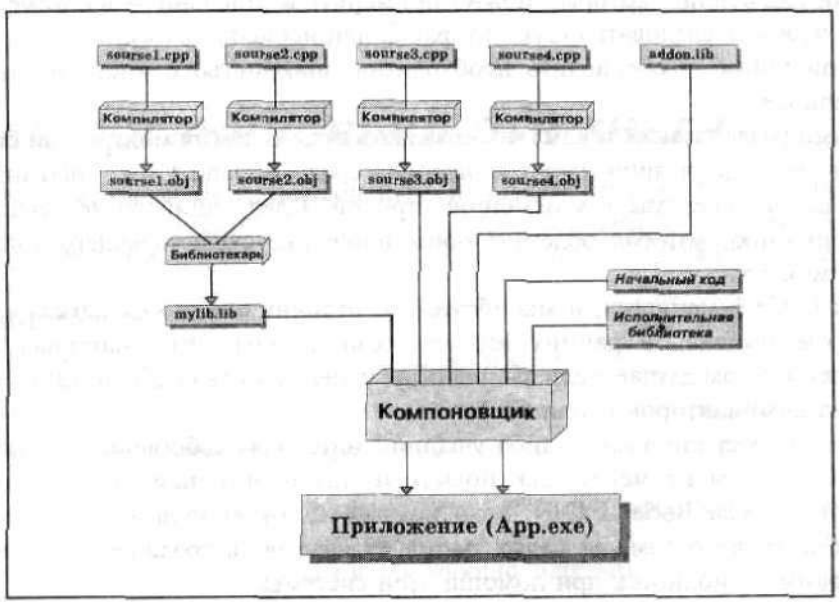


Рис. 1.1. *Этапы создания простой прикладной программы.*

Другими словами, чтобы получить прикладную программу, нужно сделать следующее:
1. Во-первых, вы должны скомпилировать объектные модули из ваших исходных файлов С или С++. Заметьте, что заголовочные файлы (,Н файлы) не компилируются; файлы С или С++ включают их в себя.
2. Затем при помощи библиотекаря из некоторых объектных модулей вы можете организовать библиотеку.
3. И наконец, вам нужно произвести редактирование связей, чтобы из объектных модулей, библиотек, дополнительных библиотек, а также начального кода (StartupCode) и исполнительной библиотеки Borland С++ собрать конечную программу.
Справка:.Начальный код и исполнительная библиотека.
Начальный код связан с процедурами инициализации, которые исполняются перед тем, как управление .будет передано входной точке вашей программы (т.е функции main, WinMain или LibMain). Эти процедуры
5выполняют ряд действий таких, как:
• Инициализация некоторых переменных в исполнительной библиотеке
• Вызов конструкторов глобальных объектов С++
• Аварийное завершение процесса, если для продолжения не хватает основных ресурсов (стека, динамической памяти)
Исполнительная библиотека (RunTime Library, RTL) содержит ряд
процедур, которые вы можете вызывать из своей программы для
выполнения широкого круга стандартных операций, таких, как:
• Управление файлами и каталогами
• Обработка строк и управление памятью Преобразование данных из одного формата в другой
• Запуск, контроль и прерывание процессов
Если вы явным образом вызываете редактор связей, чтобы создать исполняемый модуль или динамическую библиотеку, то должны явно *указать модуль начального кода (c0* .jbj),* а также и исполнительную
библиотеку (c*.lib)

Следующий раздел проведет вас через этапы, необходимые для построения простого приложения в интегрированной среде Borland С++ (Integrated Development Environment — IDE).

Работа в интегрированной среде Borland C++

Интегрированная среда Borland С++ (IDE) соединяет в себе менеджер проекта, редактор, компилятор языка С/С++, библиотекарь, редактор связей, интегрированный отладчик и многое другое. Используя IDE, вы можете создавать библиотеки и прикладные программы для DOS, 16-6ит-Windows, Win32s, и Windows NT.

в нем используется только некоторое подмножество программного интерфейса Win32.

Термины *программа (приложение) для Win32* и *программа для Windows NT* используются как взаимозаменяемые и относятся к программе, которая работает в среде Windows NT или Win32s. Аналогично, *программа для Win16* и *программа для 16-бит-Windows* являются взаимозаменяемыми выражениями и означают приложение, предназначенное для Windows v3.x или Windows for Workgroups. Программа для Win32s — это программа для Win32, но, чтобы работать правильно, она должна использовать только подмножество интерфейса Win32, поддерживаемое Win32s.

Справка: Разновидности среды Windows: NT, Win32, Win32s

Windows NT — это операционная система фирмы Microsoft. Она работает на процессорах типа Intel 80386 (или более мощных) и характеризуется такими особенностями, как, например, плоская 32-битная модель программирования, приоритетная диспетчеризация задач, средства защиты.

Win32 — это т.н. API (Application Program Interface, интерфейс прикладных программ), поддерживаемый Windows NT. Основу его составляет 32-битная модификация API Win16. Кроме того, здесь возможен доступ к первичной подсистеме Windows NT.

Под 16-бит-Windows имеются в виду системы Windows v3.x и Windows for Workgroups. Это графическая среда, реализованная поверх DOS; она делает возможным мультипрограммирование (неприоритетное), разделение данных между приложениями и проч.

Win16 представляет собой API для 16-бит-Windows. Он включает в себя функции управления памятью, окнами, графическими объектами, а также множество дополнений для работы со стандартными диалогами, шрифтами TrueType, динамического обмена данными и т.д.

Win32s является эмулятором, который позволяет программам для Windows NT работать в среде 16-бит-Windows. Реализуется это с помощью ряда динамических библиотек (DLL) и драйвера виртуального устройства. (Эти файлы прилагаются к комплекту Borland C++.) Хотя Win32s позволяет запускать в среде Windows v3.x любую программу для Win32,

Интегрированная среда — программа из одного модуля

Как было показано на рис. 1.1, чаще всего исполняемые модули генерируются из нескольких исходных файлов. Но нередко утилита или небольшая прикладная программа содержит всего один исходный модуль. Вот шаги, которые нужно выполнить для построения такой программы:

1. Если исходный файл уже существует, откройте его, выбрав Open (Открыть) в меню File (Файл). В противном случае в меню File выберите New (Новый). Введите исходный код в новое окно редактора и запишите его в файл, выбрав Save (Сохранить) в меню File. (Вы можете использовать единственную процедуру из HELLO.CPP, показанную на рис. 1.2, если у вас нет готового файла.)
2. Активируйте локальное меню редактора, нажав правую кнопку мышки в середине окна (или Alt-F10 на клавиатуре) и выберите опцию TargetExpert (Эксперт по цели). Появится окно диалога Target Expert.
3. Выберите для вашей программы подходящие параметры и нажмите "кнопку" OK. Например, можно построить HELLO.CPP как стандартное приложение DOS, как приложение типа EasyWin для Windows 3.x (16) или типа Console для Win32. На рис. 1.3 показана установка параметров для приложения типа EasyWin*.

*Все рисунки в книге показывают окна компилятора версии 4.5; если вы работаете с Borland C++ 4.0, изображение на вашем дисплее может незначительно отличаться от показанного. — *Прим. перев.*

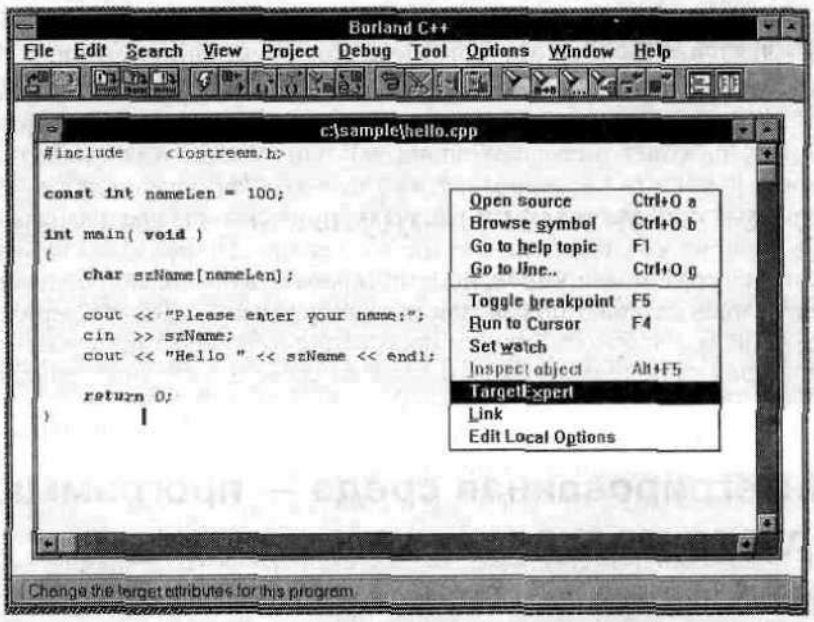


Рис.1.2. *Интегрированная среда Borland C++.*

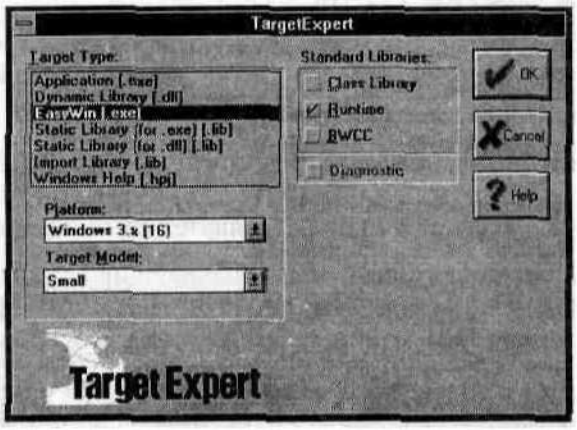


Рис. 1.3. *Установка параметров в TargetExpert.*

4. В меню Debug (Отладка) выберите Run (Запуск).
Появится окно диалога Compile Status (Состояние компиляции), и начнется компиляция и сборка программы. Если не возникнет сообщений об ошибках, программа будет запущена.

Проекты и многомодульные программы

Если для создания программы или библиотеки используется несколько исходных модулей, вы должны организовать проект. *Проект* — это файл, содержащий все необходимое для построения конечного продукта: установочные параметры, информацию о целевой среде и о входных файлах.
- Файлы проекта в Borland C++ имеют расширение .IDE. Конечный продукт, создаваемый с помощью проекта - не обязательно приложение или библиотека. На самом деле, возможности интегрированной среды Borland C++ не ограничиваются работой с программами на C или C++. Как IDE, так и менеджер проекта могут поддерживать расширения, которые позволят вам обрабатывать произвольные файлы и генерировать произвольные целевые продукты.
На рис. 1.4 показаны два модуля C++: main.cpp и greet.cpp.

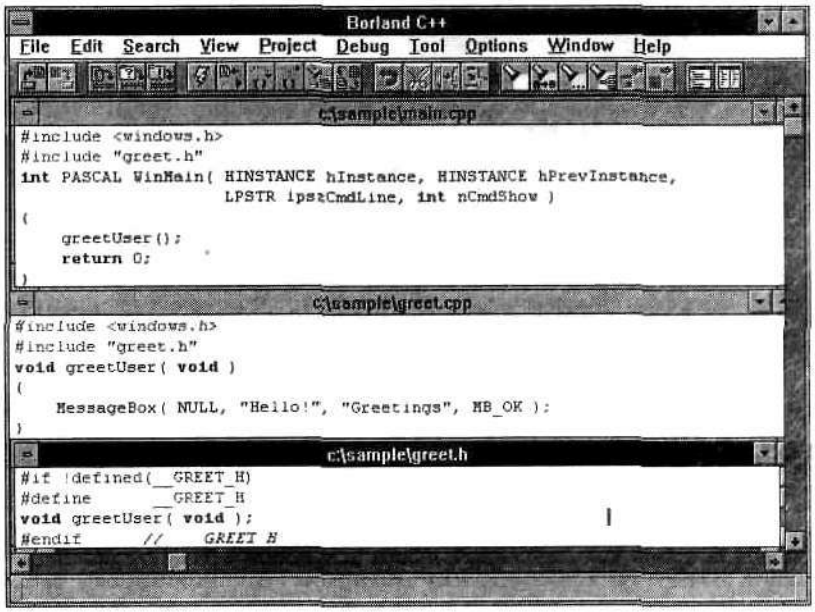


Рис. 1.4. *Файлы простой программы.*

Следующие шага демонстрируют, как организовать проект, который строит программу из двух этих модулей:

1. В меню Project (Проект) выберите New Project (Новый проект).

Появится окно диалога New Project (в версии 4.5 оно называется New Target). Это, собственно, расширенный вариант окна TargetExpert,

которое мы уже обсуждали.

2. В этом окне вы указываете имя проекта, имя и тип выходного модуля и требуемые спецификации библиотек Borland C++. Можно, например, построить вышеупомянутый пример для графической среды пользователя Windows 3.x (16) или для Win32. Рис. 1.5 показывает установку

параметров для среды Windows 3.x (16).

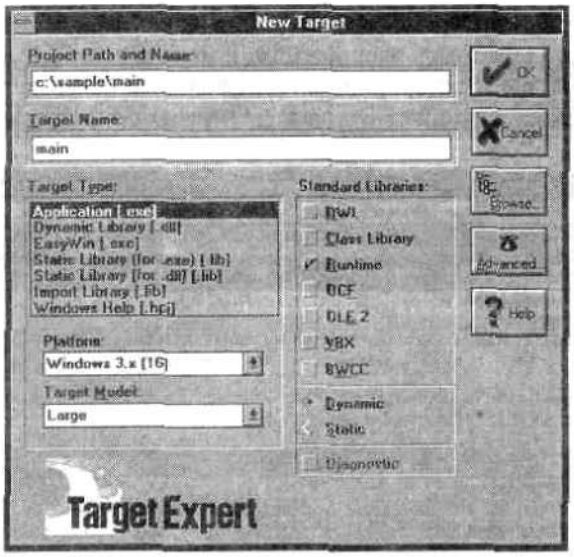


Рис. 1.5. *Пример установок параметров проекта.*

3. Так как программа не применяет управление ресурсами и не включает в себя файл **.DEF,** выберите кнопку Advanced (Расширенный), чтобы уточнить проект.

Появится окно диалога Advanced Options.

4. Выключите селекторы .rc и .def, как показано на рис. 1.6.

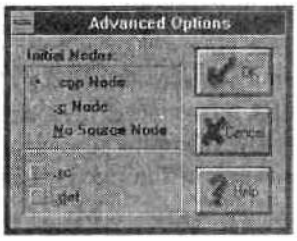


Рис. 1.6. *Окно диалога Advanced Options.*

5. Закройте окна Advanced Options и New Project, нажав соответствующие кнопки OK. Появится окно Project, содержащее узлы main [.exe] и main [.cpp].

6. С помощью правой кнопки мышки укажите узел main [.exe], чтобы активировать его локальное меню. Чтобы включить в проект модуль greet.cpp, выберите в меню опцию Add Node (Добавить узел). Появится окно диалога Add to Project List (Добавить к списку проекта), которое позволит вам найти и указать те файлы, которые нужно включить в Проект.

7. После того, как вы подключите узел greet.cpp, с помощью правой кнопки мышки укажите на узел main [.exe]. Теперь, чтобы построить эту простую программу, выберите опцию Make Node (Создать узел).

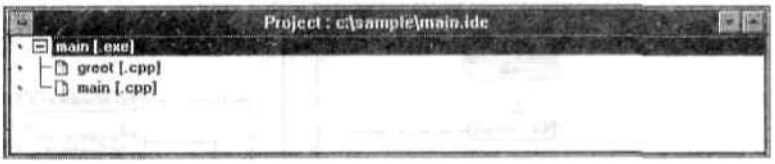


Рис. 1.7. *Пример проекта из двух модулей.*

8. Для исполнения программы выберите Run в меню Debug.

Построение нескольких целевых модулей

Чтобы построить одиночную программу из нескольких исходных модулей, вы можете следовать той общей процедуре, которая была описана в

предыдущем разделе. В проект также часто включаются дополнительные файлы, такие, как сценарии ресурсов (файлы .rc) или файлы определения модулей (файлы .def). Однако, иногда вам может потребоваться построить более одного целевого модуля. Например, в проектах для среды Windows вам понадобится создать динамическую библиотеку (Dynamic Link Library, файл .DLL) или файл Help (.HLP), которые будут использоваться главной программой. Рис. 1.8 иллюстрирует этапы проекта и функции инструментов пакета при создании динамической библиотеки и использующего ее приложения.

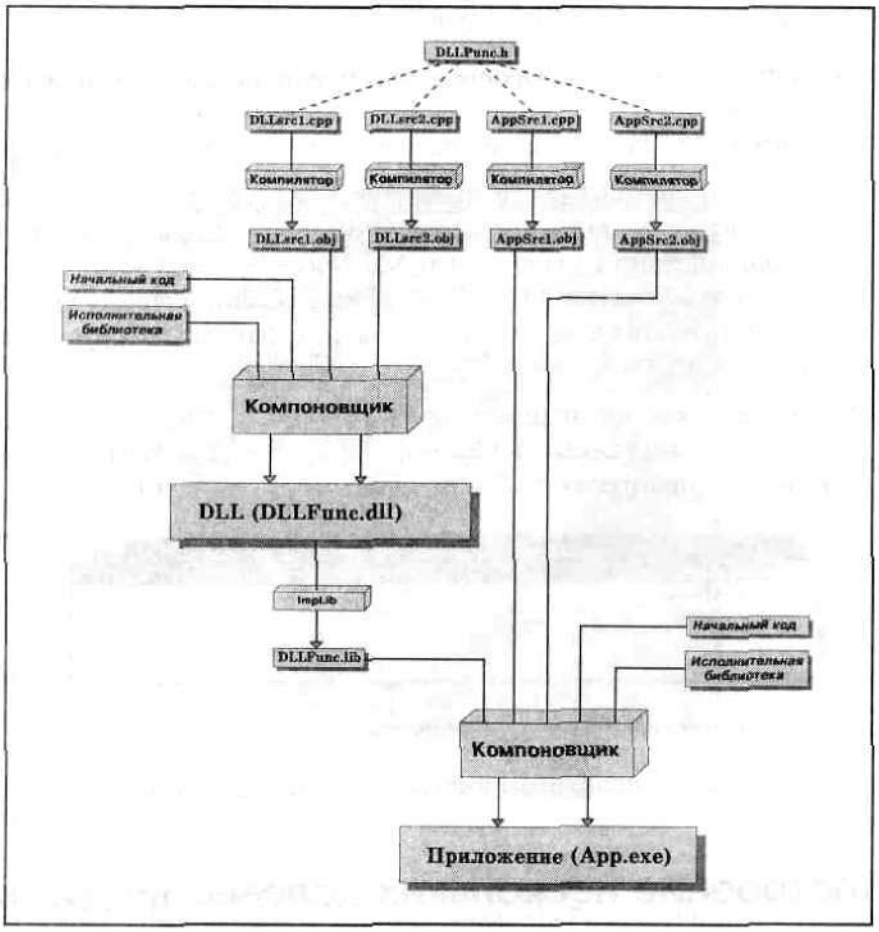


Рис. 1.8. *Приложение, использующее DLL*

Шаги для построения динамической библиотеки схожи с теми, которые предпринимаются при создании приложений (хотя опции компилятора и редактора связей отличаются). Однако, чтобы использовать DLL в прикладной программе, вам нужно сделать следующее:

• Вы должны написать (или приобрести) файл заголовка с прототипами функций и переменных, входящих в DLL.
• Включить его в модули приложения, которые должны иметь доступ к функциям и переменным DLL.
• Создать (или приобрести) библиотеку импорта для данной DLL.
• Включить библиотеку импорта в число входных файлов при редактировании связей (сборке) приложения, использующего функции или переменные из DLL.

Следующий раздел показывает, как организовать проект для двух целевых объектов. Это модификация проекта, созданного ранее (см. рис. 1.7).

Проект для DLL и исполняемого модуля

Предположим, вы захотите переделать примерный проект, ранее созданный в этой главе, чтобы он включал в себя построение библиотеки DLL (greet.dll) и исполняемого модуля (main.exe), ее использующего. Для этого выполните следующие шаги:

1. Модифицируйте файлы greet.cpp и greet.h, введя в них код для поддержки DLL, как показано на рис. 1.9.
2. В меню Project выберите New Target (Новая цель). Появится окно диалога (см. рис. 1.10).
3. Введите имя целевого объекта (GREET), установите тип цели Standard (Стандартный) и нажмите кнопку OK. Появится окно Add Target (New Target в версии 4.5).
4. Установите требуемые параметры типа и библиотек для новой DLL. Например, рис. 1.11 иллюстрирует конфигурацию для среды Windows 3.x (16).

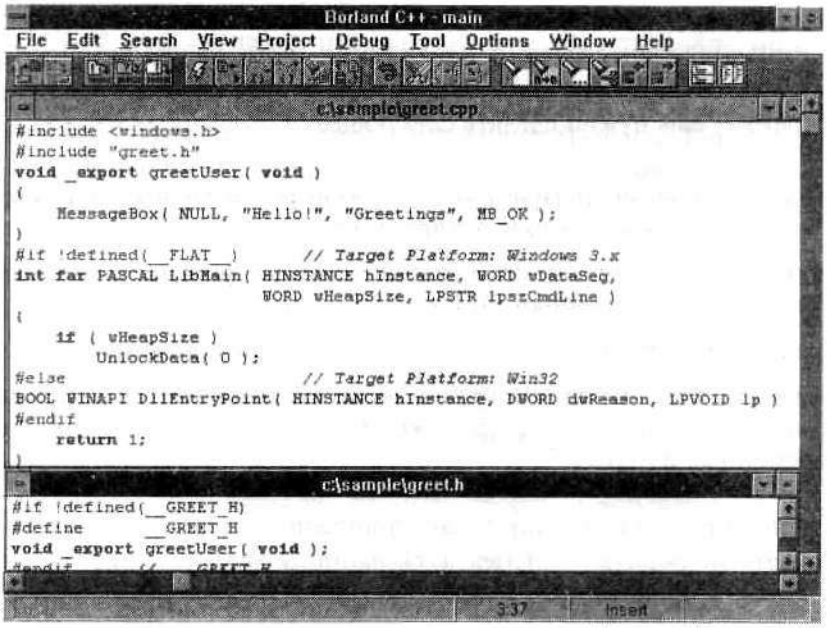


Рис. 1.9. *Модифицированные файлы greet.cpp и greet.h.*

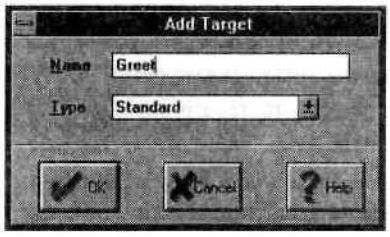


Рис. 1.10. *Окно диалога New Target.*

5. Выберите кнопку Advanced. Появится окно диалога Advanced Options.
6. Выключите селекторы .rc и .def и закройте окна с помощью кнопок OK.
7. Правой кнопкой мышки вызовите локальное меню узла greet[.cpp], соединенного с main[.exe], и укажите Delete Node (Удалить узел).
8. С помощью левой кнопки мышки "перенесите" узел greet[.dll] и "положите" его на main[.exe] (см. рис. 1.12).

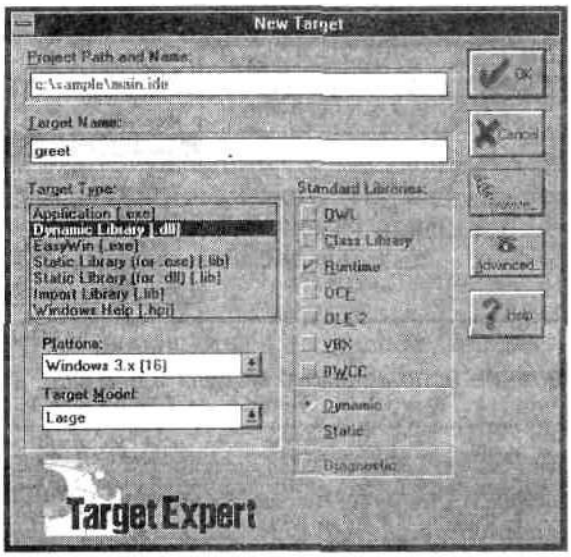


Рис. 1.11. *Установка параметров новой DLL*

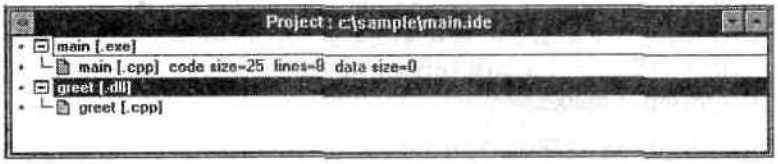


Рис. 1.12. *Перемещение узла DLL на узел .exe.*

9. Вызовите правой кнопкой меню узла main[.exe] и выберите Build Node (Построить узел), чтобы построить сразу и DLL, и исполняемый модуль, который ее использует. На рис. 1.13 показано окно проекта с двумя целевыми модулями.

10. Чтобы запустить программу, укажите Run в меню Debug.

Вы можете следовать этой процедуре, чтобы создавать проекты с несколькими целями, которыми могут быть приложения, файлы Help (справочные файлы) или библиотеки. В проекте может быть несколько целей верхнего уровня, подобно тому, как показано на рис. 1.12. Напротив, рис. 1.13 показывает многоцелевой проект, в котором всего одна цель верхнего уровня. Перемещая целевые узлы, вы задаете менеджеру проекта те или иные правила зависимостей между вашими целевыми объектами.

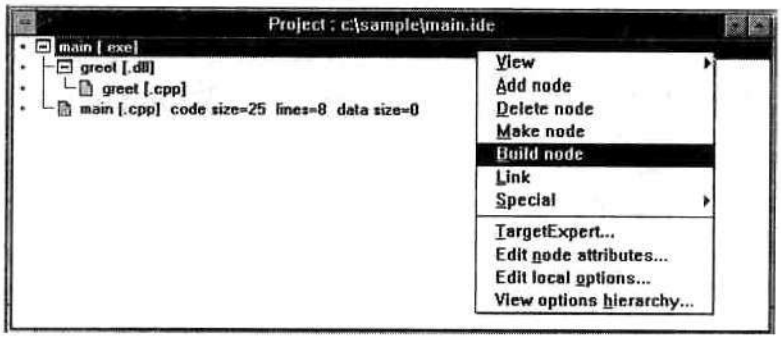


Рис. 1.13. *Построение исполняемого модуля и его DLL*

Справка: Библиотеки статические, динамические и библиотеки импорта

Статическая библиотека является собранием объектных модулей. Это удобный метод, чтобы держать вместе несколько связанных друг с другом функций из различных модулей. Когда вы присоединяете статическую библиотеку, редактор связей извлекает из нее только те модули, ссылки на которые имеются в других модулях приложения. Статическая библиотека создается с помощью программы-библиотекаря, например, tlib.exe.

Динамически присоединяемая библиотека (DLL) является модулем, содержащим функции, переменные или ресурсы, которые соответственно вызываются, адресуются или загружаются прикладной программой или другой DLL во время исполнения. Динамическая библиотека очень похожа на исполняемый модуль. Она создается редактором связей, таким, как tlink.exe или tlink32.exe.

Библиотека импорта содержит ряд записей, идентифицирующих функции и переменные, доступные в DLL. Такая библиотека дает возможность редактору связей обрабатывать имеющиеся в вашей программе ссылки на эти символы, предотвращая, таким образом, появление сообщений об ошибках типа undefined symbol xxxx (неопределенный символ). Библиотеки импорта создаюттся импортирующим библиотекарем, например implib.exe. Следует заметить, что можно вводить объектные модули в уже существующую библиотеку импорта с помощью tlib.exe.

Использование инструментальных средств с командной строкой

Если вы предпочитаете работать с командной строкой Windows NT или DOS, Borland C++ предложит вам соответствующие инструменты для создания библиотек и приложений. Таблица 1.1 описывает имеющиеся в пакете компиляторы, редакторы связей и библиотекари.

Таблица 1.1. Основные средства с командной строкой *Имя файла Описание*

BCC.EXE 16-битовый компилятор. Транслирует файлы C или C++ в объектные модули для целевой среды DOS или 16-бит-Windows.

BCC32.EXE 32-битовый компилятор. Транслирует файлы для использования в среде Windows NT или Win32s.

TLINK.EXE 16-битовый редактор связей (компоновщик). Создает приложения для DOS и приложения или динамические библиотеки (DLL) для 16-бит-Windows. Обычно TLINK.EXE вызывается автоматически из BCC.EXE. (TLINK.EXE может вызывать RLINK.EXE для подключения ресурсов к файлам .EXE или .DLL.)

TLINK32.EXE 32-битовый компоновщик. Создает DLL или приложения для Windows NT и Win32s. Обычно вызывается автоматически из BCC32.EXE (TLINK32.EXE может использовать RLINK32.EXE для подключения ресурсов к целевому файлу).

TLIB.EXE Библиотекарь. Создает и поддерживает статические библиотеки для DOS, !6-бит-Windows, Win32s и Windows NT. TLIB.EXE может также применяться для добавления или удаления модулей библиотек импорта.

IMPLIB.EXE Библиотекарь. Создает библиотеку импорта из DLL или файла определения модуля (файла .DEF). Применяется как для 16-битовой, так и 32-битовой среды Windows.

Файлы конфигурации

Компиляторы с командной строкой (bcc.exe и bcc32.exe) и компоновщики (tlink.exe и tlink32.exe) автоматически ищут файлы конфигурации в текущем каталоге и в том, из которого они были загружены. Таблица 1.2 перечисляет конфигурационные файлы, соответствующие каждому инструменту.

Таблица 1.2. Файлы конфигурации компиляторов и компоновщиков *Инструмент Описание Файл конфигурации* BCC.EXE 16-битовый компилятор TURBOC.CFG BCC32.EXE 32-битовый компилятор BCC32.CFG TLINK.EXE 16-битовый компоновщик TLINK.CFG TLINK32.EXE 32-битовый компоновщик TLINK32.CFG

Файл конфигурации обычно содержит предпочтительные установочные параметры для соответствующего инструмента. Например, файлы TURBOC.CFG и BCC32.CFG идентифицируют каталоги для ваших библиотечных и include-файлов. Следующая выдержка показывает пример того, что можно обычно найти в файле конфигурации компилятора:

```
-IE:\BC4\INCLUDE
-LE: \BC4\LIB
-ml
-v
```

Похожим образом, типичный файл конфигурации компоновщика содержит информацию, идентифицирующую библиотечные каталоги:

```
/LE\BC4\LIB
```

Справка: Модификация файлов конфигурации

Процедура установки Borland C++ создает файлы конфигурации с опциями библиотечных и заголовочных каталогов, используемых по умолчанию. Если вы собираетесь регулярно использовать заголовочные файлы и библиотеки, находящиеся в других каталогах, то вы можете модифицировать конфигурацию. Вы можете также добавить некоторые предпочтительные параметры для компилятора или компоновщика (эти параметры, если необходимо, всегда можно отменить в командной строке). Например, вот модифицированное содержание файла TURBOC.CFG:

```
-IE:\BC4\INCLUDE;C:\PXENG30\WINDOWS
-LE:\BC4\LIB;C:\PXENG30\WINDOWS
-ml
-v
```

Создание целевых модулей для Windows

Результатом вашей работы, ориентированной на среду Windows, обычно является приложение (EXE) или динамическая библиотека (DLL). Следующий раздел показывает этапы создания программного продукта для Win16 или Win32 с применением средств, управляемых командной строкой.

Создание приложения для Widows

Нижеследующий код просто выдает приветственное сообщение:

```
//////////////////////////////////////////////////////
//SIMPWIN.CPP: Простая Windows-программа... //
//////////////////////////////////////////////////////
#define STRICT
#include <windows.h>
//
//Переменные для приветствия...
```

```
//
const char szMsg[] = "Привет от Windows!;
const char szCap[] = "Простое сообщение";
//
// Входная точка программы
//
#pragma argsused
int PASCAL WinMain( HINSTANCE hlnstance,
HINSTANCE hPrevInstance,
LPSTR IpCmdLine, int nCmdShow)
```

// Приветствуем пользователя! MessageBox(NULL, szMsg, szCap, MB_OK); // Завершение... return 0; }

Чтобы построить приложение SIMPWIN.EXE, вы должны сначала откомпилировать исходный файл, а затем связать полученный объектный модуль с компонентами исполнительной библиотеки C++.

Windows 3.1 (16)

Следующая команда создает вариант SIMPWIN.EXE для Winl6:

BCC -v -W -ml SIMPWIN.CPP

Команда запускает компилятор Borland C++ с командной строкой BCC.EXE, который компилирует SIMPWIN.CPP, создавая объектный модуль SIMPWIN.OBJ, Затем компилятор автоматически вызывает компоновщик, TLINK.EXE, который, комбинируя объектный модуль с модулями исполнительной библиотеки, создает исполняемый модуль SIMPWIN.EXE.

Справка: Раздельный вызов 16-битных компилятора и компоновщика

Чтобы предотвратить автоматический вызов компоновщика, можно использовать опцию компилятора -c. Затем, после компиляции файлов C или C++, вы должны явным образом вызвать TLINK.EXE. Следующие команды это иллюстрируют:

```
BCC   -c -v -W -ml SIMPWIN.CPP

TLINK /c /v /Tw @RSPAPP.TXT
```

RSPAPP.TXT — это текстовый файл с таким содержимым:

```
c0wl.obj simpwin.obj
simpwin.exe
simpwin.map
import.lib mathwl.lib cwl.lib
```

Win32

Следующая команда строит вариант SIMPWIN.EXE для Win32:

BCC32 -V -W SIMPWIN.CPP

Команда вызывает компилятор BCC32.EXE, который компилирует SIMPWIN.CPP и создает SIMPWIN.OBJ. Затем компилятор автоматически вызывает компоновщик, TLINK32.EXE, который комбинирует объектный модуль с модулями исполнительной библиотеки и создает SIMPWIN.EXE.

Справка: Раздельный вызов 32-битных компилятора и компоновщика

Чтобы предотвратить автоматический вызов компоновщика, можно использовать опцию компилятора -c. Затем, после компиляции файлов .CPP, вы должны явным образом вызвать TLINK32.EXE. Следующие команды это иллюстрируют:

```
BCC32   -c -v -W  SIMPWIN.CPP

TLINK32 /c /v /Tw @RSPAPP32.TXT
```

Текстовый файл RSPAPP32.TXT содержит следующее:

```
c0w32.obj simpwin.obj
simpwin.exe
simpwin.map
import32.lib cw32.lib
```

Создание динамической библиотеки для Windows

Чтобы создать динамическую библиотеку, вы должны откомпилировать исходный файл и скомпоновать его с модулями исполнительной библиотеки Windows. Следующая команда выполняет обе эти операции:

```
BCC -v -WD -ml GREET.CPP
```

Команда запускает компилятор BCC.EXE, который генерирует объектный файл GREET.OBJ и затем автоматически вызывает компоновщик, TLINK.EXE, собирающий из объектного модуля и модулей исполнительной библиотеки файл GREET.DLL.

Компиляторы и компоновщики ресурсов

В большинстве случаев пользовательские интерфейсы Windows-программ определяются сценариями ресурсов, которые компилируются и присоединяются к прикладной программе. Например, с помощью редактора ресурсов вы можете разработать окно диалога и дать ему уникальное имя. Редактор ресурсов создает сценарий окна, который затем компилируется и компонуется с вашей программой. Во время исполнения программа, используя имя диалога и соответствующие вызовы функций, сможет получить к нему доступ и отобразить его на экране. Ресурсы позволяют вам легко модифицировать интерфейс пользователя, не изменяя основного кода программы. Другими возможными видами ресурсов являются ускорители, меню, битовые карты (bitmaps), пиктограммы (icons), курсоры, шрифты, информация о версии, строковые таблицы.

Рис. 1.14 иллюстрирует последовательность действий и функции компилятора и компоновщика при использовании ресурсов в прикладной программе.

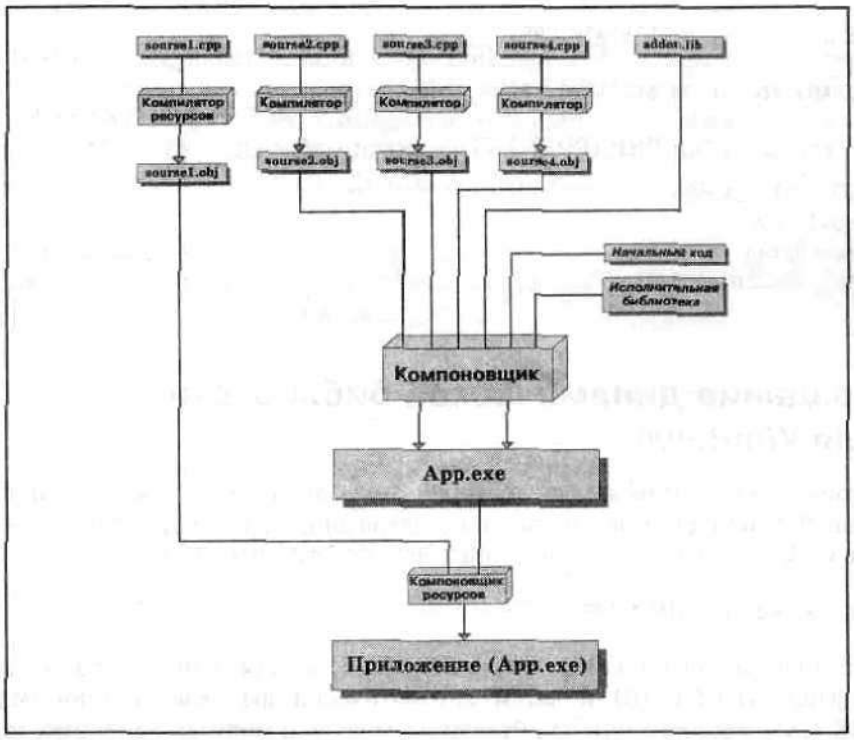


Рис. 1.14. *Последовательность использования ресурсов*

Таблица 1.3 описывает компиляторы и компоновщики ресурсов, имеющиеся в пакете Borland C++.

Таблица 1.3. Компиляторы и компоновщики ресурсов, вызываемые командной строкой
Имя файла Описание
BRCC.EXE 16-битовый компилятор ресурсов. Транслирует файлы .RC в файлы .RES, которые могут присоединяться к программам для среды 16-бит-Windows.
RLINK.EXE 16-битовый компоновщик ресурсов. Присоединяет один или несколько файлов .RES к приложению (.EXE) или динамической библиотеке (.DLL) для 16-бит-
Windows.

BRC.EXE Оболочка, вызывающая BRCC.EXE или RLINK.EXE.
BRCC32.EXE 32-битовый компилятор ресурсов. Транслирует файлы RC в файлы .RES, которые могут присоединяться к программам для 32-битовой среды Windows.
RLINK32.DLL 32-битовый компоновщик ресурсов. Присоединяет один или несколько файлов .RES к приложению (.EXE) или динамической библиотеке (.DLL) для 32-битовых Windows.

Компилятор и утилиты для файлов Help (Помощь)

В комплект Borland C++ входит компилятор Help фирмы Microsoft, HC31.EXE, который позволяет генерировать файлы Help (.HLP) для Windows. Также имеются компилятор битовых карт с различным разрешением (Multiple Resolution Bitmap Compiler, MRBC.EXE) и редактор "горячих точек" (Hot Spot Editor, SHED.EXE). Таблица 1.4 дает краткое описание этих средств.

Таблица 1.4. Компилятор Help и сопутствующие утилиты. *Имя файла Описание*
HC31.EXE Компилятор Help'a. Читает файл проекта Help, файлы RTF и любые указанные файлы битовых карт (.BMP) или горячих точек (.SHG) и генерирует файл поддержки (.HLP) для Windows.

MRBC.EXE Компилятор битовых карт с различным разрешением. Позволяет вам скомбинировать битовые карты различного разрешения в единый файл. Во время исполнения программа поддержки (Windows Help Engine, WINHELP.EXE) автоматически, в зависимости от разрешения экрана, загружает подходящую битовую карту.
SHED.EXE Редактор "горячих точек" (Hotspot). Позволяет вам создавать битовые карты, некоторые области которых ассоциированы с определенным действием; когда пользователь указывает на такую область, программа поддержки выполняет связанное с ней действие, например, открывает окно.

Модификация существующего кода для Borland C++

В этом разделе дается обзор отличий языка и библиотек Borland C++ от предыдущих версий пакета.

Три типа *char*

Borland C++ теперь рассматривает простой тип *char, char без* знака и *char* со знаком как три различных типа. Ранние версии компилятора считали простой *char* имеющим знак. Посмотрите на следующее описание классов:

```
class base
{
 public:


virtual void f( char );
class derived:  public base{
public:
virtual void f( signed char;
};
```

В ранних версиях компилятора функция derived: :f(signed char) переопределяла base::f(char). Теперь же *char* ведет себя так, что derived: :f(signed char) скрывает base::f(char).
Вы можете восстановить старое поведение этих типов, задав опцию -K2 компилятору, управляемому командной строкой. Или же, при работе с интегрированной средой (IDE), можно вызвать меню Options и указать последовательно Project, C++, C++ Compatibility (Совместимость) и активировать опцию Don't Treat Char As A Dictinct Type (Не рассматривать Char как отдельный тип).

Варианты с массивами операторов new и delete

Для выделения и освобождения памяти для массивов из объектов Borland C++ теперь использует операторы new[] и delete[]. Если вы предусмотрели свои собственные версии new и delete, вам, вероятно, потребуется дать новые определения этих операторов и для вариантов с массивом.

Оператор new и исключительные ситуации (Exceptions)

Стандартный new_handler, имеющийся в Borland C++, ведет себя в соответствии со спецификацией ANSI C++ и выбрасывает исключение xalloc, чтобы указать на неудачу при распределении памяти. Поэтому код, в котором для проверки результата операции new используется сравнение с нулем, нуждается в модификации. Следующий пример иллюстрирует необходимые изменения:

```
//////////////////////////////////////////////

/ / / / / / / / / / /
//    NEWNEW.CPP:    исключение    xalloc    и    оператор    new                //
//////////////////////////////////////////////////////
```

```
#include <new.h>          const int size = 0x100;
//
//Старый метод проверки отказов при выделении памяти...
//       void old_func( void )
{
char *p; if(  I( p = new char[size] ) ) {
//Неудача при выделении памяти...
else
{
//Успешное выделение памяти...
//...
// Новый метод проверки на отказ...
//
void new_func( void )
{
char *p;
try
{
p = new char[size]; //Успешное выделение
//...
}
catch( xalloc& xx )
{
//Отказ при выделении памяти...
}
//  ...
```

Применение функций *longjmp* и *setjmp*

Определение типа jmp_buf, используемого функциями longjmp и setjmp, было изменено так, чтобы оно соответствовало средствам управления исключениями. Нижеследующий листинг показывает старое и новое определения:

```
//
//
// Раннее определение //Новое определение
//типа jmp_buf //типа jmp_buf                    //
//
typedef struct _jmp_buf {      typedef struct _jmp_buf {
unsigned j_sp; unsigned j_sp;
unsignedj_ss; unsigned j_ss;
unsignedj_flag; unsigned j_flag;
unsignedj_cs; unsigned j_cs;
unsignedj_ip; unsigned j_ip;
unsignedj_bp; unsigned j_bp;
unsignedj_di; unsigned j_di;
unsignedj_es; unsigned j_es;
unsignedj_si; unsigned j_si;
unsignedj_ds; unsigned j_ds;
}  jmp_buf[1]; unsigned j_excep;
 unsigned j_context; }   Jmp_buf[1];
```

Переименованные глобальные переменные

В Таблице 1.5 перечислены некоторые глобальные переменные из исполнительной библиотеки, названия которых были изменены.

Таблица 1.5. Переименованные глобальные переменные

Старые имена Новые имена

daylight	_daylight
directvideo	_directvideo
environ	_environ
timezone	_timezone
tzname	_tzname
sys_errlist	_sys_errlist
sys_nerr	_sys_nerr

Совет: Использование библиотеки obsolete.lib

Если ваши программы используют переменные, перечисленные в Таблице 1.5, вы должны модифицировать ссылки на них в соответствии с новыми именами. Если, однако, у вас есть объектный модуль или библиотека, которые ссылаются на старые имена и не могут быть модифицированы, присоедините библиотеку obsolete.lib к вашему проекту (или к списку библиотек компоновщика с командной строкой), чтобы разрешить эти неопределенные ссылки.

Заключение

Из этой главы вы узнали, как применять инструментарий Borland C++ для создания прикладных программ из исходного кода на языках C или C++. Последующие главы этой книги сосредоточивают внимание на синтаксисе, конструкциях и идиомах, обычно встречающихся в коде на C и C++. У вас будет достаточно материала, чтобы попрактиковаться в компиляции проектов или отдельных фрагментов программ.

Глава 2

Borland C++ в полном объеме поддерживает стандарт языка C в том виде, как он определяется Американским институтом национальных стандартов (ANSI). Данная глава познакомит вас с принципами и синтаксисом ANSI C на примере небольших программ или фрагментов кода. Учитывая, что C++ развился из языка C, хорошее понимание C представляется необходимым для хорошего программирования на C++. Мы начнем с простой программы, которая по традиции бывает первой для каждого начинающего программировать на C.

Простая программа на C

Следующий короткий пример иллюстрирует структуру программы на C.

```
/* HELLO. C: Пример C-программы */
#include <stdio.h>
int main( void )
{
printf( "Hello, World" );
return 0;
```

Если вы используете IDE, то вам нужно будет создать проект для построения этой программы. Это можно сделать с помощью такой последовательности шагов:

1. Укажите в меню Project опцию New Project. Появится окно диалога New Project.
2. Введите имя проекта в рубрику Project Path and Name (Каталог и имя проекта). Можно напечатать, например, просто hello или же c:\bc4\examples\hello.

В поле Target Name автоматически будет установлено hello, последнее слово из имени проекта.

3. В списке Target Type (Тип целевого модуля) выберите EasyWin (.exe).

Менеджер проекта автоматически установит целевую среду (Platform) Windows 3.x (16). Модель проекта устанавливается Small (Малая), а стандартные библиотеки - Static (статические), как показано на рис. 2.1.

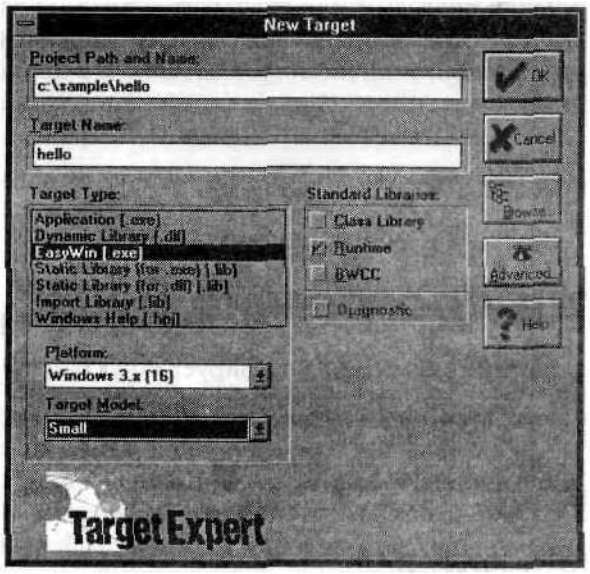


Рис. 2.1. *Окно диалога New Project*

4. Укажите кнопку Advanced в правой части окна для дальнейшей детализации проекта. Появится новое окно диалога.
5. Включите только селектор узла .c в списке начальных узлов (Initial Nodes), как показано на рис. 2.2.

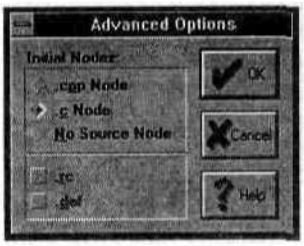


Рис. 2.2. *Окно Advanced Options*

6. Выберите OK и затем снова OK, чтобы закрыть оба окна диалога.
Выполнив указанные шаги, вы начали новый проект, в окне которого содержится несколько узлов, в том числе и hello.c. Рис. 2.3 показывает окно проекта для нашего примера.

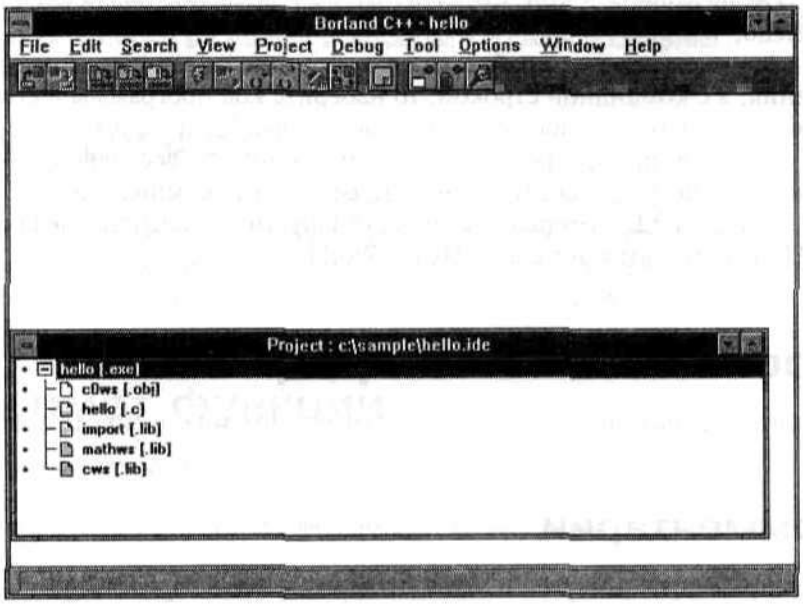


Рис. 2.3. *Окно проекта*

Подсказка: Планировка окна проекта

Вы можете не увидеть в окне проекта все те узлы, которые видны на рис. 2.3, если не разрешено отображение модулей исполнительных библиотек или проекта. Похожим образом, каждый из узлов может отображать информацию о модулях проекта с той или иной степенью детализации, что зависит от установленных параметров видимости (Project View). Чтобы спланировть вид, который имеет окно проекта, выберите Environment в меню Options и затем Project View.

Укажите на узел hello.c, дважды нажмите кнопку мыши и введите вышеприведенный текст примера в окно редактора. Сохраните текст на диске, использовав опцию Save в меню File.
Теперь вы готовы к тому, чтобы скомпилировать, собрать и запустить программу. В меню Project выберите опцию Make All (Построить все модули). Появится окно состояния компиляции. Если вы столкнетесь с сообщениями об ошибках, проверьте, правильно ли был введен текст программы.
После успешной компиляции и компоновки, программу нужно запустить. В меню Debug выберите Run. Когда программа запущена, она печатает в окне слова Hello, World.

Если вы предпочитаете работать не с интегрированной средой программирования, а с командной строкой, то наберите код программы и сохраните его с помощью какого-либо текстового редактора. Если вы работаете в DOS, то для компиляции программы выполните команду **bcc hello.c,** а если в Windows NT, команду **bcc32 hello.c.** В результате компиляции получится модуль HELLO.EXE, который вы сможете запустить, напечатав **hello** и нажав Enter. Программа выведет слова Hello, World.

Составные части программы

Давайте рассмотрим подробнее, из чего состоит эта маленькая программа.

Комментарии

Наш пример начинается с комментария. *Комментарий* является частью кода, который игнорируется компилятором. Комментарий начинается с дробной черты, за которой следует звездочка (/*). Компилятор пропускает все последующие символы, пока не встретит комбинацию звездочки и дробной черты (*/).

Справка: Комментарии в стиле C++

Borland C++ допускает использование комментариев в стиле C++, когда вы пишете код на C. В C++ комментарий начинается с двойной дробной черты (//) и продолжается до конца текущей строки. Следует избегать использования коментариев в стиле C++, если вы намерены переносить свой код в другую среду программирования, так как другие компиляторы C могут не допускать комментариев такого вида.

Директива #include и заголовочные файлы

Директива #include, следующая за комментарием, говорит компилятору, что нужно включить в текст примера содержимое файла stdio.h. Это один из большого числа заголовочных файлов, поставляемых вместе с компилятором Borland C++. *Заголовочные файлы* снабжают компилятор необходимой информацией о данных и функциях, которые вы можете использовать в своей программе.

Функция main

Следующая часть нашего примера — функция main. Строго говоря, каждая программа на C содержит функцию main, которая является начальной точкой программы. Однако программы, ориентированные на работу в среде Windows, часто содержат вместо нее функцию WinMain.

Формат функции

В общем случае функция имеет следующий формат:

```
тип возвращаемого значения ИмяФункции( параметры )
{
// Здесь следует тело функции,
// состоящее из операторов
}
```

Если рассматривать программу на C как некоторое эссе, то функция будет в нем чем-то вроде параграфа. В свою очередь функция состоит из операторов, напоминающих предложения. Каждый оператор заканчивается точкой с запятой (;).

Наша простая программа состоит всего из одной функции, содержащей два оператора. Первый из них печатает приветственное сообщение, используя функцию printf. Второй оператор возвращает нулевое значение (0) вызывающей процедуре.

Справка: Функции исполнительной библиотеки

Printf является одной из стандартных библиотечных функций, которые вы можете использовать в своих программах. Исполнительная библиотека Borland C++ предоставляет в ваше распоряжение функции для выполнения самых разнообразных задач, таких, как обработка кодов ASCII, преобразование данных, операции с файлами и каталогами, управление памятью, математические процедуры и т.д.

Функции часто возвращают результат вычислений. Функция main обычно возвращает нулевое значение, чтобы сообщить об отсутствии ошибок при выполнении.

Представление информации в языке С

Строго говоря, любая программа обрабатывает информацию или данные. Программа hello обрабатывает приветственное сообщение, выводя его на экран. Язык С использует различные способы представления информации или данных.

Константы

Термин *константа,* в общем случае, относится к значению, которое не может быть изменено. В языке С константы могут быть строковыми, символьными, целыми и вещественными (с плавающей точкой).

Строковая константа представляет собой последовательность символов, заключенную в кавычки. В hello использована константная строка "Hello, World".

Также могут применяться *символьные константы, целые константы* и *вещественные константы.* Таблица 2.1 описывает соответствующие каждому виду форматы.

Таблица 2.1 Константы в языке С

Константа	*Формат*	*Примеры*
Символьная	Символ, заключенный в апострофы	'a', '!'
Целая	Десятичный: последовательность цифр, не начинающаяся с 0	23, 989
	Восьмиричный: 0, за которым следуют восьмиричные цифры	077, 023
	Шестнадцатиричный: 0x или 0X, за которыми следуют шестнадцатиричные цифры	0xAF, 0x9B
Вещественная	Десятичный: [цифры].[цифры]	1., .34, 2.5
	Экспоненциальный: [цифры]E|e[+|-]цифры	4e7, 5.1e+8
Строковая	"символ,символ,..."	"Hello", "C\n"

Совет: Применение макросов для представления констант

Для представления констант вы можете использовать макроопределения (макросы). Макрос ассоциирует имя с определенным значением. Разумное применение макроимен может сделать ваш код более надежным. Если вы используете одно и то же константное значение в различных местах программы, макроопределение позволит очень просто его модифицировать — *нужно только переопределить макрос.* В следующем фрагменте кода с константами связываются имена, отражающие их смысл.

```
/*********************************************************/
/*  CONSTANT.C: Применение макросов                      */
/*  для представления констант                           */
/*********************************************************/
```

```
#define CHARACTER_B     'B'   // Макро для символьной
                              // константы
#define VERSION_OCT     020   // Для восьмиричного числа
#define VERSION_DEC     16    // Для десятичного числа
#define VERSION_HEX     0x10  // Для шестнадцатиричного числа
```

Простые типы данных

В C можно использовать различные типы данных для представления хранимой и обрабатываемой вами информации. Данные каждого типа занимают определенное количество байт памяти и могут принимать значения в известном диапазоне. Размер и допустимый диапазон для них в различных реализациях языка могут отличаться. В таблице 2.2 дан обзор основных типов, доступных для программ, ориентированных на среду DOS или 16-бит-Windows.

Таблица 2.2. Простые типы данных

Тип данных	*Размер (бит)*	*Диапазон*
char	8	−128 ... 127
signed char	8	−128 ... 127
unsigned char	8	0 ... 255
short int	16	−32768 ... 32767
unsigned int	16	0 ... 65535
int	16	−32768 ... 32767
long	32	−2147483648 ... 2147483647
unsigned long	32	0 ... 4294967295
float	32	$3.4x10^{-38}$ ... $3.4x10^{38}$
double	64	$1.7x10^{-308}$ ... $1.7x10^{308}$
long double	80	$3.4x10^{-4932}$ ... $3.4x10^{4932}$

Подсказка: Применение limits.h и float.h для определения допустимого диапазона

Вы можете использовать макросы, находящиеся в файлах limits.h и float.h, чтобы определять допустимый диапазон значений для данных различных типов во время компиляции. Это сделает ваш код более пригодным для перенесения его в другую программную среду.

Переменные

Чтобы выделить память для данных конкретного типа, вы определяете *переменную*. Вначале указывается тип данных, а затем имя переменной, как показано ниже:

int i; // определение целой переменной i long 1=10; // определение и инициализация
// длинного целого 1 double d1, d2, d3=1.25 // выделение памяти для трех
// вещественных чисел двойной длины

Определяя переменную, вы можете присвоить ей начальное значение. Можно также определить несколько переменных одного типа, перечислив их через запятую. Как и другие операторы C, определение должно заканчиваться точкой с запятой (;).

Типизованные константы

Типизованные константы — это переменные, значение которых нельзя изменить. Вы можете создать такую константу, написав определение для переменной с добавлением ключевого слова *const* перед типом. Ниже следует пример использования типизованной константы:

```
/* CONST.C: Этот пример использует как макро, */
/* так и типизованную константу... */
#include <stdio.h>
#define GERMAN_PRICE 10    // Это макроопределение
const float xchngRate = 1.60; //А это типизованная константа
int main( void )
{
printf( "Этот продукт стоит в США $X.2f!", GERMAN_PRICE * xchngRate);
return 0;
```

Совет: Применяйте не макросы, а типизованные константы

Применение типизованных констант предпочтительнее, так как макросы являются просто текстовыми подстановками и могут не давать компилятору достаточной информации о желательном представлении данной величины. В отличие от имен типизованных констант, макроимена не включаются в отладочную информацию. Вы не можете использовать их в выражениях во время отладки программы, а имена констант при отладке доступны.

Функции

Вы уже в какой-то мере познакомились с двумя функциями - main и printf. Вы написали первую из них и использовали вторую для вывода сообщений. При изучении языка программирования бывает полезно хорошо знать процедуры ввода и вывода (I/O). Распечатка сообщений и содержимого переменных с помощью этих процедур позволяет вам следить за поведением и состоянием ваших программ.

Ввод и вывод

Вы видели, как используется функция printf для вывода сообщений. При вызове printf обязательно передается в качестве аргумента хотя бы одна строка. Функция просматривает строку и выводит каждый символ как он есть, буквально, пока не встретит спецификацию преобразования.

Спецификация преобразования

Спецификация преобразования начинается со знака процента (%) и имеет следующий формат:

%[флаг][ширина][.точность][размер]тип

Каждая спецификация заставляет функцию printf искать дополнительный аргумент, который затем преобразуется и выводится в соответствии с заданным преобразованием. Вы должны быть уверены, что число дополнительных аргументов в вашем вызове printf соответствует числу спецификаций.

В таблице 2.3 описаны элементы спецификаций преобразования.

Таблица 2.3. Спецификация преобразования для функции printf

Элемент	*Необязательный*	*Символ*	*Значение*
флаг	да	–	Прижать число при выводе к левому краю поля
		0	Заполнить лишнее пространство нулями вместо пробелов
		+	Всегда выводить знак (+ или -)
		пробел	Пробел на месте знака, если значение положительно
		#	Выводить 0 перед восьмиричным или 0x перед шестнадцатиричным значением
ширина	да		Минимальная ширина поля
точность	да		Максимальное число знаков; для целых — минимальное число выводимых цифр
размер	да	F	Аргумент является дальним указателем
		N	*Ближний указатель*
		h	Короткое целое (short int)
		l	Длинное целое (long int)

Таблица 2.3. Продолжение

Элемент	*Необязательный*	*Символ*	*Значение*
		L	Аргумент является числом типа long double
тип	нет	d	Представить в виде десятичного целого числа со знаком
		i	То же, что и d
		o	Представить в виде восьмиричного целого без знака
		u	Представить в виде десятичного целого без знака
		x	Шестнадцатиричное целое без знака, цифры в нижнем регистре
		X	То же, что x — в верхнем регистре
		f	Число с плавающей точкой в форме [-]dddd.dddd
		e	Число с плавающей точкой [-]d.dddd или e[+/-]ddd
		g	Число с плавающей точкой (использовать форму f или e)
		E	Число с плавающей точкой [-]d.dddd или E[+/-]ddd
		G	Использовать форму f или E
		c	Вывести одиночный символ
		s	Вывести строку, оканчивающуюся нулем
		n	Вывести указатель на целое
		p	Вывести указатель в виде SSSS:OOOO или OOOO

Следующий пример даст вам хорошее представление о возможностях и гибкости функции printf.

```
/*************************************************************/
/* PRINTF.C:  Пример, демонстрирующий многосторонние         */
/*     возможности функции printf...                         */
/*************************************************************/
#include <stdio.h>
int main( void )
{
   int i=11;
   float f=500.78653;
   printf( "Hello, world!\n" );  /* Вывести строку */
   /* Вывести несколько строк... */
   printf( "%s, %s!\n", "Hello", "again" );
   /* Вывести целое число в различных формах... */
   printf( "Integer = (dec)%d, (oct)%#o, (hex)%#x\n", i, i, i );
   /* Управление шириной поля и точностью... */
   printf( "Float = %f, %6.2f\n", f, f );
   /* Выравнивание по левому краю и заполнение нулями... */
   printf( "%-7.2f+ %d = %08.2f\n", f, i, f+i );
    return 0;
```

Escape-последовательности

Обратная косая черта (\) имеет в языке C специальное значение. Ее называют *escape-символом* (эскейп). Ее применяют для представления символов или чисел, которые нельзя непосредственно ввести с клавиатуры. Например, когда вы редактируете текст, клавиши Backspace и Enter связаны с особыми функциями. Чтобы использовать такие коды в программе, можно применить *escape-последовательность,* т.е. escape-символ, за которым следует escape-код. В таблице 2.4 перечислены допустимые в C escape-последовательности.

Таблица 2.4. Escape-последовательности

Последовательность	*Название*	*Функция*
\a	Звонок	Подает звуковой сигнал
\b	Backspace	Возврат на один символ
\f	Перевод страницы	Начало нового экрана
\n	Новая строка	Переход к началу новой строки
\r	Возврат каретки	Возврат к началу текущей строки
\t	Табуляция	Переход к следующей позиции табуляции
\v	Вертикальная табуляция	Переводит курсор вниз на несколько строк
\\	Обратная черта	Выводит обратную косую черту
\'	Апостроф	Выводит апостроф
\"	Кавычка	Выводит двойную кавычку

Можно также использовать escape-символ для представления символов восьмиричном или шестнадцатиричном формате. В таблице 2.5 показан такое применение escape-последовательностей.

Таблица 2.5. Представление символов значением

Формат	*Основание*	*Описание*
\OOO	8	От одной до трех восьмиричных цифр, следующих за escape-символом
\xHH или \XHH	16	Одна или две шестнадцатиричных цифры, следующих за \x или \X

Функция printf будет преобразовывать escape-последовательности, входящие в строку формата, в соответствующие коды, что расширяет возможности управления форматом. Вы уже видели, как применялась последовательность '\n' в нескольких вызовах printf.

Функции scant, gets, atoi, atol и atof

Функция scanf является противоположностью printf. Точно так же, как printf, эта функция ожидает в качестве аргумента строку, содержащую одну или несколько *спецификаций формата,* указывающих формат и тип данных, которые должны быть прочитаны. Дополнительные аргументы, следующие за строкой формата, должны быть адресами переменных, в которых данные будут сохраняться.

Однако, часто программисты избегают пользоваться функцией scanf. Если данные, прочитанные с помощью scanf, не соответствуют строке формата, то функция может вести себя непредсказуемо. Так как нельзя, вообще говоря, ожидать, что пользователь вашей программы будет вводить данные в точном соответствии с форматом, вы можете сделать выбор в пользу функции gets вместо мощной, но требовательной scanf. Функция *gets* читает вводимые данные в указанный вами буфер. Данные представляются в виде строки. Если должно быть введено число, то вы можете затем вызвать функцию atoi, atol или atof для преобразования строки соответственно в целое, длинное целое или вещественное число. Вот пример:

```
/**************************************/
```

```
/* IN_OUT.C: Пример, иллюстрирующий ввод/вывод...    */
#include <stdio.h> /* В stdio.h описаны printf/gets */
#include <stdlib.h> /* В stdlib.h описана atoi     */
int main( void )
{
char name[80],   /* Массив char для имени       */
ageStr[80];   /* Массив char для возраста     */
int age=0;    /* Целое для прочитанного возраста */ /* Попросите пользователя
ввести имя... */
printf( "Пожалуйста, введите ваше имя: " ) ; /*
Прочитайте имя в массив */ gets( name );
/* Приветствуйте пользователя; спросите о возрасте */
printf( "Привет, %s ! Сколько вам лет? ", name ); /*
Прочитайте возраст в виде строки символов... */
gets( ageStr );
/* Преобразование строки в целое число... */
age = atoi( ageStr );
/* Проверьте, правильно ли введено число
*/ if ( age != 0 )



/* Напечатайте возраст */
printf( "Ничего себе, %d\" ); else
/* Скажите, что мы все понимаем... */
printf( "Конечно, не будем об этом говорить" )
return 0;
```

Пример функции

Теперь, когда вы знакомы с методами чтения и отображения информации, мы попробуем объединить в одном примере несколько простых функций.

Прототип функции

Нередким является случай, когда функция описывается до того, как она будет определена. Описание информирует компилятор о существовании функции, о типе возвращаемого значения, а также о типе параметров, которые ей передаются. Описание функции часто называют *прототипом функции*. Описание функции имеет следующий вид:

<возвращаемый тип> ИмяФункции(параметры);

Некоторые характеристики прототипов функций перечислены ниже:

• Тип возвращаемого значения может быть одним из тех типов данных, с которыми вы уже знакомы по предыдущим разделам (или типом данных, определенным пользователем). Функции, не возвращающие значения, обычно имеют тип void (пустой).

• В списке параметров обычно указываются тип и имя для каждой переменной; элементы списка разделяются запятыми. Указание имени переменной в прототипе не обязательно, но, как правило, применяется.

• Как и в случае возвращаемого значения, функции, которые не предполагают передачи параметров, описываются прототипом с ключевым словом void на месте списка аргументов.

Справка: Прототип и определение функции

Определение функции будет одновременно являться прототипом, при условии, что функция определяется до того, как будет вызвана, и при условии, что она определена с void в качестве параметра, если не предполагается передачи аргументов.

Предположим, вы столкнулись с необходимостью написать функцию, которая возвращает среднее значение трех величин. Прежде всего, вам нужно решить, какой тип данных использовать для представления чисел. Сверившись с таблицей типов, вы решаете перестраховаться и выбираете длинные целые, потому что значения чисел могут выходить за пределы диапазона обычных целых. Итак, вы пишете следующий прототип:

```
long Average( long val1, long val2, long val3 );
```

Прототип информирует компилятор, что функция *Average* предполагает передачу ей трех параметров типа long и что возвращает она также значение типа long. Далее вы определяете тело функции.

Определение функции

Ниже приводится возможная реализация функции Average:

```
/* Определение функции Average */
long Average( long val1, long va!2, long va!3 )
{
long sum = val1+val2 + val3;
return sum/3; }
```

Чтобы убедиться, что функция написана правильно, вы решаете запросить ввод трех чисел и затем показать их среднее. Хорошо. Из осторожности вы отказываетесь от использования scanf в пользу более безопасной комбинации gets/atoi. Так как вам придется запрашивать ввод числа несколько раз, то представляется разумным локализовать процедуру чтения в отдельной функции, назвав ее, например, ReadLong. В результате получается следующее:

```
/**************************************************/
/*     AVGNUM.C:     Функция,     вычисляющая     среднее     трех     чисел     */
/**************************************************/
#include <stdio.h>
#include <stdlib.h>
/* Прототипы функций */
long Average( long val1, long val2, long val3 );
long ReadLong( void );
/* Определение функции Average */
long Average( long val1, long val2, long val3 )
{
long sum = val1 + val2 + val3;
return sum/3; }
/* Определение функции Headlong */ long ReadLong( void ) {
char buffer[80];
gets( buffer );  /* прочитать строку */
return atoi( buffer ); /* строка - long  */ }
int main( void ) {
long 11, 12, 13, avg;
printf( "Пожалуйста, введите первое число: " );
11 = ReadLong();
printf( "Пожалуйста, введите второе число: " );
```

```
12 = ReadLong();
printf( "Пожалуйста, введите третье число: " );
13 = ReadLong(); avg = Average( 11, 12, 13 );
printf( "Среднее значение трех чисел = %ld \n", avg ); return 0; }
```

Выражения и операции

Функция обычно содержит одно или несколько выражений, которые обрабатывают данные или вычисляют значение. В функции Average для сложения чисел используется операция +. Язык C предоставляет в распоряжение программиста богатый набор операций, с помощью которых вы описываете манипуляции с данными, которые должна производить ваша функция. Когда выражение содержит более, чем одну операцию, порядок их выполнения определяется соотношением приоритетов. Предполагается, что операция с более высоким приоритетом выполняется раньше. Операции с одинаковым приоритетом обрабатываются в соответствии с их ассоциативностью. Таблица 2.6 дает краткое описание операций с указанием их приоритета и ассоциативности.

Таблица 2.6. Операции языка C

ПП

Операция	*Описание*	*Пример*	*Приоритет/ Ассоциативность*
	Первичные и постфиксные		
[]	индекс массива	columns[5]	16, слева направо
()	вызов функции	printf(msg)	16, слева направо
.	элемент структуры	time.tm_hour	16, слева направо
->	элемент структуры	time->tm_hour	16, слева направо
++	постфиксное приращение	counter++	15, слева направо
--	постфиксное уменьшение	counter--	15, слева направо
	Унарные операции		
++	префиксное приращение	++counter	14, справа налево
--	префиксное уменьшение	--counter	14, справа налево
sizeof	размер в байтах	sizeof(custRec)	14, справа налево
(тип)	преобразование типа	(float)i	14, справа налево
~	побитовое НЕ	~WS_VISIBLE	14, справа налево
!	логическое НЕ	!EOF	14, справа налево
-	унарный минус	-i	14, справа налево
&	адрес	&aVar	14, справа налево

Таблица 2.6. Продолжение

Операция	*Описание*	*Пример*	*Приоритет/ Ассоциативность*
*	разыменование	*ptr	14, справа налево
	Бинарные и тернарные операции		
	Мультипликативные		
*	умножение	aVar*10	13, слева направо
/	деление	aVar/10	13, слева направо
%	взятие по модулю	aVar%10	13, слева направо
	Аддитивные		
+	сложение	aVar+20	12, слева направо
-	вычитание	aVar-20	12, слева направо
	Побитовый сдвиг		
<<	сдвиг влево	aVar<<1	11, слева направо
>>	сдвиг вправо	aVar>>1	11, слева направо
	Операции отношения		
<	меньше, чем	i<counter	10, слева направо
>	больше, чем	i>counter	10, слева направо
	Равенство		
==	равно	value==0	9, слева направо
!=	не равно	value!=0	9, слева направо
	Битовые		
&	побитовое И	style & WS_BORDER	8, слева направо
^	побитовое исключающее ИЛИ	flag^msk	7, слева направо
|	побитовое ИЛИ	style | WS_VISIBLE	6, слева направо

Таблица 2.6. Продолжение

ППП

Операция	*Описание*	*Пример*	*Приоритет/ Ассоциативность*
	Логические		
&&	логическое И	IEOF&& sizeReadX)	5, слева направо
¡	логическое ИЛИ	a==0 ‖ Ь==0	4, слева направо
	Условные		
?:	при условии	a>Ь? 1:0	3, справа налево
	Присваивание		
=	присваивание	x = 10	2, справа налево
*=	присвоение произведения	x*= 10	2, справа налево
/=	присвоение частного	x /= 10	2, справа налево
%=	присвоение остатка	x %= 10	2, справа налево
+=	присвоение суммы	x += 10	2, справа налево
-=	присвоение разности	x -= 10	2, справа налево
<<=	присвоение левого сдвига	v a r « = 10П	2, справа налево
» =	присвоение правого сдвига	var »= 100	2, справа налево
&=	присвоение И	i & = j	2, справа налево
Л_	присвоение исключающего ИЛИ	i ^ = j	2, справа налево
f=	присвоение ИЛИ	i | = j	2, справа налево
	запятая	x=2, y=3	1, слева направо

П

```
/*  OPERATOR.C: Пример, использующий некоторые из операций...   */

(include <stdio.h>
(include <stdiib.h>
lnt inain( void }
{
   int il = 10, 12 =20;
   /* Операция sizeaf...  */
   prlntf( "Размер int равен lkd байтам \n", sizeof( int ) );
   printf( "Размер long равен Xd байтам \n", sizeof( long ) );
   /* Операции сдвига...  */
   printf{ "Xd • 2 = Xd  и  id « 1 = Xd\n", i1, 11*2,
                                              11. 11 « 1 );
   printfC "Xd / 2 = Xd  и  Jd » 1 = Xtt\n", 12. 12/2.
                                              12, 12 » 1 );
   return 0;
)
```

Условные операторы и циклы

До сих пор все наши примеры программ выполнялись в линейной последовательности. Все операторы исполнялись ровно по одному разу, в том порядке, как они встречались в тексте программы. Вы, вероятно, заметили, что такой подход обладает весьма ограниченными возможностями. Большинство задач, которые вам придется решать, требуют, чтобы ваша программа "принимала решения". Язык С предусматривает различные конструкции, позволяющие вам управлять потоком исполнения операторов в программе. Вы можете, например, определить, нужно ли выполнять данный оператор, проверив некоторое условие. Возможности управления расширяются тем, что можно составить блок из нескольких операторов и обращаться с ним как с одиночным оператором.

Блок начинается с открывающей фигурной скобки ({) и заканчивается закрывающей скобкой (}). Функция, собственно, является блоком, и нередко можно встретить выражение *функциональный блок*.

Применение операторов if и else

Оператор if осуществляет условное ветвление программы, проверяя истинность выражения или комбинации выражений. Он имеет следующий вид:

if (*выражение*) *оператор исполняемый_если_выражение_истинно;*

Следующий код иллюстрирует применение оператора if с простым исполняемым оператором.

```
/*********************************************************/
/* IF.C:  Условное ветвление с использованием if(...)     */
/*********************************************************/
#include <stdio.h> // Описывает printfQ
tfinclude <dos.h> // Описывает _dos_gettime()   int main( void )
struct time t;
_dos_gettime ( &t );
if ( t.ti_hour < 12 )
printf(  "Доброе утро!   " );
if ( t.tijiour >= 12 )
printf( "Добрый день!  " );                                    .
```

```
printf( "Сейчас %02d:%02d:!l!02d\n",
t.ti_hour, t.ti_min, t.ti_sec ); *\
return 0;
```

При необходимости в комбинации с if можно использовать ключевое слово else, позволяющее выполнить альтернативный оператор, если выражение в условии неистинно. Ниже следует тот же самый пример, упрощенный за счет применения комбинации if/else.

```
/*****************************************.**************/
/* IFELSE.C: Условное ветвление с if / else... */
/*************************************************/
#include <stdio.h>          // Описывает printf()
#include <dos.h>             // Описывает _dos_gettime() int main( void )
{


struct time t;
_dos_gettime (&t);
if ( t.ti_hour < 12 ) printf( "Доброе утро! " );
else
printf( "Добрый день! " );
printf( "Сейчас %02d: %02d: %02d\n", t.ti_hour, t.ti_min, t.ti_sec );
return 0;
}
```

Операторы if и else могут быть вложенными. Если такая конструкция является двусмысленной, компилятор ставит каждое else в соответствие ближайшему if. Рассмотрим следующий пример:

```
/* IFELSE2.C: Иллюстрирует вложенные операторы if/else    */
#define NOWORK_RATE 0 #define
NORMAL_RATE 1
#define DOUBLE_RATE 2 ...
const float hrlyRate = 6.0;
float GetDayEarnings( int hours )  {
int rate = NORMAL_RATE;   /* Предположим обычную ставку */
if ( hours > 0 )    /* Проверим присутствие...   */
if ( hours > 8 )      /* Сверхурочная работа?    */
rate = DOUBLE.RATE;
rate = NOWORK_RATE;
return hrlyRate * hours * rate; }
```

Этот пример требует применения скобок, так как else будет поставлено в соответствие внутреннему if, что приведет к неправильному решению. Вот исправленный вариант:

```
/* IFELSE2.C:  Иллюстрирует вложенные операторы if/else */
#define NOWORK_RATE 0


#define NORMAL_RATE 1
#define DOUBLE_RATE 2
const float hrlyRate = 6.0;
float GetDayEarnings( int hours )
{
int rate = NORMAL_RATE;      /* Предположим обычную ставку */ if ( hours > 0 ) /* Проверим присутствие...      */ { if ( hours >
8 )             /* Сверхурочная работа?          */
rate = DOUBLE_RATE; } else
rate = NOWORK_RATE; return hrlyRate * hours * rate; }
```

Применение операторов switch и case

В некоторой точке вашей программы может оказаться несколько (более двух) возможных путей ветвления. Например, вы можете разработать меню, предлагающее пользователю выбор из нескольких команд. Для выбора нужного варианта в такой ситуации можно использовать последовательность операторов if/else, как показано ниже:

```
void ProcessBBSMenu()
{
char option = GetSelectedOption(); option = toupper( option ); if   ( option

== 'F' )
FileMenu(); else if ( option == 'M' )
MessageMenu(); else if ( option == 'G' )
LogOffUser(); else
ShowOptions();
}
```

В качестве альтернативы, для сложного условного ветвления язык С предоставляет конструкцию с ключевыми словами switch и case. Синтаксис ее следующий:

switch (*выражение*)
{
case константное_выражение: оператор или группа операторов case константное_выражение: оператор или группа операторов
default: оператор или группа операторов }

Результат вычисления *выражения* сравнивается с каждым из *константных_выражений*. Если находится совпадение, то управление передается оператору или операторам, связанным с данным case. Заметьте, что исполнение продолжается до конца тела оператора switch или пока не встретится оператор break, который передает управление из тела switch вовне.

Операторы, связанные с ключевым словом default, выполняются, если выражение не соответствует ни одному из константных выражений в case. Default является опциональным и не обязательно располагается в конце.

Пример с функцией ProcessBBSMenu можно переписать, использовав преимущества, которые дает оператор switch, например, так:

```
void ProcessBBSMenu() {
switch ( GetSelectedOption () ) {
default: ShowOptions (); case 'f' case 'F'   FileMenu ();
break; case 'm' case 'M' MessageMenu ();
break; case 'g' case 'G' LogOffUser ();
break;
}
```

Оператор while

Ключевое слово while позволяет выполнять оператор или блок до тех пор, пока условие не перестанет быть истинным. Синтаксис его следующий:

while (*выражение*) *оператор*

Оператор или тело блока, связанного с while, не будет выполняться, если выражение изначально ложно. Следующий пример иллюстрирует применение оператора while в процедуре, которая просто ждет, когда закончится заданный отрезок времени.

```
    /************************************************/
I    /* WHILE.C: Пример, иллюстрирующий применение    */
/*      цикла while... */
#include <stdio.h> #include <time.h> void WaitFor( int secs ) {
time_t start = time( NULL ), now = start;
while( now < start+secs )
now = time( NULL ); }
int main( void ) {
```

```
printf( "Ждем в течение 5 секунд..." );
WaitFor( 5 );
printf( "\nГотово!" );
return 0;
}
```

Подсказка: Копирование строк с помощью оператора while

Ветераны программирования на C часто приводят следующий цикл while в качестве эффективного метода копирования строк, оканчивающихся нулем:

```
void CopyString( char *dest, char *src )
{
   while( dst++ = src++ );
}
```

Пустой оператор

Вы, вероятно, заметили, что в примере CopyString в цикле while нет никакого оператора или блока. Но он есть! Одинокая точка с запятой (;) является оператором и называется *пустым оператором*. Используйте пустой оператор там, где наличие оператора предполагается, но вам не нужно выполнять никаких действий.

Оператор for

Цикл for похож на цикл while, но дает вам две дополнительных возможности:

• Вы можете включить в оператор инициализирующее выражение, исполняемое один раз перед тем, как будет произведена оценка условия.

• Вы можете указать выражение, которое будет исполняться после каждой итерации оператора или блока, связанного с циклом for.

Синтаксис оператора следующий:

for *(инициализация;* условное.выражение; *конвчное_выражение') оператор или блок операторов*

Нужно иметь в виду, что все три выражения являются необязательными. Инициализирующее выражение, если есть, всегда будет выполняться; вычисление конечного выражения может не производиться, если условие ложно с самого начала. Следующий пример является модифицированным вариантом функции WaitFor, в котором использован цикл for.

```
/*****************************.*******************/ /* FOR.C: Пример, иллюстрирующий применение */ /* цикла for... */
/*************************************************/
#include <stdio.h> #include <time.h> void WaitFor( int secs ) {
time_t start;
for( start=time( NULL ); time( NULL ) < start+secs;   )


;   // Пустой оператор }
int main( void ) {
printf( "Ждем в течение 5 секунд..." );
WaitFor( 5 );
printf( "\nГотово!" );
return 0; }
```

В данном примере в цикле for отсутствует конечное выражение, а телом цикла является пустой оператор.

Подсказка: Бесконечные циклы с применением for или while

Вы можете написать цикл, которыйо не кончается никогда, опустив все три выражения в операторе for. Можно также написать бесконечный цикл, использовав в операторе while в качестве условия выражение, которое всегда истинно! Конечно, программу, выполнение которой не завершается, нельзя назвать корректной. Однако, как вы увидите в дальнейшем, существуют способы прекратить исполнение самых некорректных циклов.

```
#include <stdio.h>
void forever1( void )
{
   for( ;; )    // Forever (навсегда)...
    printf( "Это никогда не кончится...\n" );
}
void forever2( void )
{
   while( 1 )   // Условие всегда истинно
    printf( "Это тоже не кончается...\n");
}
```

Цикл do/while

В цикле do/while оценка условия производится после исполнения тела цикла. Это означает, что оператор или блок операторов, связанный с циклом,

будет обязательно исполняться хотя бы один раз. Синтаксис оператора имеет следующий вид:
do оператор или блок операторов while (выражение)
Вы увидите, что цикл do/while весьма полезен, когда вам нужно спросить у пользователя, следует ли повторять определенную операцию. Вот пример:
/**./ /* DOWHILE.C: Пример использует цикл do/while, чтобы */ /* выполнить действие и запросить указание */ /**/
#include <stdio.h>
#include <conio.h>
#include <dos.h>
/* Прототип */
```
void Showlnfo( void );
/* Showlnfo: Выводит некоторую информацию о системе */
void Showlnfo( void )
{
clrscr();
printf( "Версия ОС: %d.%02d \n", _osmajor, _osmlnor );
printf( "PSP     : %%\n", _psp ); }
int main( void ) {
int answer;
do {
ShowInfo();
printf( "Хотите завершить исполнение? " ); answer = getch();
} while( answer != 'Y' && answer != 'Y' );
printf( "Сделано...\n" );
return 0; }
```

Прерывание выполнения блока

Бывают ситуации, когда необходимо прервать выполнение блока операторов независимо от каких-либо условий. Язык С предусматривает для этой цели три ключевых слова: break, continue и return. Оператор break прекращает выполнеие того оператора while, do/while, for или switch, в котором он непосредственно находится. Вы можете, например, организовать обработку ошибок, исполняя оператор break, если обнаружилось нечто непредвиденное. Предположим, вам нужно ввести очень длинный список имен и фамилий. Удобно использовать для этой цели бесконечный цикл while, в котором у пользователя будет запрашиваться информация, и применить оператор break для завершения цикла, когда ввод закончен. Вот возможная реализация:

```
/*******************************************************/
/* BREAK.C: Использует 'break' для завершения цикла   */
/*******************************************************/
#include <stdio.h>
/* Сохраняет данные */
void Savelnput( char *firstName, char *lastName )
{
/* Пустышка; пока нереализовано */ }
/* Вводит имена и фамилии, пока не встретится пустая строка */ void
Processlnput( void ) {
char fname[80]; char lname[80]; while( 1 ) {
printf( "Введите имя   : " ); if ( !gets( fname ) || lfname[0] ) /* Проверить
ввод */
break;      /* Выйти их цикла */ printf( "Введите фамилию: " ); if ( !gets(
lname ) || !lname[0] ) /* Проверить ввод */
break;      /* Выйти из цикла */ Savelnput( fname, lname ); } }
int main( void )
{


ProcessInput();
printf( "Конец..." );
return 0;
```

Подсказка: Прерывание оператора switch

Применение оператора break в контексте блока switch является обычным. Без прерывания управление передавалось бы дальше, всем вариантам case, которые следуют за тем, который соответствует значению управляющего выражения. Такое поведение этой конструкции может оказаться удобным, если для двух или большего числа вариантов нужно выполнить сходные действия.

Оператор continue очень похож на оператор break, за исключением двух моментов:

- Оператор continue не влияет на выполнение операторов switch.
- continue возвращает управление к началу цикла, пропуская оставшуюся его часть.

Следующий пример вызывает функцию random, пока не будет найдено 10 нечетных чисел. Каждый раз, когда генерируется четное число, выполняется оператор continue, который передает управление обратно на начало цикла.

```
/* CONTINUE.C:  Пример,  иллюстрирующий применение 'continue'     */
•include <stdlib.h> #include <stdio.h> #include <time.h> /* Прототип функции */ void InitRoutlne( void ); void InitRoutine()
{
```

```
printf( "Производим инициализацию... \n" );
randomize();
}


int main( void ) {
int count = 0;
for( InitRoutine(); count<10; ) {
int val = random( 100 ); if ( val % 2 == 0 ) {
putchar( '.' ); continue; }
count++;
printf( "\nНайдено нечетное число: %d ", val ); }
return 0; }
```

Чтобы прервать выполнение функции, вы можете использовать оператор return, с необязательным выражением в качестве возвращаемого значения. Оператор имеет следующий вид:

return *выражение;*

Функции, описанные как void, не возвращают значения. Если оператор return отсутствует, то предполагается, что он следует за последним оператором функции. Однако возвращаемое значение в этом случае не определено.

Применение операторов goto и меток

Оператор goto осуществляет безусловную передачу управления на метку в пределах текущей функции. Синтаксис его следующий:

goto *метка*

Метка - это идентификатор с двоеточием (:), который помещается перед помечаемым оператором.

Совет: Неизбежное зло — goto

Хотя goto и является допустимым ключевым словом в языке С, вы должны иметь в виду, что использование его не рекомендуется; предпочтительне применять условные конструкции или операторы цикла. На самом деле, нужно по возможности избегать даже применения операторов break и continue. Программы с безусловными переходами часто трудно понимать. Однако, goto часто используют, когда непредвиденное состояние или ошибка может возникнуть внутри многократно вложенных циклов. В такой ситуации goto может значительно упростить код.

Область действия переменных

Под областью действия переменной понимают область программы, в которой переменная доступна для использования.

Локальные переменные

Функции, с которыми вы к этому моменту уже познакомились, содержали объявления переменных внутри определения функции или функционального блока. Такие переменные называются локальными и доступны только в пределах функции или блока, с которых они определены. В следующем примере var1 доступна только внутри функции fimc(), а var2 активна только в main().

```
/* SCOPE1.C: Иллюстрирует область видимости локальных */ /* переменных... */
/*******************************************************/
#include <stdio.h>
void func( void )
{
int varl =11;   /* Локальная переменная var1 */
printf( "Внутри func() значение varl = %d\n", varl );


int maln( void ) {
int var2 = 22;   /* Локальная переменная var2 */ printf( "Внутри main()
значение var2 = %d\n", var2 ); func();
/* Следующий оператор закомментирован, т.к. main() не может обращаться к
varl и возникает ошибка 'Undefined symbol varl'...
printf( "Внутри main() значение varl = %d\n", varl );
*/
return 0;
}
```

Глобальные переменные

Переменная, объявленная вне любой из функций, называется *глобальной переменной* и доступна в области от точки ее объявления до конца файла. Например, чтобы сделать varl доступной как в func(), так и в main(), вы можете описать эту переменную в начале файла, как показано в следующем примере:

```
/*******************************************************/
/* SCOPE2.C: Иллюстрирует область видимости локальных  */
/*   и глобальных переменных... */
y*******************************************************/
#include <stdio.h>
int varl =11;    /* Глобальная переменная varl */
void func( void )
{
printf( "Внутри func() значение varl = %d\n", varl ); }
int main( void ) {
int var2 = 22;   /* Локальная переменная var2 */
printf( "Внутри main() значение var2 = %d\n", var2 );
func();
printf( "Внутри main() значение varl = %d\n", varl );
return 0;
}
```

Видимость переменных

Описание локальных переменных возможно не только в начале функции. Можно объявить локальную переменную в начале блока. Переменная, объявленная в блоке, "прячет" любую другую переменную с таким же именем, описанную вовне. В следующей программе, например, вызовы printf() для переменной var1 выводят три различных значения.

```
/* SCOPE3.C: Иллюстриует область видимости локальных */ /* и глобальных переменных... */
/*******************************************************/
#include <stdio.h>
int var1 = 12; /* Глобальная переменная var1 */
int main( void )
```

```
{
if ( printf( "Входим во внешний блок if() \n" ) )
{
int var1 = 34;
printf( "Во внешнем блоке if() var1=%d", var1 ); if
{
printf( "Входим во внутренний if()\n" ) )
{
var1=56;
printf( "Во внутреннем if() var1=%d\n", var1 ); }
}
printf( "В main() значение var1 = %d\n", var1 );
return 0;
}
```

Время жизни переменной

Память для локальных переменных выделяется, когда начинается выполнение функции. Как только происходит возврат из функции, эта память снова становится доступной и может быть использована другими функциями. Таким образом, нельзя ожидать, что данные, содержащиеся в локальной переменной, останутся неизменными при последующих вызовах функции. Однако, если вам нужно, чтобы значение локальной переменной сохранялось в промежутках между вызовами функции, вы можете описать переменную с модификатором static. Ключевое слово static выделяет для переменной постоянную память.

Модификаторы переменных

В языке С имеется несколько модификаторов, таких, как static, которые изменяют область действия и время жизни переменных. В таблице 2.7 описаны различные модификаторы.

Таблица 2.7. Модификаторы переменных

Модификатор	*Применение*	*Область действия*	*Выделение памяти*	*Объяснение*
auto	локальная переменная	блок	временное	Для локальных переменных применяется по умолчанию
register	локальная переменная	блок	временное	Предполагается, что переменная размещается в машинном регистре
extern		блок		Информирует компилятор, что переменная определяется в другом файле
static	локальная переменная	блок	постоянное	Область действия соответствует локальной переменной, а время жизни — глобальной
static	глобальная переменная	файл	постоянное	Ограничивает область действия глобальной переменной текущим файлом

Изменяющиеся переменные

Еще одним ключевым словом, часто используемым при описании переменных, является volatile. Этот модификатор сообщает компилятору, что значение переменной может изменяться периферийным устройством или некоторой фоновой процедурой. Поэтому компилятор не будет пытаться оптимизировать вашу программу, помещая значение переменной в регистр.

Следующий пример иллюстрирует применение ключевого слова volatile:

```
/*****************************************************/
/* VOLATILE.C: Описание переменных как volatile...   */
/**.**************************************************/
volatile int vInt; /* Изменяющееся целое */
const  int cInt =10; /* Целая константа */
volatile const vcInt = 100; /* Изменяющаяся константа*/
```

Подсказка: volatile и const

Заметьте, что переменная volatile может быть одновременно объявлена как константа. В таком случае компилятор не допустит, чтобы ваша программа изменяла значение переменной, и в то же время не будет делать никаких предположений о ее содержимом.

Массивы

Очень часто вашей программе может потребоваться сохранять и обрабатывать некоторое множество значений одного и того же типа. Предположим, вам нужно хранить температуру воздуха для каждого дня недели. Вы можете определить семь целых переменных для этих значений. Однако язык С предлагает лучший метод — использовать массив. *Массив* представляет собой набор данных одного типа. Формат определения массива следующий:

тип_данных имя_массива[размер_массива];

Для представления ваших данных подойдет такое описание:

```
int temperature[7];        // Без инициализации
```

или

```
int temperature[7] = { 78,  79,  90,  99,  86,  75,  81 };
```

Для доступа к элементу массива используется имя массива, за которым следует индекс в квадратных скобках. Вот программа, вычисляющая среднюю температуру:

```
/.*************************************************/
/*ARRAY.C: Пример применения массива... */
/************************************************/
#include <stdio.h> #define NUMDAYS 7 int main( void ) {
int temperature[NUMDAYS] = { 78, 79, 90, 99,  86, 75, 81 };
```

```
int indx, sum;
for( indx=0; indx<NUMDAYS; indx++) printf( "День %d-й,
температура %d F\n",
indx+1,
temperature[indx] );
for( indx=0, sum=0;
indx < sizeof( temperature )/sizeof( temperature[0] );
indx++ )
{
sum += temperature[indx];
}
printf( "Средняя температура Xd F\n", sum/NUMDAYS );
return 0;
}
```

Вы можете сделать массив многомерным, задав несколько размеров, заключенных в скобки. Двумерные массивы обычно используются для представления матриц, координат на плоскости и шахматных досок.

Указатели

Указатель является переменной, которая содержит адрес другой переменной или функции. Описание указателя определяет тип данных, на которые указатель ссылается. Описание указателя имеет следующий вид:

тип_указываемых_данных *имя_указателя; Вот возможные описания указателей:

```
int    *int_ptr;    /* Указатель на целое   */
double *dbl_ptr;    /* Указатель на тип double */
```

Для доступа к объекту, на который ссылается указатель, используется имя указателя со звездочкой перед ним. Например, *dbl_ptr представляет собой число двойной точности, на которое указывает dbl_ptr.

Вот пример, демонстрирующий применение указателей:

```
/*************************************************************************/
/* POINTER.C: Иллюстрирует применение указателя... */
/*************************************************************************/
#include <stdio.h>
#include <stdlib.h>
int main( void )
{
int 11 = 100;
/*
* Определить указатель на целое i1
*/
int *iptrA = &i1,
*iptrB;
/*
* Показать, что iptrA указывает на i1
*/
printf( "Адрес i1 равен       %p \n", &i1 );
printf( "iptrA указывает на адрес %p \n", iptrA );
/*
* Разыменовывая указатель, можно получить доступ
* к значению переменной i1
*/
```

```
printf( "Значение i1   = %d \n", i1 ); printf( "Значение
*iptrA = %d \n", *iptrA ); /*
* Использование указателя
* для работы с динамической памятью
*/
iptrB = malloc( sizeof( int ) );
/*
* Поместить значение i1 в выделенную память
*/
*iptrB = i1;
/*
* Показать содержимое динамической переменной
*/
```

```
printf( "Значение *iptrB = Хпрd \n", *iptrB );
free( iptrB );      /* Очистка... */
return 0;
}
```

Подсказка: Перед использованием указателя

Прежде чем разыменовывать указатель, его надо инициализировать. Одним возможным способом является присвоение указателю адреса переменной. Или, в качестве альтернативы, вы можете присвоить ему адрес, возвращаемый функцией распределения памяти malloc. Если память была выделена успешно, ее следует освободить, вызвав функцию free, когда она больше не будет нужна.

Указатель типа void может указывать на объект любого типа; его обычно называют *пустым указателем*. Функция malloc возвращает пустой указатель. Это позволяет присваивать адрес выделенной памяти указателям любого типа. Следующий пример показывает применение указателя на функцию.

```
/* PTRFUNC.C: Применение указателей на функции... */
/*******************************************************/
#include <stdio.h>
int ShowMsg( char *msg )
{
return printf( msg ); }
int main( void ) {
/* Определить указатель на функцию... */
int (*ptrFunc)( char* );
/* Инициализировать указатель на функцию... */
ptrFunc = ShowMsg;
/* Вызвать функцию через указатель... */
(*ptrFunc)( "Привет! \n" );
return 0;
}
```

Подсказка: Вызов функции с помощью указателя

Чтобы вызвать функцию через указатель_на_функцию, вы можете написать:

```
(*указатель_на_функцию)([параметры]);
```

или просто

```
указатель_на_функцию([параметры]);
```

Хотя вторая форма проще и изящнее, предпочтительнее все же первая, так как она ясно показывает, что указатель_на_функцию является указателем, а не функцией.

Массивы и указатели

В языке С массивы и указатели тесно связаны друг с другом. Имя массива соответствует адресу его первого элемента. Поэтому можно присваивать указателю адрес первого элемента, используя просто имя массива.

```
int   iarray[10];
int *iptr = iarray;   /* То же,  что iptr = &iarray[0] */
```

В С указатели можно увеличивать или уменьшать. Указатель, увеличенный на 3, будет указывать на четвертый элемент массива (эквивалентно &iarray[3]). Другими словами, увеличивая указатель на единицу, вы в деиствительности увеличиваете адрес, который он представляет, на размер объекта связанного с ним типа. Так как указателям типа void не соответствует никакой тип данных, к ним нельзя применять арифметические операции.

Указатель можно индексировать точно так же, как массив. Компилятор, \ на самом деле, преобразует индексацию в арифметику указателей. Например,

```
iarray[3] = 10; представляется как
*( iarray + 3 ) = 10;
```

```
/*********************************************************/
/* PTRARRAY.C: Иллюстрирует связь указателей с массивами  */
/*********************************************************/
#include <stdio.h> #include <stdlib.h> #define NUM_ELEMENTS 10 int main(
void ) {
int  iarray[NUM_ELEMENTS];
int *iptr;
int indx;
/* Применение индексации к массиву      */
for( indx=0; indx<NUM_ELEMENTS; indx++ )
iarray[indx]= indx*10;
/* Применение к массиву арифметики указателей */
for( indx=0; indx<NUM_ELEMENTS; indx++ )
printf( "Элемент %d = %d\n", indx, *( iarray + indx ) );
/* Присвоение указателю адреса массива   */
iptr = iarray;
```

```
    /* Применение индексации к указателю      */
    for( indx=0; indx<NUM_ELEMENTS; indx++ )
    printf( "Элемент %d = %d\n", indx, iptr[indx] );
    return 0;
}
```

Подсказка: Вызов функции с помощью указателя

Чтобы вызвать функцию через указатель_на_функцию, вы можете написать:

```
(*указатель_на_функцию)([параметры]);
```

или просто

```
указатель_на_функцию([параметры]);
```

Хотя вторая форма проще и изящнее, предпочтительнее все же первая, так как она ясно показывает, что указатель_на_функцию является указателем, а не функцией.

Типы, определяемые пользователем

Базовые типы данных С, а также массивы и указатели, являются тем фундаментом, на котором строится обработка реальной информации, но их недостаточно для представления некоторых сложных совокупностей данных, с которыми вам придется иметь дело. Коммерческой программе могут потребоваться типы данных, описывающие заключенную сделку. Научной программе - тип данных, представляющий температуру и давление для определенного момента времени. Язык С позволяет вам определять, или создавать, типы данных.

Переименование типов

Вы можете использовать ключевое слово typedef, чтобы приписать типу данных новое имя. Применение новых имен может сделать текст программы более понятным. Определение типа с typedef имеет следующий вид:

typedef *<тип> <новое_имя_типа>* или
typedef *<тип> <новое_имя_типа>[размер _массива][...*]

Следующий пример определяет два новых типа, Msg и Msglndx.

```
/* TYPEDEF.C:  Иллюстрирует применение 'typedef */
#include <stdio.h>
#include <string.h>
#define MAX_MSG_LEN    100
typedef short Msglndx;
typedef char    Msg[MAX_MSG_LEN];
int main( void )
{
Msglndx indx;
/* Создать массив из Msg */


Msg _msgs[3];
/* Инициализировать сообщения   */
strcpy( _msgs[0], "Привет!\n" );
```

```
strcpy( _msgs[1],   "C - это здорово!\n" );
strcpy( _msgs[2],   "До свидания!\n" );
/* Показать сообщения */
for( indx=0;
indx<sizeof( _msgs )/sizeof( _msgs[0] );
indx++ ) {
printf( _msgs[indx] ); }
return 0; }
```

Перечисляемые типы

Перечисляемый тип представляет собой набор целых чисел, определенный с помощью ключевого слова enum. Каждому числу приписывается имя. Определение перечисляемого типа имеет следующую форму:

```
enum имя_типа { имя_константы [= целое_значение], ... };
```

Указывать значение необязательно; по умолчанию первой константе в списке приписывается значение 0. Последующие константы по умолчанию принимают значение, на единицу большее предыдущей.

Например,

```
enum Computer_Drives{ floppy=1, harddrives=2, cd_rom=4, tape_backup=8 };
```

или

```
enum Seasons {Fall, Winter, Spring, Summer};
```

Переменные перечисляемого типа могут определяться с помощью ключа enum и имени типа, за которыми следуют имена переменных:

```
enum Computer_Drives drvA, drvB, drvC;
```

. Можно также применить определение typedef, чтобы присвоить перечисляемому типу более короткое имя:

```
/*************************************************************/
/* ENUM.C: Пример создания и использования типов enum... */
/.***********.*******.***********************************/
enum Chess_Pieces {
King, Queen, Rook, Bishop, Knight, Pawn };
typedef enum Chess_Pieces Pieces; #define NUM_ROW_COL    8
typedef short Board[NUM_ROW_COL][NUM_ROW_COL]; int main( void 0 {
Board brd; brd[0][0] = Rook;
/*
и т.д.
*/
return  0;
}
```

Совет: Используйте перечисляемые типы вместо директив #define

Если вам нужно определить ряд связанных по смыслу констант, предпочтительнее использовать перечисляемый тип, а не макроопределения #define.

Структуры

В отличие от массивов или перечисляемых типов, *структуры* позволяют определять новые типы путем логического группирования переменных различных типов. Следующая структура, например, может применяться для представления информации о заказчиках:

```
struct customer_information
{
char lastName[25];
char firstName[25];
char midInitial;
char address[80];
char city[25];
char state[2];
char zipCode[10];
long accntNum; };
```

Как и другим типам, структурам можно давать другое имя с помощью ключа typedef. Для доступа к элементам структуры используется имя представителя (переменной) структурного типа, за которым следует точка и имя элемента. Если структура адресуется указателем, то точка заменяется на ->. Следующий пример использует упрощенный вариант структуры customer_information:

```
information:
/*************************************************/
/* STRUCT.C: Использование структур... */
#include <stdio.h>
#include <string.h> typedef struct
{
char Name[80];
long accntNum;
} custInfo;
int main( void )
{
custlnfo newCust;
strcpy( newCust.Name, "Жан-Клод Пайетт" );
newCust. accntNum = 100;
printf( "Информация о новом заказчике.\n" );
printf( "Имя : %s \n", newCust.Name );
printf( "Счет: %d \n", newCust.accntNum );
return 0;
}
```

Объединения

Объединения напоминают структуры, но все элементы объединения занимают одно и то же пространство в памяти. Компилятор выделяет под объединение память, достаточную для размещения наибольшего элемента. Например, под представитель объединения

```
union payment
{
char poNumber[25];
char creditCrd[25];
long chkNumber;
};
```

компилятор выделит пространство как для массива из 25 символов. Обычно | объединение включают в состав структуры, так как это позволяет задать специальный элемент структуры, помогающий определить, какая секция объединения используется в каждом конкретном случае. Вот образец применения такого объединения:

```
/*******************************************************/
/* UNION. C: Использование объединений... */
/*******************************************************/
#include <stdio.h> :
#include <string.h>
typedef enum payment_type { PO, CC, CHK } pmntType;
struct transaction
{
pmntType pType; /* Вид платежа... */
union /* Информация о платеже */
{
char poNumber[25];
char creditCrd[25];
long chkNumber;
} Info;
};
typedef struct transaction SalesRec;
void ShowSalesInfo( SalesRec s )
{
switch ( s.pType )

{
case P0:
printf( "Почтовый перевод: Xs\n", s.Info.poNumber );
break;
case CC:
printf( "Кредитная карта : Xs\n", s.Info.creditCrd );
break;
case CHK;
printf( "Оплачено чеком : Xld\n", s.Info.chkNumber );
break; }
}
int main( void )
{
SalesRec s1, s2;
sl.pType = P0;
strcpy( s1.Info.poNumber, "#PFT-34982-56" );
s2.pType = CHK; s2.Info.chkNumber = 34763L;
ShowSalesInfo( s1 );
ShowSalesInfo( s2 );
return 0;
}
```

Битовые поля

Битовые поля являются элементами структуры, для которых указывается ширина в битах. Эта особенность языка С позволяет вам плотно упаковать элементы структуры. Предположим, вам требуется описать в виде структуры текущее время. Можно использовать для этой цели два типа данных:

```
struct simpleTime
{
unsigned tm_min; /* Минуты 0-59   */
unsigned tm_hr ; /* Часы     */
unsigned isAM ; /* Флаг AM (PM) */
};
struct simpleTimeBF
{


unsigned tm_min :  6; /* Минуты 0-59   */ unsigned :  2;  /* Не используется
*/ unsigned tm_hr    :  4;  /* Часы     */ unsigned isAM     :  1; /* Флаг AM (PM)
*/ unsigned              :  3;  /* Не используется    */ };
```

Вторая структура, определяющая битовые поля, требует в три раза меньшей памяти, так как занимает всего 16 бит.

Заключение

Вы познакомились с употреблением 32-х ключевых слов языка С стандарта ANSI: auto, break, case, char, const, continue, default, do, double, else enum, extern, float, for, goto, if, int, long, register, return, short, signed, sizeof, static, struct, switch, typedef, union, unsigned, void, volatile и while.

Как вы узнаете из последующих глав, Borland C++ поддерживает некоторые расширения стандарта ANSI. Если вы пишете код, который собираетесь переносить на машины, отличающиеся от PC, то, вероятно, захотите избежать использования расширений или ограничить его небольшими участками программы.

Подсказка: Конфигурация компилятора под стандарт ANSI

Вы можете добиться того, чтобы компилятор воспринимал только ключевые слова, соответствующие ANSI, указав опцию -A, если вы используете компилятор с командной строкой. Или, если вы работаете с интегрированной средой, укажите Project в меню Options; затем выберите в рубрике Compiler подрубрику Source. Активируйте опцию ANSI в разделе Language Compliance (Совместимость языка).

Директивы препроцессора

Директивы препроцессора начинаются с символа # и обрабатываются во время первой фазы компиляции. В отличие от ранних компиляторов Borland C++ не генерирует промежуточного файла после обработки препроцессорных директив. Однако можно использовать отдельный препроцессор, CPP.EXE, чтобы получить такой файл. В этой главе рассматриваются директивы препроцессора и макросы, предопределяемые компилятором. Правильное применение директив может сделать ваш код более универсальным, переносимым и разборчивым. Раздел в конце главы посвящен общепринятому применению директив препроцессора.

Макросы: #define

С помощью директивы #defme можно связать имя, или идентификатор, некоторой лексемой или последовательностью лексем. Идентификаторы, служащие для представления констант, называют обычно *объявленными* или *символическими константами.* Идентификаторы, представляющие операторы, называются *макросами.* Однако термин *макрос* часто применяется и к символическим константам. Макросы могут воспринимать параметры. Следующий пример показывает некоторые макросы:

```
/*******************************************************/
/* MACRO.C:   Иллюстрирует директиву #define... */
/*******************************************************/

«#include <stdio.h>
/* Простой макрос... */
#define PIE   3.14159
/* Макрос, принимающий аргументы... */
#define SQR(v)    ( (v) * (v) )
#define AREA( r, a ) ( 0.5 * SQR(r) * (a) )
int main( void )
{
double angle = 15;
double radius = 321;
angle = angle * PIE/180; /* Преобразует градусы в радианы */
printf( "Площадь сектора: %If \n", AREA( radius, angle ) );
return 0;
}
```

Вложенные макросы

Как видно из предыдущего примера, макросы могут быть вложенными. После каждого расширения результат сканируется заново с целью обнаружения идентификаторов, требующих дальнейшей обработки. Исключением является случай, когда расширение содержит собственный идентификатор макроса или оно является препроцессорной директивой.

Символ продолжения строки

Для длинных макросов, занимающих несколько строк, можно использовать обратную дробную черту (\) в качестве символа продолжения строки, как показано ниже:

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

```
#define ERROR( msg )
printf( "Ошибка: " msg "\n" ); \
abort( );


int main( void )
{
ERROR( "Невозможно запустить программу..." );
return 0;
}
```

Аннулирование макроса

Макрос может быть объявлен неопределенным с помощью директивы #undef. После этого ссылка на идентификатор макроса будет вызывать при компиляции ошибку.

Макрос как аргумент компилятора

Borland C++ позволяет определить простой макрос с помощью опции -D в командной строке или меню Options | Project | Compiler | Defines. На рис. 3.1 показано, как в IDE определяется макрос PIE из предыдущего примера.

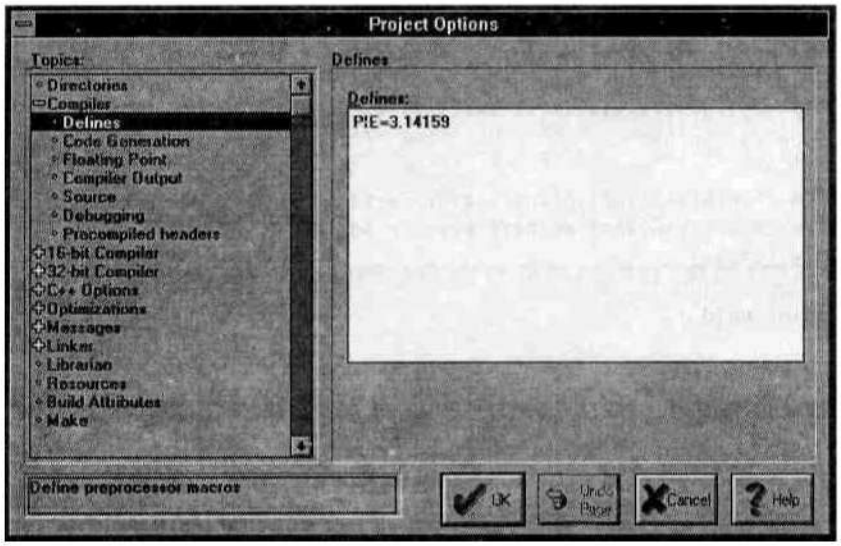


Рис. 3.1. *Определение макроса в интегрированной среде.*

Пользователи, работающие с командной строкой, могут определить этот макрос при вызове компилятора:

```
BCC -DPIE=3.14159 MACRO.C
```

В командной строке макрос можно также аннулировать с помощью опции -U. В интегрированной среде такой возможности нет.

Преобразование в строку *(#)*

Аргументу макроса может предшествовать оператор #, который показывает компилятору, что после расширения аргумент нужно преобразовать в строку. Как и вообще любая строка, результат этой операции может объединяться со смежными строками, если отделяется от них только пробелами.

Применение оператора # в отладочных макросах

Преобразование в строку часто применяется в целях отладки. Ниже-следующий код, например, использует два макроса, которые легко позволяют отслеживать значения переменных.

```
/*********************************************************/
/* STRINGIZ.C: Применение в макросах оператора #...      */
/*********************************************************/
#include <stdio.h>

#define SHOWINTVAR(var) printf( #var "= %d \n", (int)(var) )
#define SHOWDBLVAR(var) printf( #var "= %d \n", \
(double)(var) )

int main( void )
{
   int    counter = 10;
   double current = 300.1459;

   SHOWINTVAR( counter );
   SHOWDBLVAR( current );

   return 0;
}
```

Склейка лексем *(##)*

Оператор склейки *## (token paste)* объединяет в одну две лексемы, между которыми он находится. Новая лексема затем проверяется на предмет возможного расширения. В следующем примере с помощью операторов # и ## определяются макросы, создающие имена переменных.

```
/*********************************************************/ /* TOKNPSTE.C: Применение оператора
# #... */
/*********************************************************/
#include <stdio.h>
#define DEF_VARi( n ) int __var_ ## n
#define USE_VARi( n ) __var_ ## n
#define SHW_VARi( n ) printf( "__var_" #n " = %d\n", \
__var_ ## n )
int main( void )
{
DEF_VARi( 1a );     /* Расширяется в 'int __var_1a;' */
DEF_VARi( 1b );     /* Расширяется в 'int __var_1b;' */
USE_VARi( 1a ) = 10; /* Расширяется в '__var_1a = 10;' */
USE_VARi( 1b ) = 10; /* Расширяется в '__var_1b = 10;' */
SHW_VARi( 1a ); /* 'printf( "__var_1a = %d\n", __var_1a );
*/ SHW_VARi( 1b ); /* 'printf( "__var_1b = %d\n", __var_1b
) ; */
return  0;
}
```

Предостережения

Макросы часто упрощают программу, заменяя последовательность операторов единственным идентификатором. Они также делают код более читаемым, так как

имя макроса часто проясняет смысл оператора или представляемого значения. Однако использованию макросов присущи следующие недостатки:

• Макросы не обеспечивают безопасного обращения с типами. Другими словами, в компиляторе нет никакого встроенного механизма проверки того, что вы передаете аргумент правильного типа.
• Макросы могут производить побочные эффекты, если обращение к аргументу происходит более одного раза. Например, следующие макросы, определяемые в файле STDLIB.H, нужно использовать с осторожностью:

```
#define max(a.b)   (((a) > (b)) ? (a) : (b))
#define min(a,b)   (((a) < (b)) ? (a) : (b))
```

Следующий пример показывает побочный эффект макросов min и max

```
/*****************************************************************************/
/* MIN_MAX.C: Иллюстрирует побочные эффекты макросов */ т
/*****************************************************************************/
#include <stdlib.h>
#include <stdio.h>
int main( void )
{

int i=10;
int j=14;
int mx, mn;
printf( "Значения перед макросом: i= %d, j = %d \n", i, j );
mx = max( i++, j-- ) ;
mn = min( i++, j-- ) ;
printf( "Значения после макроса: i= %d, j= %d \n", i, j );
return 0;
}
```

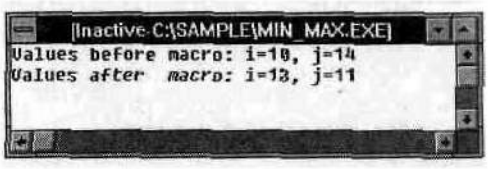


Рис. 3.2. *Результат программы min_max.exe.*

В коде программы значение i увеличивается лишь дважды. Однако конечное значение оказывается на три единицы больше первоначального! Аналогично, конечное значение j на 3 меньше исходного, хотя оно уменьшалось только два раза.

Директива #include

Вы видели, что во многих показанных нами примерах использовалась директива *#include.* Эта директива предписывает компилятору поместить на ее место содержимое другого файла. Обычно директива применяется к включаемым, или заголовочным, файлам *(header or include files);* однако вы можете включить в текст программы любой другой исходный файл. Включаемые файлы обычно содержат определения типов, макросов, описания

внешних переменных и функций C++ типа inline, совместно используемые несколькими модулями. Синтаксис директивы следующий:

#include <*имя_файла*>

#include *"имя файла"*

Если имя файла не является полным именем, в первом случае поиск файла происходит только в пределах специфицированных каталогов включаемых файлов *(include directoriës)*. При использовании второй формы сначала просматривается текущий каталог, а затем каталоги включаемых файлов.

Условная компиляция

Можно избирательно компилировать части файла в зависимости от значения некоторого константного выражения или идентификатора. Для

этой цели служат директивы #if, #elif, #else и #endif. Синтаксис условной конструкции имеет такой вид:

```
#if выражение_1
// Компилируется, если выражение_1 истинно
#elif выражение_2
// Компилируется, если выражение_2 истинно
// и выражение._1 ложно
#elif выражение.3
// Компилируется, когда выражение_1 и выражение_2
// ложны и выражение_3 истинно
#else
// Компилируется, когда все выражения ложны
#endif
```

Для условных директив *#if/#elif/#else* и *#endif* выполняются следующие правила:

• Для каждого *#if* должна присутствовать соответствующая директива *#endif*.

• Директивы *#elif и #else* являются опциональными.

• Число директив *#elif* между *#if* и *#endif не* ограничено, в то время как директива #else должна быть единственной и находиться перед *#endif*.

• Аналогично операторам C if/else, компилируется та секция, которая соответствует первому истинному выражению.

• Если ни одно из выражений не истинно, компилируется секция, следующая за *#else*.

• Значение выражения должно быть целой константой; в выражении нельзя использовать операцию sizeof.

Использование операций в директивах условной компиляции

В выражении могут применяться операции ==, >, >=, < и <=. Например, в программах для Windows часто оценивают символ WINVER, который определен как 0x0300 или 0x030a для версий v3.0 и v3.1 соответственно, как показано ниже:

```
/*********************************************************/
/* WIN3031.C: Проверка значения WINVER в препроцессоре   */
/*********************************************************/

#include <windows.h>

#if (WINVER < 0x030a)   /* Совместимо с WINDOWS.H в v3.0 */
   typedef WORD WINMSG;
#else                   /* v3.1 и выше     */
   typedef UINT WINMSG;
#endif
```

Оператор defined

В директивах *#if* и *#elif* может применяться оператор *defined*. Он позволяет вам проверить, был ли определен идентификатор или макрос. Следующий код иллюстрирует это:

```
void ShowMessage( char *msg )
{
#if defined(DOS_TARGET) puts( msg );
#else
MessageBox( NULL, msg, "MSG", MB_OK|MB_TASKMODAL );
#endif
```

Если макрос DOS_TARGET определен, функция ShowMessage использует для вывода сообщения функцию puts; в противном случае используется функция MessageBox.

Можно применять логическую операцию отрицания '!' для проверки тощ что идентификатор не определен. Функцию ShowMessage( ) можно переписать таким образом:

```
void ShowMessage( char *msg )
{
#if   !deflned(DOS_TARGET)
MessageBox( NULL, msg, "MSG", MB_OK|MB_TASKMODAL );
#else
puts( msg ); #endif
}
```

Директивы #ifdef и #ifndef

Директивы *#ifdef u #ifndef* эквивалентны *#if defined u #if !defined*. Они позволяют проверить, определен идентификатор в данный момент или не определен. Применение оператора *defined* предпочтительнее, так как он позволяет

проверять сразу несколько макросов в сложных логических выражениях. Например, такой код:

```
#ifdef     DOSTARGET
#ifndef NDEBUG
puts( msg );
#endif
#endif
```

можно упростить до

```
#if defined(DOSTARGET) && ! defined(NDEBUG)
puts( dbmsg );
#endif
```

Директива #error

Директива *#error* вызывает во время компиляции сообщение об ошибке. Она имеет следующий вид:

```
#error сообщение_об_ошибке
```

Сообщение может включать в себя идентификаторы макросов, которые будут расширены препроцессором. Директива обычно применяется, когда не был определен необходимый идентификатор, как показано ниже:

```
/* ERROR.C: Применение #error... */
/* Проверить целевую среду... */
«if !defined(DOS) && ! defined(WINDOWS)
#error Вы должны определить либо DOS, либо WINDOWS
#endif
#if defined(DOS) && defined(WINDOWS)
#error Нельзя одновременно определять DOS и WINDOWS
#endif
```

Директива #line

С помощью директивы *#line* можно изменить внутренний счетчик строк компилятора. Она имеет следующий вид:

```
#line  номер_строки ["имя_файла"]
```

Номер строки должен быть целой константой. Может присутствовать опциональное имя файла. Эта директива изменяет предопределенный макрос _LINE_. Если присутствует имя файла, модифицируется также макрос _FILE_ .

Директива ttpragma

Директива #pragma позволяет управлять специфическими возможностями компилятора. Синтаксис директивы следующий:

```
#pragma директива_pragma
```

В таблице 3.1 описаны директивы pragma, поддерживаемые компилятором Borland C++.

Таблица 3.1. Директивы pragma компилятора Borland C++ *Директива*
Описание

argsused: Подавляет предупреждающее сообщение Parameter *xxxx* never used in function *ffff* (Параметр xxxx не использован в функции *ffff*) для функции, следующей за директивой.

exit: Позволяет указать функцию, которая должна быть вызвана перед завершением программы. Синтаксис: *#pragma exit ИмяФункции [приоритет]*

extref: Заставляет компилятор включить ссылку на неиспользованную внешнюю переменную или функцию. См. пример, помещенный после таблицы.

hdrfile: Специфицирует имя заранее откомпилированного файла-заголовка.
hdrignore: Так как макросы и типы, определяемые в заголовочном файле, могут изменяться, когда определяется другой макрос, компилятор не использует информацию из прекомпилированного заголовка, если встречает директиву условной компиляции. Директива указывает, что заголовок должен использоваться, если специфицированный идентификатор встречается в директиве условной компиляции. Синтаксис: *#pragma hdrignore идентификатор Идентификатор* означает символ, который оценивается в условной директиве препроцессора.
hdrstop: Предписывает компилятору не включать дальнейшую информацию в прекомпилируемый заголовочный файл.

Таблица 3.1. Продолжение *Директива Описание*
inline: Указывает, что компиляция текущего модуля должна производиться через ассемблер. Другими словами, из вашего исходного модуля на C/C++ компилятор генерирует код ассемблера и затем вызывает TASM (или другой ассемблер, если указано), который создает объектный файл.
intrinsic: Эта директива может быть использована, чтобы разрешить или запретить генерацию inline-кода для встроенной функции. Встроенная функция — это библиотечная процедура, для которой компилятор генерирует inline-код вместо вызова библиотеки). Синтаксис:
#pragma intrinsic [-]ИмяФункции
Если имени функции предшествует -, вызовы функции не расширяются как inline. Компилятору Borland C++ известны следующие функции:
memchr memcmp memcpy memset
strcat strchr
strcmp strncpy strnset
strrchr strset rotl
rotr fabs alloca
obsolete: Директива obsolete имеет следующий вид:
#pragma obsolete ИмяФункции
Она приводит к тому, что компилятор генерирует сообщение Warning *имя_файла номер_строки: ИмяФункции'* is obsolete in function xxxx (устаревшая функция). Директиву можно использовать для того, чтобы сообщить другим программистам, что вы улучшили ваш код и предусмотрели для данной задачи новую функцию.
option: Эта директива позволяет вам включить в ваш код опции командной строки компилятора. Некоторые опции не могут быть использованы в этой директиве; другие должны помещаться в самом начале исходного текста. Используйте директиву option, когда вам совершенно необходимо, чтобы для данного модуля использовалась именно такая установка параметров.

Таблица 3.1. Продолжение
Директива Описание
startup: Эта директива является дополнительной к *pragma exit.* Она позволяет указать функцию, которая должна исполняться перед *main, WinMain, LibMain* или *DLLEntryPoint.*
warn: Директива *warn* позволяет вам выборочно разрешать или подавлять предупреждающие сообщения. Синтаксис: *#pragma warn +\-\.xxx,*

где *'xxx'* представляет определенное сообщение. Если символу предшествует +, то выдача сообщения разрешается, если -, то запрещается. Предшествующая точка восстанавливает для данного сообщения исходное состояние.

Использование #pragma option с исходным кодом исполнительной библиотеки

Иногда бывает полезно включить в ваш проект один или несколько файлов из исполнительной библиотеки Borland C++, чтобы отладить неработающую программу. Так как исполнительная библиотека построена с определенной установкой параметров, то, чтобы быть уверенным в совместимости ваших установок с теми, которые требуются для компиляции кода RTL, вы можете вставить в начало модуля RTL директиву

```
Spragwa option -a- -k- -zC_TEXT -zFLDATAV-zTOATA
```

```
/*********************************************************/
/* EXTREF.C: Иллюстрирует #pragma extref... */
/*********************************************************/
#include <stdio.h>
extern int unusedInt1;
extern int unusedInt2;


void funcNotCalled1( void );
void funcNotCalled2( void );
#pragma extref unusedInt2
#pragma extref funcNotCalled2
int main( void )
{
printf( "Hello, world \n" );
return 0;
}
```

Компилятор включает в выходной файл .OBJ запись, показывающую, что используется внешняя функция *print/.* Обычно для функций (или внешних переменных), которые объявлены, но ни разу в данном модуле не вызываются, такие записи не генерируются. С помощью *#pragma extref* можно заставить компилятор генерировать записи внешнего определения (обычно называемые EXTDEF) для неиспользуемых внешних функций и переменных. Объектный файл, генерируемый для приведенного примера, будет содержать EXTDEF для переменной 'unusedInt2' и функции 'funcNotCalled2'.

Предопределенные макросы

Компилятор автоматически определяет некоторые макросы.

Макросы ANSI

В таблице 3.2 описаны макросы ANSI, автоматически определяемые компилятором.

Таблица 3.2. Предопределенные макросы ANSI *Макрос Описание*

__DATE__: Строка, представляющая в форме *mmm dd yyyy* дату, когда данный файл обрабатывался препроцессором.

FILE: Строка, представляющая имя текущего файла в двойных кавычках. (Заметьте, что директива *#line* может изменять значение _FILE_.)
LINE: Целое, представляющее текущий номер строки исходного файла. (Директива *#line* может изменять значение _LINE_.)
STDC: Значение является целой константой 1, если установлена совместимость со стандартом ANSI (опция -A в командной строке BCC или BCC32 и опция меню Options | Project | Compiler | Source | Language_Compliance | ANSI в IDE.) В противном случае макрос не определен.
TIME: Строка, представляющая в форме *hh:mm:ss* время, когда данный файл обрабатывался препроцессором.
Следующая программа показывает обычное использование этих макросов.

```
/*******************************************************/
/* ANSIMAC.C: Использование макросов ANSI...           */
/*******************************************************/
#include <stdio.h>
char buildDate[] = __DATE__;
char buildTime[] = __TIME__;
int main( void )
{
FILE *fp;
printf( "SMPL Version 2.01 Copyright (c) 1994"
" A.B. co. Ltd\n" );
printf( "Дата создания %s @ %s\n", buildDate, BuildTime );


if ( (fp = fopen( "\\FILE\\SAMPLE.NOr,   r+t )) == NULL ) {
printf( "Ошибка fopen() в файле %s строка %d\n", __FILE__, __LINE__ );
return 0; } /*
*/
return 0; }
```

Макросы Borland C++

Помимо макросов ANSI, Borland C++ предопределяет некоторые другие идентификаторы. Таблица 3.3 описывает эти макросы.

Таблица 3.3. Макросы Borland C++ *Макрос Описание*

_cplusplus Определен, когда компилятор находится в режиме C++.
_Windows Определен, когда компилируеется код для 16-бит-Windows и для Console или GUI Windows NT.
BCPLUSPLUS Значение равно 0x320 в режиме компиляции C++.
BORLANDC Значение равно 0x400.
CDECL Значение равно 0x1, если используется протокол вызова функций, принятый в C (по умолчанию).
CONSOLE Определен, если компилируется 32-битное приложение типа Console.
DLL Определен, если компилируется 16- или 32-битная DLL.
__FASTCALL_ Определен, если используется протокол вызова Register.
FLAT Определен при построении 32-битного приложения.

Таблица 3.3. Продолжение
Макрос Описание
MSDOS Определен при построении 16-битного приложения.

_MT_ Определен, если в 32-битном компиляторе используется опция multi-thread.
_OVERLAY__ Определен, если разрешена компиляция перекрытий (например, использован ключ -Y при вызове BCC.EXE).
PASCAL Определен, если используется протокол вызова языка Pascal.
TCPLUSPLUS Определен в режиме компиляции C++ (аналогично _BCPLUSPLUS_).
TEMPLATES Показывает, что Borland C++ поддерживает шаблоны.
TLS Определен в 32-битном компиляторе для консольных и GUI-приложений.
TURBOC Значение равно 0x400 (аналогично _BORLANDC_).
WIN32 Определен при использовании 32-битного компилятора для консольных и GUI-приложений.
При компиляции 16-битных приложений Borland C++ также определяет один из следующих макросов, показывающих используемую модель памяти _TINY_, __SMALL_, _MEDIUM__, _COMPACT__, _LARGE_. _HUGE__.

Типичное применение директив препроцессора.

Теперь, когда вы познакомились с различными препроцессорными директивами, давайте рассмотрим некоторые из их типичных применений.

Предотвращение многократных включений файла-заголовка

Вы можете предотвратить многократное включение в текст содержимого файла-заголовка, поставив в файл "часового". Следующий заголовочный файл иллюстрирует это:

```
#if !defined(__SAMPLE_H)
#define    __SAMPLE_H
//
// Здесь идут описания, макросы,
// определения типов и т.д...
//
#endif // __SAMPLE_H
```

Приведенный код гарантирует, что только одна копия содержимого файла будет включена в текст, даже если файл несколько раз косвенно включается в текущий модуль.

Простое исключение секций кода

Очень часто применяют директивы *#if0/#endif* для исключения некоторой секции .кода. На самом деле, с помощью комбинации директив #if0*/#else/#endif* можно легко выбирать одну из двух секций программы. Этот прием очень полезен при экспериментировании. Следующий пример показывает, что имеется в виду:

```
/*****************************************************/
/* IF_0_1.C: Применение #if для исключения кода... */
/*****************************************************/
void SortData()
{
#if 0 // Замените на '1', чтобы использовать более быстрый
```

```
// алгоритм, когда Бруно исправит фатальную ошибку.
//    (Билл)


//
// Здесь находится код быстрого алгоритма
//
#else
//
//   Здесь находится медленный алгоритм
//
#endif
}
```

Обеспечение правильной установки параметров компиляции

Неплохо быть уверенным в том, что ваш код всегда будет компилироваться с правильно установленными опциями. Проверяя различные макросы, предопределяемые компилятором, вы можете обнаруживать некоторые типы неправильных установок. Следующий пример демонстрирует несколько макросов, часто применяемых с этой целью.

```
//////////////////////////////////////////////////////
// MAC_CHK.CPP: Применение макросов для проверки /
// параметров компилятора /
//////////////////////////////////////////////////////
#if ! defined(_cplusplus)
#error   Этот файл должен компилироваться в режиме C++
#endif
#if  !defined(__Windows)
«error Этот файл можно использовать только в приложениях
Windows
«endif
#if !defined(__WIN32__)
#error Этот файл использует VBX и не поддерживает Win32
#endif
#if ! defined(__COMPACT__) && !defined(__LARGE__)
#error Для этого файла требуются длинные указатели данных
#endif
```

Диагностические макросы

Для упрощения макросов, выводящих диагностические сообщения, можно употреблять оператор #. Различные диагностические макросы могут также использовать идентификаторы _FILE_ и _LINE_ для уточнения сообщений об ошибках. Следующий код содержит два примера организации диагностики с помощью препроцессора.

```
#include <stdio.h>
#define INFO(msg) printf( "INFO: " #msg "\n" )
#define WARN(msg)  printf( "WARN:  " #msg " ("_„FILE_ ")" \
" в строке %d \n", _LINE_  );
int main( void )
{
INFO( Входим в main );
//
```

```
// Что-то не работает...
WARN( Вызов функции xxxx не проходит );
return 0;
}
```

Заключение

С появлением C++ программисты все меньше и меньше полагаются на макросы препроцессора (константы C++, функции inline, шаблоны предлагают лучшие альтернативы). Однако директивы препроцессора не теряют своей ценности и, как показывает материал этой главы, дают вам средства для того, чтобы сделать ваш код более универсальным, ясным и переносимым.

Глава 4
Расширения языка C

Если бы мы жили в идеальном мире, язык программирования высокого уровня, такой, как C, совершенно скрывал бы от программиста особенности процессора и операционной системы, полностью используя, тем не менее, все их преимущества. Вы могли бы писать переносимый ANSI-код, который воспринимался бы другими компиляторами и даже работал бы на компьютерах с другой архитектурой. Однако такое идеальное положение дел достижимо только для приложений, требующих минимального взаимодействия с пользователем или доступа к системным ресурсам. Для того, чтобы можно было в полной мере использовать особенности среды программирования или обходить какие-либо ограничения, ей накладываемые, компиляторы предлагают программисту расширения языка. Эта главы познакомит вас с расширениями языка Borland C++, связанными с особенностями процессоров INTEL 80x86. В основном материал главы освещает вопросы, относящиеся к приложениям для 16-битной среды DOS и WINDOWS. Расширения, имеющиеся в Borland C++, представлены дополнительными ключевыми словами и опциями компилятора.

Формы ключевых слов: keyword, _keyword и __keyword

Компилятор, *как правило*, распознает эти ключевые слова, если им предшествует один или два символа подчеркивания или он отсутствует. В этой главе применяется форма с двумя подчеркиваниями, так как существует соглашение, что *идентификаторы с двойным подчеркива-*нием резервируются для использования компилятором и употребление их программистами нежелательно.

```
/* Следующий код определяет три длинных указателя */
/* с помощью модификаторов far, _far и __far...   */

int   far *loopPtr;
int  _far *countPtr;
int __far *indexPtr; /* <- Формат, применяемый в данной главе */
```

Сегменты

У большинства из вас машины имеют несколько мегабайт памяти и вы привыкли думать о ней как простирающейся от 0 до xMB. Другими словами, вы ее представляете себе организованной линейно. С точки зрения программы память представляется в форме сегментов. *Сегмент* является куском памяти, который начинается с адреса, делящегося на 16. Каждый сегмент имеет длину 64K. Чтобы обратиться к ячейке памяти, нужно указать значение сегмента и смещение внутри сегмента. На самом деле, сначала сегментное значение загружается в сегментный регистр, а затем процессор выполняет некоторые инструкции, оперируя смещением внутри сегмента. В CPU имеется четыре базовых *сегментных регистра,* с помощью которых доступ к памяти осуществляется следующими способами:

• CS (сегмент кода). Регистр CS содержит адресный сегмент всех исполняемых инструкций или функций.

• DS (сегмент данных). Регистр DS содержит сегментное значение по умолчанию для доступа к глобальным и статическим переменным.

• SS (сегмент стека). Регистр SS содержит сегментное значение стека программы, в котором выделяется пространство под локальные переменные.
• ES (дополнительный сегмент). ES является вспомогательным сегментным регистром, обычно используемым для доступа к глобальным и статическим переменным.
Когда происходит доступ к памяти, CPU аппаратно комбинирует значение сегментного регистра с указанным смещением, чтобы получить требуемый линейный адрес. Это краткое описание способа адресации CPU относится к среде DOS, но не вполне верно для случая 16-битных Windows. При работе под 16-бит-Windows сегмент не содержит реальный адрес памяти;

вместо этого он является индексом (или *селектором),* указывающим на пуни в таблице *(таблице дескрипторов),* где этот адрес хранится. Тем не менее, в обеих конфигурациях адрес составляется из пары сегмент:смещение. Вы можете представлять себе имя функции как смещение блока инструкций в пределах определенного сегмента кода. Аналогично, имя переменной является просто смещением внутри сегмента, в котором компилятор выделил некоторое пространство.
Для доступа к переменной в регистр DS сначала загружается соответствующее базовое значение, а затем процессору передается ее смещение. Аналогичным образом при вызове функции может потребоваться загрузив в CPU новое значение кодового сегмента перед передачей управления по адресу, представленному смещением функции. Каждая новая спецификация сегмента связана с небольшими издержками, так как необходимо загрузить сегментный регистр новым значением. В идеальном случае все данные вашей программы должны умещаться в единственный сегмент, в котором для нахождения каждой переменной достаточно будет одного смещения. Эта исключит необходимость перезагрузки регистра DS. Похожим образом размещение всего кода программы в одном сегменте, внутри которого каждая функция идентифицируется уникальным смещением, устранило бы необходимость постоянного обновления регистра CS. Однако, так как длина каждого сегмента не может превышать 64K, это ограничило бы размер вашей программы 64K байтами кода или данных! Такое ограничение было 6ы приемлемым несколько лет назад, но сегодня не является чем-то исключительным, если приложение занимает мегабайты памяти. Чтобы имелась возможность выбора между эффективностью односегментной конфигурации и возможностями, предоставляемыми большими объемами кода и данных Borland C++ предусматривает различные *модели памяти.*

Модели памяти

Модели памяти являются параметрами компиляции, которые управляю выделением ресурсов для вашей программы. Приложение DOS должно компилироваться с использованием одной из шести доступных моделей (Tiny, Small, Medium, Compact, Large и Huge). Приложения Windows могут использовать только одну из четырех возможных моделей памяти: Small Medium, Compact и Large. Кроме спецификации требований, предъявляемым вашей программой к объему памяти, выделяемой под код и данные, выбор модели влияет на размер указателей, размер динамической памяти (ДП) и в случае DOS, на размер стека программы. Таблица 4.1 дает представление об атрибутах шести моделей памяти для приложений DOS.

Таблица 4.1. Модели памяти DOS

Модель	*Код*	*Данные*	*Стек*	*Тип указателей данных и ДП по умолчанию*
Tiny	код +	данные +	стек+ДП до 64K	(near)(короткий)
Small	64K	данные +	стек+ДП до 64K	(near)
Medium	64K/файл	данные +	стек+ДП до 64K	(near)
Compact	64K	64K	64K	far (длинный)
Large	64K/файл	64K	64K	far
Huge	64K/файл	64K/файл	64K	far

В таблицах 4.2 и 4.3 приведены атрибуты четырех моделей памяти, доступных для исполняемых модулей и динамических библиотек Windows.

Таблица 4.2. Модели памяти Windows (EXE)

Модель	*Код*	*Данные*	*Стек*	*Тип указателей данных и ДП по умолчанию*
Small	64K	данные +	стек+ДП до 64K	(near)
Medium	64K/файл	данные +	стек+ДП до 64K	(near)
Compact	64K	данные +	стек до 64K	far
Large	64K/файл	данные +	стек до 64K	far

Таблица 4.3. Модели памяти Windows (DLL)

Модель	*Код*	*Данные*	*Стек*	*Тип указателей данных и ДП по умолчанию*
Small	64K	~ 64K	—	far
Medium	64K/файл	~ 64K	—	far
Compact	64K	~ 64K	—	far
Large	64K/файл	~ 64K	—	far

Короткие и длинные указатели

С моделями памяти тесно связан размер указателей, используемых для адресации кода и данных. *Короткий (ближний) указатель* содержит только адресное смещение, но предполагается, что сегментная часть адреса находится в одном из сегментных регистров CPU. Для моделей с ближней динамической памятью (Tiny, Small и Medium) по умолчанию используются короткие указатели данных, а сегмент обычно представлен значением в регистре DS. Модели, поддерживающие доступ не более чем к 64K кода (Tiny, Small и Compact), по умолчанию используют короткие указатели кода или функций. Сегментная часть адреса находится в регистре CS.

Длинные (дальние) указатели содержат как сегментную часть адреса, так и смещение. В моделях памяти Compact, Large и Huge применяются длинные

указатели данных, а модели Medium, Large и Huge по умолчанию используют длинные указатели функций. Однако, вне зависимости от модели памяти вы можете, если хотите, отменить размер по умолчанию, специфицировав для указателя ключевое слово **_near** или **__far.** Синтаксис:

```
тип _near *имя_переменной;
или
тип _far   *имя_переменной;
```

Обратите внимание на то, что ключевое слово _near или _far предшествует звездочке. Если поместить модификатор между * и именем переменной, он может изменить местонахождение указателя, но не повлияет на его размер. Следующий пример иллюстрирует размер и содержимое ближних и дальних указателей. Вы должны откомпилировать и построить его по крайней мере для моделей Small и Large, если хотите заметить разницу в размерах указателей.

```
/*********************************************************/
/* PTRSIZE.C:  Пример,  показывающий размер указателей   */
/*********************************************************/
#include <sdtio.h>
char       Var[ ] = "Привет!"; 1


int main( void )
{
char *p = Var;          /* Стандартный указатель */
char far*fp = Var;      /* Дальний указатель... */
char near *np = Var;    /* Ближний указатель... */
printf( "Глобальная переменная расположена в %p\n", Var );
printf( "Размер стандартного указателя    %d байт\n", sizeof( p ) );
printf( "Стандартный указатель ссылается на %p\n", p );
printf( "Размер дальнего указателя       %d байт\n", sizeof( fp ) );
printf( "Дальний указатель ссылается на    %Fp\n", fp );
printf( "Размер ближнего указателя %d байт\n",sizeof( np ) );
printf( "Ближний указатель ссылается на    %\n", np );
return 0;
}
```

Помните, что компилятор всегда может преобразовать ближний указатель в дальний. Дальний указатель, однако, не может быть преобразован в ближний. Например, если вы компилируете приведенный пример в модели с дальними данными, такой, как LARGE, то заметите, что последний вызов *printf* выводит недействительный адрес.

Указатели типа Huge

Когда над указателями производятся арифметические операции, у длинных указателей модифицируется только смещение. Это значит, что при увеличении указателя смещение вернется к нулевому значению, если перейдет границу сегмента. Чтобы компилятор генерировал код, позволяющий указателям выходить за текущий сегмент, можно применить при определении указателя модификатор **__huge.** Синтаксис такой же, как для коротких или длинных указателей:

```
тип _huge *имя_указателя;
```

Методы, используемые компилятором для работы с этими указателями, отличаются в конфигурациях DOS и Windows; однако результат один и

тот же — не происходит "заворачивания" при переходе границы сегмента.

Указатели huge в DOS

Для программ, ориентированных на среду DOS, компилятор будет постоянно *нормализовать* указатель, когда вы увеличиваете или уменьшаете его значение. Другими словами, компилятор гарантирует, что смещение указателя не превышает 15 (0x0f) байт, соответствующим образом изменяя сегмент. Следующий пример для DOS демонстрирует нормализацию указателей.

```
/*******************************************************/
/* HPTRDOS.C: Показывает нормализацию указателей huge  */
/*******************************************************/
#include <stdio.h>
int main( void )
{ long _huge *hp = NULL;
int count;
for( count=0; count<10; count++ )
{
printf( "hp[%d] = адресу %Fp\n", count, &hp[count] );
}
for( count=0; count<10; count++ )
{
printf( "hp содержит адрес %Fp\n", hp );
hp++;
}
return 0;
}
```

Указатели huge в Windows

В среде Windows нормализация производится только при переходе смещения через границу сегмента. Для корректной нормализации указателей должны быть выполнены следующие условия:

• Активирована опция быстрых указателей huge (ключ -h для компилятора с командной строкой).

• Арифметические операции над указателями должны производиться только с применением степеней 2. Если ваш указатель ссылается на тип, определенный пользователем, может потребоваться дополнение этого типа пустыми байтами. Например, если переменная указывает на тип

```
struct CustInfo {
char   lastName[13];
char   firstName[13];
long   accountNum; };
struct CustInfo _huge *pBigCustList;
```

вам нужно будет дополнить структуру до 32 байт, как показано ниже, чтобы обеспечить правильную нормализацию указателя при выполнении над ним арифметических операций:

```
struct CustInfo {
char   lastName[13];
char   firstName[13];
long   accountNum;
short _padding_to_make_struct_32_bytes; };
```

```
struct Custlnfo _huge *pBigCustList;
```

Макросы для обращения с указателями

Заголовочный файл DOS.H определяет три макроса, облегчающих обращение с указателями.

- FP_OFF(fp) Макрос возвращает смещение указателя.
- FP_SEG(fp) Макрос возвращает сегмент указателя.
- MK_FP(s, o)Макрос возвращает длинный указатель, составленный из сегмента и смещения, переданных в качестве параметров.

Не обязательно ограничивать употребление термина "указатель" переменными-указателями. Например, FP_OFF и FP_SEG могут применяться к адресу переменной.

```
/*******************************************************/
/* PTRMAC.C: Применение макросов FP_SEG и FP_OFF... */
/*******************************************************/
#include <stdio.h> #include <dos.h>
int main( void )
{
int i;
printf( "Адрес локальной переменной: %p \n", &i ); printf(
"Адрес локального значения : %04X: %04X \n",
FP_SEG( &i ),
FP_OFF( &i ) ); return 0; }
```

Который из сегментов?

Указатель _near содержит только смещение указателя. Сегментное значение, используемое компилятором при разыменовании указателя, зависит от типа указателя: короткие указатели данных используют содержимое регистра DS, а короткие указатели на функцию подразумевают использование регистра CS. Можно, однако, изменить регистр, назначаемый по умолчанию для короткого указателя, с помощью следующих ключевых слов:

- _cs: использовать для короткого указателя регистр CS.
- _ds: использовать для короткого указателя регистр DS.
- _es: использовать для короткого указателя регистр ES.
- _ss: использовать для короткого указателя регистр SS.

В следующем примере DLL получает доступ к первому слову сегмента стека с помощью модификатора *_ss для* короткого указателя данных. Этот технический прием часто применяется, когда DLL требуется доступ к переменным прикладной программы, так как сегмент стека DLL на самом деле является сегментом данных приложения, вызывающего процедуру динамической библиотеки.

```
#include <windows.h>
short _ss *appTaskHdrPtr;
void CALLBACK _export" DLLFunction( void )
{
short s1 = *appTaskHdrPtr;
}
```

Вы можете определить переменную-указатель, которая содержит только сегментную часть адреса, применив модификатор *_seg.* Полный указатель можно получить позднее, сложив переменную-сегмент с коротким указателем, представляющим адресное смещение. В примере, показанном ниже, сегмент

массива сообщений сохраняется в переменной-сегменте. Смещение внутри массива потом вычисляется, исходя из индекса сообщения. Смещение и сегмент комбинируются и передаются функции *printf*. Поскольку в *printf* передается длинный указатель, пример должен компилироваться в модели памяти Compact или Large.

```
/* SEG.C: Применение сегментных указателей... */
#include <stdio.h>
#include <dos.h>


char *msg[] = { "Это первое сообщение",
"Следующее...",
"И, наконец, последнее"
}; char _seg *msgSeg;
void ShowMsg( int indx ) {
char _near *msgOff = ( char _near* )msg[indx];
printf( msgSeg+msgOff ); }
int maln( void ) {
/* Инициализация сегмента... */
msgSeg = ( char _seg* )msg;
ShowMsgC 1 );
return 0; }
```

Модификаторы переменных

Из таблиц 4.1, 4.2 и 4.3 видно, что все модели памяти, кроме Huge, имеют только один сегмент для всех глобальных данных прикладной программы. Это может стать серьезным ограничением, если вашей программе требуется большое количество глобальных и статических переменных. С помощью ключевых слов *_far* или *_huge* вы можете определить переменные, которые не будут размещены в сегменте данных, подразумеваемом по умолчанию. Этот прием обычно применяется в приложениях, требующих более 64K памяти для глобальных или статических переменных.

_far

Модификатор *_far* предписывает компилятору разместить переменную в дальнем сегменте данных. Обратите внимание, что модификатор должен

помещаться перед именем переменной. Это особенно важно при применении его к указателям. Рассмотрите следующий пример.

```
/*******************************************************/
/* FARVAR.C:  Показывает применение модификатора _far */
/*******************************************************/
/* Переменная,  размещаемая в сегменте по умолчанию       */ int     i = 0;
/* Переменная,  размещаемая в дальнем сегменте данных    */ int far  j = 0;
/* Указатель,  размещаемый в сегменте по умолчанию */ int    *ip1 = &i;
/* Дальний указатель в сегменте данных по умолчанию      */ int far   *ip2 = &i;
/* Стандартный указатель в дальнем сегменте данных       */ int * far   ip3 = &i;
/* Дальний указатель в дальнем сегменте данных */ int far * far ip4 = &i;
```

Если при компиляции или сборке программы выдается сообщение, показывающее, что используется слишком много глобальных данных, опишите несколько больших объектов с добавлением модификатора *_far*. *н*апример,

следующий файл генерирует сообщение об ошибке Too much global data defined in file (Файл определяет слишком много глобальных данных).

```
/* Пример файла со слишком большим объемов данных */
int intTable1[32000];
int intTable2[32000];
char msg[ ] = "Производятся вычисления, пожалуйста подождите!";
long counter"1, counter2;
```

Чтобы устранить возникающую ошибку, добавьте модификатор _far к определению больших массивов.

```
int _far intTable1[32000];
int _far intTable2[32000];
char msg[ ] = "Производятся вычисления, пожалуйста подождите!";
long counter1, counter2;
```

_huge

Ограничения, связанные с размером сегментов, становятся очевидными, когда вам нужно определить переменную, размер которой превышает 64K. Например, следующее определение

```
long Table[20000];
```

вызывает сообщение Array size too large (Размер массива слишком велик), Можно применить модификатор *_huge,* чтобы предписать компилятору выделить под одну переменную два (или более) сегмента. Другими словами, переменная __huge похожа на переменную _far, за исключением того, что размер первой может превосходить 64K. Как и в случае с ключом *_far,* модификатор должен находиться перед именем переменной, как показано ниже:

```
long „huge Table[20000];
```

Модификаторы функций

Как и указатели данных, функции и указатели на функцию по умолчанию являются ближними или дальними в зависимости от применяемой модели памяти. Как и прежде, можно изменить подразумеваемый размер указатель на функцию с помощью ключевых слов *_near* или *_far.* Например:

```
/*********************************************************/
/* FARFUNC.C:  _far и _near функции и указатели...      */
/*********************************************************/


№include <stdio.h>
void _far farFunction( int i ) {
printf( "farFunction приняла аргумент %d\n", i ); }
void _near nearFunction( char *msg ) {
printf( "nearFunction приняла аргумент %s\n", msg ); }
/* Объявить и инициализировать короткий указатель на функцию
*/ void (_near *nFuncPtr )( char* ) = nearFunction;
/* Объявить и инициализировать длинный указатель на функцию
*/ void („far *fFuncPtr )( int  ) = farFunction;
int main( void )
{
/* Вызвать функции через их указатели... */
fFuncPtr( 10 );
nFuncPtr( "Приветственное сообщение!" );
return 0;
}
```

Функции, вызываемые операционной системой или драйвером устройства, должны явно определяться как *_far,* если только не применяется модель памяти с длинными указателями для кода. Например, если вы пишете программу для среды DOS, то можете использовать программный интерфейс мыши и специфицировать некоторую функцию в качестве процедуры обработки событий, генерируемых драйвером. Функция может вызываться, когда нажимается кнопка мыши или мышь перемещается. Если вы находитесь в модели памяти с ближним кодом (Tiny, Small, Compact), функция, обрабатывающая события, должна быть объявлена как*_far,* так как она вызывается драйвером мыши. Программы Windows широко применяют возвратно вызываемые (callback) функции. Эти функции вызываются непосредственно системой Windows и также должны объявляться с ключевым словом *_far* или *FAR.*

_interrupt

Некоторые события заставляют CPU временно приостановить исполнение текущей операции и передать управление другому участку кода. Эти события называются *прерываниями* и могут быть инициированы как оборудованием, так и программно. Например, если вы что-то делаете с клавиатурой в то время, когда программа производит длинные вычисления, каждое нажатие клавиши приводит к временному переходу управления к *обработчику прерывания, или процедуре обслуживания прерывания (interrupt service routine, ISR),* которая сохраняет код нажатой клавиши в буфере. Если работающая программа не проверяет, нажимаются ли клавиши, то в конце концов обработчик прерывания начинает пищать, информируя вас, что буфер переполнен. Процедуры обработки прерываний обычно пишутся на языке ассемблера, так как он предоставляет лучшие возможности управления процессором и свободен от накладных расходов, свойственных языкам высокого уровня. Однако, вы можете написать функцию на С, которая будет обрабатывать прерывания, применив" модификатор *_interrupt.* Этот модификатор приводит к тому, что компилятор генерирует код, который

• сохраняет и восстанавливает регистры соответственно при входе в функцию и выходе из нее.

• устанавливает сегментный регистр данных на адрес сегмента данных по умолчанию, предоставляя процедуре обслуживания доступ к глобальным и статическим переменным.

• завершает выполнение функции инструкцией *IRET.*

В библиотеке Borland C++ имеется функция *setvect(),* которая позволяет установить и активировать процедуру обслуживания прерывания во время исполнения программы. Если только вы не пишете резидентное приложение *(Terminate and Stay Resident, TSR),* неплохо сохранить адрес предыдущей процедуры перед тем, как вы установите свою. Можно использовать библиотечную функцию *getvect(),* которая возвращает адрес текущего обработчика прерывания. Перед завершением вашей программы восстановите старый обработчик. Следующий пример демонстрирует основные действия, связанные с установкой процедуры обслуживания прерывания.

```
#include <dos.h>
#define INT_NUMBER0x60


void interrupt ( *oldHandler )( void );
```

```
void interrupt newHandler( void ) {
/* Сначала вызывает старый обработчик */
oldHandler();
/*
Выполняет некоторые действия перед возвратом...
*/
}
void interrupt AnotherHandler( void ) {
Л
Сначала производит обработку...
*/
/* Цепной вызов старого обработчика */
chain_intr( oldHandler );
}
void InstallHandler( void ) {
/* Сохранить адрес текущего обработчика */
oldHandler = getvect( INT_NUMBER );
/* Установить новый обработчик */
setvect( INT_NUMBER, newHandler ); }
void RestoreHandler( void )
{
setvect( INT_NUMBER, oldHandler );
return 0;
}
```

Обработчик прерывания может вызывать старый обработчик перед выполнением каких-либо действий. В качестве альтернативы для перехода к старому обработчику можно применить функцию *chain_intr();* эта функция не возвращает управления вызывающей процедуре.

_saveregs, _loadds

Генерируя для ваших функций машинный код, компилятор С придерживается определенного набора правил, относящихся к использованию регистров. Полученный код сохраняет и восстанавливает содержимое некоторых регистров, если они используются внутри функции. Значение других регистров не сохраняется никогда. Такое соглашение о регистрах гарантирует, что важные данные будут помещаться только в регистры, которые сохраняются перед вызовом другой процедуры. Это позволяет также свободно пользоваться некоторыми регистрами без затрат на сохранение и восстановление их содержимого. К сожалению, эти правила могут соблюдаться не всеми процедурами, которые вызывают ваш код. Возможно, вам потребуется обеспечить взаимодействие с функцией, написанной на языке ассемблера или языке высокого уровня с другими соглашениями о регистрах. Ваш код может даже вызываться драйвером устройства или другими приложениями. Для таких случаев в Borland C++ имеются модификаторы функций *_saveregs* и *_loadds*.

Модификатор *_saveregs* предписывает компилятору безусловно сохранять все регистры перед выполнением тела функции и восстанавливать их по завершении выполнения. Используйте этот модификатор, когда вам нужно обеспечить интерфейс с процедурами, которые предполагают, что ваша функция будет сохранять значения всех регистров.

Модификатор *_loadds* указывает, что перед выполнением тела функции необходимо сохранить текущее значение сегментного регистра данных и установить его на адрес сегмента данных вашей программы. Первоначальное

значение сегментного регистра данных восстанавливается при выходе из функции. Этот модификатор полезен, если ваша функция может вызываться процедурой, которая изменяет исходное значение вашего регистра DS.

_export

Модификатор функции _export напоминает *_loadds,* но специфичен да программ, ориентированных на среду 16-бит-Windows. Он предписывает компилятору генерировать код, который обеспечивает функции доступ к глобальным переменным вашей программы в случае, когда она вызывается из другого приложения или непосредственно из Windows. Функции, объявленные с модификатором *__export,* должны также иметь модификатор *far*, если используется модель памяти Small или Medium.

Соглашения о вызове

Порядок, в котором параметры некоторой функции помещаются в стек, и способ очистки стека при возврате из функции составляют то, что называется *соглашением о вызове*. Соглашение, которому следует компилятор при вызове функции, зависит от установленного на данный момент соглашения и от определения функции. Функция может специфицировать одно из нескольких ключевых слов, указывающих метод, посредством которого она должна вызываться.

_cdecl

По умолчанию компилятор применяет соглашение о вызове языка C для тех функций, которые не указывают в явном виде предполагаемый ими способ передачи параметров. Однако вы можете изменить конфигурацию компилятора и установить другое соглашение, применяемое по умолчанию. Поэтому некоторые библиотеки явно требуют применения соглашений C, специфицируя для функций модификатор *_cdecl*. Например:

```
void  _cdecl  DisplayError( int ErrCode );
```

Ключевое слово *_cdecl* влияет также на внутреннее имя, присваиваемое функции компилятором. Модификатор предписывает использовать соглашение C о именах, в соответствии с которым создается копия имени (с различением верхнего и нижнего регистров), к которому спереди добавляется символ подчеркивания. Таким образом, внутренним именем функции DisplayError является *_DisplayError*.
Хотя соглашение C используется по умолчанию, можно быть уверенным, что оно активно, если вы

- укажете опцию -p- или -pc в командной строке компилятора

или

- в меню диалога Project Options укажете 16-bit Compiler или 32-bit Compiler | Calling Convention | C.

_pascal

Соглашение языка Pascal предполагает, что параметры передаются в порядке, обратном по отношению к C; они помещаются в стек слева направо. Кроме того, вызываемая функция ответственна за очистку стека. Соглашение Pascal может оказаться более эффективным, особенно если функция вызывается много раз и из разных мест. Однако функция, использующая это соглашение, не может быть объявлена со списком параметров переменной длины. Для функции, вызываемой в стиле Pascal, укажите в определении модификатор *_pascal*.

```
void _pascal DisplayErrorP( int ErrCode );
```

При таком определении внутреннее имя, присвоенное функции, будет совпадать с указанным вами именем, за исключением того, что символы будут преобразованы к верхнему регистру. Внутренним именем функции DisplayErrorP, таким образом, будет *DISPLAYERRORP*.
Вы можете активировать соглашение Pascal, если
• укажете опцию -p в командной строке компилятора или
• в меню диалога Project Options укажете 16-bit или 32-bit Compiler | Calling Convention | Pascal.
При изменении соглашения, применяемого по умолчанию, нужно соблюдать осторожность. Например, следующий код вызовет сообщение об ошибке Undefined symbol _main..., если вы укажете Pascal как соглашение о вызове по умолчанию.

```
#include <stdio.h>
void ShowMessage( char *msg )
{
printf( msg );
}
int main( void )
{
ShowMessage( "Привет! \n" );
return 0;
}
```

Ошибка является последствием соглашения языка Pascal о именах, в соответствии с котором функция main() получает внутреннее имя MAIN. Библиотеке компилятора, однако, требуется функция с именем _main, соответствующее соглашению C. Чтобы предотвратить появление ошибки, в определении main() следует указать ключ *_cdecl*.

```
int _cdecl main( void ) {
/*...*/
return 0;
}
```

_fastcall

Параметры обычно передаются через стек. Вы можете, однако, написать функцию, которая предполагает, что параметры передаются в регистрах, применив модификтор *__fastcall*. Существуют ограничения относительно числа и типа параметров, передаваемых в регистрах. В качестве общего правила, используйте соглашение _fastcall только для функций, которые требуют не более трех параметров типа char, int, unsigned, long или коротких указателей.

```
/*********************************************************/
/* FASTCALL.C: Функция, использующая соглашение _fastcall */
/*********************************************************/
#include <stdio.h>
#include <conio.h>
int _fastcall Sum( int a, int b, int c )
{
return a+b+c;
}
int main( void )
{

int x=10, y=20, z=30;
printf( "Сумма равна %d\n", Sum( x, y, z ) );
```

```
return 0;
}
```

Внутренним именем функции при соглашении _fascall является имя, определенное программистом, с добавлением префикса '@'. Таким образом, внутренним именем функции Sum будет *@Sum*.
Вы можете сделать соглашение _fascall подразумеваемым по умолчанию, если
• укажете опцию -pr в командной строке компилятора или
• в меню диалога Project Options укажете 16-bit или 32-bit Compiler Calling Convention | Register.

_stdcall

Стандартное соглашение о вызове, специфицируемое модификатором *_stdcall,* может применяться только при построении 32-битных приложений,

```
void _stdcall SaveThreadData( void *p )
{
/*...*/
}
```

Оно представляет собой гибрид методов C и Pascal. Параметры помещаются в стек справа налево; однако за очистку стека отвечает вызванная функция. Нет разницы между именем функции, определенной с модификатором _stdcall, и внутренним именем, присвоенным компилятором. Внутренним именем функции SaveThreadData является *SaveThreadData*.
Вы можете сделать соглашение _stdcall подразумеваемым по умолчанию, если
• укажете опцию -ps в командной строке компилятора или
• в меню диалога Project Options укажете 32-bit Compiler | Calling Con vention | Standard Call.

Встроенный код ассемблера

Ничто не может сравниться с языком ассемблера, когда нужны эффективность и непосредственный контроль. Код ассемблера, однако, обычно не переносим и требует утомительного, для большинства из нас, внимания к деталям. В качестве компромисса вы можете, с помощью ключевого слова *_asm,* вставлять в ваши функции на C операторы языка ассемблера. Секция кода на языке ассемблера может даже взаимодействовать с вашими переменными и функциями на C. Если вы хотите вставить несколько ассемблерных операторов, заключите их в фигурные скобки.

```
/* BASM1.C: Пример, использующий встроенный ассемблер */
#include <stdio.h>
int num1 = 125; int num2 = 300;
int main( void )
{
printf( "num1=%d, num2=%d, num1, num2 );
/* Быстрая ассемблерная процедура для */
 /* загрузки двух чисел в регистры и их */
/* обмена с применением операции XOR */
asm {
mov ax, num1
mov dx, num2
xor ax, dx
xor dx, ax
```

```
xor ax, dx
mov num1, ax; mov num2, dx }
printf( "num1=%d, num2=%d, num1, num2 );
return 0;
}
```

Можно создавать метки внутри блока _asm и использовать их для ветвления. Можно также вызывать из блока _asm другие функции С.

у**********/ /* BASM2.C: Пример, использующий встроенный ассемблер */

```
/*****************************************************/
#include <stdio.h>
void _near ShowMsg() {
printf( "Привет от близкой функции!\n" ); }
void „far SHowMsgF() {
printf( "Привет от далекой Функции! \n" ); }
int maln( void ) {
asm {
mov cx, 3 LBL01:
call near ptr ShowMsg call far ptr ShowMsgF loop LBL01
}
return 0; }
```

Вызывая функцию, принимающую параметры, убедитесь, что перед исполнением инструкции call аргументы помещаются в стек. Может также потребоваться очистить стек после вызова. Если вы собираетесь компилировать код в разных моделях памяти, можно применить' препроцессор для условного использования ближних или дальних вызовов функций.

/**********/
/* BASM3.C: Пример, использующий встроенный ассемблер */
/**********/

```
#include <stdio.h>
#define COUNT 55
void ShowCount( int cnt )
{
printf( "Счетчик равен %dl\n", cnt );
}
int main( void )
{
asm {
mov ax, COUNT /* Поместить параметр */
push ax   /* в стек...  */
#if defined(__TINY__) || defined(__SMALL__) \
|| dafined(__MEDIUM__)
call near ptr ShowCount
#else
call far ptr ShowCount
#endif
pop cx   /* Очистить стек... */
```

```
}
return 0;
}
```

Псевдо -регистры

Borland C++ предусматривает следующие ключевые слова для непосредственных операций над регистрами и флагами CPU:
_AX _AL _AH _SI _ES _BX _BL _BH _DI _SS _CX _CL _CH _BP _CS _DX _DL _DH _SP _DS _FLAGS

Эти ключевые слова могут рассматриваться как глобальные переменные и применяться так же. Если при компиляции разрешено использование набора инструкций процессора i386, можно применять следующие ключевые слова, соответствующие специфическим регистрам i386:
_EAX _EBX _ECX
_EDX _ESI _EDI _ESP _EBP

Помните о побочных эффектах при использовании псевдо-регистров

Нужно быть очень осторожным, работая с псевдо-регистрами, так как некоторые операторы могут приводить к побочным эффектам, разрушающим содержимое других регистров. Например, такой код:

```
#include <dos.h>

char msg[] = "Hello!"

void f()
{
   _AH = 0x09;
   _DX = FP_OFF( msg );
   _DS = FP_SEG( msg );
   geninterrupt( 0x21 );
}
```

не будет работать, так как присвоение значения регистру DX связано с временным использованием регистра AX, при котором разрушается содержимое AH. Поэтому часто лучше применять вместо псевдо-регистров встроенный код ассемблера.

Заключение

Лучше всего ограничить применение расширений языка, описанных в этой главе, немногими функциями или модулями проекта, чтобы код мог легко компилироваться для различных конфигураций, поддерживаемых пакетом Borland C++ (DOS, 16-бит-Windows, 32-бит-Windows), и чтобы было достаточно несложным перенесение кода на процессоры, отличные от Intel. Выигрыш в эффективности, достигаемый при использовании встроенного кода

ассемблера, и удобство применения нормализованных указателей huge характеризуют диапазон возможностей, отвечающих требованиям самых разных приложений для персонального компьютера.

Глава 5
Переходим к C++

Язык C++ основывается на C. Это означает, что ваши программы на C могут быть легко перекомпилированы как программы C++. Есть, однако, несколько конструкций, которые в C и C++ обрабатываются по-разному, так что вам может потребоваться внести небольшие изменения в существующий код на языке C. С другой стороны, C++ предоставляет дополнительные возможности программистам, пишущим на C. Некоторые расширения, имеющиеся в C++, не связаны с использованием объектов и могут естественно вводиться в код ваших программ на C. Эта глава подразделяется на три части. В первой рассматриваются случаи, когда код по-разному интерпретируется в C и в C++. Если у вас есть определенное количество уже написанных строк на языке C и вы думаете над тем, не стоит ли перевести их на C++, первый раздел главы предоставит вам список пунктов, на которые нужно обратить внимание при просмотре вашего кода. Во второй части вы узнаете о некоторых чертах языка C++, которые, как мы надеемся, могут сделать для вас переход от C к C++ чем-то привлекательным и захватывающим. Последний раздел содержит информацию о том, как организуется взаимодействие кода на C++ с кодом на других языках. Это особенно важно, если вы пользуетесь библиотеками, написанными не на C++ или работаете над проектом совместно с другими программистами, пишущими на C, Паскале или языке ассемблера.

Чем C++ отличается от ANSI C

Лучший метод перевести программу с C на C++ - просто перекомпилировать ее как программу на C++ и проследить все появляющиеся по ходу дела предупреждения и возможные сообщения об ошибках. Это дает компилятору возможность обнаружить недействительные конструкции и двусмысленные выражения. Данный раздел освещает те сложности, с которыми вам, вероятно, придется встретиться и предлагает решения, способные сгладить неровности на этом пути.

Ключевые слова C++

C++ расширяет список зарезервированных слов, имеющийся в C. Если вы преобразуете существующий код на C в код C++, то должны убедиться, что никакие из ключевых слов языка C++ не использованы в вашем коде в качестве идентификаторов. Следующий список перечисляет ключевые слова C++, использование которых нужно проверить:

asm
catch
class
delete
friend
inline
new
operator
private
protected
public
template
this

throw
try
virtual

Прототипы функций

ANSI C поддерживает применение прототипов функций, но не считает его обязательным. Для программы на C Borland C++ обычно выдает предупреждающее сообщение, если вызову функции не предшествует прототип. Напротив, для C++ будет выдано сообщение об ошибке: You must declare a function before using it! (Функция должна быть объявлена перед ее использованием.)

void*

В C можно присваивать значение типа void* переменной-указателю любого типа. В C++ для такого присваивания необходимо явное преобразование типа. Следующий код годится для C, но вызывает сообщение об ошибке: cannot cast 'void*' to 'char*', если компилируется как код C++.

```
#include <stdlib.h> char *p = NULL;
void AllocateMemory() {
p = malloc( sizeof( char ) * 1024 ); }
```

Если ввести в присваивание явное преобразование типа, такой ошибки в режиме C++ возникать не будет:

```
p = ( char* )malloc( sizeof( char ) * 1024 );
```

Глобальные константы и внешние связи

Глобальная переменная, определенная как const, в C по умолчанию доступна для внешних обращений. В C++ определению константы должно предшествовать ключевое слово extern, если необходимо сделать ее доступной для глобальных связей.

Использование заголовочных файлов C++, содержащих const, файлами C

Интересно отметить, что это различие может явиться причиной ошибок duplicate symbol (дублирование символа) при компоновке программы, если заголовочный файл C++ используется несколькими C-файлами проекта.

Тип символьных констант

Типом символьной константы в C++ является char. В C для этой цели используется тип int. Это различие вряд ли может приводить к ошибкам, однако вы должны проверить, что ваш код не предполагает, что sizeof('c') равен sizeof(int).

Обход инициализации

Язык C допускает некоторые конструкции, которые могут обходить инициализацию переменной; в C++ они недопустимы. Например, следующий код C вызовет сообщение об ошибке Case bypasses initialization of a local variable (Case обходит инициализацию...), если компилируется как код C++.

```
#include <windows.h>
LRESULT CALLBACK _export WndProc( HWND hwnd, UINT msg,
WPARAM wParam, LPARAM lParam )
{
```

```
switch( msg )
{
unsigned key = 0;
case WM_LBUTTONDOWN:
key = ( unsigned )wParam;
if ( key & MK_SHIFT )
{
/*...*/


}
case WM_MOUSEMOVE:
break; }
return 0; }
```

Чтобы устранить сообщение об ошибке, можно определить переменную внутри блока после строки case WM_LBUTTONDOWN:.

C++ *как улучшенный C.*

Этот раздел рассматривает те особенности C++, которые делают его "лучшим C".

Аргументы, используемые по умолчанию

В прототипе функции C++ можно указывать значения по умолчанию для некоторых параметров. Это позволяет вам пропускать соответствующие аргументы при вызове функции, на место которых компилятор подставит указанные в прототипе значения. Параметры по умолчанию должны быть последовательными. Если вы приписываете значение по умолчанию некоторому параметру, то необходимо указать и значения для всех тех параметров, которые стоят справа от него. Соответственно, если при вызове функции параметр опускается, то должны быть опущены и все параметры справа. Аргументы по умолчанию специфицируются только в прототипе, но не в определении функции. Посмотрите несколько примеров:

```
//
// Прототип функции 'ShowMessage' с двумя
// параметрами, используемыми по умолчанию
//
void ShowMessage( char *msg, int x=0, int y=0 );


//
// Следующее неверно; всем параметрам справа от того, // для
которого указано значение по умолчанию, также // должны
приписываться значения
//
void ShowMsgC int x=0, int y=0, char *msg );
//
// Этот вызов ShowMessage неверен; если для второго
// параметра выбрано значение по умолчанию, то для третьего
// тоже должно использоваться значение по умолчанию
//
ShowMessage( "Error: Out of Memory", , 10 );
//
// Эти вызовы ShowMessage являются корректными
//
```

```
ShowMessage( "Error: Out of Memory" );
ShowMessage( "Error: Out of Memory", 10, 10 );
ShowMessage( "Error: Out of Memory", 10 );
```

Значения по умолчанию: не только константы!

Значение аргумента функции, используемое по умолчанию, не обязательно должно быть константой; оно может быть *глобальной* переменной или даже значением, возвращаемым некоторой функцией.

```
//////////////////////////////////////////////////////
// DEFARG.CPP: Пример аргументов по умолчанию   //
//////////////////////////////////////////////////////

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

//////////////////////////////////////////////////////
// Использовать как значение по умолчанию
// глобальную переменную errno...
void ShowError( int errValue = errno );
```

```
//////////////////////////////////////////////////////
void ShowError( int errValue )
{
   printf( "ERROR: %s\n", sys_errlist[errValue] );
}

//////////////////////////////////////////////////////
int main( void )
{
   fopen( "\\DOES\\NOT\\EXIST.TXT", "r" );
   ShowError();    // Использовать умолчание - errno!

   ShowError( ENOMEM );// Ошибка "недостаточно памяти"

   return 0;
}
```

Ссылки

В C++ имеется несколько видоизмененная форма указателя, называемая *ссылкой*. Ссылка на некоторую переменную может рассматриваться как указатель, который при употреблении всегда разыменовывается. Однако для ссылки не требуется дополнительного пространства в памяти: она является просто другим именем, или псевдонимом переменной. Для определения ссылки применяется унарный оператор &. Следующий пример иллюстрирует сказанное:

```
////////////////////////////////////////////////////// // REF.CPP: Применение ссылок C++... //
//////////////////////////////////////////////////////
include <stdio.h>
```

```
int value = 10;
int &refval = value; // Ссылка на value
int main( void )
{


printf( "value = %d \n", value );
refval += 5;  // Модификация через ссылку
printf( "value = %d \n", value );
printf( "Адрес value равен ) %p \n", &value );
printf( "Адрес refval равен %p \n", &refval );
return  0;
}
```

Ссылки применяются, прежде всего, для параметров функции и для типов возвращаемых функциями значений. Однако ссылка может быть определена и как псевдоним переменной, как показано в предыдущем примере. В этом случае ссылка при определении должна быть инициализирована, если только она не объявлена как extern. Ссылке после инициализации не может быть присвоена другая переменная.

Параметры-ссылки

В C параметры передаются значением. Это означает, что компилятор создает для вызываемой функции копию переменных, специфицированных в вызове. Если вызываемая функция должна изменять значение переменной, то при вызове передается указатель на последнюю. Функция может разыменовать указатель, чтобы модифицировать значение переменной в вызывающей функции. В C++ можно передавать параметры ссылкой. Вызов функции, ожидающей в качестве параметра *ссылочную переменную,* синтаксически не отличается от вызова, в котором параметр передается значением. Чтобы указать, что функция ожидает передачу параметра ссылкой, в прототипе перед именем переменной ставится модификатор &. Следующий пример демонстрирует использование параметров-ссылок.

```
////////////////////////////////////////////////////// // REF_PARM.CPP: Использование ссылочных переменных     //
////////////////////////////////////////////////////// include <stdio.h>
void Inc_val( int  i ) // Получает параметр-значение
{


i++; // Модификация не влияет на оригинал
}
void Inc_ptr( int *i ) // Получает адрес оригинала
{
(*!)++;          // Модифицирует оригинал
// путем косвенной адресации
}
void Inc_ref( int &i ) // Получает параметр-ссылку
{
i++;      // Модифицирует оригинал!
}
int main( void )
{

int  j=10;
```

```
printf( "J равняется %d\n", j );
Inc_val( j );
printf( "После Inc_val(j) j=%d\n", j );
Inc_ptr( &j );
printf( "После Inc_ptr(&j) j=%d\n", j );
Inc_ref( j );
printf( "После Inc_ref(j) j=%d\n", j );
return 0;
}
```

Когда вы вызываете функцию, ожидающую передачи ссылки, компилятор неявно передает адрес специфицированного параметра. Внутри самой функции компилятор транслирует все обращения к параметру, разыменовывая переданный адрес. Все это происходит за кулисами! Главным преимуществом ссылок по сравнению с указателями является простота кода:

- Вызывающей функции не требуется применять операцию взятия адреса

Ссылки также весьма полезны при передаче больших структур, так как передача адреса более эффективна, чем копирование всей структуры.

Передача аргументов ссылкой

При передаче значения с помощью ссылки нужно иметь в виду возможные побочные эффекты. За исключением случаев, когда вызываемая функция должна явным образом изменять значение параметра, лучше всего использовать ссылку на константу. Это позволяет компилятору считать ошибкой попытку функции модифицировать значение переменной.

Функция, возвращающая значение типа ссылки

Ссылка может быть использована как тип, возвращаемый функцией. Это позволяет присваивать функции значение. Например, рассмотрим следующее:

```
//////////////////////////////////////////////////////
// REF_RET.CPP:  Ссылка как возвращаемый функцией тип //
//////////////////////////////////////////////////////
#include <stdio.h>
const    int arraySize = 0xF; static int valArray[arraySize];
int& valueAt( int indx ) {
return valArray[indx]; }
int main( void ) {
for( int i=0; i<arraySlze; i++ )

valueAt( i ) = 1<< i;
for( 1=0; i<arraySize; i++ ) printf( "Значение [%02d] = %-6d\n", i, valueAt( i ) );
return 0; }
```

Функцию *valueAt(int)* можно применять как для чтения элемента с определенным индексом, так и для присваивания нового значения этому элементу.

Абстрагирование данных с помощью возвращения ссылки

Вы можете использовать функцию, которая возвращает ссылку на переменную, для того, чтобы *скрыть* переменную и контролировать доступ к ней. Например, функция *valueAt* могла бы быть определена следующим образом:

```
#include <assert.h>

int& valueAt( int indx )
{
   assert( indx >= 0 );
   assert( indx < arraySize );
   return valArray[indx];
}
```

Сделав массив *(valArray)* статическим и ограничив доступ к переменной функцией *valueAt()*, можно быть уверенным, что все обращения будут происходить в пределах границ массива. Такой прием может оказаться полезным для сокращения числа ошибок в проекте.

Встроенные (inline) функции

Ключевое слово *inline* является модификатором определения функции. но предписывает компилятору помещать расширение тела функции везде,

где происходит обращение к ней, вместо того, чтобы генерировать код вызова. Встроенные функции, вообще говоря, делают ваш код более быстрым, так как не требуют затрат на выполнение вызова. Однако использование больших inline-функций может привести к увеличению размера ваших приложений. Обыкновенно встроенными делают маленькие функции, такие, как процедура *swap* в следующем примере.

```
////////////////////////////////////////////////////// // INLINE.CPP: Пример кода со встроенной функцией... // //////////////////////////////////////////////////////
#include <stdio.h>
inline void swap( int &i,  int &j  ) {
i  =j  =i  =j ;
}
int main( void )
{
int a = 10,  b = 20;
printf( "a = %d ,  b = %d \n",  a,  b ); swap( a,  b );
printf( "a = %d ,   b = %d \n",  a,   b );
return 0;
}
```

Ограничения на использование inline-функций

Borland C++ не допускает inline-расширения для функций:

- применяющих операторы while, do/while, switch/case, for и goto;
- имеющих тип не void и не содержащих оператора return;
- применяющих встроенный код ассемблера.

Определение inline-функций

Определение встроенной функции должно обязательно предшествовать ее вызовам. Логично поэтому разместить определения inline-функций в заголовочном файле. В следующем коде, например, функция *min()* вызывается до ее определения. Компилятор выдаст сообщение об ошибке Function defined inline after use as extern (Функция определена inline после использования ее как внешней).

```
int i, j;
int min( int i1, int i2 );
int main( void ); {
return min( i, j ); }
inline int min( int i1, int i2 )
{
return i1 > i2 ? i2 : i1;
}
```

Сообщение об ошибке можно устранить, если включить ключ inline в прототип функции. Однако если функция вызывается до определения, компилятор генерирует вызов вместо ее inline-расширения. Следующий пример это иллюстрирует:

```
int i, J, k, 1;
inline int max( int i1, int i2 );
int func1( void )
{
return max( 1, j ); // Не расширяется как inline!
}
inline int max( int i1, int i2 )
{
return i1 > i2 ? i1 : i2;
}


int func2( void ) {
return max( k, 1 ); // Расширяется inline! }
```

Операция ::

В языке С локальная переменная скрывает любую глобальную переменную с тем же именем. Все обращения к имени в пределах области действия локального определения относятся к локальной переменной. C++ позволяет вам получить доступ к глобальной переменной, применив к имени префикс , называемый *операцией разрешения видимости*. Следующий пример демонстрирует использование этой операции. Заметьте, что операция :: позволяет обращаться к глобальной переменной, даже если переменная с таким же именем определена во внешней локальной области.

```
//////////////////////////////////////////////////////
// SCOPE. CPP: Иллюстрирует оператор разрешения видимости //
//////////////////////////////////////////////////////
#include <stdio.h>
int total =10; // Глобальная переменная
int main( void )
{    //
int total = 100; // Локальная переменная
// во внешней области действия
// if ( total > 0 )
```

```
{ // int total = 1000;  // Локальная переменная
// во внутренней области
//
printf( "Локальная total : %d \n",  total );
printf( "Глобальная total : %d \n", ::total );
}
return 0;
}
```

Перегруженные функции

В языке C каждая функция должна иметь уникальное имя. В C++ можно определять функции с одинаковым именем, но уникальными типами аргументов. Эта особенность способствует улучшению ясности текста, так как одно и то же *перегруженное* имя описывает одни и те же действия, которые могут производиться над данными различного типа. Компилятор выбирает нужную функцию по соответствию типов аргументов. Следующий пример перегружает функцию *ShowMessage,* чтобы она воспринимала в качестве аргумента либо целое, либо строку.

```
//////////////////////////////////////////////////////
// OVERLOAD.CPP: Использование перегруженных функций //
//////////////////////////////////////////////////////////////////////////
#include <stdlib.h> #include <stdio.h>
void ShowMessage( int ); void ShowMessage( char *msg );
void ShowMessage( int errCode ) {
printf( "MSG:  %s \n",  sys_errlist[errCode] ); }
void ShowMessageC char *msg ) {
printf( "MSG: %s \n",  msg ); }
int main( void )
{
ShowMessage( 1 ); // 1: Недействительный номер функции!
ShowMessage( "Ошибка исправлена!" );
return 0;
}
```

Ограничения

• Функции, отличающиеся только типом возвращаемого значения, не могут быть перегружены. Например, следующий код вызовет ошибку Type mismatch in redeclaration (Несоответствие типа в повторном описании), так как компилятор принимает во внимание только аргументы:

int getCustInfo(char *name);// Возвращает номер счета char *getCustInfo(char *name);// Возвращает адрес

• Функции не могут быть перегружены, если их параметры различаются только применением модификаторов *const* или *volatile,* или использованием ссылки. Ниже показан пример описаний и ошибки, которые он генерирует:

void DelRec(int indx);
void DelRec(int &indx);
void DelRec(const int indx);
void DelRec(volatile int indx);
'DelRec(int &)' cannot be distinguished from 'DelRec(int)'
(... невозможно отличить от ...)

'DelRec(const int)' cannot be distinguished from 'DelRec(int)' 'DelRec(volatile int)' cannot be distinguished from 'DelRec(int)'

Реализация: декорированное имя

В основе перегрузки функций лежит особенность C++, обычно называемое *декорированием имени.* Для внутреннего употребления компилятор добавляет к имени несколько символов, показывающих тип и порядок параметров, воспринимаемых функцией. В следующей таблице, например, показаны внутренние имена некоторых перегруженных функций:

void func(int i); @func$qi
int func(int i); @func$qi
void func(char i); @func$qc
void func(char *p); @func$qpc

Заметьте, что внутренние имена первых двух функций идентичны, хотя типы возвращаемых ими значений различаются. В приложении А приведена таблица, в которой описаны различные последовательности символов, применяемые в Borland C++ для декорирования имен.

Определения переменных

В языке С локальные переменные должны определяться в начале блока. Это предполагает, что определения должны предшествовать исполняемым операторам. C++ позволяет вам определять переменные в любом месте. Это уменьшает вероятность ошибок, так как переменную можно определить в непосредственной близости от кода, который ее использует. Рассмотрим следующий пример:

```
////////////////////////////////////////////////////// // LOCALVAR.CPP:  Гибкие определения переменных... //
////////////////////////////////////////////////////// #include <stdio.h>
int main( void )
{
printf( "Привет! \n" )
int i;
printf( "Значение i = %d \n",  i );
for( int j=0; j<10; j++ ) {
printf( "J = %d \n", j ); }
printf( "Текущее значение j = %d \n", j );
return 0;
}
```

Этот пример применяет обычную в C++ конструкцию, в которой счетчик цикла *for* определяется в самом операторе. Обратите внимание, что счетчик (j) остается доступным за пределами блока *for.*

Константные значения

В С модификатор *const* означат, что значение переменной не может быть изменено после того, как она инициализирована. C++, напротив, рассматривает переменные с *const* как истинные константные выражения. Поэтому отличие от С в C++ можно применять переменную-константу для спецификации размера массива.

```
////////////////////////////////////////////////// // CNSTARRY.CPP: Применение переменной const для // // размерности массива // //////////////////////////////////////////////////
const unsigned numSqr = 8*8; unsigmed board[numSqr];
```

Имена-этикетки в enum, struct и union

В C++ имя-этикетка в описании enum, struct или union является именем типа. Поэтому в определениях переменных нет необходимости употреблять ключевые слова *enum, struct* или *union*. Например:

```
enum Account { edu, corp, persnl };
struct custInfo {
char      name[80];
long      accntNum;
Account AccType;   // Вместо 'enum Account accType' };
custInfo c = { "FD, Ltd.", 100, corp }; // Без ключа struct!
```

Анонимные объединения

В C++ имеется специальная форма объединений, называемая *анонимным объединением*. Такие объединения не имеют имен-этикеток. Элементы объединения занимают одно и то же место в памяти и доступ к ним осуществляется как к переменным. Глобальные анонимные объединения должны определяться как статические. Следующий пример иллюстрирует глобальные и локальные анонимные объединения.

```
////////////////////////////////////////////////////////////////////////////////////////////
// ANOUNION.CPP: Иллюстрирует анонимные объединения  //
////////////////////////////////////////////////////////////////////////////////////////////
№include <string.h>
//
// custName и custId занимают одну и ту же область памяти
//
static union {
char custName[80];
long custId;
};
int main( void )
{
//
// Локальное анонимное объединение - newId и counter
// занимают одну и ту же область памяти
//
union {
int newId;
int counter;
};
for( counter=0;  counter<10;  counter++ ) custId = counter;
strcpy( custName, "NEW" ); newId = 0x100;
return 0;
}
```

Гибкие операторы распределения памяти

Для управления распределением динамической памяти C++ предоставляет операторы *new* и *delete* (или *new[ ]* и *delete[ ]*). По традиции в программах

на С применяют для этой цели библиотечные процедуры *malloc, calloc и free.* Как и все функции С, они могут вызываться и в С++. Однако операции С++ обладают расширенными возможностями:

• new возвращает указатель на тип, для которого выделяется память, в то время как malloc возвращает пустой указатель. Таким образом, при использовании new преобразование типа не требуется. Вот пример:

```
#include <stdlib.h> #include <stdio.h>
long *lptr;
void f1( void ) {
lptr = ( long* )malloc( sizeof( long ) );
*lptr = 0x1000;
printf( "Значение равно %ld \n", *lptr ); free( lptr ); }
void f2( void ); {
lptr = new long;
*lptr = 0x1000;
printf( "Значение равно %d \n", *lptr ); delete lptr; }
```

• Наряду с операцией new С++ предусматривает библиотечную функцию *set_new_handler,* которую можно использовать для установки определяемой пользователем процедуры обработки ошибок. Она будет вызываться в тех случаях, когда при распределении памяти возникает ошибка. Следующий пример демонстрирует определяемый пользователем обработчик:

```
////////////////////////////////////////////////////////////////////////////////////////////
// NEWHNDLR.CPP: Применение определенного пользователем //
// обработчика ошибок new... //
//////////////////////////////////////////////////////////////////////////////////////////////

#include <stdio.h> #include <stdlib.h> #include <new.h>
void MyNewHandler() {
printf( "Нет памяти! \n" ); }
int main( void )
{
// Установить новый обработчик set_new_handler( MyNewHandler
);
//...//
return 0;
}
```

new_handler и обработка исключений

С введением обработки мёклвдчитеЛБцш. ситуаций сцанев^тся" необхй- *щмд$* процедура^ *nЫ^handier* для яыбрасьшания *^щтчЬШя хаИдс* Раапи'ч'им& библиотеки, такие,, как ObjectWindows, нспаль<з.утот^этот метод. .

• Гибкость операций new и delete становится очевидной при использовании классов, так как класс может определять свой собственный вариант этих операций. Операции new и delete также вызывают для классов специальные процедуры инициализации и очистки, если они имеются. Глобальный вариант операций new и delete, однако, может быть переопределен для не-классовых объектов без new и delete. Этот прием часто применяется в библиотеках С++ для отладочных целей. Рассмотрите четыре операции:

```
void* operator new( size_t );
void* operator new[] ( size_t );
```

Операция new используется во всех случаях, кроме тех, когда вы распределяете память под массив. Для размещения массивов используется операция new[].
void operator delete(void*); void operator delete[] (void*);
Аналогично, оператор delete используется для всех запросов на удаление, кроме удаления массивов. Для удаления массивов используется delete[].
Операцию new можно перегрузить так, что она будет воспринимать дополнительные параметры. Следующий пример показывает переопределение глобальных операций new и delete. Он также содержит перегруженный вариант new, принимающий два дополнительных параметра.

```
//////////////////////////////////////////////////////
// NEWDEL.CPP: Переопределение операций new и delete //
//////////////////////////////////////////////////////
#include <stdio.h>
#include <stdlib.h>
//
// Перекрыть глобальную операцию new
//
void* operator new( size_t size )
{
printf( "new() запрашивает %u байт \n", size );
return malloc( size ); }
//
// Перекрыть глобальную операцию delete
//
void operator delete( void *p )
{
printf( "delete() \n" );
free( p );
}



// Перекрыть глобальную операцию new[]
//
void* operator new[]( size_t size )
{
printf( "new[] запрашивает %u байт \n", size );
return malloc( size ); }
//
// Перекрыть глобальную операцию delete[]
//
void operator delete[]( void *p )
{
printf( "delete[] () \n" );
free( p ); }
//
// Перегрузить глобальную операцию new...
//
void* operator new( size_t size, char *fname, int line )
{
printf( "Вызов new() из %s в строке %d \n", fname, line );
return malloc( size ); }
```

```
int main( void ) {
int *p = new int; // Вызывает глоб. new

*p  = 10;
delete p;      // Вызывает глоб. delete] for( int i=0; i<10;
p = new int[10]; // Вызывает глоб. new[
i++ )
p[i] = i*10; delete[] p;     // Вызывает глоб. delete[]
// Вызывает перегруженное глоб. new
p = new(_FILE_, _LINE_) int;


*p  = 10;      // Вызывает глоб. delete
delete p;
return 0;
}
```

Перегрузка new для контроля над распределением памяти

Показанный технический прием может применяться для слежения за распределением памяти и удалением объектов. Перегрузив *new* и *delete*, вы можете добавить процедуры для обнаружения повторных удалений, записи в занятую память, утечек памяти и т.п.

Операция new и помещающий синтаксис

Вы можете перегрузить операцию new таким образом, что память будет выделяться по указанному адресу. Такую конструкцию называют *помещающей формой* операции:

```
//////////////////////////////////////////////////////////
// PLACENEW.CPP: Помещающая форма new...                //
//////////////////////////////////////////////////////////
#include <new.h>

int __i = 0;

//
// Операция new, допускающая помещающий синтаксис
//
void * operator new ( size_t, void *p )
{
   return p;
}

int main( void )
```

```
// Поместить целое по адресу _i //
int *pi = new(&_i) int;

*pi = 20;

return 0;
```

Примечание: *Помещающая форма* обычно применяется для "вызова" конструктора для существующего объекта. Подробности см. в главе 6.

Сопряжение C++ с C, Паскалем и языком ассемблера

При разработке проекта может потребоваться использовать код C++ совместно с библиотеками на C, языке ассемблера или Паскале. Возможно, вам также придется написать некоторые функции C++, которые будут вызываться кодом не на C++. При решении подобных задач *декорирование имен* должно быть запрещено. Для этой цели C++ предусматривает *спецификации внешних связей.*

Спецификация внешней связи

Спецификация связи может применяться в двух случаях:

• Чтобы запретить компилятору декорирование имени для определяемой вами функции C++.

• Чтобы информировать компилятор, что внешняя функция, которую вы вызываете, не использует внешнюю спецификацию C++.

Следующий пример иллюстрирует оба случая:

```
//////////////////////////////////////////////////////////
// TYPELINK.CPP: Спецификация связи... //
//////////////////////////////////////////////////////////
#include <stdio.h>
extern "C" void CppFuncCalledFromC( void );
extern "C" {
void FunctionInCLibrary( void ); void
FunctionlnASMlibrary ( int );
}
void CppFuncCalledFromC( void )
```

Спецификация внешней связи
и "оглашение о вызове

```
derude Windows.дд

// InitCallBack не будет декорировано
efctern "C" BOOL FAR PASCAL _export InitCallBack{ UIHT j;

// InfoCallBack будет декорировано!
BOOL FAR PASCAL „export IrifoCallBack( UIHT );
```

В этом примере внутренними именами, которые будет *хщерщдт*ть компилятор:для возвратно вызываемых функций — LmtCallBaek *v.*! InfoCallBack — являются соответственно INITCALLBACK и @]NFO-CAbLBACKSQUI.

Заголовочные файлы

Если вы работаете с языком C++, но должны вызывать функции, написанные на C, можно включить заголовок C в блок внешних спецификаций C, как показано ниже:

```
////////////////////////////////////////////////////// //CPP_N_C.CPP: Использование библиотек C... //
//////////////////////////////////////////////////////
extern "C"
{
#include "clibfunc.h"
#include "asmfunc.h"
}
```

Если вы пишете функции C++, которые будут вызываться из кода не на C++, то можете организовать файл заголовка, специфицирующий внешние связи C при компиляции в режиме C++. Вот примерный заголовок:

```
«if defined(_cplusplus) // Режим компиляции C++?
extern "C" { // Да, использовать специф. "C"
#endif // (_cplusplus)
// Здесь находятся прототипы функций


#if defined(__cplusplus) // Режим C++?
} // Закрыть блок внешней спецификации
#endif //(_cplusplus)
```

Приведенный заголовок проверяет макрос *_cplusplus,* определенный в режиме компиляции C++.

Заключение

Вы познакомились с разнообразными элементами языка C++, с помощью которых можно легко усовершенствовать уже существующий код на C. В следующей главе будут рассмотрены основания объектно-ориентированного программирования на языке C++: классы C++!

Глава 6
Объектно-ориентированное программирование *на* C++

В этой главе изучаются объектно-ориентированные аспекты C++. Вы познакомитесь *с* элементами языка, которые позволяют создавать новые типы данных, писать утилизируемый код и помогают лучше организовать структуру вашей программы. Также рассматриваются термины, связанные с объектно-ориентированным программированием, такие, как *полиморфизм* или *наследование.* Хотя объектно-ориентированные особенности C++ не имеют отношения к процедурным аспектам C, вы должны ознакомиться с основами языка C, прежде чем приступите к чтению данной главы.

Класс в C++

Класс является фундаментальным механизмом, с введением которого язык C приобретает объектно-ориентированные черты и становится C++. Класс есть расширение понятия структуры языка C. Он позволяет создавать типы и определять функции, которые задают поведение типа. Каждый представитель класса называется *объектом.* В некотором смысле можно рассматривать класс как средство расширения языка.

Определение класса

Определение *класса* напоминает определение структуры в C, за исключением того, что:

• оно обычно содержит одну или несколько *спецификаций доступа,* задаваемых с помощью ключевых слов *public, protected* или *private;*

• вместо ключевого слова *struct* могут применяться *class* или *union;*

• оно обычно включает в себя функции *(функции-элементы,* или *методы)* наряду с элементами данных;

• обычно в нем имеются некоторые специальные функции, такие, как *конструктор* (функция с тем же именем, что и сам класс) и *деструктор* (функция, именем которой является имя класса с префиксом-тильдой (~).

В следующем примере показаны несколько определений классов.

```
//////////////////////////////////////////////////////
// CLASSDEF.CPP: Примеры определения классов... //
//////////////////////////////////////////////////////
//
// Класс Rect (аналогичен struct в C)
//
struct Rect
{
int x1;  //
int y1;  // Элементы данных Rect
int x2;  //
int y2;  // };
//
// Класс Point: содержит как элементы данных,
//    так и элементы-функции...
//    Также использует спецификаторы доступа
//
```

```
struct Point
{
private:  // Спецификатор 'частного' доступа


int x;    // Элементы данных класса int y;    // типа
'Point'
public:     // Спецификатор 'публичного'доступа int GetX();
//
int GetY();      // Элементы-функции класса void SetX( int
);  // void 8etY( int );  // };
//
// Класс Line
//
class Line
{
Point pt1;  //
Point pt2;  // Элементы-данные...
int width;  //
```
public: // Спецификатор 'публичного' доступа Line(int _x, int _y); // Функция-элемент: конструктор! "Line(); // Функция-элемент: деструктор! }

В приведенном примере определение класса *Rect* совпадает с определением структуры в языке С. Класс *Point* содержит *спецификаторы доступа* и функции-элементы. В отличие от *Rect* или *Point* класс *Line* определяется с помощью ключевого слова *class* и содержит функции-элементы специального вида. В следующих разделах рассматриваются спецификаторы доступа, три ключевых слова, определяющих класс, и функции-элементы.

Управление доступом

В языке С элементы структуры доступны для любой функции в пределах области действия структуры. Это часто приводит к изменению данных из-за невнимательности. В C++ можно ограничить видимость данных и функций класса при помощи меток *public, protected* и *private.* Метка-спецификатор доступа применяется ко всем элементам класса, следующим за ней, пока не встретится другая метка или кончится определение класса. Следующая таблица описывает три спецификатора доступа.

Таблица 6.1. Спецификаторы доступа к классу

Метка доступа	*Описание*
private:	Элементы данных и функции-элементы доступны только для функций-элементов этого класса.
public:	Элементы данных и функции-элементы класса доступны для функций-элементов и других функций программы, в которой имеется представитель класса.
protected:	Элементы данных и функции-элементы доступны для функций-элементов данного класса и классов, производных от него

Классы, структуры и объединения

В C++ *структура, класс* и *объединение* рассматриваются как типы классов. *Структура* и *класс* походят друг на друга, за исключением доступа по умолчанию: в структуре элементы имеют по умолчанию доступ *public,* в то

время как в классе - *private*. Аналогично структуре, *объединение* по умолчанию предоставляет доступ *public;* аналогично объединениям в С, его элементы данных размещаются начиная с одного и того же места в памяти.
В следующей таблице перечислены эти отличия:

Таблица 6.2 Различие между классом, структурой и объединением

	Классы	*Структуры*	*Объединения*
Ключевое слово:	class	struct	union
Доступ по умолчанию:	private	public	public
Перекрытие данных:	Нет	Нет	Да

Элементы класса

Элементы класса делятся на две основных категории:
• данные, называемые элементами-данными (data members)
• код, называемый элементами-функциями (member functions), или методами

Элементы данных

Элементы-данные классов C++ похожи на элементы структур языка С с некоторыми дополнениями. Следующий список характеризует свойства элементов-данных класса:
• Элементы-данные не могут определяться как *auto, extern* или *register.*
• Элементами данных могут быть перечислимые типы, битовые поля или представители ранее объявленного класса.
• Элементом данных класса не может быть представитель самого этого класса.
• Элемент данных класса может являться указатель или ссылка на представитель этого класса.

Элементы-функции

Элемент-функция является функцией, *объявленной (описанной)* внутри определения класса. Тело функции может также *определяться* внутри определения класса; в этом случае функция называется *встроенной (inline)* функцией-элементом. Когда тело функции определяется вне тела класса, перед именем функции ставится префикс из имени класса и операции разрешения видимости (::). Следующий пример иллюстрирует встроенные и невстроенные функции- элементы.

```
//////////////////////////////////////////////////////////
// MEMFUNC.CPP:  Класс с элементами-функциями... //
//////////////////////////////////////////////////////////
#include <assert.h>
const int MAX_X = 0x100; const int MAX_Y = 0x100;
struct Point
{
private:
int x;
int y;
public:
int GetX()
{
return x;
}
int GetY()
{
return y;
}
void SetX( int );
```

```
void SetY( int );
};
void Point::SetX( int _x )
assert ( _x >= 0 ); assert( _x <= MAX_X );
x = _x;
}
void Point::SetY( int _y )
{
assert( _y >= 0 );
assert( _y <= MAX_Y );
y = _y;
}
```

Функции-методы *GetX* и *GetY* определены как встроенные, а *SetX* и *SetY* встроенными не являются. Можно определить встроенную функцию-элемент и вне тела класса, указав в заголовке определения ключевое слово *inline*. Следующий пример это иллюстрирует:

```
//////////////////////////////////////////////////
// INLINE.CPP: Встроенные функции-элементы... //
//////////////////////////////////////////////////
class Time
{
int hr,  min;


public:
void SetTime( int, int ); void GetTime( int&, int& ); };
inline void Time::SetTime( int _hr, int _min ) {

hr = _hr;
min = _min; }
inline void Time::GetTime( int &hour, int &minute ) {
hour = hr;
minute = min; }
```

Класс как область действия

В языке С имеется четыре разновидности области действия: функция, файл, блок и прототип функции (см. Главу 2). В С++ вводится понятие *класса* как области действия: имена всех элементов класса находятся в области класса — они могут использоваться функциями-элементами класса. Имена элементов класса могут также использоваться в следующих случаях:

• С представителем класса (или выводимых из него классов), за которым следует операция-точка
[представитель.имя_элемента].

• С указателем на представитель класса (или выводимых из него классов), за которым следует операция ->
[указатель->имя_элемента].

• С именем класса, за которым следует операция :: разрешения видимости
[имя_класса: :имя_элемента].

Доступ к элементам данных

Функции-элементы находятся в области действия класса, в котором они определены. Таким образом, они могут обращаться к любому элементу класса, используя просто имя переменной. Обычные функции или функции-элементы другого класса могут получить доступ к элементам данных с помощью операций .

или ->, применяемых соответственно к представителю или указателю на представитель класса. Следующий пример это иллюстрирует.

```
////////////////////////////////////////////////// // DATAMEM.CPP: Доступ к элементам данных... //
//////////////////////////////////////////////////
class Coord
{
public:
int x,  y; };
int main( void )
{
Coord org; // Создать локальный объект Coord     *orgPtr = &org;     // Создать указатель на
объект
org.x = 0; // объект. элемент orgPtr->y = 0;                  // указатель->элемент
return 0; }
```

Вызов функций-элементов

Функции-элементы класса могут вызывать другие функции-элементы того же класса, используя имя функции. Обычные функции или функции-элементы других классов могут вызывать элементы класса с помощью операций . и ->, применяемых к представителю или указателю на представитель класса, как показано в следующем примере.

```
////////////////////////////////////////////////////////////////////////////
// CODEMEM.CPP:  Вызов функций-элементов... //
////////////////////////////////////////////////////////////////////////////
class Coord {
int x,  y; public:
void SetCoord( int _x,  int _y )
{ x = _x;  y = _y;  }
void GetCoord( int &_x,  int &_y ) { _x = x;  _y = y;   } };
int main( void )
{
Coord org; // Создать локальный объект Coord     *orgPtr = &org;     // Создать указатель на
объект
org.SetCoord(10,  10); // объект.элем_функция()
int col, row;
orgPtr->GetCoord(col,  row); // указатель->элем_функция
return 0; }
```

Использование указателей на функции-элементы

Вы можете определить указатель на функцию-элемент класса. Синтаксис определения следующий:

возвр_тип(имя_класса::*имя_указателя)(параметры);

Чтобы использовать такой указатель, можно применить либо операцию .* , либо операцию ->* . Следующий пример показывает, как используется указатель на функцию-элемент.

```
////////////////////////////////////////////////////////////////////////////////////////////////
// MEMPTR.CPP: Применение указателей на функцию-элемент  //
////////////////////////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
//
// Простой класс A
```

```
//
class A
{
int i; public:
A( int _i ) : i(_i) {}
void Func()
{
printf( "Hello!" );
}
//
// Функция, вызывающая функцию-элемент,
// адрес которой передается как параметр
//
void CallMemberPtr( void (A::*funcPtr)() )
{
(*this.*funcPtr)();
}
};
//
// Функция, вызывающая функцию-элемент для определенного //
объекта: и объект, и функция-элемент передаются // как
параметры...
//
void UseMemFuncPtr( A *aObjectPtr, void(A::*funcPtr)() )
{
(aObjectPtr->*funcPtr)();
}
//

// main: Вызывает функции, определенные выше...
//
int main( void )
{
void (A::*funcPtr)() = &A::Func;
A a1( 1965 ) ;
UseMemFuncPtr( &a1, funcPtr );
A a2( 3435 ); a2.CallMemberPtr( funcPtr );
return 0;
}
```

Для указателей на функцию-элемент существуют следующие правила:

• Указатель на функцию-элемент не может ссылаться на статическую функцию-элемент класса.

• Указатель на функцию-элемент не может быть преобразован в указатель на обычную функцию (не-элемент класса).

Указатель this

Каждая не-статическая функция-элемент имеет доступ к объекту, для которого вызвана, через ключевое слово *this*. Типом *this* является тип_класса*. Пример иллюстрирует использование указателя *this:*

```
//////////////////////////////////////////////////////////////////////
// THISPTR.CPP: Применение указателя this... //
//////////////////////////////////////////////////////////////////////
```

```
class Simple
{
public:
Simple();
void Greet()
{


printf("Hello!\n"); } };
Simple::Simple()
{
Greet(); // Все операторы
this->Greet(); // этой функции вызывают
(*this).Greet();       // функцию Greet()
}
```

Так как функции-элементы могут обращаться ко всем элементам класса просто по имени, в основном указатель *this* используется для возврата указателя (return this;) или ссылки (return *this) на подразумеваемый объект.

Применяйте статические функции-элементы для возвратно вызываемых процедур

Часто программистам, пишущим для Windows, приходится предусматривать требуемые интерфейсом Windows возвратно вызываемые функции. Программистам, работающим с DOS, может потребоваться установить обработчик событий для мыши или процедуру обслуживания прерывания. Функция-элемент не может быть использована для этих целей, так как при ее вызове не будет записан в стек скрытый параметр *this*. Однако статические функции-элементы не имеют параметра *this* и могут быть использованы в качестве возвратно-вызываемых или процедур обработки прерывания.

Специальные функции-элементы

Выражение *специальные функции-элементы* относится к некоторым функциям класса, от которых зависят способы создания, копирования, преобразования и уничтожения представителей класса. Часто эти функции неявно вызываются компилятором. Следующий список дает краткое описание этих функций.

Таблица 6.3. Описание специальных функций-элементов

Функция	*Описание*
Конструктор:	Инициализирует представитель класса
Конструктор копии:	Инициализирует новый представитель, используя значение уже существующего представителя класса
Операция присваивания:	Присваивает содержимое представителя другому представителю
Деструктор:	Производит очистку представителей класса
Операция *new:*	Распределяет память под динамические представители класса
Операция *delete:*	Освобождает динамическую память, занятую представителем класса
Функции преобразования:	Преобразуют представители класса к другому типу

Конструктор

Конструктор является функцией-элементом с тем же именем, что и ее класс. Она вызывается компилятором всегда, когда создается представитель класса. Если вы не определили никаких конструкторов, компилятор генерирует

конструктор по умолчанию (не имеющий параметров). Для конструкторов выполняются следующие правила:

- Для конструктора не указывается возвращаемый тип.
- Конструктор не может возвращать значение.
- Конструктор не наследуется.
- Конструктор не может быть объявлен как *const, volatile, virtual* или *static*.

Для ошибок применяйте обработку исключений

Конструктор не может йозвратить значение, чтобы сообщить об ошибке во время 'инициал изащщ. Однако для возврата ошибки из конструктора можно использовать механизм *обработки исключительных ситуаций* C++. (См. Главу 9 для более подробной информации.):

Вызов конструктора для существующего объекта

Применяя *помещающую форму*, можно вызывать конструктор для уже существующего объекта. Этот прием обычно применяется для глобальных объектов, которые должны быть инициализированы после того, как будет выполнена определенная процедура. Пример это иллюстрирует:

```
//////////////////////////////////////////////////////////////
// CALLCTR.CPP: Применение помещающей формы вызова констр.  //
//////////////////////////////////////////////////////////////
#include <new.h>

//
// Операция new, допускающая помещающий синтаксис
//
inline void * operator new( size_t, void *p )
{
   return p;
}

//
// Простой класс
//
class SomeObject
{
   public:
    SomeObject();
};
```

```
//
// Глобальный представитель класса
//
SomeObject gblInst;


int main( void )
{
   CheckSystem();    // Проверка конфигурации системы

   //
   // Теперь вызывается конструктор существующего объекта
   //
   new (&gblInst) SomeObject;

   return 0;
}
```

Список инициализации элементов

Элементы-данные класса обычно инициализируются в теле конструктора. Однако, определение конструктора может также содержать *список инициализации элементов*. *Список инициализации элементов* отделяется двоеточием (:) от заголовка определения функции и содержит элементы данных (и *базовые классы),* разделенные запятыми. Для каждого элемента указывается один или несколько параметров, используемых при инициализации.

Следующий пример показывает два сходных класса с конструкторами, применяющими два метода инициализации. Первый использует список инициализации, а второй присваивает элементам значения в теле конструктора.

```
////////////////////////////////////////////////////////////////////////////////
// MEMINIT.CPP: Инициализация или присваивание? //
////////////////////////////////////////////////////////////////////////////////
class XYValue
{
int x, y;
public:


// Использует список инициализации XYValue( int _x, int _y )
: x(_x), y(_y) { } };
class XYData {
int x, y; public:
// Присвоение в теле конструктора XYData( int _x, int _y ) {

x = _x;
y = _y; } };
```

Константы, ссылки и объекты — элементы данных

Хотя для большинства элементов данных применение *списка инициализации данных* не является обязательным, он является единственным методом инициализации элементов -*констант* и *ссылок.* Если элемент класса является объектом, конструктор которого требует спецификации одного или нескольких параметров, то *список инициализации* также будет для него единственно возможным механизмом инициализации.

Конструктор копии

Конструктор копии является конструктором специального вида: он воспринимает в качестве аргумента константную ссылку на объект класса *(const тип_класса&}* или простую ссылку на объект *(тип_класса&)*.

```
//////////////////////////////////////////////////////////////////////////
// COPYCON.CPP: Конструктор копии класса... //
//////////////////////////////////////////////////////////////////////////


class Coord
{
int x, y; public:
Coord(const Coord &src); //Конструктор копии
};
Coord::Coord(const Coord &src)
{

x = src.x;
Y = src.y;
}
```

Ссылка передается всякий раз, когда новый объект инициализируется значениями уже существующего. Если вы не предусмотрели конструктор копии, компилятор генерирует *конструктор копии по умолчанию*. Такой конструктор создает буквальную копию объекта. Это, скорее всего, будет непригодным для объектов, содержащих указатели или ссылки.

Объявление конструктора копии как private

Если ваш класс содержит указатели или ссылки, вам, вероятно, потребуется определить конструктор копии. Однако возможно другое решение - объявить конструктор копии в разделе *private*, чтобы лишить пользователей возможности копирования представителей класса. В результате компилятор будет генерировать сообщение об ошибке, если кто-то попытается создать копию объекта.

Операция присваивания

Операция присваивания является функцией-элементом с именем *operator=,* который воспринимает единственный аргумент типа *const тип_класса&* (или *тип_класса&)*.

Возвращение ссылки на объект

Сделайте вашу операцию присваивания возвращающей ссылку на объект, для которого она вызвана; тогда сохранится семантика присваивания (operator=), допускающая последовательные присвоения.

Операция присваивания вызывается компилятором, когда объекту присваивается значение другого объекта. Компилятор генерирует присваивание по умолчанию, если вы не определили операцию присваивания. Присваивание по умолчанию производит *побитное копирование*.

```
//////////////////////////////////////////////////////////////////////////
// ASSGOPER.CPP: Операция присваивания класса... //
//////////////////////////////////////////////////////////////////////////
class Coord
{
```

```
int x, y; public:
Coord& operator=(const Coord &src);
};
Coord& Coord::operator= (const Coord &src)
{
x = src.x;
y = src.y;
return *this;
}
```

Проверка на само-присвоение

В целях безопасности операция присваивания, с помощью указателя *this*, должна проверить возможность присвоения объекта самому себе. Например:

```
Coord& Coord::operator= (const Coord &src)
{
   if( &src == this )
```

Деструктор

Деструктор является дополнением конструктора. Он имеет то же имя, что и класс, но с префиксом-тильдой (~). Он вызывается всякий раз, когда уничтожается представитель класса. Для деструктора существуют следующие правила:

- Деструктор не может иметь аргументов.
- Деструктор не может возвращать значение.
- Деструктор не наследуется.
- Деструктор не может быть объявлен как *const, volatile* или *static*.
- Деструктор может быть объявлен как *virtual*.

Компилятор генерирует деструктор по умолчанию, если вы его не определяете.

Операция класса new

Класс может предусматривать применение своих собственных вариантов операции *new:*

- Функция *тип_класса::operator new* (если предусмотрена) вызывается всякий раз, когда создается динамический объект.
- Функция *тип_класса::operator newf]* (если предусмотрена) вызывается, когда создается динамический массив объектов.

Операция класса delete

Класс может определять свои собственные варианты операции delete:

- Функция *тип_класса:-.operator delete* (если предусмотрена) вызывается всякий раз, когда уничтожается динамический объект.
- Функция *тип_класса::operatordelete[]* (если предусмотрена) вызывается, когда уничтожается динамический массив объектов.

В следующем примере показан класс с операциями *new* и *delete*.

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////
// CLASSMEM.CPP: Класс, предусматривающий операции new/delete //
//////////////////////////////////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
//
// Макрос TRACE(msg): выводит параметр как строку
//
#if ! defined(NDEBUG)
#define TRACE(msg) printf( #msg "\n" )
#else
#define TRACE(msg) ((void)0)
#endif
//
// Максимальная длина имени
//
const int MAX_NAME = 30;
//
// Примерные данные для имен заказчиков
//
char *custName[] = {
"Maritum Hotel, Ltd.",
"Merville Hotel, Ltd.",
"Villa Carolina, Ltd."
};
//
// Класс 'custRec'
//
class custRec
{
int recNum; // Номер записи char name[MAX__NAME];
// Имя заказчика
public: custRec(); // Конструктор


~custRec(); // Деструктор
void «operator new ( slze_t ); // Операция new
void *operator new[] ( size_t ); // Операция new[]
void operator, delete ( void* ); // Операция delete
void operator delete[] ( void* ); // Операция delete[]
void InitData( char *newName, int newRec ); void
ShowData();
};
// Конструктор custRec::~custRec()
{
TRACE( Конструктор custRec );
}
// Деструктор
custRec::"custRec()
{
TRACE( Деструктор custRec );
}
// Операция new()
void* custRec::operator new(size_t size )
{
TRACE( Операция new );
```

```
return malloc( size );
}
// Операция delete()
void custRec::operator delete( void *p )
{
TRACE( Операция delete );
free( p );
}
// Операция new[]()
void* custRec::operator new[](size_t size )
{
TRACE( Операция new[] );
return malloc( size );
}


// Операция delete[]()
void custRec::operator delete[]( void *p )
{
TRACE( Операция delete[] );
free( p );
}
// Инициализация данных класса
void custRec::InitData( char *newName, int newRec )
{
strcpy( name, newName );
recNum = newName;
}
// Выводит данные класса void custRec::ShowData()
{
printf( "Заказчик: %s \t Запись #: %d \n",name, numRec );
}
int main( void )
{
custRec *pCust;
// Создать динамический представитель класса 'custRec'
pCust = new custRec;
// Инициализировать и показать данные
pCust->InitData( "Le Chaland, Ltd.", 0 );
pCust->ShowData();
// Удалить динамический объект
delete pCust;
int arraySize = sizeof(custName)/sizeof(custName[0]);
// Создать динамический массив 'custRec'-ов
pCust = new custRec[arraySize];


// Инициализировать и показать данные for( int i=0;
i<arraySize; i++ ) pCust[i].InitData( custName[i], i+1 );
for( i=0; i<arraySize; i++ ) pCust[i].ShowData();
// Удалить динамический массив delete []pCust;
return 0;
```

Применяйте к динамическим массивам delete []ptr

Нельзя безопасно удалять массивы с помощью операции *delete*! Для удаления любых массивов, в том числе символьных, вы должны использоваать операцию *delete[]*:

```
char *p = new char[100];

// ...

delete[]p;
```

Функции преобразования

Представители класса могут быть преобразованы к другому типу с помощью *конструкторов преобразования* или *операций приведения*.

Конструкторы преобразований

Если конструктор класса A воспринимает единственный аргумент типа B, то говорят, что B может быть приведен к A с помощью *преобразования конструктором*. Другими словами, компилятор может использовать конструктор класса A с аргументом типа B, чтобы получить A из B. Вот пример:

```
//////////////////////////////////////////////////////////////////////////////
// CONVCONS.CPP: Преобразование с помощью конструктора    //
//////////////////////////////////////////////////////////////////////////////
#include <string.h>
class Record {
public:
Record( char *newName );
//
// ...
// };
int raain( void ) {
//
// Следующая инициализация вызывает
// 'Record::Record(char*)', преобразуя
// "Mervin C." к классу 'Record'
//
Record customer = "Mervin C.";
// ... return 0; }
```

Операции приведения

Вы можете определить функции-элементы, которые будут осуществлять явное преобразования типа класса к другому типу. Эти функции называют *операциями приведения* или *процедурами преобразования* типа. Они имеют следующий вид:
operator *имя_нового_типа();*
Процедуры преобразования подчиняются следующим правилам: • Процедура преобразования не имеет аргументов.

• Процедура преобразования не имеет явной спецификации типа возвращаемого значения (подразумевается тип, указанный после ключевого слова *operator).*
• Процедура преобразования может описываться как *virtual.*
• Процедура преобразования наследуется.
Следующий пример иллюстрирует *процедуру преобразования* класса.

```
//////////////////////////////////////////////////////////////////////////////
// CONVERSN.CPP:  Процедура преобразования класса...     //
//////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
class Value
{
int val;
public:
operator int();
};
Value::operator int()
{
return val;
}
int main( void )
{
Value v;
//
// Следующая операция вызывает Value::operator int();
// vvvvv
printf( "Значение равно %d \n", int(v) );
return 0;
}
```

Друзья

Спецификаторы доступа класса позволяют указывать, могут ли функции вне определенного вами класса обращаться к его элементам. Может, однако, случиться, что вам потребуется обеспечить определенной функции или классу доступ к элементам вашего класса, специфицированным как *private* или *protected.* Язык C++ позволяет вам особо разрешить доступ к любым элементам класса другому классу или функции с помощью ключевого слова *friend.*

Дружественные классы

Вы можете разрешить элементам другого класса *(anotherClass)* полный доступ к элементам вашего класса *(myClass),* объявленным как *private* или *protected,* включив в определение вашего класса описание *friend.*

```
class myClass
{
friend class anotherClass;
 };
```

Дружественные функции

Вы можете разрешить обычной функции или функции-элементу другого класса полный доступ к элементам класса, объявленным *private* или *protected,* с помощью описания *friend* в вашем определении класса.

```
class myClass
{
friend void anotherClass::MemberFuncName(int);
friend void regularFuncName(double);
};
```

Правила относительно друзей

К *друзьям* и *дружественности* приложимы следующие правила:

• На описания *friend* не влияют спецификаторы *public, protected* или *private.*

• Описания *friend* не взаимны: если A объявляет B другом, то это не означает, что A является другом для B.

• *Дружественность* не наследуется: если A объявляет B другом, классы, производные от B, не будут автоматически получать доступ к элементам A.

• *Дружественность* не является переходным свойством: если A объявляет B другом, классы, производные от A, не будут автоматически признавать дружественность B.

Перегрузка функций-элементов

Функции-элементы класса могут быть перегружены; две или несколько функций-элементов могут иметь одно и то же имя, при условии, что списки их аргументов не совпадают. Компилятор следит за тем, чтобы вызывалась нужная функция, проверяя соответствие аргументов. Довольно обычной является перегрузка *конструкторов*. Например:

```
//////////////////////////////////////////////////////////////////////////////
//OVERLD.CPP: Перегруженные функции-элементы... //
//////////////////////////////////////////////////////////////////////////////
#include <time.h>
const int TIME_STR_LEN = 30;
class Time
{
char  timeStr[TIME_STR_LEN];
public:
Time();
Time( char *str );
};


int main( void )
{
Time T1;        // Вызывает Time::Time();
time_t t;
time( &t );
Time T2( ctime( &t ) );    // Вызывает Time::Time(char*);
// ...
return 0; }
```

Перегрузка операций

Язык C++ позволяет вам определять и применять к вашему классу обозначения операций. Эта особенность, называемая *перегрузкой операций,* дает вашему классу возможность вести себя подобно встроенному типу данных. Вы можете перегружать для ваших классов любые из следующих операций:

Таблица 6.4. Операции, допускающие перегрузку

+	-	*	/	%	^	&	|	~
!	=	<	>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=	<<=	==	!=
<=	>=	&&	||	+	--	->*	->	
()	[]							

Следующие операции не допускают перегрузку

. .* :: ?:

Правила

Функции-операции и перегрузка операций подчиняются следующим правилам:

• Приоритеты операций и правила ассоциации, принятые для встроенных типов данных, остаются неизменными при оценке выражений с перегруженными функциями-операциями.

• Функция-операция не может изменить поведение операции по отношению к встроенным типам данных.

• Функция-операция должна быть либо элементом класса, либо воспринимать один или несколько аргументов, имеющих тип класса.

• Функция-операция не может иметь аргументов по умолчанию.

• За исключением *operator=(),* функции-операции наследуются.

Примеры

Следующие примеры демонстрируют перегрузку операций / и *[].*

```
//////////////////////////////////////////////////////////////////
// OPEROVR1.CPP: Перегрузка операций... //
//////////////////////////////////////////////////////////////////
#include <string.h>
#include <stdio.h>
const int MAX_LEN =100;
//
// Класс для хранения имени...
//
class Name
{
char data[MAX_LEN];
public:
Name();
void SetData( char *newData ); int operator!();
};
```

```
//
// Конструктор: инициализация нулями
//
Name::Name()
{
memset( data, '\0', sizeof(data) );
}
//
// Сохранить специфицированное имя
//
void Name::SetData( char *newData )
{
strcpy( data, newData );
}
//
// Перегрузка операции !() : показывает, имеются ли данные
//
int Name::operator!()
{
return ( data[0] == '\0' );
}
int main( void )
{
Name name;
if ( !name )  // Неявный вызов operator!() printf(
"Данные еще не присвоены! \n" )
name.SetData( "Beau-Bassin" );
•// Теперь явный вызов функции operator!() if (
!name.operator!() ) printf( "Данные присвоены...\n" );
return 0;
}


////////////////////////////////////////////////////////////////////
//OPEROVR2.CPP: Перегрузка операций... //
////////////////////////////////////////////////////////////////////
#include <string.h>
#include <stdio.h>
const int MAXJ.EN = 100;
//
// Класс для хранения имени шпиона и ключа к шифру
//
class SecretInfo
{
int decoderKey;
char codeName[MAX_LEN];
public:
SecretInfo( char *_spyName, int _spyKey );
int operator []( const char *_spyName );
};
//
// Конструктор: записать информацию о шпионе...
```

```
//
SecretInfo::SecretInfo( char *_spyName, int _spyKey )
{
strcpy( codeName, _spyName );
decoderKey = spyKey;
}
//
// Операция индексации перегружается, чтобы использовать
строку
// в качестве индекса!!
//
int SecretInfo::operator []( const char *_spyName )
{
if ( !strcmp( codeName, _spyName ) )
return decoderKey; else
return 0;
}
int main( void )
{
SecretInfo agent( "Дж. Бонд", 7 );


char temp[MAX_LEN];
printf( "Введите ваше кодовое имя: " );
gets( temp );
if ( agent[temp] )    // Вызывает operator[](const *char);
printf( "Ваш ключ шифра %03d",
agent.operator[]( temp ) );
else
printf( "Сожалею, Бонд! Неправильное кодовое имя." );
return 0;
}
```

Статические элементы

Можно определить элемент данных или элемент-функцию класса как *static*. Статический элемент класса может рассматриваться как глобальная переменная или функция, доступная только в пределах области класса.

Статические элементы данных

Элемент данных, определенный как *static,* разделяется всеми представителями данного класса: существует только один экземпляр переменной, независимо от числа созданных представителей. На самом деле, память под статический элемент выделяется, даже если не существует никаких представителей класса. Класс со статическим элементом должен как *объявлять (описывать),* так и *определять* статический элемент данных.

• Объявление статического элемента данных:

```
class myClass
{
static int count;
}
```

• Определение статического элемента данных:

int *myClass::count* = 0;

Доступ к публичному статическому элементу

Хотя к статическому элементу данных, объявленому как *public*, можно обращаться с помощью конструкций

представитель_класса.элемент

или

указатель_на_представитель->элемент,

лучше использовать форму *имя_класса::элемент*, так как она показывает, что переменная является единственной для всего класса.

Статические элементы-функции

Статические элементы-функции класса не ассоциируются с отдельными представителями класса. Другими словами, при вызове им не передается указатель *this*. Из этого следует, что

- статическая функция-элемент может вызываться независимо от того, существует или нет какой-либо представитель класса;
- статическая функция-элемент может обращаться только к статическим элементам данных класса и вызывать только другие статические функции-элементы класса;
- статическая функция-элемент не может объявляться *как virtual*.

Константные объекты и константные элементы-функции

Можно создать представитель класса с модификатором *const*. Ключевое слово *const* информирует компилятор, что содержимое объекта не должно изменяться после инициализации. Чтобы предотвратить изменение значений элементов *константного* объекта, компилятор генерирует сообщение об ошибке, если объект используется с *не-константной* функцией-элементом.

Константная функция-элемент:

- Объявляется с ключевым словом *const,* которое следует за списком параметров.
- Не может изменять значение элементов данных класса.
- Не может вызывать не-константные функции-элементы класса.
- Может вызываться как для константных, так и не-константных объектов класса.

Следующий пример демонстрирует константные объекты и элементы-функции.

```
//////////////////////////////////////////////////////////////////////////////////////////////////
// CONSTOBJ.CPP: Константные объекты и элементы-функции        //
//////////////////////////////////////////////////////////////////////////////////////////////////
class Coord
{
int x, y;
public:
```

```
Coord(int _x,  int _y); void SetVal(int _x,  int _y);
void GetVal(int & x,  int &_y) const;
};
//
// Конструктор
//
Coord::Coord(int _x,  int _y)
{
x = _x;
y = _y;
}
//
// Установка значений x, y / не-константная функция
//
void Coord::SetValint _x, int y)
{
x = _x;
y = _y:
}


//
// Получает значения x, y / константная функция
//
void Coord::GetVal( int &_x, int &_y ) const
{
_x = x;
_y = y;
}
//
// main()
//
int main( void )
{
Coord   c1(10, 10);
const Coord c2(20, 20);
int x, y; c1.GetVal(x, y);
// Неверно: вызов не-константной функции
// с константным объектом
//c2.SetVal(x, y);
c2.6etVal(x, y);
return 0;
}
```

Наследование классов

Язык C++ позволяет классу *наследовать* элементы данных и элементы-функции одного или нескольких других классов. Другими словами, новый класс может получать атрибуты и поведение от уже существующего класса.

Новый класс называют *производным* классом. Класс, элементы которого наследуются производным классом, называется *базовым* классом. В свою очередь производный класс может служить базовым для другого класса.

Наследование дает вам возможность заключить (абстрагировать) некоторое общее или схожее поведение различных объектов в одном базовом классе. Несколько классов будут затем наследовать это поведение, являясь производными от базового.

Наследование также позволяет вам немного изменить поведение существующего класса. Производный класс может переопределить некоторые функции-элементы базового, наследуя, тем не менее, основной объем свойств и атрибутов базового класса.

Синтаксис наследования имеет следующий вид:

class Base {
// ... };
class *Derived* : *ключ_доступа Base [,ключ_доступа Base2, ...] {*
II ... };

Ключ_доступа не является обязательным и может быть *private, protected* или *public.* Если не указан, доступ принимается по умолчанию *private* для классов и *public* для структур.

Базовые классы: public, protected и private

При наследовании ключ доступа определяет уровень доступа к элементам базового класса внутри производного класса. В таблице 6.5 описаны возможные варианты доступа.

Таблица 6.5. Наследование: public, protected и private

Доступ в Наследование	*Доступ в базовом классе*	*производном классе*
public:	public	public (не меняется)
	protected	protected (не меняется)
	private	private (не меняется)
protected:	public	protected
	protected	protected (не меняется)
	private	private (не меняется)
private:	public	private
	protected	private
	private	private (не меняется)

В следующем примере

- B1 является публичным базовым классом для Derived.
- B2 является защищенным базовым классом для Derived.
- B3 является частным базовым классом для Derived.

```
//////////////////////////////////////////////////////////////////
// BASECLASS.CPP:   Базовые классы... //
//////////////////////////////////////////////////////////////////
class B1
{
// ...
};
class B2
```

```
{
// ...
};
```

```
class B3
{
// ...
};
class Derived : public B1, protected B2, private B3
{
// ...
};
```

Частное наследование и публичные базовые элементы

Если базовый класс наследуется как *private*, его публичные элементы будут являться *private* в производном классе. Однако вы можете выборочно сделать некоторые из элементов базового класса публичными в производном классе, указав их в секции *public* производного класса:

```
class Base
{
   public:
    void f1();
    void f2();
};

class Derived : private Base
{
   public:
    Base::f1;   // Делает void Base::f1() доступной
          // как public
};
```

Простое наследование

Простым наследованием называется случай, когда производный класс имеет всего один базовый класс. Следующий пример иллюстрирует простое наследование.

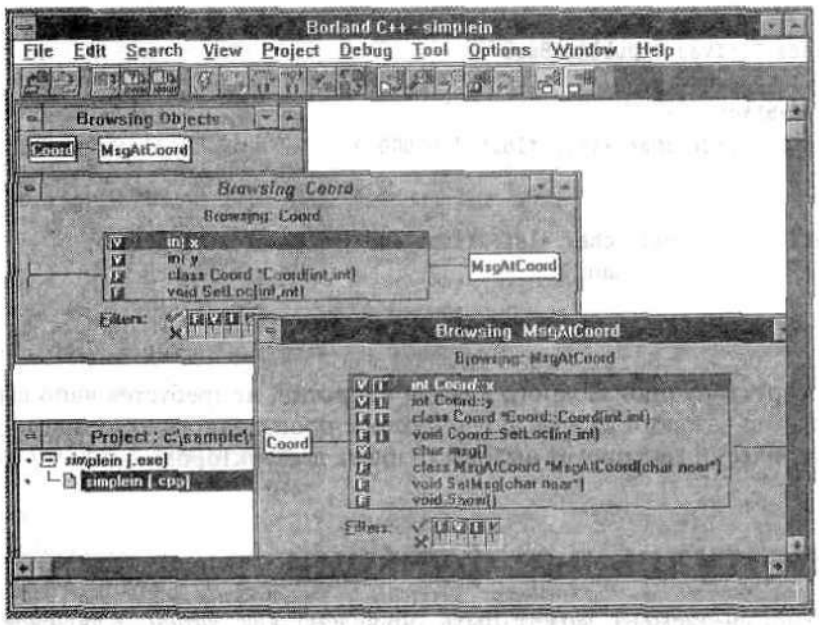
```
//////////////////////////////////////////////////////////////////////////////
// SIHPLEIN.CPP: Простое наследование...          //
//////////////////////////////////////////////////////////////////////////////
#include <stdio.h>
#include <conio.h>
#include <string.h>
const int MAX_LEN = 10;
//
// Coord: заключает в себе значения x и y.
//
```

```
class Coord
{
protected: int x, y; public:
Coord(int _x=0, int _y=0);
void SetLoc(int _x, int _y);
};
//
// Coord, конструктор (инициализирует координаты)
//
Coord::Coord(int _x, int _y)
{
SetLoc(_x, _y);
}
//
// Coord::SetLoc: меняет значения x и y
//
void Coord::SetLoc(int _x, int _y)
{

x = _x;
y = _y;
}
//
//MsgAtCoord: Содержит сообщение
//        Наследует от Coord...

//
class MsgAtCoord : public Coord
{
char msg[MAX_LEN]; public:
MsgAtCoord( char *_msg = "NO MSG" );
void SetMsg(char *_msg);
void Show();
};
//
// MsgAtCoord: конструктор
//
MsgAtCoord::MsgAtCoord( char *_msg )
{
SetMsg( _msg );
}
//
// MsgAtCoord::SetMsg, записывает сообщение
//
void MsgAtCoord::SetMsg(char *_msg)
{
strcpy( msg, jnsg );
}
//
// MsgAtCoord::Show, выводит сообщение в точке x, y
//
```

```
void  MsgAtCoord::Show()
{
gotoxy(x, y);
printf( msg );
}
//
// main: создает и использует MsgAtCoord
//
int main( void )


 {
MsgAtCoord greeting;
greeting.SetLoc( 10, 10 ); greeting.SetMsg( "Hello..." ); greeting.Show();
return 0;
}
```

Просмотр иерархии классов в IDE

В интегрированной среде программирования имеется инструмент для визуального представления иерархии классов *(Browser).* На рис. 6.1 показан возможный вариант отображения классов из примера простого наследования:



Конструкторы, деструкторы и наследование

Конструкторы не наследуются. Если конструктор базового класса требует спецификации одного или нескольких параметров, конструктор производного класса должен вызывать базовый конструктор, используя *список инициализации элементов*.

```
//////////////////////////////////////////////////////////////////////////////
// CTR_BASE.CPP:  Конструктор и наследование... //
//////////////////////////////////////////////////////////////////////////////
#include <string.h>
class Base
```

```
{
public:
Base(int, float);
};
class Derived :  public Base
{
public:
Derived( char *lst, float=1.00000 );
};
Derived::Derived( char *lst, float amt ) : Base( strlen(lst),
amt ) { }
```

Деструктору производного класса, напротив, не требуется явно вызывать деструктор базового класса. В деструкторе производного класса компилятор автоматически генерирует вызовы базовых деструкторов.

Виртуальные функции

Функция-элемент может быть объявлена как *virtual*. Ключевое слово *virtual* предписывает компилятору генерировать некоторую дополнительную информацию о функции. Если функция переопределяется в производном

классе и вызывается с указателем (или ссылкой) базового класса, ссылающимся на представитель производного класса, эта информация позволяет определить, какой из вариантов функции должен быть выбран: такой вызов будет адресован функции производного класса.

Следующий пример демонстрирует разницу между виртуальными и не-виртуальными функциями.

```
//////////////////////////////////////////////////////////////////////////
// VIRTUAL.CPP: Виртуальные функции-элементы... //
//////////////////////////////////////////////////////////////////////////
#include <stdio.h>
//
// Базовый класс с виртуальной
// и не-виртуальной функциями
//
class Base
{
public:
virtual void virt()
{
printf( "Hello from Base::virt \n" );
}
void nonVirt()
{
printf( "Hello from Base::nonVirt \n"
);
 }
};
//
// Производный класс: заменяет обе функции
// класса Base
//
class Derived : public Base
{
public:
void virt()
```

```
{
printf( "Hello from Derived::virt \n" );
}
void nonVirt()

{
printf( "Hello from Derived::nonVirt \n" ); } };
//
// Функция main
//
int main( void )
{
Base *bp = new Derived; // Базовый указатель, реально
// ссылающийся на производный
// объект
bp->virt();    // Вызов виртуальной функции bp->nonVirt();
// Вызов не-виртуальной функции
return 0;
}
```

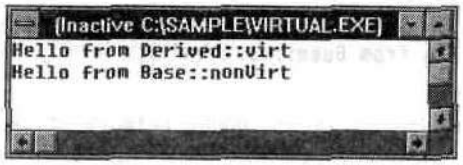


Рис. 6.2. *Результат выполнения VIRTUALCPP*

Результат выполнения приведенной программы показывает, что при обращении к виртуальной функции вызывается, как и должно быть, функция производного класса. Обращение к не-виртуальной функции *nonVirt,* однако, вызывает *Base::non Virt,* хотя объект на самом деле принадлежит к производному классу.

Виртуальный механизм требует применения указателей или ссылок

Чтобы использовать механизм виртуальных функций, вы должны применять указатели или ссылки. Только указатель (или ссылка) на класс может ссылаться на представитель производного класса.

Для виртуальных функций существуют следующие правила:

- Виртуальную функцию нельзя объявлять как *static.*
- Спецификатор *virtual* необязателен при переопределении функции в производном классе.
- Виртуальная функция должна быть определена, либо должна описываться как чистая.

Выключение виртуального механизма

Вы можете обойти механизм виртуальных функций, если укажете при вызове имя класса с операцией разрешения видимости.

```
//////////////////////////////////////////////////////////
// NO_VIRT.CPP: Обход виртуального механизма...         //
//////////////////////////////////////////////////////////
#include <stdio.h>

class Base
{
   public:
    virtual void virt()
    { printf( "Hello from Base::virt \n" ); }
};

class Derived : public Base
{
   public:
    void virt()
    { printf( "Hello from Derived::virt \n" ); }
};

int main( void )
{
   Base *bp = new Derived;     // Базовый указатель на
                               // производный объект
   bp->virt();                 // Использует виртуальный механизм
   bp->Base::virt();           // Обходит виртуальный механизм

   return 0;
}
```

Реализация

Виртуальные функции реализуются с использованием *таблицы переходов*. Ниже дается обзор основных моментов реализации виртуального механизма:

• Для каждого класса, содержащего виртуальные функции, компилятор строит таблицу адресов этих функций. Такую таблицу обычно называют *таблицей виртуальных методов* или *vtable*.

• Каждый представитель класса с виртуальными функциями содержит (скрытый) указатель на его *таблицу виртуальных методов*. Обычно его называют *указателем виртуальных методов, указателем виртуальной таблицы* или *vptr*.

• Компилятор автоматически вставляет в начало конструктора класса фрагмент кода, который инициализирует *указатель виртуальных методов* класса.

• Для любой данной иерархии классов адрес некоторой виртуальной функции имеет всегда одно и то же смещение в *таблицах виртуальных методов* каждого класса.

• Если вызывается виртуальная функция, код, генерированный компилятором, прежде всего находит *указатель виртуальной таблицы*. Затем код обращается к *таблице виртуальных методов* и извлекает из нее адрес виртуальной функции. Наконец, код производит косвенный вызов функции.

Полиморфизм и позднее связывание

Термины *полиморфизм* и *позднее (динамическое) связывание* обычно применяются, когда говорят о механизмах наследования и виртуальных функций языка C++. *Полиморфизмом* называют свойство базового класса, позволяющее ему обеспечивать взаимодействие с виртуальными функциями, которые могут по-разному определяться несколькими производными классами. Другими словами, это

способность указателей (или ссылок) на базовый класс *"принимать различные формы, характер или стиль"* (определение слова *полиморфный* у Вебстера) в контексте наследования и виртуальных механизмов.

Термин *позднее связывание* относится к тому обстоятельству, что компилятор не может определить заранее, какая функция должна вызываться на самом деле, если обращение к виртуальной функции использует указатель или ссылку на базовый класс. Хотя переменная и определяется как указатель на базовый класс, она в действительности может указывать на объект производного класса. Эта особенность виртуального механизма приводит к тому, что адрес функции может быть найден только во время исполнения программы.

Используйте в базовых классах виртуальный деструктор

Рассмотрим следующую ситуацию:

```
class Base
{
   public:
    ~Base()
    {
       // Освобождение ресурсов
    }
};

class Derived : public Base
{
   public:
    ~Derived()
    {
       // Освобождение ресурсов
    }
};

int main( void )
{
   Base *bp = new Derived;
   // ...
   delete bp;

   return 0;
}
```

Так как деструктор не является виртуальным, операция *delete* вызовет только *Base::~Base,* что может привести к потере ресурсов, используемых классом *Derived.* Если же сделать функцию *Base::~Base* виртуальной, вместо нее будет вызван нужный деструктор.

Множественное наследование

В C++ допускается *множественное наследование,* когда класс является производным от нескольких базовых классов. Это позволяет вам в одном производном классе сочетать поведение нескольких классов. Следующий пример иллюстрирует технику такого наследования. Класс *Coord* отслеживает значения координат *к/у,* класс *Message* хранит сообщение. Класс *MessageXY,* производный от

двух этих классов, наследует контроль как над координатами, так и над сообщением. Новый класс просто добавляет функцию-элемент *Show* для вывода сообщения в позиции x/y.

```
//////////////////////////////////////////////////// // MULTIIN.CPP:  Множественное наследование... //
////////////////////////////////////////////////////
#include <stdio.h> #include <conio.h> #include <string.h>
const int MAX_LEN = 10;
//
// Coord: заключает в себе значения x и y.
//
class Coord
{
protected: int x, y;
public:
Coord(int _x=0, int _y=0); void
SetLoc(int _x, int _y);
};
// Coord, конструктор (инициализирует координаты)
//
Coord::Coord(int _x,  int _y)
{
SetLoc(_x,  _y); }
//
// Coord::SetLoc: меняет значения x и y
//
void Coord::SetLoc(int _x, int _y)
{
x = _x;
y = _y;
}
//
// Класс Message: сохраняет строку
//
class Message
{
protected: char msg[MAX_LEN];
public:
void SetMsg(char *_msg)
{
strcpy( msg, _msg );
}
};
//
// Класс MessageXY: наследует от Coord и Message..
//
class MessageXY : public Coord, public Message
{
public:
void Show();
};
```

```
//
// MessageXY::Show, выводит сообщение в текущей позиции
//
void MessageXY::Show()
{
gotoxy(x, y);
printf( msg );
}
//
// main: создает и использует MessageXY
//
int main( void )
{
MessageXY greeting;
greeting.SetLoc( 10, 10 ); greeting.SetMsg( "Hello..." );
greeting.Show();
return 0;
}
```

Неоднозначность и разрешение видимости

В иерархии с множественным наследованием класс может косвенно наследовать два экземпляра базового класса.

```
////////////////////////////////////////////////////////////////////////////////
// AMBIG.CPP: Неоднозначность при сложном наследовании //
////////////////////////////////////////////////////////////////////////////////
class A
{
protected: int data; public: void func()


{
//...
}
};
class B : public A
{
// ...
};
class C : public A {
// ... };
class D : public B, public C
{
// ...
};
int main( void )
{
D d;
d.func(); d.data = 10;
return 0;
}
```

При компиляции приведенного примера для обращений *к /unc()* и *data* в *main* будут выданы следующие сообщения об ошибках:

```
Error ambig.cpp 34: Member is ambiguous: 'A::func' and 'A::func' in function main()
(Неоднозначный элемент:...)
Error ambig.cpp 35: Member is ambiguous: 'A::data' and 'A::data' in function main()
*** 2 errors in Compile ***
```

Вы можете разрешить конфликт, связанный с неоднозначностью, применив *операцию разрешения видимости*. Например, чтобы устранить приведенные выше сообщения об ошибках, замените *d.func()* на *d.B::func()* [или *d.C::func()},* а *d.data* замените на *d.B::data* [или *d.C::data].*

Виртуальный базовый класс

Хотя операция разрешения видимости и предоставляет обходной путь, позволяющий избежать неоднозначностей при сложном наследовании, вы тем не менее можете захотеть, чтобы ваш производный класс наследовал только один экземпляр некоторого базового класса. Этого можно достичь применением ключевого слова *virtual* при спецификации наследуемого класса. Следующий пример является модифицированным вариантом файла AMBIG.CPP, использующим класс A в качестве *виртуального базового класса.*

```
//////////////////////////////////////////////////////////////////////////////
// VIRTBASE.CPP:  Пример виртуального базового класса     //
//////////////////////////////////////////////////////////////////////////////
class A
{
protected:
int  data;
public:
void func()
{
// ...
}
};
class B : virtual public A  //A является виртуальным
// базовым классом
{
// ...
};
class C : virtual public A  //A является виртуальным
// базовым классом
{
//  ...
};


class D : public B, public C // D содержит только один
// экземпляр A {
// ... };
int main( void ) { D d;
d.func(); // d.data = 10; Элемент данных недоступен
return 0; }
```

Следующий список показывает последовательность шагов при конструировании производного класса:

1. Вызывается конструктор базового класса.
2. Вызываются конструкторы тех элементов, которые являются объектами какого-либо класса, в том порядке, в котором производный класс их объявляет.
3. Исполняется тело конструктора производного класса.

Деструкция производного класса выполняется в порядке, обратном конструированию:
1. Вызывается деструктор класса.
2. Вызываются деструкторы элементов, являющихся объектами классов.
3. Вызывается деструктор базового класса.

Абстрактные классы и чистые виртуальные функции

Абстрактный класс является классом, который может использоваться только в качестве базового для других классов. *Абстрактный класс* содержит одну или несколько *чистых виртуальных функций. Чистая виртуальная функция* может рассматриваться как встроенная функция, тело которой определено как *=0 (чистый спецификатор)*. Для чистой виртуальной функции не

нужно приводить действительное определение; предполагается, что она переопределяется в производных классах.
К абстрактным классам применимы следующие правила:
• Абстрактный класс не может использоваться в качестве типа аргумента функции или типа возвращаемого значения.
• Абстрактный класс нельзя использовать в явном преобразовании.
• Нельзя определить представитель абстрактного класса (локальную/глобальную переменную или элемент данных).
• Можно определять указатель или ссылку на абстрактный класс.
• Если класс, производный от абстрактного, не определяет все *чистые виртуальные* функции абстрактного класса, он также является *абстрактным*.

```
////////////////////////////////////////////////////////////////////////////
// ABSTRACT.CPP: Пример абстрактного класса... //
////////////////////////////////////////////////////////////////////////////
//
// Bird - абстрактный класс с чистой
//  виртуальной функцией Sing().
//
class Bird
{
public:
void virtual Sing() = 0; // Чистая виртуальная функция
};
//
// Eagle - также абстрактный класс, т.к. не определяет
// чистую виртуальную функцию из Bird
//
class Eagle : public Bird
{
};
//
// GoldenEagle - определяет чистую виртуальную функцию
// базового класса и, таким образом, может иметь
представителей
//
```

```
class GoldenEagle : public Eagle {
void Sing()
{ // ...
}  };
```

Заключение

Программа на C++ может и не использовать объектно-ориентированные возможности языка, представленные в этой главе. Однако могут быть случаи, когда определенный аспект языка предлагает именно то, что кажется идеальным средством описания и решения проблемы. В следующей главе вы увидите, как некоторые из представленных методов применяются в библиотеке входных/выходных потоков C++.

Глава 7
Классы потоков языка C++

Библиотека потоков C++ (библиотека *iostream)* предоставляет набор классов для управления вводом/выводом. Эти классы имеют несколько преимуществ по сравнению с традиционными средствами ввода/вывода (I/O):

• Надежность. Семейства функций *printf* и *scanf* не предусматривают никакой проверки типа. Компилятор не может сигнализировать о несоответствии спецификаций формата передаваемым аргументам. Механизм потоков C++ основывается на *перегрузке функций (операций),* что обеспечивает для каждого типа передаваемых данных вызов соответствующей функции.

• Расширяемость. Применение процедур C ограничено файловыми потоками (и некоторыми устройствами, доступ к которым возможен как к *предопределенным потокам).* Эти процедуры не допускают расширения. Классы C++, благодаря *полиморфизму,* позволяют одним и тем же процедурам работать с потоками различных типов. Например, тот же интерфейс, что используется стандартным вводом/выводом, применим к файловым и резидентным потокам. Вы можете также предусмотреть собственные перегруженные функции, которые позволят библиотеке потоков C++ работать с определяемыми вами типами.

• Простота и последовательность. Широкое использование перегруженных функций позволяет библиотеке потоков поддерживать единообразный интерфейс ввода/вывода. Такой интерфейс делает ваш код более разборчивым и способствует лучшему абстрагированию данных. Кроме того, применение в 1/O-классах перегруженных *операций* приводит к более простому и интуитивно понятному синтаксису.

В настоящей главе дается практический обзор библиотеки потоков C++ с примерами для часто применяемых ее функций.

Заголовочные файлы

Чтобы обеспечить своей программе доступ к библиотеке потоков C++, вам нужно включить в нее заголовочный файл *iostream.h;* вам также могут понадобиться файлы *fstream.h* (файловый ввод/вывод), *iomanip.h* (манипуляторы) и *strstream.h* (резидентные потоки).

Предопределенные объекты-потоки

Библиотека iostream имеет четыре предопределенных объекта-потока. Они ассоциированы со стандартным вводом и выводом. Эти объекты описаны в следующей таблице.

Таблица 7.1. Предопределенные потоки C++ *Имя Тип класса Описание*

cin istream_withassign	Ассоциируется со стандартным вводом (т.е. клавиатурой)
cout ostream_withassign	Ассоциируется со стандартным выводом (т.е. экраном)
cerr ostream_withassign	Ассоциируется со стандартным устройством ошибок (экраном) с небуферизованным выводом
clog ostream_withassign	Ассоциируется со стандартным

устройством ошибок (экраном) с
буферизованным выводом

Переадресация ввода и вывода

Вы можете переназначить имена *cin* или *cout* вашим собственным объектам-потокам. Такое назначение позволяет вашей программе легко переадресовать стандартный ввод *или* вывод. Например:

```
//////////////////////////////////////////////////////////
// REDIR.H Переадресация стандартных потоков...         //
//////////////////////////////////////////////////////////
#include <iostream.h>
#include <fstream.h>

const int MAX_LINE = 80;

//
// Переадресованный входной поток
//
ifstream ifs;

int main( int argc, char *argv[] )
{
   // Если указан аргумент...
   if ( argc > 1 )
   {
    // Попытка открыть файл
    ifs.open( argv[1] );

    // Если успешна, переадресовать ввод
    if ( ifs )
      cin = ifs;
   }

   // Запросить ввод...
   cout << "Введите строчку текста : ";

   // Прочитать данные со стандартного ввода
   char line[MAX_LINE];
   cin.getline( line, sizeof( line ) );

   // Показать введенные данные...
   cout << endl << "Вы ввели : " << line;

   return 0;
}
```

Операции помещения и извлечения

Библиотека потоков C++ предусматривает два основных класса для ввода и вывода: соответственно *istream* и *ostream*. Класс *ostream* использует для вывода операцию левого сдвига (<<). Если эта операция применяется объектам-потокам, ее называют операцией *помещения* (в поток). Следующий пример

выводит приветствие, применяя операцию *помещения* к предопределенному объекту *cout*.

```
#include <iostream.h>
int main( void )
{
cout << "Hello!  ";
return 0;
}
```

Класс *istream* использует для ввода операцию правого сдвига (>>). В таком контексте ее часто называют операцией *извлечения* (из потока). Следующий пример применяет операцию *извлечения* к предопределенному объекту *an,* чтобы прочитать строку с клавиатуры:

```
#include <iostream.h>
int main( void )
{
char  name[100];
cout << "Пожалуйста, введите ваше имя: ";
cin >> name;
```

```
cout << "Привет, "; cout << name; return 0; }
```

Перегрузка операций для встроенных типов

Классы *istream* и *ostream* перегружают соответственно операции *извлечения* и *помещения* для всех встроенных типов данных. Такая перегрузка позволяет использовать единообразный синтаксис для ввода и вывода символов, строк, целых и вещественных чисел. Следующий пример иллюстрирует тождественность синтаксиса при выводе переменных различных типов:

```
//////////////////////////////////////////////////////////////////////////////
// COUT.CPP: Иллюстрирует вывод встроенных типов //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
int main(void )
{
char    c = ' A'; signed char sc= 'B'; unsigned charuc= 'C';

int   i = Oxd; float  f = 1.7; double   d = 2.8;
cout << c; // Вызывает operator << (      char)
cout << sc; // Вызывает operator << ( signed char)
cout << uc; // Вызывает operator << (unsigned char)
cout << i; // Вызывает operator << ( int )
cout << f; // Вызывает operator << ( float )
cout << d; // Вызывает operator << ( double)
return  0;
}
```

Встроенные (inline) варианты функций

Различные функции библиотеки IOStream (в том числе некоторые перегруженные операции помещения/извлечения) в файле *iostream.h* определяются в варианте inline. Однако они недоступны, пока не определен макрос _BIG_INLINE. Если вам требуется более эффективный код и вы не возражаете против увеличения его размера, то можете определить этот макрос перед включением *iostream.h*.

Сцепленные вызовы операций

Перегруженные операции << и >> над классами *ostream* и *istream* возвращают ссылку на объект соответствующего типа. Это позволяет вам последовательно соединять несколько операций. Таким образом, приведенный выше пример можно упростить:

```
//////////////////////////////////////////////////////////////////////////////
// COUT.CPP: Иллюстрирует вывод встроенных типов //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
int main(void )
{
char c = 'A';
signed char   sc= 'B';
unsigned charuc= 'C';
int        i = 0xd;
float      f = 1.7;
double       d = 2.8;
cout   << c << sc << uc<< i << f << d;
return 0;
}
```

Расширения потоков для типов, определяемых пользователем

Вы легко можете расширить библиотеку iostream, чтобы приспособить операции *помещения/извлечения* к вашим собственным типам данных. Для этого вы должны определить две функции со следующими заголовками:

```
// Чтение данных из потока
istream& operator >> ( istream& is, имя_типа &varName );
// Запись данных в поток
ostream& operator << ( ostream& os, имя_типа &varName );
```

Следующий пример это иллюстрирует:

```
//////////////////////////////////////////////////////////////////////////////////////////
// IOTYPE.CPP:  I/O для типа,  определенного пользователем    //
//////////////////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <assert.h>
struct NewType
{
int x;
int y;
};
//
// Чтение: предполагаемый формат '(##.»#)'
//
istream& operator >> ( istream &is, NewType &nt )
{
char c;
```

```
cin >> c;
assert( c = ' (' );
cin >> nt.x;


cin >> c;
assert( c = ',' );
cin » nt.y;
cin >> c;
assert(c==' ) ');
return is; }
//
// Запись: формат '(##,##)'
//
ostream& operator << ( ostream &os, NewType &nt )
{
OS << ' ( '
<< nt.x
« ' , '
<< nt.y << ')';
return os;
}
int main( void )
{
cout << "Введите два числа: ";
int ix; cin >> ix;
int iy; cin >> iy;
NewType nt; nt.x = ix; nt.y = iy;
cout << "Введенное значение: " << nt << endl;
return 0;
}
```

Определяйте операции помещения и извлечения как дружественные

При использовании классов обычно принято объявлять операции извлечения и помещения друзьями вашего класса. Такое объявление обеспечивает операции доступ к частным элементам данных при форматировании вывода.

```
//////////////////////////////////////////////////////////////
// IO_TYPE.CPP: I/O для определенного пользователем типа    //
//////////////////////////////////////////////////////////////
#include <iostream.h>

class TPiece
{
    // ...(Частные данные)

  public:
    //
    // ...
    //
    friend istream& operator >> (istream&, TPiece& );
    friend ostream& operator << (ostream&, const TPiece& );
};

//
// Поддержка выходного потока
//
ostream& operator << ( ostream &os, const TPiece &p )
{
   //
   // Может использовать для форматирования
   // частные данные...
   //
   return os;
}

//
// и т.д.
//
```

Форматирование

Библиотека потоков C++ предусматривает три способа управления форматом выходных данных: вызов *форматирующих функций-элементов,* использование *флагов* и применение *манипуляторов*. Следующий раздел описывает каждый из методов, иллюстрируя их примерами.

Форматирующие функции-элементы

Функции для форматирования, имеющиеся в классе *ios,* перегружены, чтобы обеспечить возможность как чтения, так и установки управляющего атрибута. Часто для атрибутов, которыми можно управлять с помощью функций, библиотека потоков C++ предусматривает также *манипуляторы.* Ниже описываются атрибуты, для управления которыми в классе *ios* имеются функции-элементы.

Ширина поля

Для чтения и установки ширины поля потока в классе *ios* имеется функция *width,* показанная в таблице 7.2.

Таблица 7.2. Метод *width* класса *ios Функция Описание*

int ios::width(); Возвращает текущее значение внутренней

переменной ширины поля потока
int ios::width(int); Устанавливает значение внутренней переменной ширины поля
Дополнительная информация:
• Применяемый при вводе, метод *width* может быть использован для задания максимального числа читаемых символов.
• Применяемый при выводе, метод задает минимальную ширину поля.
• Если ширина поля меньше заданной, выход дополняется символами *fill*.

• Если выходное поле больше указанного, значение *width* игнорируется.
• По умолчанию значение *width* равно 0 (выход не дополняется и не обрезается).
• *width* обнуляется после каждого помещения данных в поток.
• См. описание родственного манипулятора *setw*.
Следующий пример показывает, как использовать *width* для ограничения числа прочитанных при вводе символов.

```
////////////////////////////////////////////////////////////////////////////////
// WIDTH1.CPP:  Использование ios::width при выводе       //
////////////////////////////////////////////////////////////////////////////////
#include <iostream.h> const int MAX_LEN = 10;
int main( void ) {
char name[MAX_LEN];
// Запросить имя пользователя cout  << "Пожалуйста, введите имя "
<< "(max " << MAX_LEN-1 // -1 для '\0'
<< " символов) : ";
// Установить максимальную ширину cin.width( MAX_LEN );
// Прочитать имя cin >> name;
// Приветствие
cout << "Привет, " << name << '!';
return 0; }
```

Следующий пример использует *width* для выравнивания правого поля при выводе чисел.

```
////////////////////////////////////////////////////////////////////////////////
// WIDTH2.CPP: Применение ios::width при выводе    //
////////////////////////////////////////////////////////////////////////////////
#include  <iostream.h>
const int FLD_WIDTH =10;
int main( void )
{
int x1 = 2867;
int y1 = 20051;
int z1 = 017;
cout.width(FLD_WIDTH); cout << x1 << '\n';
cout.width(FLD_WIDTH); cout << y1 << '\n';
cout.width(FLD_WIDTH); cout << z1 << '\n';
return 0;
}
```

Заполняющий символ

Для чтения или изменения текущего заполняющего символа можно применять функции *ios::fill* (таблица 7.3).

Таблица 7.3. Метод *fill* класса *ios Функция Описание char ios::flll();*
Возвращает текущий символ заполнения
char ios::fill(char); Устанавливает внутренний заполняющий символ потока и возвращает его предыдущее значение

Дополнительная информация:
• Символом заполнения по умолчанию является *пробел.*
• См. родственный манипулятор *setfill.*
Пример использует метод *fill* для дополнения поля нулями при выводе чисел:

```
//////////////////////////////////////////////////////
// FILL.CPP: Применение ios::fill... //
//////////////////////////////////////////////////////
#include    <iostream.h>
// Константа ширины поля
const int    FLD_WIDTH =10;
// Заполняющий символ
const char FILL_CHAR = '0';
int maln( void )
{
int x1 = 2867;
int y1 = 20051;
cout.fill(FILL_CHAR);
cout.width(FLD_WIDTH);
cout << x1 << '\n';
cout.width(FLD_WIDTH);
cout << y1 << '\n';
return 0;
}
```

Число цифр (точность) вещественных чисел

Функции *ios::precision* могут применяться при выводе чисел с плавающей точкой, позволяя читать или устанавливать текущее число значащих цифр (таблица 7.4).
Таблица 7.4. Метод *precision* класса *ios*

Функция	*Описание*
int ios::precision(int);	Устанавливает внутреннюю переменную точности вещественных чисел потока и возвращает прежнее значение
int ios::precision();	Возвращает текущее значение точности

Дополнительная информация:
• По умолчанию точность равна 6 цифрам.
• Если установлен флаг *scientific* или *fixed, precision* задает число цифр, выводимых после десятичной точки.
• Если не установлен ни один из флагов *scientific или fixed, precision* задает общее число значащих цифр.
• См. родственный манипулятор *setprecision*. Пример иллюстрирует применение *precision*.

```
//////////////////////////////////////////////////////////////////////
// PRECISN.CPP: Применение ios::precision... //
//////////////////////////////////////////////////////////////////////
#include    <iostream.h>
```

```
int main( void ) {
float f = 3456.141592;
double d = 50.2345639101;
cout.precision( 4 );
cout << d << '\n';    // Выводит '50.23'


cout << f << '\n';      // Выводит '3456'
cout.precision( 3 );
cout << f << '\n';      // Выводит '3.46e+3'
// Установить флаг fixed
cout.setf( ios::fixed, ios::floatfield );
cout << f << '\n';      // Выводит '34563.142
return 0;
}
```

Флаги форматирования

В потоках C++ имеются *флаги формата*. Они задают, каким образом форматируется ввод и вывод. Флаги являются *битовыми полями,* хранящимися в переменной типа *long.* В таблице 7.5 показаны функции, предусмотренные в классе *ios* для управления флагами формата.

Таблица 7.5. Методы управления форматирующими флагами потока

Функция — *Описание*

long ios::flags(); Возвращает текущие флаги потока

long ios::flags(long); Присваивает флагам значение, сообщаемое параметром, и возвращает прежнее значение флагов

long ios::setf(long, long); Присваивает флагам, биты которых установлены во втором параметре, значения соответствующих бит первого параметра. Возвращает прежнее значение всех флагов.

long ios::setf(long); Устанавливает флаги, биты которых установлены в параметре; возвращает прежнее значение всех флагов.

long ios::unsetf(long); Сбрасывает флаги, биты которых установлены в параметре; возвращает прежнее значение всех флагов.

Следующая таблица описывает флаги форматирования.

Таблица 7.6. Флаги формата *ios*

Флаг	*Умолчание*	*Описание*
ios::skipws	X	Если установлен, при вводе игнорируются предшествующие пробелы или эквивалентные им символы.
ios::left		Если установлен, данные при выводе выравниваются по левой границе поля.
ios::right	X	Если установлен или если сброшены ios::left и ios::internal, данные при выводе выравниваются по правой границе поля.
ios::internal		Если установлен, знак числа выводится с левого края поля, а число выравнивается по правому краю. Промежуток заполняется символами *fill.*
ios::dec	X	Если установлен, числа выводятся по основанию 10 (как десятичные).
ios::oct		Если установлен, числа выводятся по основанию 8 (как восьмиричные).
ios.hex		Если установлен, числа выводятся по основанию 16 (как шестнадцатиричные).

ios::showbase	Если установлен, при выводе чисел добавляется индикатор основания ("Ox" для 16-тиричных и "0" для восьмиричных).
ios::showpoint	Если установлен, при выводе чисел типа *float, double* и *long double* показывается десятичная точка.
ios::uppercase	Если установлен, буквы от A до F в 16-тиричных числах выводятся в верхнем регистре. Экспонента E в научной нотации чисел также выводится в верхнем регистре.
ios::showpos	Если установлен, выводится знак + для положительных значений.

Таблица 7.6. Флаги формата *ios*

Флаг	*Умолчание*	*Описание*
ios:scientific		Если установлен, вещественные числа выводятся в научной нотации (т.е. n.xxxEy, 1.2345e2).
ios::fixed		Если установлен, вещественные числа выводятся в нотации с фиксированной точкой (т.е. nnn.ddd, 123.45).
ios::unitbuf		Если установлен, буфер потока опорожняется после каждой операции помещения.
ios::stdio		Если установлен, потоки *stdout* и *strerr* опорожняются после каждой операции помещения.

Дополнительная информация:

• ios::left, ios.right и ios.internal взаимно исключают друг друга. В каждый момент времени может быть установлен только один из флагов.

• ios::dec, ios::oct и ios.hex взаимно исключают друг друга. В каждый момент времени может быть установлен только один из флагов.

• При модификации основания можно использовать константу *ios::basefleld* в качестве второго параметра функции *setf*.

• При задании способа выравнивания можно использовать константу *ios::ajustfield* в качестве второго параметра в *setf*.

• При задании нотации вещественных чисел можно использовать константу *ios:floatfteld* в качестве второго параметра в *setf*.

Следующие примеры демонстрируют некоторые из флагов.

```
////////////////////////////////////////////////////////////////////////////////
// FLAGS1.CPP: Применение флагов формата iostream...    //
////////////////////////////////////////////////////////////////////////////////
#include    <iost ream.h>
#include    <limits.n>
int main( void )

{

int i=0;
// Попросите ввести число... cout << "Введите целое с
необязательными " "пробелами впереди: ";
// При чтении пробелы будут пропущены cin >> i;
// Показать число...
```

```
cout << "Вы ввели " << i << endl;
// Удалить оставшиеся в потоке символы cin.ignore( INT.MAX,
'\n' ) ;
// Попросите снова ввести число... cout << "Введите целое с
необязательными " "пробелами впереди: ";
// Сбросить флаг skipws cin.unsetf( ios::skipws );
// Эта операция не будет игнорировать пробелы cin >> i;
// Проверить, имелись ли пробелы (или просто недопустимые //
символы) и информировать пользователя... cin.good() ? (cout <<
"Вы ввели " << i     << end1): (cout << "Неправильный ввод
..." << endl);
return 0;
}
```

```
//////////////////////////////////////////////////////////////////////////////
// FLAGS2.CPP: Применение флагов формата iostreara... //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>


{
int x = 1678;
// Показать значение
cout << "Значение x = " << x << '\n';
// Сохранить значения флагов long savedFlags = cout.flags;
// Установить основание 16 с индикацией cout.setf( ios::showcase | ios::hex
);
// Вывести значение снова
cout << "Значение x = " << x << '\n';
return 0;
}
//////////////////////////////////////////////////////////////////////////////
// FLAGS3.CPP: Применение флагов формата iostream... //
//////////////////////////////////////////////////////////////////////////////
#include  <iostream.h>
int main( void ) {
float  f = 2.3456789e6;
double d = 3.Oe9;
// Вывести значения
cout << "Значение f = " << f << '\n';
cout << "Значение d = " << d << '\n';
// Выводить знак + для положительных значений cout.setf( ios::showpos );
// Вывести значения снова
cout << "Значение f = " << f << '\n';
cout << "Значение d = " << d << '\n';
return 0;
}
```

Манипуляторы

Манипуляторы являются функциями, которые можно включать в цепочку последовательных операций помещения и извлечения. Это удобный способ

управления флагами потока. Однако применение манипуляторов не ограничивается модификациями формата ввода/вывода. За исключением *setw,* изменения, внесенные манипуляторами, сохраняются до следующей установки.

Простые манипуляторы

Манипуляторы, не требующие указания аргументов, называются *простыми*. Предопределенные простые манипуляторы описаны в таблице 7.7.

Таблица 7.7. Простые манипуляторы *Манипулятор Описание*

endl Помещает в выходной поток символ новой строки (\n) и вызывает манипулятор *flush*

ends Помещает в выходной поток нулевой символ (\0). Обычно используется для указания конца строки

flush Принудительно записывает все выходные данные на соответствующие физические устройства

dec Устанавливает основание 10 (см. ios::dec)

hex Устанавливает основание 16 (см. ios::hex)

oct Устанавливает основание 8 (см. ios::oct)

ws Заставляет игнорировать при вводе ведущие пробельные символы (см. ios::skipws)

Следующий пример иллюстрирует использование простых манипуляторов.

```
//////////////////////////////////////////////////////////////
// MANIP1.CPP: Применение простых //
// предопределенных манипуляторов //
//////////////////////////////////////////////////////////////


#include <iostream.h>
#include <iomanip.h>
int main( void )
{
int i;
// Запросить ввод
cout << "Пожалуйста, введите число: ";
// Извлечь число cin >> i;
// Проверить корректность введенного
if ( !cin )
{
// Информировать об ошибке
cout << "Ошибочный ввод..." << end1;
}
else
{
// Применить манипуляторы hex, oct и dec
//
cout << "Hex: " << hex << i << end1 << "Oct: " << oct << i << endl << "Dec: "
<< dec << i << endl;
}
return 0;
}
```

Параметризованные манипуляторы

Параметризованные манипуляторы требуют спецификации аргументов. Предопределенные параметризованные манипуляторы описаны в таблице 7.8.

Таблица 7.8. Параметризованные манипуляторы

Манипулятор	*Описание*
setbase(int _b);	Задает основание преобразования в соответствии с указанным параметром: Параметр Основание 0 Основание по умолчанию: • При вводе используется 10, если числа не восьмиричные или шестнадцатиричные. • При выводе используется 10. 8 Ввод/вывод используют 8. 10 Ввод/вывод используют 10. 16 Ввод/вывод используют 16.
resetiosflags(long _b);	Сбрасывает флаги, биты которых установлены в переданном параметре.
setiosflags(long _b);	Устанавливает флаги, биты которых установлены в переданном параметре.
setfflll(int _f);	Задает заполняющий символ.
setprecision(int _n);	Задает значение внутренней переменной точности вещественных чисел.
setw(int _w);	Задает значение внутренней переменной ширины поля: • Для входного потока задает максимальное число символов, которые должны быть прочитаны. • Для выходного потока указывает минимальную ширину поля. • Если ширина поля меньше указанной, выходной поток дополняется символами *fill*. • Если ширина поля больше указанной, спецификация ширины игнорируется. • Ширина поля устанавливается на 0 после каждой операции помещения.

Следующий пример служит иллюстрацией параметризованных манипуляторов.

```
////////////////////////////////////////////////////////////////////////////////
// MANIP2.CPP: Параметризованные манипуляторы… //
////////////////////////////////////////////////////////////////////////////////
#include <iostream.h> #include <iomanip.h>
int main( void )
{
double dbls[] = {
1.245, -12.99133, 134.007804, -2.345, 0.000003
};
cout << setfill( '.' )
<< setprecision( 4 )
<< setiosflags( ios::showpoint )
```

```
ios::fixed   |
ios::right   );
for( int i=0; i<sizeof(dbls)/sizeof(dbls[0]); i++ ) cout << "Результат" <<
setw(20) << dbls[i] << endl;
return 0;
 }
```

Ошибки потоков

Все объекты-потоки происходят от класса *ios* и наследуют элемент данных *state.* Этот элемент представляет состояние потока в виде битового множества. Флаги, индицирующие возможные состояния, перечисляются классом *ios,* который определяется в файле-заголовке *iostream.h:*

```
class _EXPCLASS ios { public:
// stream status bits
enum io_state{
goodbit    = 0x00,       // no bit set:  all is OK
eofbit     = 0x01,       // at end of file
failbit   = 0x02,       // last I/O operation
failed badbit     = 0x04, // invalid operation attempted
hardfail = 0x08   // unrecoverable error
};
// …
// …
// …
};
```

Биты состояния описаны в таблице 7.9.

Таблица 7.9. Ошибочные состояния потока

Перечислитель	*Описание*
ios::goodbit	Все в порядке.
ios::eofbit	Показывает, что был достигнут конец файла.
ios::failbit	Индицирует ошибку форматирования или преобразования: например, прочитанные данные имеют несоответствующий формат. Использование потока может продолжаться (после того, как бит будет сброшен).
ios::badbit	Сигнализирует о серьезной ошибке, обычно относящейся к буферным операциям: например, ошибке при извлечении символов, или попытке поиска, выходящей за пределы файла, или ошибке при записи символов в буфер потока. Потоком, скорее всего, больше нельзя пользоваться.
ios::hardfail	Сигнализирует о неисправимой ошибке, обычно связанной с неисправностью оборудования.

Опрос и установка состояния потока

Существуют различные функции и операции, позволяющие вам читать состояние потока, а также функции для установки или очистки состояния. В таблице 7.10 дано их краткое описание.

Таблица 7.10. Методы опроса состояния потока

Метод	*Описание*
int rdstate();	Возвращает текущее состояние.
int eof();	Возвращает ненулевое значение, если установлен флаг ios::eofbit.
int fail();	Возвращает ненулевое значение, если установлен один из флагов ios::failbit, ios:;badbit или ios::hardfail.
int bad();	Возвращает ненулевое значение, если установлен один из флагов ios::badbit или ios::hardfail.
int good();	Возвращает ненулевое значение, если сброшены все биты ошибок.
void clear(int=0);	Если параметр равен 0 (по умолчанию), все биты очищаются. В противном случае параметр принимается в качестве состояния ошибки.
operator void*();	Возвращает нулевой указатель, если установлен один из битов ios::failbit, ios::badbit или ios::hardfail (так же, как MO).
int operator!();	Возвращает ненулевое значение, если установлен один из битов ios::failbit, ios::badbit или ios::hardfail (так же, как fail()).

Применение ios::operator void*()

Операция *void*()* неявно вызывается всякий раз, когда вы сравниваете поток с нулем. Таким образом, вы можете писать выражения такого вида:

```
while( strmObj )
{
   // С потоком все в порядке; можно производить I/O
}
```

Обычные действия над состоянием потока

В таблице 7.11 приведены распространенные операции, которые вы можете производить с флагами состояния потока.

Таблица 7.11. Операции с флагами io_state

Действие	*Пример*
Проверить, установлен *ли flag:*	if(strm.rdstate() & *ios::flag)*
Сбросить flag	strm.clear(rdstate() & ~*ios::flag)*
Установить *flag:*	strm.clear(rdstate()|*ios'.'flag)*
Установить *flag*	strm.clear(ios: *:flag)* (сбрасывает другие флаги)
Сбросить все флаги:	strm.clear()

Файловый ввод/вывод с применением потоков C++

Библиотека C++ содержит три специализированных класса для файлового ввода/вывода. Это следующие классы:

ifstream: Для операций с входным дисковым файлом.

ofstream: Для операций с выходным дисковым файлом.

/stream: Для входных и выходных операций с файлом.

Эти классы являются производными соответственно от *istream, ostream* и *iostream*. Таким образом, они наследуют все их функциональные особенности, описанные выше (перегруженные операции << и >> для встроенных типов, манипуляторы, флаги формата, состояния потока и т.д.).

Конструкторы файловых потоков

Для каждого из трех классов файловых потоков предусмотрено четыре конструктора. Они позволяют вам делать следующее:

• Конструировать объект, не открывая файла:

```
ifstream(); ofstream(); fstream;
```

• Конструировать объект, открыть файл и прикрепить объект к файлу:

```
ifstream(   const char *name, int omode = ios::in, int prot   = filebuf::openprot );
ofstream(   const char *name, int omode = ios::out, int prot   = filebuf::openprot );
fstream( const char *name, int omode, int prot   = filebuf::openprot );
```

• Конструировать объект и прикрепить его к уже открытому файлу; указывается дескриптор файла:

```
ifstream( int f ); ofstream( int f ); fstream( int f );
```

• Конструировать объект, ассоциированный с указанным буфером; объект прикрепляется к уже открытому файлу; специфицируется дескриптор файла:

```
ifstream( int f, char *b, int len );
ofstream( int f, char *b, int len );
fstream ( int f, char *b, int len );
```

Открытие файла

Чтобы открыть файл, можно использовать конструкторы *ifstream, ofstream* или *fstream*. Следующий пример показывает конструкторы *ifstream* и *ofstream*.

```
////////////////////////////////////////////////////////////////////////////////
// FILEUPPR.CPP: Применение ifstream и ofstream...   //
////////////////////////////////////////////////////////////////////////////////
#include <iostreara.h>
#include <fstream.h>
#include <stdlib.h>
#include <ctype.h>
int main( void )
{
// Запросить имя входного файла
cout << "Введите имя входного файла: ";
// Прочитать имя char fname[_MAX_PATH]; cin >> fname;
// Открыть входной файл ifstream ifs( fname );
// Проверить поток...
if ( !ifs )
{
// Завершение в случае ошибки...
cout << "Невозможно открыть файл...";
return 0;
}
// Запросить имя выходного файла
cout << "Введите имя выходного файла: ";
// Прочитать имя cin >> fname;
```

```
// Попытка открыть выходной файл ofstream ofs( fname );
// Проверить поток... if ( !ofs ) { // Завершение в случае ошибки…


• cout << "Невозможно открыть файл...";
return 0;
}
char c;
// Пока не произойдет ошибки
while( ifs && ofs )
{
// Прочитать символ
ifs.get( c );
// Перевести в верхний регистр c = toupper( c );
// Записать в вых. поток ofs.put( c );
// Показать, что мы еще живы…. cout << '.';
}
// Задача выполнена... cout << endl
<< "Выходной файл является копией входного "
<< "в верхнем регистре..."
<< endl;
return 0;
}
```

Вы можете также открыть файл, применив метод *open.* Следующий пример использует функцию *open* потока *fstream.*

```
//////////////////////////////////////////////////////////////////////////////
// OPEN.CPP: Открытие файла с помощью потока C++     //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>


int main( void )
{
// Неоткрытый объект-поток
fstream fin;
// Запросить имя файла... cout << "Введите имя файла: ";
// Прочитать имя char name[_MAX_PATH]; cin >> name;
// Открыть входной файл fin.open( name, ios::in );
// Проверить состояние потока
if ( fin )
{
// Информировать об успехе
cout << "Файл открыт успешно." <<endl;
*
// Закрыть файл
fin.close();
}
else
```

```
{
// Ошибка…
cout << "Невозможно открыть файл " << name
<< endl;
}
return 0;
}
```

Параметры, требуемые функцией *open,* имеют тот же смысл, что и параметры конструкторов потока.

Режимы доступа

При открытии файла вы можете специфицировать параметр *режима доступа,* чтобы указать, каким образом файл должен открываться. Параметр можно составить из перечислителей битовых масок, определяемых классом *ios:*

```
// stream operation mode
enum open_mode    {
in   = 0x01, // open for reading
out = 0x02, // open for writing
ate = 0x04, // seek to eof upon original open
app = 0x08, // append mode:  all additions at eof
trunc         = 0x10,  // truncate file if already exists
nocreate   = 0x20, // open fails if file doesn't exist
noreplase = 0x40,  // open fails if file already exists
binary       = 0x80   // binary (not text) file
};
```

Параметр *mode* в конструкторах и методах *open* классов *ifstream* и *ofstream* имеет значения по умолчанию. Это соответственно *ios::in* и *ios::out.*

Примечание:

- ios::app подразумевает ios::out.
- ios::out подразумевает ios::trunc, если не специфицированы также ios::in, ios::ate или ios::app.

Применение различных режимов открытия

Следующий пример иллюстрирует использование параметра *mode:*

```
//////////////////////////////////////////////////////////////////////////////
// MODE.CPP:  Использование режимов доступа… //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <fstream.h>


const int MAX_LEN = 80;
const char fname[] = "NEWFILE";
int main(void )
```

```
{
// Создать новый файл, если только он уже не существует!
ofstream ofs( fnatne, ios: :out|ios: :noreplace );
// Проверить состояние потока
if ( !ofs )
{
// Сообщить об ошибке и завершить программу cout << "Ошибка! " <<
fname << "уже существует." << endl;
return 0;
}
else
{
// Записать строку в новый файл
ofs << "Привет, я - новый файл..."
// Закрыть файл ofs.close();
// Определить новый объект
fstream fs;
// Открыть файл и установить на EOF
fs.open( fname, ios::out|ios::ate );
// Дополнить сообщение
fs << " к которому сделано добавление!";
// Закрыть файл
fs.close;
// Снова открыть как входной
fstream ifs( fname );


// Проверить состояние потока
if ( ifs )
{
cout << "И старый файл сказал: " << endl;
// Высказывание старого файла...
char  line[MAX_LEN];
ifs.getline( line, sizeof(line) );
cout << line; )
else {
// Сообщение об ошибке cout << "Ошибка при повторном открытии " <<
fname << endl; } }
return 0; }
```

Замена буфера потока

Можно управлять буферизацией потока с помощью метода *setbuf.* Эта функция ассоциирует с потоком указанный буфер:

```
void setbuf( char *p,  int len );
```

p: адрес буфера len: длина буфера.

Закрытие файла

В классах файловых потоков имеется метод *close,* который:

- Опорожняет поток
- Закрывает закрепленный за потоком файл.

Примечание:

- Если при попытке закрыть файл происходит ошибка, устанавливается флаг *failbit* состояния потока.
- Предполагается, что деструктор файлового объекта (или его базового класса) автоматически закрывает файл.

Неформатируемый ввод/вывод

Библиотека IOStream предусматривает различные функции для неинтерпретирующего ввода и вывода. Эти функции позволяют вам читать и записывать байты данных без модификации и часто применяются при работе с бинарными (не текстовыми) файлами. В данном разделе рассматривается ввод/вывод бинарных файлов и описываются связанные с ним функции неформатируемого ввода/вывода.

Бинарный ввод/вывод файлов

В бинарном режиме данные при вводе/выводе не интерпретируются. Байты читаются и записываются без какой-либо модификации. Чтобы открыть файл в бинарном режиме, включите флаг *ios::binary в* параметр *open_mode,* передаваемый конструктору потока или функции *open.* В следующем примере файл открывается в текстовом и бинарном режимах.

```
//////////////////////////////////////////////////////////////////////////////
// BIN TXT.CPP:  Открытие в режимах text и binary //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
int main( void ) {
cout << "Введите имя входного файла: ";


char name[_MAX_PATH]; cin >> name;
ifstream ifbin( name, ios::in|ios::binary );
ifstream iftxt( name, ios::in );
if ( !iftxt || !ifbin )
{
cout << "Ошибка при открытии файла "
<< name
<< endl;
return 0;
}
int countInBin = 0; do { if ( ifbin.get() != EOF )
countInBin++; } while( ifbin );
cout << "Число символов, прочитанное в режиме binary = " << countInBin
<< endl;
int countInTxt = 0;
do
{
if ( iftxt.get() != EOF )
countInTxt++;
```

```
} while( iftxt );
cout << "Число символов, прочитанное в режиме text  = "
<< countInTxt
<< endl;
return 0;
}
```

Текстовый и бинарный режимы

Когда вы открываете файл в текстовом режиме (т.е. не специфицируете ios::binary), происходит следующее:

- Каждая пара символов возврат каретки/перевод строки при вводе преобразуется в единственный символ перевода строки ('\n').
- Каждый перевод строки при выводе преобразуется в пару возврат каретки/перевод строки.

Смотри пример BIN_TXT.CPP

Чтение сырых данных

Функция *read* позволяет вам извлекать из потока указанное число символов, записывая их в буфер:

```
istream& ist ream::read( char* p, int len);
istream& istream::read(   signed char* p, int len);
istream& istream::read(unsigned char* p, int len);
```

p: Буфер для записи прочитанных символов

len: Максимальное число символов, которые должны быть прочитаны.

Запись сырых данных

Функция *write* позволяет вам поместить в поток указанное число символов из буфера:

```
ostream& ostream: ;write(const                char* s, int n);
ostream& ostream: :write(const     signed char* s, int n);
ostream& ostream: :write(const unsigned char* s, int n);
```

s: Буфер-источник

n: Число символов, которые должны быть записаны.

Применение функций *read* и *write* при бинарных операциях демонстрирует следующий пример.

```
//////////////////////////////////////////////////////////////////////////////////////////////
// BIN_IO.CPP: Сопоставление бинарного и текстового I/O         //
//////////////////////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <fstream.h>
int       1   = 12345; long      1   = 98765432; float    f   = 4.536271; double
d   = 2.4e12; char   msg[]= "Hello";
const char bFname[] = "_IO.BIN"; const char tFname[] = "_IO.TXT";
int main( void )
{
//
```

```
// Сначала произвести операции в текстовом режиме
//
ofstream ofs( tFname );
if ( ofs )
{
ofs << i << '\f
<< l << '\t'
<< f << '\t'
<< d << '\t'
<< msg
<< endl;
ofs.close();
}
ifstream ifs( tFname );
if ( ifs )
{


ifs >> i >> 1 >> f >> d >> msg;
ifs.close();
}
//
// Теперь использовать бинарный режим…
//
ofs.open( bName, ios::out|ios::binary );
if ( ofs )
{
ofs.write( (char*)&i, sizeof(i) );
ofs.write( (char*)&l, sizeof(l) );
ofs.write( (char*)&f, sizeof(f) );
ofs.write( (char*)&d, sizeof(d) );
ofs.write( msg, sizeof(msg) );
ofs.close();
}
ifs.open( bName, ios::in|ios::binary );
if ( ifs )
{
ifs.read( (char*)&i, sizeof(i) );
ifs.read( (char*)&l, sizeof(l) );
ifs.read( (char*)&f, sizeof(f) );
ifs.read( (char*)&d, sizeof(d) );
ifs.read( msg, sizeof(msg) );
}
return 0;
}
```

Чтение символа

Для извлечения из потока одиночного символа можно использовать метод *int istream::get().* Пример иллюстрирует его применение:

```
//////////////////////////////////////////////////////////////////////////////
// GET1.CPP:  Чтение символа с помощью int get() //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
int main( void )
{
int ch;
cout << "Введите число,  за которым следует #:  ";
while( ( ch = cin.get() )!='#')
{
if ( ch = EOF ) break;
cout << (char)ch;
}
return 0;
}
```

Вы можете применить также один из следующих перегруженных вариантов функции *get.*

istream& istream::get(char&);
istream& istream::get(signed char&);
istream& istream::get(unsigned char&);

Например:

```
//////////////////////////////////////////////////////////////////////////////
// GET2.CPP: Использует 'istream& get(char&)'       //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
int main( void )
{


cout << "Введите число, за которым следует #: ";
while( cin.get( ch ) ) // Вызывает operator void*(); {
if ( ch = '#' ) break;
cout << (char)ch;
}
return 0;
}
```

Чтение строки

В библиотеке C++ имеются функции *get* и *getline* для чтения символов вплоть до ограничителя. Они часто используются при чтении строк.

get

Метод *istream& istream::get(char* _p, int _l, char _t)* извлекает символы из потока до тех пор, пока не

- будет найден ограничитель _t (по умолчанию \n).
- будет прочитано _l символов.
- встретится конец файла.

Примечание: Функция *get* не извлекает ограничитель из потока и не помещает его в буфер (см. *getline*).

Пример иллюстрирует функцию *get.*

```
//////////////////////////////////////////////////////////////////////
// GET3.CPP: Применение get(char*, int, char); //
//////////////////////////////////////У///////////////////////////////
#include <iostream.h>
#include <limits.h>


const Int MAX_LEN = 0x50;
int main( void )
{
char  fname[MAX_LEN];
char  lname[MAX_LEN];
// Спросить имя
cout << "Введите ваше имя:  ";
// Прочитать имя с помощью get() cin.get( fname,  sizeof( fname ) );
// Приветствовать...
cout << "Привет,  " << fname << endl;
// Спросить фамилию
cout << "Введите вашу фамилию:  ";
// Очистить входной поток:  ограничитель не был извлечен! cin.ignore(
INT_MAX,  '\n'  );
// Прочитать фамилию с помощью get() cin.get( lname,  sizeof( lname ) );
// Вывести приветствие...
cout   << "Привет,  "    << fname << '
<< lname << '!'     << endl;
return 0;
}
```

getline

Этот метод тоже может быть использован для извлечения из потока символов до тех пор, пока не будет найден ограничитель *_t* (по умолчанию \n), не будет прочитано определенное число символов _/ или не встретите конец файла:

```
istream& istream::getline(char* p,  int _ l ,  char _t);
```

Примечание: Функция *getline* извлекает ограничитель, но не записывает его в буфер.

Пример иллюстрирует функцию *getline.*

```
//////////////////////////////////////////////////////////////////////
// GETLINE.CPP:  Ввод с помощью getline... //
//////////////////////////////////////////////////////////////////////////////
include <iostream.h> const int MAX_LEN = 0x50;
int main( void )
{
char  fname[MAX_LEN];
```

```
char  lname[MAX_LEN];
// Спросить имя
cout << "Введите ваше имя: ";
// Прочитать имя
cin.getline( fname, sizeof( fname ) );
// Спросить фамилию
cout << "Введите вашу фамилию: ";
// Прочитать фамилию
cin.getline( lname, sizeof( lname ) );
// Вывести приветствие... cout << "Привет, " << fname << ' ' << lname  << '!'
<< endl;
return 0;
}
```

Часто применяемые функции

Помимо уже описанных перегруженных операций *помещения* и *извлечения,* а также функций бесформатного ввода/вывода, библиотека потоков C++ предлагает широкий выбор самых различных функций. В этом разделе описываются наиболее часто применяемые из них.

Пропуск символов при вводе

Чтобы при вводе пропустить некоторое число символов, вы можете применить следующий метод:

```
istream& istream::ignore(int n=1,  int d=EOF);
```

Эта функция извлекает символы из потока (не более n символов, по умолчанию 1), пока не встретит ограничитель *d* (по умолчанию EOF). Ограничитель также извлекается из потока.

Проверка счетчика извлечения

Функция *gcount, int istream::gcount();,* возвращает число символов, извлеченных последней функцией неформатированного ввода. Это число может измениться, поскольку некоторые процедуры, связанные с форматированным вводом, вызывают бесформатные функции.

Заглядывание вперед

Функция *peek* возвращает значение очередного символа, не извлекая его из входного потока:

```
int istream:: peek();
```

Функция возвращает EOF, если флаги состояния потока имеют ненулевое значение.

Возврат символа в поток

Функция *putback* возвращает во входной поток последний извлеченный символ.

```
istream& istream::putback( char ch );
```

Позиционирование потока

Функции *seekg* и *seekp* могут использоваться для позиционирования указателя извлечения/помещения соответственно входного и выходного потока:

istream& istream:: seekg(streampos);
istream& istream::seekg(streamoff, ios: :seek_dir);
istream& ostream: :seekp(streampos);
istream& ostream::seekp(streamoff, ios::seek_dir);

Выяснение текущей позиции потока

Функции *tellg* и *tellp* позволяют найти текущую позицию соответственно входного и выходного потока.

streampos istream::tellg(); streampos ostream::tellp();

Форматирование в памяти

Библиотека потоков C++ предусматривает операции ввода и вывода над данными в памяти (резидентными потоками), реализуемые классами *istrstream* и *ostrstream.* Третий класс, *strstream,* поддерживает оба типа

istrstream

Этот класс обеспечивает интерфейс для форматных извлечений из памяти. Вы можете создать объект класса *istrstream,* задавая буфер и его размер. Указание размера не требуется, если буфер оканчивается нулевым символом.

```
//////////////////////////////////////////////////////////////////////////
// ISTRSTRM.CPP:  Использование istrstream... //
//////////////////////////////////////////////////////////////////////////
#include <strstrea.h>
int main( void ) {
char info[] ="Симфоний 9 "
"Фортепианных_концертов 5";
istrstream stat( info );
char muslcType[20]; int number;
for( int i=1; i<2; i++ ) {
stat >> musicType;
stat >> number;
cout << "Л. Бетховен написал " << number << '\t' << musicType << endl; }
return 0;
}
```

ostrstream

Класс *ostrstream* предоставляет интерфейс для форматируемых помещений в память. Вы можете создать поток *ostrstream,* задавая буфер и его размер.

В классе имеется также конструктор по умолчанию, который выделяет буфер и динамически изменяет его размер во время исполнения. Следующий пример демонстрирует оба конструктора:

```
//////////////////////////////////////////////////////////////////////////
```

```
// OSTRSTR1.CPP: Создание потоков ostrstream... //
////////////////////////////////////////////////////////////////////////////
#include <strstrea.h>
int main( void )
{
//
// Создать ostrstream с динамическим
// перераспределяемым буфером…
//
ostrst ream   osstr_a;
char  buffer[100];.
//
// Создать поток с заданным буфером…
//
ostrstream osstr_b( buffer, sizeof( buffer ) );
// ...
return 0;
}
```

Помимо конструкторов и деструкторов, в ostrstream имеются два полезных метода:

• *char* ostrstream: :str();* этот метод возвращает указатель на буфер ostrstream. Кроме того, он "замораживает" массив. При использовании динамических объектов вызов *str()* делает динамический буфер вашей собственностью. В дальнейшем вы должны будете или удалить буфер, или вернуть его в собственность потока ostrstream, вызвав:

```
oss->rdbuf()->freeze(0);
```

• *int ostrstream::pcount();* метод возвращает число байт, которые были записаны в буфер.

Следующий пример показывает использование ostrstream с динамическим буфером.

```
////////////////////////////////////////////////////////////////////////////////////////////
// OSTRSTR2.CPP:  ostrstream с динамическим буфером         //
////////////////////////////////////////////////////////////////////////////////////////////
#include <strstrea.h>
int main( void )
{
//
// Создать ostrstream с динамическим
// перераспределяемым буфером…
//
ostrstream oss;
int i = 10;
char *str = "Значение равно ";
//
// Форматировать oss несколькими помещениями
//
oss << str
<< i
```

```
<< ends;
//
// Вывести результат форматирования
// с помощью метода str()
//
cout << oss.str();
//
// Вернуть буфер в собственность объекта ostrstream
// для надлежащей очистки
//
oss.rdbuf->freeze(0);
return 0;
}
```

Ostrstream не ограничивает буфер нулем автоматически

При работе с объектами ostrstream вы должны в явном виде добавить к потоку нулевой символ. В противном случае метод *str()*, скорее всего, возвратит указатель на строку, не ограниченную нулем.

Использование режима append для конкатенации строк

Вы можете создать поток ostrstream в режиме добавления, *установив* бит *ios::ate* или *ios::app* в опциональном третьем параметре конструктора. Это позволит вам передать в *качестве* буфера ограниченную нулем строку, и помещения в буфер начнутся с позиции нулевого символа:

```
//////////////////////////////////////////////////////////
// OSS_CAT.CPP: ostrstream в режиме добавления...       //
//////////////////////////////////////////////////////////

#include <strstrea.h>
#include <time.h>

int main( void )
{
   char msg[100] = "Текущее время: ";

   //
   // Создать ostrstream с заданным буфером.
   // Открыть в режиме добавления; помещение в
   // поток начнется с позиции ограничителя...
   //
   ostrstream oss( msg, sizeof(msg), ios::out|ios::app );

   time_t t_var;
   time( &t_var );

   oss << ctime( &t_var )
    << ends;

   cout  << oss.str()
    << endl;

   return 0;
}
```

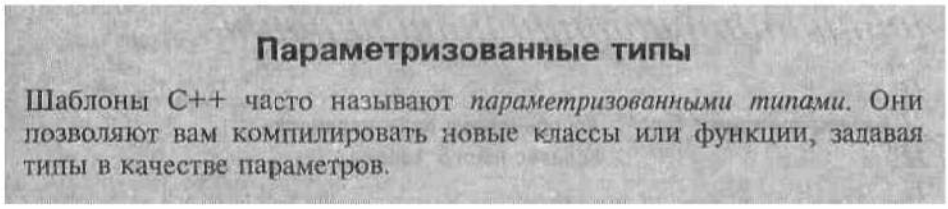
Заключение

Библиотека потоков C++ предоставляет программисту простой и последовательный интерфейс наряду с мощной и расширяемой реализацией. Она включает в себя классы, осуществляющие операции ввода/вывода со стандартными устройствами, дисковыми файлами и массивами символов. Все эти классы наследуют функциональные свойства корневых объектов иерархии и допускают

единообразное использование. В Borland C++ имеется класс *constream,* который служит иллюстрацией к тому, как может быть расширена стандартная библиотека. Если вы любите приключения, то, возможно, захотите пролистать файл *constrea.h* и посмотреть, не сможете ли вы усовершенствовать библиотеку lOStream путем добавления своих собственных классов.

Глава 8
Шаблоны C++

Шаблоны позволяют вам давать обобщенные, в смысле произвольности используемых типов, определения классов и функций. Эти определения затем могут служить компилятору основой для классов или функций, создаваемых для конкретного типа данных. Как следствие, шаблоны являются эффективным способом реализации процедур, которые раньше обычно переписывались много раз для данных различного типа. В данной главе рассматривается синтаксис и применение шаблонов и приводятся примеры, иллюстрирующие изложение. Также обсуждаются связанные с шаблонами установочные параметры компилятора.



Параметризованные типы

Шаблоны C++ часто называют *параметризованными типами*. Они позволяют вам компилировать новые классы или функции, задавая типы в качестве параметров.

Шаблоны функций

Шаблон функции представляет собой обобщенное определение, из которого компилятор может автоматически генерировать код (создать представитель) функции.

Шаблон функции: синтаксис

Синтаксис шаблона функции имеет следующий вид:

```
template <список_аргументов_шаблона> возвр_тип имя_функции
(параметры ... ) {
// Тело функции }
```

За ключевым словом *template* следуют один или несколько аргументов, заключенных в угловые скобки и отделенных друг от друга запятыми. Каждый аргумент состоит из ключевого слова *class* и идентификатора, обозначающего тип. Затем следует определение функции. Оно похоже на обычное определение функции, за исключением того, что один или несколько параметров используют типы, специфицированные в списке аргументов шаблона.

Определение шаблонов функции

Следующий пример показывает несколько определений шаблонов функции:

```
////////////////////////////////////////////////////////////////////////////////
// FUNCTMPL.CPP: Примеры шаблонов функции,..        //
////////////////////////////////////////////////////////////////////////////////
//
// Шаблон функции 'func': воспринимает один параметр // произвольного
типа...
//
template <class T>
void func( T t )
```

```
{
// Тело функции
}
//
// Шаблон функции 'Swap': воспринимает массив произвольного


//       типа и два целых числа…
//       Обменивает содержимое элементов
//       массива с указанными индексами
//
template <class T>
void Swap( T t[], int indx1, int indx2 )
{
T tmp   = t[indx1];
t[indx1] = t[indx2];
t[indx2] = tmp;
}
//
// Шаблон функции 'func_1': воспринимает два параметра
// одного типа…
//
template <class T>
void func_1.( T t1, T t2 )
{
// Тело функции
}
//
// Шаблон функции 'func_2': воспринимает два параметра
// различного (или одного и того же)
// типа.
template <class T1, class T2>
void func_2( T1 t1, T2 t2 )
{
// Тело функции
}
```

Часто приходится сортировать элементы массива. Вы можете использовать шаблон функции и написать обобщенное определение для сортировки массивов любых типов. Следующий пример демонстрирует возможный вариант шаблона функции, использующей алгоритм пузырьковой сортировки.

```
////////////////////////////////////////////////////////////////////////////////
//   SORTTMPL.H:  Шаблон функции для сортировки массива    //
////////////////////////////////////////////////////////////////////////////////


#if  !defined(__SORTTMPL_H)  #define       __SORTTMPL_H
//
// Шаблон функции для обобщенной сортировки массивов.
// ** NB: Для сортировки массива определенного пользователем
```

```
//     типа должна быть перегружена операция '>'.
//
template <class T> void sort( T array[], size_t size )
{
for( int 1=0;  i<size-1;  i++ )
for( int j=size-1; i<j; j - - )
if ( array[J-1] > array[j] )
{
T tmp = array[j]; array[j]    = array[j-1]; array[j-1] = tmp;
}

#endif//___SORTTMPL_H _____
```

Прототип шаблона .функции,

Вы мажете создатБ!?тототип шаблона функции в виде его;яредвар£'-тельного объявления. ТакЖ.объявление- информирует компилятор о наличии шаблона, а также*сообщает об ожклаемых параметрах.

*Яада!ймер:

```
//
// Предва^тельЯбе йб'&явлеШвийлона фушии 'sort'
//
templata <c1a^в т> void aoft( T аггауЦ&/$1гь_*«81гв.)4
```

п

Использование шаблона функции

После определения шаблона функции вы можете сразу ее вызывать: компилятор автоматически создаст представитель тела функции для указанных в вызове типов. Например, чтобы использовать определенный выше шаблон функции *sort,* вы можете просто вызвать ее с массивом и размером в качестве параметров. Пример это иллюстрирует:

```
////////////////////////////////////////////////////////////////////////////////
// SORT1.CPP: Использование шаблона функции sort //
////////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include "sorttmpl.h" // Шаблон функции sort
int main( void )
{
//
// Создать массив целых чисел
//
int iarray[] = { 30, 27, 45, 10, 23, 7, 4, 88 };
//
// Сортировать массив: компилятор автоматически создаст
// шаблонную функцию:
// " void sort( int[], size_t );
//
sort( iarray, sizeof( iarray )/sizeof( iarray[0] ) );
//
// Показать содержание отсортированного массива
```

```
//
for (int i=0;
i< sizeof( iarray )/sizeof( iarray[0] );
i++)
cout << "iarrayf << i<< "] = " << iarray[i]
<< end1;
return 0;
}
```

Шаблон sort можно применять и для классов, определенных пользователем, при условии, что класс перегружает операцию '>', используемую в шаблоне для сравнения элементов. Вот пример, показывающий применение шаблона sort к типу, определенному пользователем:

```
//////////////////////////////////////////////////////////////////////////////
// SORT2.CPP: Использование шаблона функции sort     //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include "sorttmpl.h"   // Шаблон функции sort
//
// Класс Amount: содержит значение в долларах/центах
//
class Amount
{
long dollars;
int cents;
public:
Amount( long _d , int _c )
{
dollars = _d ;
cents  = _c;
}
T
//
// Перегруженная опрерация '>'
// для сравнения представителей Amount...
//
int operator > (const Amount& ) const;
//
// Функция-друг, которая имеет доступ к частным
// элементам данных для форматирования вывода
//
friend ostream& operator << ( ostream&, Amount& );
};
int Amount::operator > ( const Amount& _amt ) const
{
return ( dollars > _amt.dollars ) ||
( dollars == _amt.dollars && cents  > _amt.cents  );
```

```
ostream operator << ( ostream& os, Amount &_anrt )
{
os << "$" << _amt.dollars << '.' << _amt.cents;
return os;
}
int main( void )
{
//
// Создать массив из Amount
//
Amount amtArray = {
Amount( 19, 10 ),
Amount( 99, 99 ),
Amount( 99, 95 ),
Amount( 19, 95 )
};
//
// Сортировать массив: компилятор автоматически создаст
// шаблонную функцию:
// void sort( Amount[], size_t );
//
sort( amtArray, sizeof( amtArray )/sizeof( amtAarray[0] ) );
//
// Показать содержание отсортированного массива
//
for (int i=0;
i< sizeof( amtArray )/sizeof( amtArray[0] );
i++)
cout << "amtArray[" << i << "] = " << amtArray[i]
<< endl;
return 0;
}
```

min и max как шаблоны

Файл *s(dUb.h,* входящий в комплект Borland C++, содержит определение шаблонов функций *min и маг,* которое используется в режиме компиляции C++:

```
template <class T> inline const T& fflin ( cNºBt т& t1,
                                           const T& t2 )
{
   return t1>t2 ? t2 : t1:.
}

template"'<class T> inline const T4 max ( const T4 tl;
                                           const J* til)
{
   return t"1>tZ ? t1 : 12;
}
```

Эти шаблоны позволяют ко>ш}Шятору автоматически создаватьварианты-представители функций.;*ЩП* и *max* всякий ?iaa, когда они вызываются с двумя параметрами одного типа.,Тип гщр;\метров'можетбыть встроенным или определенным пользоватсеяем е перегруженной функ*. > цией *opemtor>{)*. Пример это иллюстрируех

i

?ш/////////////////мм1/////////////m////ц/////////

/J MAX.CPP: Приквнвн^е*"шаЙдона функции=|вх().^Г *(i*

iriijiuinnfiiJiiniirniitinuiitffiniitirHiiHnn

```
«include <stdlib.h>
sif)clude-'<io'streeni.ti>
«include ^claGslib\date.h>


int main{' void )
{
   int 11 =*10?
   int 12 = 20;
```

```
    //
    // Для следующего оператора компилятор создает и вызывает
    // шаблонную функцию:
    // const int& max( const int&, const int& );
    //
    cout  << "Max целое равно "
          << max( i1, i2 )
          << endl;

    float f1 = 3.456;
    float f2 = 3.789;

    //
    // Для следующего оператора компилятор создает и вызывает
    // шаблонную функцию:
    // const float& max( const float&, const float& );
    //
    cout  << "Max вещественное равно "
          << max (f1, f2 )
          << endl;

    TDate d1( 8, 8, 65);
    Tdate d2(10, 3, 69);

    //
    // Для следующего оператора компилятор создает и вызывает
    // шаблонную функцию:
    // const TDate& max( const TDate&, const TDate& );
    //
    TDate d3 = max( d1, d2 );

    cout  << "Max дата равна " << d3
          << endl;

    return 0;
}
```

Шаблоны функций и функции шаблона

Шаблон функции является определением шаблона, из которого компилятор может создавать функции. Функции, созданные по шаблону, называются функциями шаблона (шаблонными функциями).

Перегрузка шаблонов функции

Точно так же, как и обычные функции, функциональные шаблоны могут быть перегружены. То есть вы можете предусмотреть несколько

шаблонов функции с одним и тем же именем, если только они имеют различные заголовки (различное число или различные типы параметров). Это показано в следующем примере:

```
//////////////////////////////////////////////////////////////////////////////
// OVRTMPL.CPP:  Перегруженные шаблоны функции... //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
 // // Возвращает больший из двух параметров
//
template <class T>
T getMax( T t1 , T t2 )
{
return t1 > t2 ? t1 : t2;
}
//
// Возвращает значение наибольшего элемента массива
//
template <class T>
T getMax( T t[], size_t size )
{
T retVal = t[0];
for( int i=0; i<size; i++ )



if ( t[i] > retVal )
retVal = t[i]; return retVal;
}

int main( void )
{
int i1 = 3;
int i2 = 5;
int iarray[] = { 3, 9, 5, 8 };
cout « "max int = " << getMax( i1 , i2 ) << endl;
cout << "max int = "
<< getMax(iarray, sizeof(iarray)/sizeof(iarray[0])) « endl;
return 0;
}
```

Специализация шаблонов функции

Специализированная функция шаблона - это обычная функция, имя которой совпадает с именем функции в шаблоне, но которая определяется для параметров специфических типов. Вы можете определять специализированные функции шаблона для случаев, когда обобщенный шаблон не годится для некоторого типа данных. Например, функция шаблона *getMax,* определенного выше, не может применяться для строк (char*, char[]), так как генерируемый компилятором код будет просто сравнивать их положение в памяти. Следующий пример предусматривает для этой цели специализированную функцию.

```
//////////////////////////////////////////////////////////////////////////
//OVRTMPL.CPP:  Перегрузка шаблона функции… //
//////////////////////////////////////////////////////////////////////////
#include <iostream.h> #include <string.h>

//
// Возвращает больший из двух параметров
//
template <class T>
T getMax( T t1 , T t2 )
{
return t1 > t2 ? t1 : t2;
}
//
// Специализированный вариант getMax
// для параметров-строк (char*, char[])
//
char* getMax( char*, s1 , char* s2 )
{
return strcmp( s1 , s2 ) > 0 ? s1 : s2;
}
int main( void )
{
int i1 = 3;
int i2 = 5;
cout << "max int = " << getMax( i1 , i2 ) << endl;
char *s1 = "Golden Eagle"; char *s2 = "Perigrine Falcon";
cout << "max str = " << getMax( s1 , s2 ) << endl;
return 0;
}
```

Разрешение ссылки на функцию

Когда компилятор встречает обращение к функции, для разрешения :ссылки используется следующий алгоритм:

• Найти функцию (не шаблонную), параметры которой соответствуют указанным в вызове.
• Если функция не найдена, найти шаблон, из которого можно генерировать функцию *с точным соответствием параметров.*
• Если никакой шаблон функции не обеспечивает точного соответствия, снова рассмотреть обычные функции на предмет возможного преобразования типов.

Заметьте, что преобразование типа параметров не производится при рассмотрении шаблонов.

Шаблоны классов

Шаблон класса дает обобщенное определение семейства классов, использующее произвольные типы или константы. Шаблон определяет элементы данных и элементы-функции класса. После определения

шаблона класса вы можете предписать компилятору генерировать на его основе новый класс для конкретного типа или константы.

Шаблон класса: синтаксис

Синтаксис шаблона класса имеет следующий вид:

template <*список_аргументов_шаблона*>
class *имя_класса*
{
// Тело класса };

За ключевым словом *template* следуют один или несколько аргументов, заключенных в угловые скобки и отделяемых друг от друга запятыми. Каждый аргумент является:

- либо именем типа, за которым следует идентификатор,
- либо ключевым словом *class,* за которым следует идентификатор, обозначающий параметризованный тип.

Затем следует определение класса. Оно аналогично определению обычного класса, за исключением того, что использует список аргументов шаблона.

Параметры типа и нетиповые параметры

Параметры шаблона, состоящие из ключевого слова class и следующего за ним идентификатора, часто называют параметрами типа. Другими словами, они информируют компилятор, что *шаблон предполагает тип* в качестве аргумента. Параметры шаблона, состоящие из имени типа и идентификатора, иногда называют нетиповыми. Они информируют компилятор, что в качестве аргумента предполагается константа.

Определение шаблонов класса

Следующий пример показывает несколько определений шаблонов класса:

```
////////////////////////////////////////////////////////////////////////////
//CLSTMPL1.H:  Определение шаблона класса… //
////////////////////////////////////////////////////////////////////////////
#if !defined(__CLSTMPL1_H) #define __CLSTMPL1_H
#include <stdlib.h>
const size_t defStackSize = ,10;
//
// TStack: Шаблон класса для простых стековых операций
//
template <class T>
class   TStack
{
public: TStack( size_t size = defStackSize )


{
```

```
numItems = 0; items = new T[size]; }
"TStack() delete [] items;
}
void push( T t );
T  pop();
protected:
int numItems;
T   *items;
};
//
// Метод push шаблона TStack
//
template <class T>
void TStack<T>::push( T t )
{
items[numItems++] = t;
}
//
// Метод pop шаблона TStack
//
template <class T>
T  TStack<T>::pop()
{
return items[--numItems];
}
#endif//__CLSTMPL1_H
```

Как видно из примера, функции-элементы шаблона класса могут определяться в теле класса (то есть как встроенные). Однако, если функция определяется как не-встроенная, ее заголовок имеет следующий формат:

template <*список_аргументов_шаблона*>
возвр_тип имя_класса<*арг_шаблона*>*::имя_функции(*п*араметр*ы*...*)
{
// Тело функции }

Встроенные методы шаблона класса

Можно применить ключевое слово *inline* для определения встроенного метода вне тела шаблона:

```
template <class Type>
inline void className <Type>::methodName( int i )
{
   // Тело функции
}
```

Как упоминалось выше, параметры шаблона класса могут быть либо типами, либо константами. Следующий пример демонстрирует шаблон класса, MemBlock, получающий константу, и другой шаблон, TypeBlock, получающий в качестве параметров и тип, и константу.

```
//////////////////////////////////////////////////////////////////////////
// CLSTMPL2.H: Определение шаблона класса... //
//////////////////////////////////////////////////////////////////////////
#if !defined(__CLSTMPL2_H) #define __CL8TMPL2_H
//
// MemBlock: Шаблон класса для обобщенного блока
//       памяти. Получает константу как параметр
//
template <int size>
class MemBlock
{
public: MemBlock()


{
p = new char[size];
}
~MemBlock()
{
delete [] p;
}
operator char* ()
{
return p;
}
protected:
char *p;
};
//
// TypeBlock: Шаблон класса, содержащего блок памяти для
//       определенного типа. Получает как параметры
//      тип и константу...
//
template <class T, int numElements>
class TypeBlock
{
public: TypeBlock();
~TypeBlock();
operator T* ();
protected:
T* p;
};
template <class T, int numElements> TypeBlock<T,
numElements>::TypeBlock()
{
p = new T[numElements];
```

```
}


template <class T, int numElements> TypeBlock<T,
numElements>::~TypeBlock()
{
delete [] p;
}
template <class T, int numElements>
TypeBlock<T, numElements>::operator T*()
{
return p;
}
#endif//__CLSTMPL2_H
```

Как и классы, шаблоны классов могут содержать статические элементы данных, статические элементы-функции, дружественные функции и классы. Нельзя определить внутренний шаблон в теле другого шаблона класса. Однако вы можете объявить внутренний класс, который использует один или несколько параметров шаблона.

Использование шаблонов класса

Чтобы создать представитель шаблонного класса, можно просто указать имя шаблона со списком аргументов, заключенным в угловые скобки, в качестве спецификатора типа. Список аргументов шаблона модифицируется следующим образом:.

• каждый аргумент, имеющий вид *'class идентификатор'* (параметр типа), заменяется именем действительного типа.

• каждый аргумент, имеющий вид *'имя_типа идентификатор'* (нетиповой параметр), заменяется константным выражением.

Следующий пример показывает порождение различных представителей шаблона класса *TStack.*

```
//////////////////////////////////////////////////////////////////////////////////////////////
// STACKTMP.CPP:  Использование шаблона класса TStack      //
//////////////////////////////////////////////////////////////////////////////////////////////


#include <cstring.h>
#include "clstmpl1.h"      // Определение TStack
TStack<int> stckInt1;     // Стек значений 'int',
// предполагающий размер по // умолчанию
TStack<int> stckInt2( 40 );  // Стек для 'int', вмещающий до 40
// элементов...
TStack<long> g *pstckLng;   // Указатель на стек для 'long' TStack<double>
dblStk[10]; // Массив стеков для 'double'
TStack<string> strStk;     // Стек для значений типа 'string',
// определенного пользователем;
// [см. cstring.h]
//
// Указатель на элемент-функцию класса 'Стек для unsigned',
// получающую long параметр и возвращающую int...
//
```

```
int (TStack<unsigned>::*mf)( long );
extern TStack<char*> cStck;  // Внешний стек указателей
// на char…
extern TStack<char*> «pcStck; // Внешний указатель на стек
// для указателей на char…
//
// Выделение памяти для переменной pcStck
//
void InitpcStck()
{
pcStck = new TStack<char*>(20);
}
```

Создав представитель шаблонного класса, можно обращаться с ним точно так же, как с принадлежащим к обычному классу. Следующий пример

демонстрирует создание и использование представителей шаблонных классов.

```
//////////////////////////////////////////////////////////////////////////////
// CLSTMPL1.CPP: Применение шаблонов класса… //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include "clstmpll.h"
int main( void )
{
TStack<int> Stack0fInt( 10 );
Stack0fInt.push( 33 );
Stack0fInt.push( 44 );
Stack0fInt.push( 55 );
cout << "Помещенные значения равны: "
<< Stack0fInt.pop() << ','
<< Stack0fInt.pop() << ','
<< Stack0fInt.pop() << endl;
TStack<double> Stack0fDouble( 10 );
Stack0fDouble.push( 2.3456 ); StackOfDouble.push( 4.8967 );
Stack0fDouble.push( -1.345 );
cout << "Помещенные значения равны: "
<< Stack0fDouble.pop() << ','
<< Stack0fDouble.pop() << ','
<< StackOfDouble.pop() << endl;
return 0;
}
```

Еще один пример использует шаблон класса *MemBlock:*

```
//////////////////////////////////////////////////////////////////////////////
// CLSTMPL2.CPP: Применение шаблонов класса… //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
```

```
#include <string.h>
#include "clstmpL2.h"
int main( void )
{
MemBlock<512> Half_K_Block;
strcpy( Half_K_Block, "Have a nice day!" ); cout << (char*) Half_K_Block <<
endl;
MemBlock<1024> One_K_Block;
strcpy( One_K_Block, "Have a royal day!" );
cout << (char*) One_K_Block << endl;
return 0;
}
```

Используйте для шаблонных классов typedef

Вы можете применить к шаблонному классу ключевое слово typedef, чтобы упростить его использование. Например,

```
#include "clstmpl1.h"

TStack<long>   lstck;
TStack<long> *plstck;
TStack<long>  alstck[10];

можно упростить до

#include "clstmpl1.h"

typedef TStack<long>  LStack;

LStack   lstck;
LStack *plstck;
LStack  alstck[10];
```

Специализация шаблонов класса

Можно специализировать шаблон класса, предусмотрев для специфических типов явную реализацию некоторых методов. Рассмотрите следующее определение шаблона класса *TArray.*

```
////////////////////////////////////////////////////////////// // ARRAY.H: Шаблон класса для
простых массивов... //
//////////////////////////////////////////////////////
#if ! defined(__ARRAY_H)
#define  _ARRAY_H
#include <fstream.h>
const int defSize = 100;
//
// Шаблон обобщенного массива: сохраняет объекты
// в массиве...
//
template <class T>
class  TArray
```

```
{
public: TArray( int size = defSize )
{
//...

TArray<int>iArray; iArray.add( i1 );
char *msg = "Look at the rose hill!"
//
// Следующее определение объекта использует
// обобщенный вариант конструктора…
//
TArray<char*> strArray;
//
// Вызов метода 'add' передается специализированному
// варианту 'add', предусмотренному
// для массивов <char*>
//
strArray.add( msg );
return 0;
}
```

Вы можете также предусмотреть полное переопределение шаблона класса, предназначенное для работы с некоторым особым типом. При специализации целого шаблонного класса вы должны сделать следующее:

• Предусмотреть определение специализированного шаблонного класса, поместив его после определения обобщенного класса.

• Предусмотреть определения для всех элементов-функций.

Следующий пример является немного модифицированным вариантом предыдущего: он определяет вариант шаблона класса *TArray,* специализированный для типа <string>.

```
#include <cstring.h> #include "array.h"
//
// Специализированный вариант шаблона TArray
// для поддержки типа <string>
//
class TArray<string>


{
public:
TArray( int size = defSize )
{
// ...
}
~TArray()
{
// ... }
void add( string str, int indx = -1 );
};
//
```

```
// Специализированный метод add
//
void TArray<string>::add( string str, int indx )
{
// ...
}
int main( void )
{
int i1 = 10;
//
// Вызывает обобщенный конструктор TArray
//
TArray<int> iList;
//
// Вызывает обобщенный метод add из TArray
//
iList.add( i1 );
string str( "Morning at Rose Hill!" );
//
// Вызывает специализированный
// конструктор TArray<string>
//
TArray<string> sList;
//
// Вызывает специализированный вариант
// TArray<string>::add
//
sList.add( str );
return 0;
}
```

Перегрузка шаблонов класса

В то время как вы можете перегружать имена шаблонов функций, *настоящая реализация* шаблонов C++ не позволяет перегрузить имя шаблона класса. Например, нельзя определить одновременно TArray<class T> и TArray<class T, int size>.

Шаблоны и конфигурация компилятора

Borland C++ имеет три селектора, которые определяют, каким образом будет генерироваться код для представителей шаблонов функций и классов. Пользователи IDE могут получить к ним доступ через меню Options:

• Выберите в меню Optons | Project; появится окно диалога, как показано на рис. 8.1.
• Раскройте рубрику C++ Options и выберите Templates. Появится диалог Template Settings.

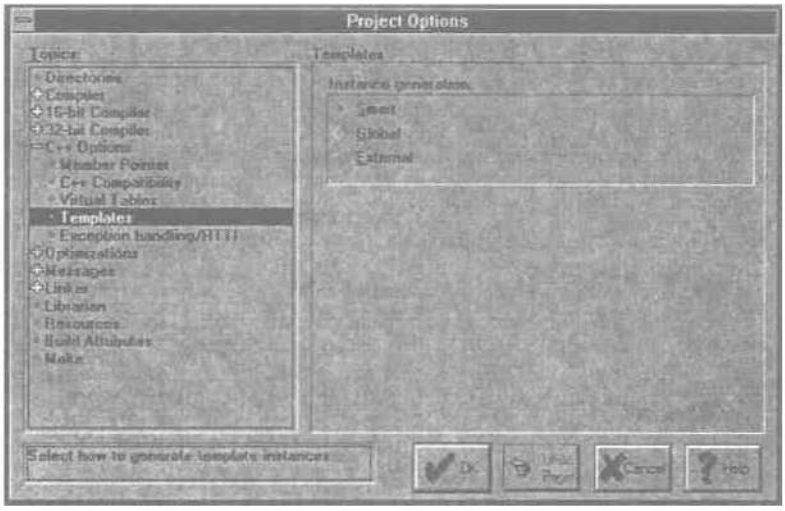


Рис. 8.1. *Диалог Project Options.*
Пользователи компилятора с командной строкой могут установить те же опции с помощью ключей -Jd (smart), -Jgd (global) и -Jgx (external).

Шаблоны Smart

Опция шаблонов *Smart* принимается по умолчанию. Она предписывает компилятору генерировать соответствующий код для тех типов, для которых создаются представители шаблона. Эта конфигурация подходит для проектов, в которых представители шаблона создаются только после того, как компилятору станет известно полное определение шаблона (определение тела шаблона функции, тела элементов-функций и определение статических элементов данных).
Хотя компилятор может несколько раз генерировать код для одной и той же комбинации шаблон/тип, если она используется в различных модулях, в конечный модуль (.EXE или .DLL) компоновщик включит только один такой экземпляр кода.

Поместите ваше определение шаблона в файл-заголовок

Создайте файл-заголовок для вашего определения шаблона(ов) функции или класса. Файл должен содержать:

- тело шаблонов функции.
- тело элементов-функций шаблона класса.
- определение статических элементов-данных шаблона класса.

Шаблоны Global и External

Если вы создаете или используете представители шаблона до того, как его полное определение станет известным компилятору, необходимо

комбинированное применение опций шаблонов *Global* и *External.* Эти опции должны применяться и в том случае, если вы хотите, чтобы компилятор генерировал код для некоторого шаблонного представителя, не создавая сам представитель. Например, разработчик библиотек, предусматривающий использование шаблонов, может построить динамическую библиотеку с распространенными представлениями шаблона, не создавая действительных его представителей.
Для правильного использования опций *Global* и *External* вы должны убедиться, что

• для каждой комбинации шаблон/тип только один модуль, создающий ее представителей, компилируется с разрешением опции шаблонов *global.*

• все остальные модули, создающие представителей данной комбинации шаблон/тип, используют опцию *external.*

Первый из приведенных ниже модулей применяет глобальную опцию шаблонов и создает представитель C<int>, в то время как второй использует внешнюю конфигурацию.

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// TMPLOPT1.CPP: Применение шаблонов в конфигурации                                                           //
//global/external                                                                                             //
////////////////////////////////////////////////////////////////////////////////////////////////////////////////

//
// Следующая директива указывает, что необходимо
// генерировать глобальные определения для представителей
// шаблона, создаваемых в данном модуле…
//
#pragma option -Jgd       // <<<<<<<<<<<
//
// Шаблон класса C
//
template <class T>
class  C
{
public: C();
};
//
// Конструктор C
//
template <class T>
C<T>::C()
{
// тело конструктора
}
C<int> CofInt;   // Приводит к глобальной генерации класса C<int>



////////////////////////////////////////////////////////////////////////////////////////////////
// TMPLOPT2.CPP: Применение шаблонов в конфигурации      //
// global/external                                       //
```

```
//////////////////////////////////////////////////////////////////////////////////////////
//
// Следующая директива указывает компилятору, что нужно
// генерировать внешние ссылки для представителей шаблона,
// создаваемых в данном модуле. Это необходимо, так как
// при компиляции модуля компилятор не имеет доступа
// к телу конструктора C<T>::C()...
//


pragma option -Jgx
//
// Шаблон класса C
//
template <class T>
class C
{
public: C();
};
int main( void )
{
C<int> iC;   // Генерирует внешние ссылки!
return 0;
}
```

Используйте typedef для генерации глобальных определений

Если применяется глобальная опция шаблонов, компилятор генерирует глобальные определения, как только ему встречается создание представителя. Однако вы можете вынудить компилятор генерировать глобальные определения шаблона, просто применив определение typedef после определения тела шаблона. Этот прием применяется в Container Class Library (библиотеке контейнерных классов) пакета Borland C++. См. файл TMPLINST.CPP в каталоге \\BC4\SOURCE\CLASSLIB.

•

Недостатки шаблонов

Шаблоны C++ приносят программисту определенные выгоды, включая утилизируемость и малые затраты на сопровождение кода. Техника шаблонов

в общем весьма эффективна по сравнению с другими методами (такими, как полиморфизм), применяемыми в обращении с различными типами данных. Шаблоны, кроме того, обеспечивают безопасное использование типов, в отличие от макросов препроцессора. Однако вы должны принимать во внимание и некоторые отрицательные стороны шаблонов:

• Ваша программа будет содержать полный код представителей шаблона для каждого из порождающих типов. Это может сильно увеличить размер исполняемого модуля.

• Часто реализация шаблона хорошо работает с некоторыми типами, но далека от оптимальной в других случаях. Примером могут служить шаблоны функций *min* и *max,* определяемые файлом STDLIB.H.

```
template <class T> inline const T& min ( const T& t1 ,
const T& t2 ) {
return t1>t2 ? t2 : t1 ; }
template <class T> inline const T& max ( const T& t1 ,
const T& t2 ) {
return t1>t2 ? t1  :  t2; }
```

Эти функции не слишком хороши в применении к встроенным типам: использование ссылок заставляет компилятор генерировать дополнительный код косвенных обращений. Однако шаблон годится для больших объектов, копирование которых было бы дорогостоящим. Поэтому имеет смысл рассмотреть *специализацию* шаблона как возможный способ повышения эффективности.

Заключение

Из этой главы вы получили представление о мощности шаблонов и преимуществах, которые дает их применение. Различные большие библиотеки (такие, как библиотека контейнерных классов Borland C++) широко используют шаблоны. Вы можете легко создавать новые классы и функции,

подставляя в уже имеющиеся шаблоны конкретные аргументы. Или же, работая над проектом, вы можете начать создание своих собственных шаблонов для определения операций, алгоритмов и функций, которые вы используете.

Глава 9
Управление исключениями

Под *управлением исключениями (exception handling)* понимают стандартный интерфейс для обнаружения и обработки необычных, непредвиденных и исключительных состояний или событий. Он предоставляет формальный способ отклонить поток управления функции на неспецифицированную секцию кода, готового принять контроль над данной исключительной ситуацией. Borland C++ поддерживает две разновидности управления исключениями:

- Управление исключениями языка C++
- Структурированное управление исключениями языка C

В данной главе рассматривается синтаксис и применение обоих механизмов и освещаются различные преимущества управления исключениями перед традиционными методами сообщения об ошибочных состояниях:

- Единообразность стиля, которая, в свою очередь, улучшает разборчивость кода и облегчает его сопровождение.
- Устранение глобальных переменных и определяемых пользователем возвратных процедур, обычно использовавшихся в подобных целях.
- Расширение возможностей отладки. Механизм исключений является интегрированной частью языка и запрашивает поддержку исполнительной библиотеки; отладчик обеспечивает специализированную поддержку, позволяя вам отслеживать исключительные ситуации.
- Невозможность для программ игнорировать ошибки и продолжать выполняться, надеясь на лучшее.

Исключения и стек

Когда функция "выбрасывает" (термин C++) или "заявляет" (структурные исключения C) исключение, управление передается некоторой секции кода, принадежащей функции, которая еще находится в *стеке вызовов.* Другими словами, управление может быть передано только функции, вызванной ранее и еще не завершившейся.

Стек представляет собой область памяти, в которой сохраняются локальные (временные) переменные и адреса возврата из функций. На компьютерах с процессорами INTEL один из регистров, SP (указатель стека), указывает на последнее значение (слово), помещенное в стек. Например, для приведенной ниже программы стек в точке (1) внутри main() может быть представлен следующим образом:

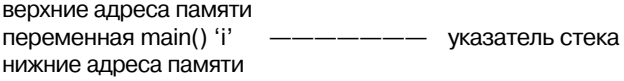


Рис. 9.1. *Представление стека*

```
//////////////////////////////////////////////////////////////////////////////
// STACK.CPP: Пример для понимания стека вызовов...   //
//////////////////////////////////////////////////////////////////////////////
#include <fstream.h>
void func2()
{
int I = 0x1000L;
// (3) Когда управление достигает этой точки, указатель
// стека указывает на переменную 'I'; если двигаться
```

```
// вверх по стеку, мы обнаружим:
// - обрамление стека func2()


// - адрес возврата в func1()
// - переменную fund 'ifs'
// - обрамление стека func1()
// - адрес возврата в main()
// - переменную main 'i'.
void func1()
{
ifstream ifs( "STACK.CPP" );
// (2) Двигаясь вверх по стеку, когда управление достигло
// этой точки, мы обнаружим следующее:
// - локальную переменную 'ifs'
// - обрамление стека func1()
// - адрес возврата в main()
// - переменную main 'i'.
func2();
}
int main( void ) int i = 0x1;
{
// (1) Когда указатель инструкций достигает этого места,
// указатель стека указывает на переменную 'i',
// последнее значение, помещенное в стек...
func1(); return i;
}
```

В точке (3) внутри функции func2() стек выглядит, как показано на рис. 9..2.

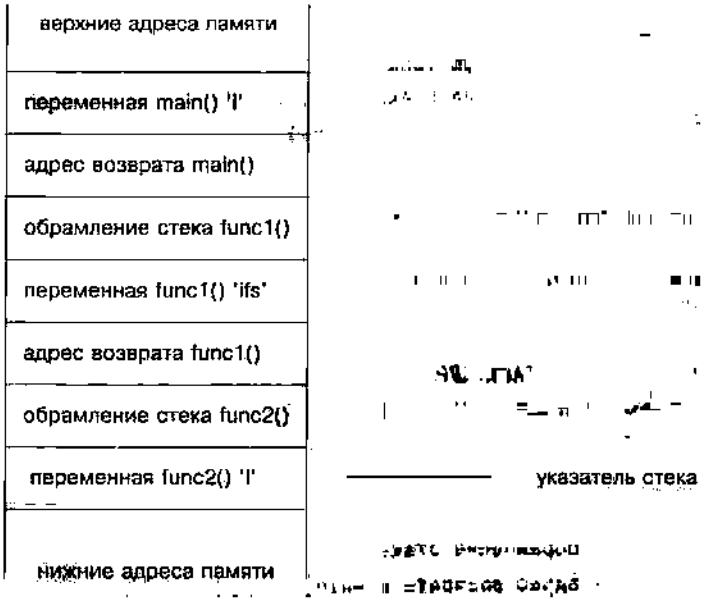


Рис. 9.2. *Стек в точке (3) внутри func2()*

Термин *стек вызовов* обозначает последовательность вызванных (но еще не завершившихся) функций, в результате которой управление достигает определенной точки программы. Рис. 9.3 показывает стек вызовов изнутри func2().
Разматыванием стека называется процесс выталкивания значений из стека, необходимый для того, чтобы уничтожить локальные переменные и возвратить управление вызывающей функции. Когда функция возвращает управление, происходит естественное разматывание стека. Однако стек может разматываться и через управление исключениями, функцию longjump() или прямую модификацию регистров CPU. В случае управления исключениями в C++ термин *разматывание стека* относится также к исполнению деструкторов локальных объектов по мере выталкивания их из стека.

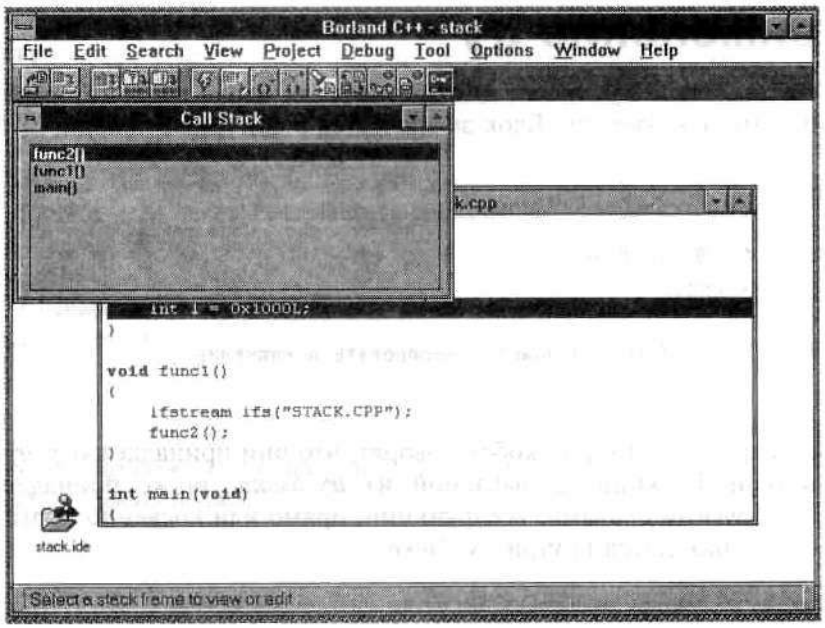


Рис. 9.3. *Стек вызовов для приведенного примера*

Работа с управлением исключениями языка C++

Управление исключениями C++ является частью стандарта ANSI C++. Этот стандарт поддерживает *окончательную* модель управления: после того, как исключение было зафиксировано, обрабатывающая процедура не может потребовать, чтобы исполнение было продолжено с точки исключения. Исключения C++ также не поддерживают обслуживание асинхронных событий, таких, как ошибки оборудования, или обработку прерываний. Обслуживаются только исключения, в явном виде сигнализированные некоторой функцией.
В контексте управления исключениями в C++ применяются три ключевых слова: *try, catch* и *throw.*

Применение *try*

Ключевое слово *try* служит для обозначения блока кода, который может генерировать исключение. Блок заключается в фигурные скобки:

```
try
{
cout << "В try-блоке…  "
<< endl;
func();      // func() может генерировать исключение
}
```

Про операторы внутри скобок говорят, что они принадлежат к *try-блоку.* Тело всякой функции, вызываемой из *try-блока,* также принадлежит к *try-блоку.* Другими словами, все функции, прямо или косвенно вызываемые из *try-блока,* находятся внутри *try-блока.*

Применение *catch*

Ключевое слово *catch* следует за *try-блоком* и обозначает секцию кода, в которую может быть передано управление в том случае, если произойдет исключение. За ключевым словом следует *описание исключения,* заключенное в скобки и состоящее из имени типа и необязательной переменной. Имя типа идентифицирует тип исключений, которые данный код может обслуживать. Можно рассматривать *описание исключения* как параметр функции.

Блок кода, обрабатывающего исключения, заключается в фигурные скобки и называется *catch-обработчиком* или *обработчиком исключения.* За *try-блоком* могут следовать несколько операторов *catch.* Следующий пример иллюстрирует оператор *catch:*

```
//////////////////////////////////////////////////////
// CATCH.CPP:  Оператор catch… //
//////////////////////////////////////////////////////
#include <iostream.h>
void func( void );     // Прототип внешней функции int main( void
}


{
try
{
cout << "В try-блоке… " « endl;
func();  // Может генерировать исключение }
// Обрабатывает исключения типа 'int'
catch( int i )
{
}
// Обрабатывает исключения типа 'const char*'
catch( const char* )
{
}
// Обрабатывает все (необслуженные) исключения
catch( …  )
{
}
return 0;
```

}
Оператор *catch* с многоточием (...) перехватывает исключения любого типа и должен быть последним из операторов *catch,* следующих за *try-блоком.*

Применение *throw*

Ключевое слово *throw* выбрасывает исключение и вызывает переход управления к обработчику. За ключевым словом может следовать выражение.

throw с операндом

Выражение, следующее за *throw,* сводится к значению переменной определенного типа. Можно рассматривать операнд *throw* как аргумент вызова

функции. Тип операнда определяет, который из обработчиков может перехватить исключение. Местоположение оператора *throw* обычно называют *точкой выброса.* См. следующий пример:

```
//////////////////////////////////////////////////////////////////////////////
// THROW.CPP:  Применение ключевого слова throw... //
//////////////////////////////////////////////////////////////////////////////
#include- <cstring.h>
void func1 ()
{
//
// Выбрасывает 'const char*'
//
if ( что-то_не_в_порядке )
throw "Обнаружена ошибка...";
}
void func2()
{
//
// Выбрасывает объект 'string'
//
if ( что-то_не_в_порядке )
{
string str( "Ой!..." );
throw str;
}
}
```

Типы в операторе throw

Можно выбрасывать как встроенные, так и *определенные* пользователем типы. Это означает, что ваши возможности не ограничиваются возвратом кода ошибки; вы можете передать обработчику объект, нагруженный информацией.

throw без операнда

Если ключевое слово *throw* применяется без операнда, то заново выбрасывается то исключение, которое обрабатывается в данный момент. Из этого следует, что такая форма оператора может быть использована только в cutfc/z-обработчике или в функции, которая явно или неявно им вызывается.

Перехват *throw*

Когда выполняется оператор *throw,* функции исполнительной библиотеки C++ производят следующие действия:

• Создают копию выброшенного объекта/переменной.

• Разматывают стек, вызывая деструкторы локальных объектов, выходящих из области действия.

• Передают управление ближайшему обработчику *catch,* принимающему параметр, совместимый по типу с выброшенным объектом. Копия объекта передается обработчику в качестве параметра.

Пример иллюстрирует последовательность происходящих событий.

```
//////////////////////////////////////////////////////////////////////////////
// XCPT1.CPP:  Соответствие параметров trow и catch…    //
//////////////////////////////////////////////////////////////////////////////
#include <fstream.h>
//
// TaleTellingClass: Простой класс, который информирует
// о своем создании и уничтожении…
//
class  TaleTellingClass
{
public:
TaleTellingClass()
{
cout << "TaleTellingClass:
"Жили-были  …."
<< endl;
}


~TaleTellingClass()
{
cout << "TaleTellingClass: "
"....  и умерли в один день." << endl; }
};
void function1( void ) {
```

```
ifstream ifs( "\\INVALID\\FILE\\NAME" );
if ( !ifs )
{
cout << "Выбрасываем исключение..." << endl;
//
// Выбросить 'const char*' //
throw "Ошибка при открытии файла..."
}
//
// В противном случае, файл открыт успешно //
}
void function2( void )
{
//
// Создать локальный объект, чтобы проверить вызов
// деструктора при разматывании стека
//
TaleTellingClass tellme;
//
// Вызвать функцию, выбрасывающую исключение
//


function1();
}
int main( void )
{
try
{
cout << "Входим в try-блок..." << endl;
function2();
cout << "Выходим из try-блока..."
<< endl;
}
catch( int i )
{
cout << "Вызван обработчик 'int' с "
<< i
<< endl;
return - 1;
}
catch( const char* p )
{
cout << "Вызван обработчик 'char*' "
<< ' ['<< p << ' ]'
<< endl;
return - 1; }
catch( ... )
{
cout << "Вызван обработчик catch_all..." << endl;
```

```
return  - 1 ;
}


return 0;   // Обошлось без приключений!
}
```

Результаты выполнения приведенного кода имеют такой вид:

```
Входим в try-блок...
TaleTellingClass: Жили-были ....
Выбрасываем исключение...
TaleTellingClass:    ....  и умерли в один день.
Вызван обработчик 'char*'  [Ошибка при открытии файла...]
```

Заметьте, что в данном примере

- Деструктор локальной переменной *tellme* функции *function2* был вызван правильно, хотя из *function1* управление было передано непосредственно обработчику исключения, находящемуся в main.
- Оператор из main, содержащий сообщение "Выходим из try-блока...", так и не был выполнен.
- Для данного исключения был вызван второй из обработчиков catch, так как тип его параметра соответствовал типу выброшенного объекта (т.е. const char*).

Тип исключения и конструкторы копии

Генерируется сообщение об ошибке, если конструктор копии выброшенного объекта в точке выброса недоступен. Например, вы не можете выбросить тип или указатель на тип с не-public конструктором копии, если только вы не являетесь *другом*. Рассмотрите такой пример:

```
//////////////////////////////////////////////////////////
// NOTHROW.CPP: Исключения и доступ...                  //
//////////////////////////////////////////////////////////

//
// Класс с не-публичным конструктором копии
//
class TCopCon
{
```

```
    public:
     TCopCon();

    protected:
     TCopCon( const TCopCon& );

    friend void funcB( void );
};


void funcA( void )
{
   if ( что-то не в порядке )
   {
    TCopCon пр;

    // Следующее выражение throw приводит к
    // сообщению об ошибке:
    //
    // Error nothrow.cpp 45:
    //   'TCopCon::TCopCon(const TCopCon&)'
    //   is not accessible in function funcA()
    //   ( ... недоступен в функции funcA()  )

    throw пр;
   }

   // ...
}


void funcB( void )
{
   if ( что-то не в порядке )
   {
    TCopCon пр;

    //
    // Здесь все правильно
    //
```

```
    throw пр;
   }

   // ...
}
```

поиск соответствующего типа исключения

Как видно из синтаксиса управления исключениями и предыдущего примера, исключение является одновременно переменной и типом данных. Можно сказать также, что оно одновременно является объектом и классом:

Вы *выбрасываете* переменную и *ловите* тип. Вы *выбрасываете* объект и *ловите* класс.

После того, как исключение выброшено, процедуры исполнительной библиотеки C++ ищут подходящий обработчик. Обработчик считается найденным, если

• Тип выброшенного объекта тот же самый, что и тип, ожидаемый обработчиком. Другими словами, если выбрасывается *T,* ему будет соответствовать обработчик, который перехватывает *T, const T, T&* или *const T&.*

• Тип обработчика является публичным базовым классом выброшенного объекта.

• Обработчик ожидает указатель, и выброшенный объект является указателем, который может быть преобразован к типу обработчика по стандартным правилам преобразования указателей.

Следите за порядком следования catch-обработчиков

Будьте .внимательны к последовательности, в которой вы; располагаете 1 обрабохЧикимс^10че>!й%Обра^о^тк, ожидающий исключение базог

;бьго класса, автоматаческилкрьгвает обработчик производного класса; *Точт'Шк* же об^ботчик! для пустого-укадателя будет автоматически' скрыш4т& обработчик для указателя любого типа. Рассмотрите следующий- пример:

```
//////////////////////////////////////////////////////////
// CATCHORD.CPP. Порядок следования сат.сЬ-айработчда«щ //
//////////////////////////////////////////////////////////

tfinolude  <iostreafl;h>

class Base
{
};

class'Derived : public-Base
<          -'
};

void  furtcAf)
'{   {,';
    Derived d;
    throf?.  .&,                 •
>     : . .                ; *

void f'uncBO
{
   thr^..  "Ошибка в funcB()"
}


int main(void )
{
   try
   {
    funcA();
   }
```

```
    //
    // Следующие обработчики catch расположены неправильно:
    // обработчик 'Derived&' должен предшествовать обработчику
    // базового класса. При теперешней их последовательности
    // обработчик 'Derived&' никогда не сможет получить
    // управление!
    //
    catch( Base& )
    {cout << "Перехвачен Base&" << endl;          }

    catch( Derived& )
    {cout << "Перехвачен Derived&" << endl;       }


    try
    {
     funcB();
    }

    //
    // Следующие обработчики catch расположены неправильно:
    // обработчик 'const char*' должен предшествовать обработчику
    // 'void*'. При теперешней их последовательности
    // обработчик 'const char*' никогда не сможет получить
    // управление!
    //
    catch( void* )
    {cout << "Перехвачен void*" << endl;          }

    catch( const char* )
    {cout << "Перехвачен const char*" << endl;    }

    return 0;
}
```

При исполнении программа выводит:

```
Перехвачен Base&
Перехвачен void*
```

Применение *terminate()* и неуправляемые исключения

Если для выброшенного исключения не найдено подходящего обработчика, вызывается функция *terminate().* Она вызывает функцию *abort(),* аварийно завершающую текущий процесс.

Вы можете установить свою собственную функцию завершения с помощью функции *set_terminate,* определенной в файле EXCEPT. H:

```
typedef void (_RTLENTRY *terminate_function)();
// ...
terminate_function _RTLENTRY set_terminate(terminate_function);
```

Эта функция возвращает адрес предыдущей функции завершения. Ваша завершающая функция не должна возвращать управление вызвавшему ее коду или выбрасывать исключение.

Функция завершения вызывается также в том случае, когда исключение выбрасывается при выполнении деструктора — если деструктор был вызван в процессе разматывания стека, инициированного ранее брошенным исключением.

Работа со спецификациями исключений

Вы можете задать список исключений, которые функция может прямо или косвенно выбрасывать, с помощью *спецификации исключений,* присоединяемой к заголовку функции. Спецификация имеет следующий формат:

throw(тип, тип, ...)

Спецификация исключений без типа предполагает, что функция не должна выбрасывать никаких исключений. Функции без спецификации, напротив, могут выбрасывать исключения любого типа. Следующий пример демонстрирует спецификации исключений:

```
////////////////////////////////////////////////////////////////////////////////////////////////
// XCPTSPEC.CPP:  Применение спецификаций исключения       //
////////////////////////////////////////////////////////////////////////////////////////////////


struct xClass
{
int i;
}
void funcA( void ) throw( int )
{
// Функция должна выбрасывать только исключение типа int
}
void funcB( void ) throw( long, xClass* )
{
// Функция должна выбрасывать только переменные long,
// указатели на xClass или на типы, производные от
// xClass
}
void funcC( void ) throw()
{
// Функция не должна выбрасывать какие-либо исключения
}
```

Работа с непредусмотренными исключениями

Спецификация исключений ни к чему, собственно, не обязывает. То есть функция может прямо или косвенно выбросить исключение, которое она обещала не использовать. Например, следующий код при компиляции не вызовет ошибки или предупреждения:

```
void func( void ) throw( int )
{
throw "ОЙ!...";
}
```

Нарушение списка допустимых исключений обнаруживается только во время исполнения. Непредвиденные исключения приводят к вызову

функции *unexpectedQ.* По умолчанию *unexpected* просто вызывает функцию *terminate().* Однако с помощью функции *set_unexpected()* вы можете установить свою собственную процедуру, которая будет вызываться, если функция выбрасывает не специфицированное исключение. Функция *set_unexpected* определена в файле EXCEPT.H:

```
typedef void (_RTLENTRY *unexpected_function)();
// ...
unexpected_function _RTLENTRY
set_unexpected(unexpected_function);
```

Функция возвращает адрес предыдущей процедуры для непредусмотренных исключений. Ваша процедура не может возвращать управление или выбрасывать исключение.

Работа с конструкторами и исключениями

Ниже описывается схема того, что происходит, если исключение прямо или косвенно выбрасывается конструктором класса.

Локальные объекты

При возникновении исключения деструкторы вызываются только для полностью конструированных локальных объектов. Это предполагает, что если исключение происходит в конструкторе объекта, только для уже сконструированных объектов данных и базовых классов будут вызваны соответствующие деструкторы. Например, в приведенной ниже программе вызываются только деструкторы *TDataClass* и *TBaseClass.*

```
////////////////////////////////////////////////////////////////////////////////
// XCPTCONS.CPP: Уничтожение конструированных объектов //
////////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <except.h>
class  TDataClass
{
public:


+TDataClass()
{
cout << "TDataClass::TDataClass()"
<< endl;
}
~TDataClass()
{
cout << "TDataClass:: "TDataClass()"
<< endl;
}
};
classTBaseClass
{
public: TBaseClass()
{
cout << "TBaseClass::TBaseClass()"
<< endl;
```

```
}
~TBaseClass()
{
cout << "TBaseClass:: "TBaseClass()"
«  endl;
}
};
class TDerivedClass : public TBaseClass
{
TDataClass data; public:
TDerivedClass()
{
cout << "TDerivedClass: :TDerivedClass()"
<< endl;


cout << "Выбрасываем исключение"
<< endl;
throw "Ой! Что-то случилось..."
}
"TDerivedClass()
{
cout  <<  “TDerivedClass:: ~TDerivedClass()"
<< endl;
}
};
int main( void )
{
try
{
TDerivedClass tds; // ... }
catch ( const char* msg )
{
cout << "Пойманное исключение: " << msg << endl;
return  - 1 ;
}
return 0;
}
```

Программа выводит следующее:

```
TBaseClass::TBaseClass()
TDataClass::TDataClass()
TDerivedClass:: TDerivedClass()
Выбрасываем исключение
TDataClass:: ~TDataClass()
TBaseClass:: ~TBaseClass()
Пойманное исключение: Ой! Что-то случилось...
```

Используйте локальные объекты для представления выделяемых ресурсов

Чтобы управление исключениями C++ приносило выгоду, вы должны упаковать инициализацию и очистку некоторого ресурса соответственно в конструктор и деструктор *класса и при выделении* ресурса создавать локальные объекты этого класса. Если конструктор вашего класса выделяет ресурсы нескольких типов или исполняет код, который может выбросить исключение *после того*, как ресурс выделен, заключите каждый ресурс в свой собственный класс. Таким образом, каждый выделенный ресурс будет всегда локализован в полностью конструированном классе.

Динамические объекты

Очистка путем вызова деструкторов ограничивается полностью конструированными локальными объектами. Однако если исключение выбрасывается из конструктора объекта, создаваемого операцией *new,* память автоматически освобождается. Рассмотрите следующий пример:

```
////////////////////////////////////////////////////////////////////////////////
// NEW XCPT.CPP:  Исключения и динамические объекты        //
////////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
//
// Простой класс, выбрасывающий исключение
// из своего конструктора…
//
class  TVBClass
{
public:
TVBClass()
{
cout  <<  "TVBClass::TVBClass()"
<< endl;


throw "Ошибка в конструкторе TVBClass"
}
~TVBClass()
{
cout  <<  "TVBClass::~TVBClass()"
<< endl;
}
};
//
// Класс с TVBClass в качестве виртуального базового класса
//
class TClass : public virtual TVBClass
{
int i;
public:
```

```
TClass();
~TClass();
void* operator new( size_t );
void operator delete( void *p );
};
TClass::TClass()
{
cout << "TClass::TClass()" << endl;
}
TClass::~TClass()
{
cout << "TClass::~TClass()" << endl;
}
void* TClass::operator new( size_t size )
{
cout << "TClass: .'operator new()" << endl;
return ::new char[size];
}
void TClass::operator delete( void *p )


{
cout << "TClass::operator delete()" << endl;
::delete( p );
}
int main( void )
{
TClass *tcp;
try
{
//
// Создать динамический представитель класса,
// виртуальный базовый класс которого
// выбрасывает исключение
//
tcp = new TClass;
//
// Следующее исполняться не будет:
//
cout << "Объект создан..." << endl;
delete tcp;
}
catch( const char *msg )
{
cout << "Исключение: " << msg << endl;
}
return 0;
}
```

Этот код выводит такие сообщения:

```
TClass::operator new()
```

TVBClass::TVBClass()
TClass::operator delete()
Исключение: Ошибка в конструкторе TVBClass

Передача значений из конструктора и деструктора

Язык C++ не позволяет конструктору или деструктору возвращать значение. Однако вы можете использовать исключения, чтобы сообщить из конструктора или деструктора о состоянии ошибки. На самом деле, вы можете передать больше, чем просто значение; вы можете возвратить объект с несколькими элементами данных и элементами-функциями.

Вот пример:

```
//////////////////////////////////////////////////////////////
// XCPT_CON.CPP: "Возвращение" значения из конструктора     //
//////////////////////////////////////////////////////////////

#include <except.h>
#include <iostream.h>


class TSomeClass
{
    // ...
  public:
    struct xSomeClass
    {
      int errCode;
      xSomeClass( int err );
      {}
    };

    TSomeClass();
};


TSomeClass::TSomeClass()
{
  // ...
  if ( error_condition )
      throw xSomeClass( err_code );
}
```

```
int main( void )
{
   try
   {
    TSomeClass tsc;
    // ...
   }
   catch( TSomeClass::xSomeClass xinfo )
   {
    cout << "Код ошибки исключения :"
         << xinfo.errCode
         << endl;
   }

   // Объект создан успешно...

   return 0;
}
```

Работа с иерархиями исключений

Так как управление исключениями C++ позволяет вам предусмотреть обработчик для базового класса, который будет перехватывать любые объекты, являющиеся публичными производными базового класса, родственные исключения часто могут быть представлены в виде иерархии классов C++. Производя классы исключений из общего базового класса, вы можете использовать свойство полиморфизма, перехватывая указатель или ссылку на базовый класс. Например, для представления ошибок ввода/вывода могут использоваться следующие классы (см. рис. 9.4).

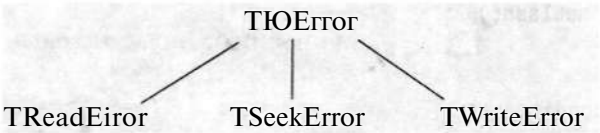


Рис.9.4. *Классы для представления ошибок ввода/вывода*

Следующий код показывает возможную реализацию:

```
////////////////////////////////////////////////////////////////////////////////
// XCPTHIER.CPP: Пример иерархии исключений...      //
////////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
class TIOError
{
public:
virtual void explain()
{
// ...объяснить возникшую проблему
}
};
class TReadError : public TIOError
{
public:
void explain();
// ...
};
```

```
class TWriteError : public TIOError
{
public:
void explain();
// ...
};
int main( void ){ try
{
//
// Выполнить I/O
//
}
catch( TIOError& ioerr )



{
ioerr.explain();
}
return 0;
}
```

Работа с предопределенными классами исключений

Существует несколько предопределенных классов, которые используются библиотекой C++ для сообщений об исключениях. В этом разделе рассматриваются классы *xmsg* и *xalloc.*

xmsg

Класс *xmsg* применяется для выдачи сообщений об исключениях. Он описан следующим образом:

```
class _EXPCLASS xmsg
{
public:
xmsg(const string _FAR &msg);
xmsg(const xmsg _FAR &msg);
"xmsg();
const string _FAR & why() const;
void raise() throw(xmsg);
xmsg& operator=(const xmsg _FAR &src); private:
string _FAR *str;
};
```

Следующий код показывает выбрасывание и перехват *xmsg.*

```
////////////////////////////////////////////////////////////////////////////////
// XMSG_.CPP:  Применение xmsg для выдачи сообщений...    //
////////////////////////////////////////////////////////////////////////////////


#include <iostream.h>
#include <cstring.h>
#include <except.h>
```

```
void func()
{
if ( something_is_wrong )
{
xmsg xx( "Зарегистрирована ошибка..." );
throw xx;
}
// ...
}
int main( void )
{
try
{
func();
// ...
}
catch( xmsg& msg )
{
cerr << "Исключение: "
<< msg.why
<< endl;
return -1;
}
return 0;
}
```

xalloc

Стандартная операция new выбрасывает исключение *xalloc,* если память не может быть выделена. Класс объявляется следующим образом:

```
class _EXPCLASS xalloc : public xmsg .
{
pudlic:
xalloc(const string _FAR &msg, size_t size);
size_t requested() const;
void raise() throw(xalloc); private:
size_t siz;
};
```

Вы должны заключать каждый запрос о выделении памяти в *try-блок* с обработчиком *catch(xalloc&).*

```
////////////////////////////////////////////////////////////////////////////////
// NEWXLLC.CPP:  Проверка xalloc при вызове new... //
////////////////////////////////////////////////////////////////////////////////
#include <except.h>
#include <iostream.h>
void func()
{
char *p = 0;
try
{
```

```
p = new char[0x100];
}
catch( xalloci xllc )
{
cerr << "Перехвачено xalloc."
}
// ..
}
```

xalloc или NULL

Прежние версии операции new при нехватке памяти возвращали значение NULL. Вы можете восстановить такое поведение этой операции, вызвав функцию *set_new_handler* с параметром NULL:

```
#include <new.h>
...
set_new_handler(NULL);
```

Перед тем, как вызывать *set_new_handler(0)*, необходимо убедиться, что никакие из библиотек, с которыми вы работаете, не применяют исключение *xalloc*. Например, версии 2.x ObjectWindows Library Борланда предполагают использование *xalloc*.

Bad cast и Bad_typeid

Исключение *Bad_cast* выбрасывается, когда операция *dynamic_cast* не может преобразовать ссылку на тип. Если операция *typeid* не может определить тип операнда, выбрасывается исключение *Badjypeid.* Более подробные сведения даются в главе "Информация о типе во время исполнения".

Использование информации о местонахождении исключения

Файл EXCEPT.H определяет три глобальных переменных, которые можно использовать для получения информации об исключении:
_throwFileName: Указатель типа char* на имя файла, из которого было выброшено исключение.
_throwLineNumer. Переменная типа unsigned с номером строки, в которой было выброшено исключение.
_throwExceptionName: Указатель типа char* на имя исключения, то есть на ограниченную нулем строку, содержащую имя типа выброшенного объекта.

Примечание: Чтобы использовать доступ к переменным имени файла и номера строки, вы должны установить опцию компилятора Exception Location Information. Для более подробных сведений см. "Исключения и опции компилятора".

Следующий пример обращается к информации о место исключения.

```
//////////////////////////////////////////////////////////////////////////////////////////////
// XCPTNAME.CPP: Доступ к информации о месте исключения   //
//////////////////////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <cstring.h>
#include <except.h>
void func( void )
{
if ( error_cond )
{
xmsg xx( "Тип ошибки: ###" )
// ...
throw xx;
}
}
int main( void )
{
try
{
func();
// ...
}
catch( ... )
{


cout << "Исключение типа "
<<  _throwExceptionName
<< " выброшено: "
<< "Строка "
<<  _throwLineNumber
<< " - Файл: "
<<  _throwFileName
<< endl;
return - 1 ; }
return 0; }
```

Пример выводит такое сообщение:

Исключение типа xmsg выброшено: Строка 14 - Файл: xcptname.cpp

Исключения и опции компилятора

Чтобы определить из IDE конфигурацию компилятора в плане управления исключениями, вы должны сделать следующее:

1. Выбрать в меню Options|Project. Появится окно диалога *Project Options.*
2. Выбрать и развернуть рубрику *C++ Options.*
3. Выбрать *Exception handling/RTTI.*

Появятся установочные параметры *Exception handling/RTTI.*

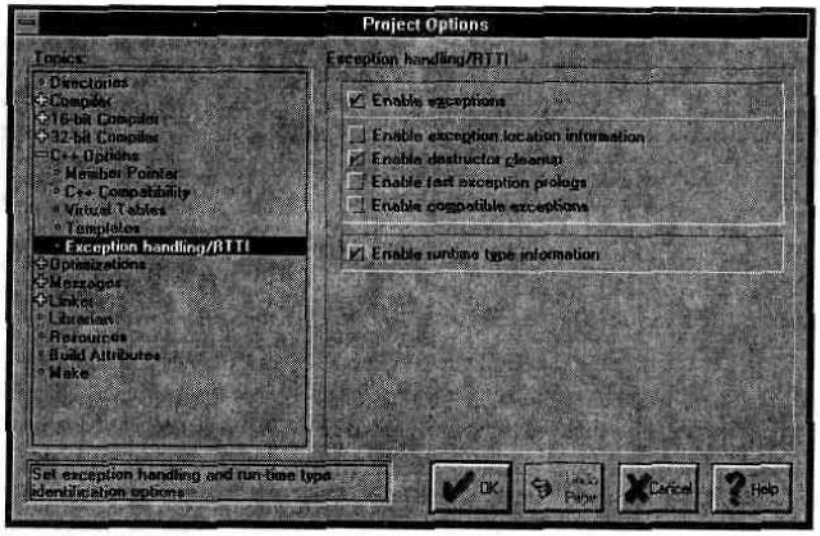


Рис. 9.5. *Опции управления исключениями C++*

В следующей таблице описывается каждая из опций и дается эквивалентный ключ командной строки.

Таблица 9.1. Опции управления исключениями

Опция	*Ключ*	*Описание*
Enable exceptions	-x	Если не установлена, компилятор генерирует сообщение об ошибке, когда встречает try-блок. По умолчанию опция установлена.
Enable exception location info	-xp	Если установлена, опция дает вам доступ к имени файла и номеру строки, где было выброшено исключение. По умолчанию — не установлена.
Enable destructor cleanup	-xd	Опция разрешает вызов деструкторов локальных объектов, созданных между точками throw и catch. По умолчанию установлена.

Примечание: При использовании компилятора с командной строкой можно в явном виде выключить эти опции, поставив '-' после соответствующего ключа. Например, *-xd-* запрещает вызов деструкторов.

Обзор структурного управления исключениями

Структурное управление исключениями в действительности является частью операционной системы Windows NT. Если строится приложение для Win32, Borland C++ обеспечивает доступ (с помощью новых ключевых слов *_try, _except* и *finally)* к механизмам операционной системы, которые управляют исключениями. Однако Borland C++ позволяет применять структурное управление исключениями и в программах, ориентированных на 16-битную среду исполнения. Другими словами, описанный ниже синтаксис можно применять в коде

приложений для Windows и DOS, так как исполнительная библиотека Borland C++ предусматривает 16-битные версии различных функций, связанных со структурным управлением исключениями.
Структурное управление предлагает две возможности: *кадрированное управление исключениями* и *завершающее управление.* Кадрированное управление позволяет вам специфицировать блок кода, который будет исполняться при возникновении исключения. Завершающее управление позволяет задать блок кода, который будет исполняться всегда, безотносительно к потоку управления.

Использование кадрированного управления исключениями (_try/_except)

Кадрированное управление исключениями включает в себя три момента:
• Во-первых, имеется блок кода, следующий за ключевым словом *_try,* заключенный в фигурные скобки. Он называется *телом защищенного кода.* Он состоит из одного или нескольких операторов, способных прямо или косвенно заявить исключение.

• За ним следует ключевое слово *_except с фильтрующим выражением* в качестве параметра.
• Наконец, имеется другой блок кода, следующий за выражением *_except* и заключенный в фигурные скобки. Его называют *блоком-обработчиком* исключения.
Ниже показан общий вид синтаксиса:

```
__try
{
/* тело защищенного кода */
}
_except( фильтрующее выражение )
{
/* блок-обработчик исключения */
}
```

Заявление исключения

Чтобы заявить исключение, можно использовать функцию *RaiseException( ).*

```
void
_cdecl _far
RaiseException(
DWORD dwExceptionCode,
DWORD dwExceptionFlags,
DWORD nNumberOfArguments,
const LPWORD lpArguments
);
```

dw ExceptionCode: Код, идентифицирующий заявленное исключение.
Dw ExceptionFlags: Показывает, является ли исключение возобновимым. Может принимать значение *EXCEPTION_CONTINUABLE* или *EXCEPTION_NONCONTINUABLE.*
nNumberOfArguments: Указывает число аргументов, передаваемых в массиве *lpArguments.*

lpArguments: Адрес массива 32-битных аргументов.

Поток управления

Если оператор в *защищенном теле кода* заявляет исключение, производится оценка *фильтрующего выражения.* Результат оценки определяет поток управления, как показано в следующей таблице:
Таблица 9.2. Возможные значения фильтрующего выражения *Значение Описание*
EXCEPTION_EXECUTE_HANDLER Управление передается на блок-обработчик исключения.
EXCEPTION_CONTINUE_SEARCH Управление на ассоциированный блок-обработчик не передается. Стек разматывается, и происходит поиск другого обработчика.
EXCEPTION_CONTINUE_EXECUTION Поиск прекращается, и управление возвращается в то место, где было заявлено исключение.
Следующий пример демонстрирует ключевые слова *_try, _except* и функцию *RaiseException.*

```
/*************************************************************************/
/* SEH.C: Иллюстрирует структурное управление исключениями */
/*************************************************************************/
#include <excpt.h>
#include <stdio.h>
#define   EXCEPTION_ERROR_CODE   0x1000L
void doSomething( void )
{
printf( "Пытаемся что-то сделать...\n" );
/*———————————*/ /* Предположим, случилось нечто
непредвиденное... */ /* Поэтому мы заявляем исключение */


printf( "Внимание! Обнаружена ошибка...\n" ); printf( "Заявляем
исключение...\n" );
RaiseException( EXCEPTION_ERROR_CODE,
EXCEPTION_CONTINUABLE, 0,     /* Число параметров */ 0 ); /* Указатель
параметров */
}
int main( void )
{
try
{
doSomething();
}
except( EXCEPTION_EXECUTE_HANDLER ) {
printf( "Исключение перехвачено...\n" ); return - 1 ;
}
return 0;
}
```

Фильтрующее выражение

Фильтрующее выражение часто вызывает функцию, называемую *функцией фильтра.* Эта функция обычно принимает один или несколько параметров, описывающих исключение, и возвращает одно из трех возможных значений *фильтрующего выражения. Фильтрующее выражение* и *блок обработки* исключения могут получить доступ к информации об исключении, вызывая следующие функции:

GetExceptionCode(): Возвращает код, идентифицирующий заявленное исключение.

Get ExceptionInformation(): Возвращает указатель на структуру, содержащую детальное описание исключения.

Пример показывает применение функции-фильтра:

```
/*************************************************************************/
/* SEH_FLTR.C: Использование фильтрующей функции     */
/*************************************************************************/
#include <excpt.h>
#include <stdio.h>
#define MY.EXCEPTION 0x0000FACE
void doSomething( void )
{
/* Предположим, случилось нечто непредвиденное...  */
/* Поэтому мы заявляем исключение */
printf( "doSomething(): Ошибочное состояние!\n" );
printf( "RaiseException() \n" );
RaiseException( MY_EXCEPTION, /* Код */ EXCEPTION_CONTINUABLE,
0,   /* Число параметров       */
0 ); /* Указатель параметров     */
printf( "Похоже, мне разрешили продолжать!" );
}
DWORD ExceptionFilter( DWORD dwCode )
{
printf( "Фильтр исключения: Код=%ld\n", dwCode );
if ( dwCode = MY_EXCEPTION )
return   EXCEPTION_EXECUTE_HANDLER;   else
return    EXCEPTION_CONTINUE_SEARCH;
/*-[Примечание]-----------------------------------*\
|  Можно  возвратить  EXCEPTION_CONTINUE_EXECUTION,  |
| что возобновит выполнение с точки исключения       |
| и прервет поиск обработчика...                     |
 \*---------------------------------------------------*/
}


int main( void )
{
try {
doSomething();
}
```

```
except( ExceptionFilter( GetExceptionCode() ) )
{
printf( "Исключение перехвачено...\n" );
}
return 0;
}
```

Перехват исключений процессора

В приложениях, работающих под Windows NT (и частично под Win32s) можно использовать кадрированное управление исключениями для перехвата процессорных исключений. Например, с помощью конструкции *_try/_except* может быть обработано печально известное *general protection fault* (общее нарушение защиты). Для этих исключений функция *GetExceptionCodeO* возвращает одно из возможных значений, определенных в файлах *excpt.h* и *winbase.h.* Вот некоторые из них:

EXCEPTION_ACCESS_VIOLATION
EXCEPTION_ARRAY_BOUNDS_EXCEEDED
EXCEPTION_FLT_DENORMAL_OPERAND
EXCEPTION_FLT_DIVIDE_BY_ZERO
EXCEPTION_FLT_INEXACT_RESULT
EXCEPTION_FLT_INVALID_OPERATION
EXCEPTION_FLT_OVERFLOW
EXCEPTION_FLT_STACK_CHECK
EXCEPTION_FLT_UNDERFLOW
EXCEPTION_INT_DIVIDE_BY_ZERO
EXCEPTION_INT_OVERFLOW
EXCEPTION_PRIV_INSTRUCTION
EXCEPTION_IN_PAGE_ERROR
EXCEPTION_ILLEGAL_INSTRUCTION
EXCEPTION_NONCONTINUABLE_EXCEPTION
EXCEPTION_STACK_OVERFLOW
EXCEPTION_INVALID_DISPOSITION

Следующий пример показывает, как с помощью кадрированного управления исключениями можно обрабатывать нарушения прав доступа.

ACS_XCPT.C: Обработка нарушений доступа...

Этот пример выдаст (в лучшем случае) сообщение 'Protection violation', если его скомпилировать, скомпоновать и запустить как 16-битное приложение. Построенный и запущенный как 32-битное приложение, он распознает нарушение доступа, вызванное копированием, использующим нулевой указатель...

Для компиляции/компоновки примените команду

BCC32 -W -v ACS_XCPT.C

```
\*****************************************************./
#if  !defined(STRICT)
#define        STRICT
#endif
#include <windows.h>
#include <excpt.h>
```

```
#include <string.h>
DWORD ExceptionFilter( DWORD dwCode,
EXCEPTION_POINTERS *pXcptInfo )
{
if ( dwCode = STATUS_ACCESS_VIOLATION )
{
return   EXCEPTION_EXECUTE_HANDLER;
}
else
{
return   EXCEPTION_CONTINUE_SEARCH;
}
}
int PASCAL WinMain( HINSTANCE hInstance,


HINSTANCE hPrevInstance, LPSTR lpszCmdline, int nCmdShow )
{
„try
{
char *p = NULL;
strcpy( p, "Oh la la!!" );
}
_except( ExceptionFilter( GetExceptionCode(),
GetExceptionInformation() ) )
{
MessageBox( NULL,
"Нарушение доступа, завершение...", "Код обработчика исключения...",
MB_OK|MB_TASKMODAL );
}
return 0;
}
```

Применение завершающих обработчиков исключений *(_try/_finally)*

Завершающее управление включает в себя две компоненты:

• Во-первых, имеется блок кода, следующий за ключевым словом *_try,* заключенный в фигурные скобки. Он называется *телом защищенного кода.* Он состоит из одного или нескольких операторов, способных прямо или косвенно заявить исключение.

• За ним следует ключевое слово *_finally с* блоком кода, заключенным в фигурные скобки и называемого *завершающим блоком.* Операторы *завершающего блока* исполняются всегда, когда управление покидает *тело защищенного кода.*

Следующий пример показывает общий синтаксис:

```
_try
{
/* тело защищенного кода */
```

```
}
„finally
{
/* завершающий блок */
}
```

Нормальное и аномальное завершение

Если управление последовательно переходит от защищенного тела к завершающему блоку, говорят, что тело защищенного кода завершилось нормально. Аномальное завершение происходит, когда управление покидает защищенное тело вследствие

- исполнения оператора goto, break или continue
- вызова функции *long/ump()*
- заявленного исключения

Завершающий блок может вызвать функцию *AbnormalTermination(),* чтобы выяснить, как завершился защищенный код. Пример демонстрирует обработку завершения:

```
/*********************************************************/
/* TRM_HNDL.C: Использование обработчиков завершения.. */
/*********************************************************/
#include <stdio.h>
#include <excpt.h>
void func( void )
{
/*
6rab_A_Resource();
*/
„try


{
/*
Что-то случается...
*/
return;
}
„finally
{
/*
Release_The_Resource();
*/
printf( "_finally: Защищенный блок завершился %s\n",
AbnormalTermination() ? "аномально" : "нормально" );
}
}
int main( void )
{
func();
return 0;
}
```

Комбинированное применение кадрированной >I завершающей обработки

|8ы може-те комбинировать оба варианта обраббткн^ йспольз^я'вяожёк-н&е-блоки:

```
void fи по/jQ     /* НВ: Если заявляется          */
{                 7*  исключение, in рядек       */
   __try          /*  исполн-ення такой:         */
   {              V*  f,' ~ Выражение фндьтра    */
    __try         /*  2.  Обработчик исключения  */
    {             /щ  3.  Обработчик завершения  */
```

```
      /* Защищенный код... */
    }
    __except( фильтрующее_выражение )
    {
      /* Обработчик исключения */
    }
  }
  __finally
  {
   /* Обработчик завершения */
  }
}
```

```
voiЛ -fiincBC)     /* NB; Если заявляется          •/
{                 •. 7*  исключение, порядок      •/
   _try           if.*  исполнения такой:         •/
   h .....         /* 1.  Выражение" фильтра      */
   "_try           /* 2.  Обработчик завершения   •/
    {              /* 3.  Обработчик исключения *' */

      /;* Защищенный код-*/
    }
    „finally
    {
      /* Обработчик завершения */
   1
  }
  __except( фильтрукщее^выражение )
  {
   /* Обработчик исключения */
  }
}
```

шшшшшяш

Использование структурного управления и управления исключениями C++

• Управление исключениями C++ *(try/throw/catch)* не может применяться в модулях на языке C.

• Также не может применяться синтаксис управления завершением *(_try/finally)* в модулях C++.

• Кадрированное управление C может быть применено в модуле на C++. Для реализации кадрированного обработчика в модуле C++ используйте пару ключевых слов *try/_except* или *try/except.*

Заключение

В этой главе вы познакомились с возможностями, которые предоставляет программистам управление исключениями. Оно включает в себя последовательный механизм для обработки и сообщения об исключительных ситуациях, а также позволяет гарантировать, что при возникновении таких ситуаций выделенные ресурсы будут возвращаться системе. Некоторые компоненты исполнительных библиотек C++ используют управление исключениями (напр., операция new, класс ANSI C++ *string),* и эта тенденция, скорее всего, будет расширяться по мере того, как комитет ANSI/ISO по C++ будет формулировать детали реализации языка. Поэтому пользуйтесь преимуществами, которые дает этот механизм, и *ловите* момент!

Глава 10
Информация о типе во время исполнения и операции приведения типа

Информация о типе во время исполнения (Runtime Type Infomation, RTTI) является механизмом, который позволяет определять тип объекта во время исполнения программы. Это может оказаться особенно полезным в иерархии классов, где указатель или ссылка на объект базового класса часто указывает на представитель производного от него класса. Средства для определения типа объекта дают возможность производить надежные приведения типов. Другими словами, имея информацию о типе, можно проверять возможные преобразования и разрешать только безопасные и осмысленные. В этой главе рассматривается реализация, синтаксис и использование RTTI, а также новый стиль приведения типов в C++. Хотя эти черты являются сравнительно недавними добавлениями к языку, они позволят вам разрешать старые проблемы, такие, как приведение типа от виртуального базового класса к производному классу.

Программирование с использованием RTTI

RTTI реализуется не для всех типов C++, но только для классов, имеющих одну или несколько виртуальных функций (т.е. для *полиморфных типов).* Тот же синтаксис, однако, может все же применяться и к не-полиморфным типам (включая встроенные, такие, как 'char' или 'double'), хотя идентификация тогда будет производиться статически, а на динамически. В этом разделе рассматривается синтаксис использования RTTI.

Операция typeid и класс Type info

Для доступа к RTTI имеются специальные операция и класс:

• Новая операция *typeid* принимает в качестве параметра имя типа или выражение и возвращает ссылку на объект типа *Type_info:*

typeid(имя_типа) typeid(выражение)

Результат:

const Type_info&

• Класс *TypeJ.nfo,* определенный в файле *typeinfo.fi,* содержит информацию о типе.

Когда передается выражение, представляющее собой указатель или ссылку на полиморфный тип, операция typeid исследует объект во время выполнения и возвращает динамическую информацию о его типе. Если операции typeid передается выражение, представляющее собой ссылку на не-полиморфный тип, она возвращает ссылку на объект Type_info, содержащий статический тип выражения (другими словами, исследуется выражение, а не тип объекта, на который оно ссылается).

Исключение *Bad_typeid*

Если операция *typeid* не может определить тип операнда, выбрасывается исключение *Bad_typeid.*

Применение typeid для сравнения типов

Можно применять операцию *typeid для* сравнения типа объектов во время выполнения программы. Следующий код показывает пример такого сравнения. Обратите внимание на разницу между полимофными и не-полиморфными типами.

```
//////////////////////////////////////////////////////////////////////////////
// TYPECMP1.CPP:  Сравнение типов с помощью typeid…     //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <typeinfo.h>
//
// Не-полиморфный тип
//
class Tbase1
{
public:
// …
};
//
// Не-полиморфный тип
//
class TDerived1 : public TBase1
{
public:
// …
};
int main( void )
 {
TDerivedl d;
TBase1  &br = d;
cout << "typeid(br) "
<< (typeid(br) == typeid(TBasel) ? "==" : "!=")


<< " typeid(TBase1)"
<< endl;
cout << "typeid(br) "
<< (typeid(br) == typeid(TDerivedl) ? "==" : "!=")
<< " typeid(TDerived1)"
<< endl;
return 0;
}
```

Пример выводит:

```
typeid(br) == typeid(TBase1) typeid(br)  != typeid(TDerived1)
```

Error! Bookmark not defined.

TBase1 не является полиморфным. Хотя *br* в действительности ссылается на класс *TDerived1,* его тип идентифицируется как *TBase1*.*

```
//////////////////////////////////////////////////////////////////////////////
// TYPECMP2.CPP:  Сравнение типов с помощью typeid…     //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <typeinfo.h>
//
// Полиморфный тип
//
```

```
class TBase2
{
public:
virtual ~TBase2()
{}
// ...
};
//
// Полиморфный тип
//
class TDerived2 : public TBase2


{
public: // ...
};
int main( void ) {
TDerived2 d;
TBase2  &br = d;
cout << "typeid(br) "
<< (typeid(br) == typeid(TBase2) ? "==" : "!=") << " typeid(TBase2)" « endl;
cout  << "typeid(br) "
<< (typeid(br) == typeid(TDerived2) ? "==" :  "!=") << " typeid(TDerived2)" << end1;
return 0; }
```

Этот пример выводит:

typeid(br) != typeid(TBase2) typeid(br) == typeid(TDerived2)

На этот раз *TBase2* — полиморфный тип. Поэтому *br* правильно идентифицируется как ссылка на представитель класса *TDerived2.*

Использование Type_info

Ниже приводится слегка отредактированное описание класса *Type_info.*

```
class  _rtti  Type_info
{
public:


tpid    _far *   tpp;
private:
_cdecl Type_info(const Type_info _FAR &);
Type_info   &_cdecl operator=(const Type_info _FAR &);
public:
virtual _cdecl~Type_info();
int _cdec1 operator==(const Type_info _FAR &) const;
int_cdec1 operator!=(const Type_info _FAR &) const;
int _cdec1 before(const Type_info _FAR &) const;
const char _FAR *_cdecl name() const;
};
```

• Конструктор копии и операция присваивания объявлены как частные, чтобы сделать невозможным случайное копирование или присваивание представителей класса.

• Виртуальный деструктор делает *Type_info* полиморфным.

• Перегруженные операции == и != позволяют вам сравнивать два объекта *Type_info* (два результата вызовов операции *typeid),* как показывают приведенные примеры TYPECMP1.CPP и TYPECMP2.CPP.

Type_info::before(const Type_info&)

Метод *before()* из *Type_info* зависит от реализации языка и отражает упорядоченность типов. Версия *before()* Borland C++ производит просто побуквенное сравнение имен двух типов.

```
if( typeid( TypeA ).before(typeid( TypeB ) ) )
```

эквивалентно

```
if(strcmp( typeId( TypeA ).name(), typeid( TypeB ).name() ) < 0 )
```

Type_info::name()

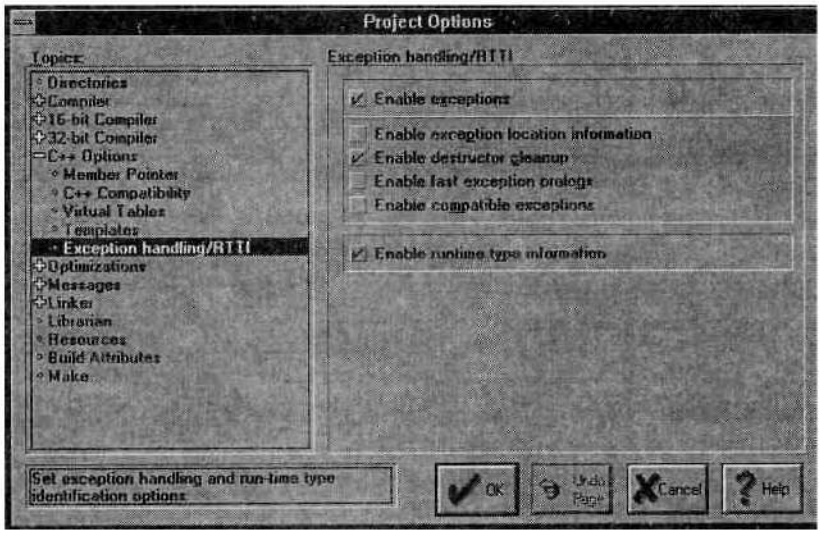
Метод *пате()* из *Type_info* возвращает строку, представляющую имя типа, описываемого объектом *Type_info.*

Использование RTTI и опции компилятора

В компиляторе Borland C++ использование RTTI по умолчанию разрешено. Если вы работаете с IDE, то можете изменить опцию RTTI, выполнив следующие шаги:

1. Выберите меню Options|Project. Появится диалог *Project Options.*
2. Выберите и разверните рубрику C++ *Options.*
3. Выберите Exception Handlmg/RTTI.

Появится окно *Exception Handling/RTTI* с селекторами опций.



Пользователи компилятора с командной строкой могут применить ключ -RT (принимается по умолчанию), чтобы разрешить опцию в явном виде. Ключ -RT- запрещает использование RTTI.

Модификатор *__rtti*

Можно обеспечить генерацию кода с RTTI для некоторого класса вне зависимости от конфигурации компилятора, если применить в определении класса модификатор *__rtti.* Например:

```
class _rtti TClass
{
```

```
 public:
virtual TClass();  // Полиморфный!
                };
```

Корректное применение RTTI

Хотя RTTI является весьма гибким *средством*, *оно* НЕ должно применяться вместо виртуальных функций. Следующий код является примером некорректного *использования* RTTI.

```
#include <typeinfo.h>

class TBaseWnd { public: virtual ~TBaseWnd(); };
class TMainWnd : public TBaseWnd {};
class TDlgWnd  : public TBaseWnd {};

void Create( TBaseWnd *pWnd )
{
   //
   // Очень расточительное применение RTTI - использование
   // виртуальных функций является более элегантным и
   // связано с меньшими издержками!
   //
   if     ( typeid(pWnd) == typeid(TMainWnd) )
    CreateWindow(...);
```

```
    else if ( typeid(pWnd) == typeid(TDlgWnd)  )
     CreateDialog(...);
    else

    // ...
}
```

Используйте RTTI, если вы расширили библиотечный класс путем создания производного класса, и было невозможно или бессмысленно добавлять виртуальные функции к базовым классам иерархии. См. следующий пример кода:

```
#include <typeinfo.h>

class TMyMainWnd : public TMainWnd
{
   public:
    virtual void myUniqueMethod();
}


void WindowActivated( TBaseWnd *pActiveWnd )
{
   if ( typeid(pActiveWnd) == typeid(TMyMainWnd) )
    ((TMyMainWnd*)pActiveWnd)->myUniqueMethod();

   // ...
}
```

Другими словами, используйте RTTI, когда невозможно определить тип объекта при компиляции и другие механизмы C++ (такие, как позднее связывание) не могут быть использованы, чтобы обеспечить правильное обращение с соответствующими типами.

новый стиль приведения типов

Borland C++ поддерживает новый синтаксис приведения типов, включающий операцию *dynamic_cast.* Эта операция, пользуясь преимуществами

RTTI, обеспечивает надежные преобразования типа, которые проверяются во время исполнения. В этом разделе рассматривается синтаксис и цели применения этих новых операций приведения.

Обзор новых форм приведения типов

В разделе 5.4 "Справочного руководства по C++ с примечаниями" (ARM) есть следующее замечание относительно традиционной нотации приведений, применяемой для явного преобразования типов:
"Во многих случаях лучше избегать явных преобразований типов. Использование приведений подавляет проверку типа, производимую компилятором, и может, таким образом, давать неожиданные результаты, если только программист не был совершенно прав в своих действиях."
Иными словами, традиционная нотация приведений позволяет программисту производить (ненамеренно) непроверяемые преобразования типов, которые будут отказывать во время исполнения.

Другим недостатком традиционной нотации приведений является то, что синтаксис не отражает действительных намерений программиста. Например, приведение типа могло быть применено, если программисту требовалось:

• Изменить точку зрения компилятора на некоторый объект в памяти, не изменяя его содержимого. В следующем примере приведение i к типу *unsigned* совершенно меняет результат оператора *if.*

```
int i = -2;
if ( i < -1 )
cout << "i < - 1 "
<< endl;
if ( unsigned(i) < -1 )
cout << "i < - 1 "
<< endl;
```

• Изменить действительный результат оценки выражения. В следующем примере адрес, передаваемый во втором вызове функции *Call4Msg(),* отличается от адреса в первом из-за приведения типа.

```
class A
{
public:
virtual void msg();
}:
class B : public virtual A
{
public:
void msg();
}:
class C : public virtual A
{
public:
void msg();
}:
class D : public B, public C
{
public:
void msg();
}:
void Call4Msg( A* ap )
{
ap->msg();
}
int main( void )
{
D d;
Call4Msg( &d ); Call4Msg( &(C)d );
return 0;
}
```

• Устранить сообщение об ошибке при компиляции. См. следующий пример:

```
const int TblSize = 0x100; const int *iTbl;
void CreateTbl( void )
{
int *ip = new int[TblSize];
// ...
iTbl = ip;
}
void DestroyTbl( void )
{
delete [ ] ( int* )iTbl;
}
```

При вызове операции *delete* необходимо приведение типа к *int*.* В противном случае генерируется сообщение об ошибке: Cannot convert 'const int*' to 'void*' in function DestroyTbl().

Традиционная нотация приведений страдает также от другого ограничения: указатель на базовый класс B не может быть явно преобразован в указатель на производный класс D, если B является виртуальным базовым классом. Рассмотрите такой пример:

```
class A {};                                  //    A
class B     public virtual A {};         //   /    \
class C      public virtual A {};       // B         C
class D     public B,  public C   {}; //    \     /
                                         //       D
void func( A *ap )
{
//
// Следующая строка генерирует сообщение об ошибке:
// "Cannot cast from 'A *' to 'D *'
//  in function func(A *)"
//


D* dp = (D*)ap; '//...
}
int main( void )
{
D d;
func( &d );
// ... return 0;
}
```

Новый стиль приведений решает проблемы традиционного приведения типов,

• давая ясный и точный синтаксис, документирующий намерения программиста;

• предусматривая обнаружение ошибок как при компиляции, так и во время исполнения;

• устраняя невозможность преобразования указателей на базовый виртуальный класс в указатели на тип, производный от базового класса.

Применение dynamic_cast

Операция *dynamic_cast* позволяет вам производить надежные преобразования типа: если приводятся типы полиморфных объектов, действительность данного преобразования проверяется во время исполнения.

Синтаксис:

dynamic_cast<T>(v)

где:

T является типом ссылки, типом указателя или *void*.*

v *является ссылкой, если* T является типом ссылки; в противном случае v является указателем.

Результат операции *dynamic_cast* имеет тип *T,* если преобразование успешно. При неудаче операция *dynamic_cast.*

- Выбрасывает исключение *Bad_cast,* если T — ссылка.
- Возвращает нулевой указатель, если T — указатель.

Примечание: *dynamic_cast* использует RTTI. Поэтому убедитесь, что:

- v является полиморфным типом
- при компиляции разрешена опция RTTI

Рассмотрение примера с *dynamic_cast*

Следующий пример сопоставляет традиционный стиль приведений и операцию *dynamic_cast.*

```
//////////////////////////////////////////////////////////////////////////////
// DYN_CAST.CPP:  Применение операции dynamic_cast //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
//
// Предположим, что TAppObject является законченным
// функциональным классом, имеющимся в библиотеке.
//
class TAppObject
{
public:
virtual ~TAppObject(){}
// ...
};
//
// Функция GetAppObject() возвращает указатель
// на объект TAppObject данной задачи.
//
TAppObject* GetAppObject( void );


//
// Скажем, нам нужно усовершенствовать класс TAppObject
// и мы, таким образом, создаем новый класс, производный
```

```
                // от TAppObject, в котором мы заменяем некоторые из
                // методов TAppObject, добавляем новые элементы данных
                // и элементы-функции.
                //
                class TMyAppObject : public TAppObject
                {
                // ...
                public:
                virtual void MyMethod();
                // ...
                };
                //
                // Эта функция иллюстрирует традиционную (рискованную)
                // нотацию приведений, так же как и новый, более безопасный
                // стиль приведения типов…
                //
                void func()
                {
                #if    !defined(__USE_CHECKED_CAST)
                //
                If Здесь мы используем старый стиль приведений типа
                //
                TMyAppObject*
                objPtr = (TMyAppObject*)GetAppObject();
                //
                // Если 'objPtr' не указывает на самом деле // на представитель класса
                'TMyAppObject', // следующий вызов позорно провалится!
                // objPtr->myMethod();
// ... #else



                //
                // Здесь мы используем новый (безопасный) стиль
                // приведения
                //
                TMyAppObject*
                objPtr = dynamic_cast<TMyAppObject*>(GetAppObject());
                if ( objPtr )// Подтвердить действительность
                // преобразования!
                {
                objPtr->myMethod();
                // ...
                }
#endif
                }
```

При нисходящем приведении типа применяйте dynamic_cast

Применяйте операцию *иуритсjcast* во всех слулаях, когда вам нужно выполнит* *нисходящее приведение* типа (т.е. -преобразование, .указателя/ссылки.: на базовый класс в указатель/ссылку на производный от негсгйтсс).

Нисходящее приведение виртуального базового класса

При традиционной нотации приведений типа язык C++ не допускает преобразования виртуального базового класса в производный класс. Однако с помощью RTTI вы можете осуществить нисходящее приведение виртуального базового класса, используя операцию *dynamic_cast,* при условии, что Тип является полиморфным и преобразование недвусмысленно. Рассмотрите следующий пример:

```
//////////////////////////////////////////////////////////////////////////////////////////////
// VB_DNCST.CPP: dynamic_cast и виртуальный базовый класс  //
//////////////////////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
#include <typeinfo.h>
class A {
public:      //
virtual ~A(){}
// RTTI поддерживается только
// для полиморфных типов.
};
//
class B : public virtual A {};
class C : public virtual A {};
class D : public B, public C {};
void func( A& refA )
{
try
{
D &d = dynamic_cast<D&>( refA );
//
// Использование d…
//
}
catch( Bad_cast& )
{
cout << "Перехвачено Bad_cast..."
<< endl;
}
}
int main( void )
{
D d;
```

```
    func( d );
    // ...
return 0;
}
```

Перекрестное приведение типа

Операция *dynamic_cast* позволяет производить *перекрестное приведение* типа; то есть вы можете безопасно преобразовывать друг в друга классы, которые при компиляции кажутся совершенно не связанными друг с другом. Рассмотрим иерархию, показанную на рис. 10.2.

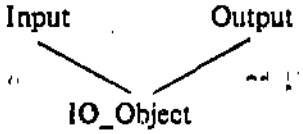


Рис. 10.2. *Перекрестное приведение типа*

Если для преобразования указателя на Input в указатель на Output используется традиционная форма приведения типа, то:

- Преобразование допустимо, но, скорее всего, недействительно.
- Результат преобразования бесполезен (даже если и действителен), так как не производится никакой настройки указателя.

Можно, однако, применить операцию *dynamic_cast,* чтобы:

- Проверить действительность преобразования.
- Произвести преобразование с необходимой настройкой указателя.

Рассмотрите следующий пример:

```
//////////////////////////////////////////////////////////////////////////////
// XIERCST.CPP:  Перекрестное приведение типа... //
//////////////////////////////////////////////////////////////////////////////
#include <iostream.h>
class Input
{
public:
virtual void readData();
};
class Output
{
public:

virtual void writeData();
};
class IO_Object : public Input, public Output
{
};
void func( Input *pi )
{
//
```

```
// Может быть, pi в действительности указывает
// на IO_Object! Давайте посмотрим…
//
Output *po = dynamic_cast<0utput*>(pi); if ( po ) po->writeData();
// …
}
int main( void )
{
Input i;
func( &i );
I0_0bject io;
func( &io );
// …
return 0;
}
```

Использование static_cast

Операцию *static_cast* можно использовать для выполнения преобразований между

- целыми типами;
- целыми и вещественными типами;
- целыми и перечисляемыми типами;
- указателями и ссылками на объекты иерархии, при условии, что преобразование однозначно и не связано с нисходящим приведением виртуального базового класса.

Синтаксис: static_cast<T>(v) где:

T является типом ссылки, указателя, арифметическим или перечисляемым типом.

v сводится к представителю указателя, ссылки, арифметического или перечисляемого типа.

Результат операции *static_cast* имеет тип *T.*

Следующий пример иллюстрирует использование операции *static_cast*

```
//////////////////////////////////////////////////////////////////////////////
// STA_CAST.CPP: Использование операции static_cast //
//////////////////////////////////////////////////////////////////////////////
void func( void )
{
int i;
long   1;
//
// ( целое <-> целое )
//
1 = static_cast<long>(i);
float f;
//
// вещественное <-> целое
//
f = static_cast<float>(i);
```

```
char c;
enum Managers { Matt, Jen, Jerry, Georgia, Jeff };
Managers mgr;
//
// enum <-> целое
//
c = static_cast<char>(mgr); }
```

Операция *static_cast* может применяться для преобразования между указателями (ссылками) на базовый и производный классы иерархии. Преобразование производится во время компиляции.

```
class Base {};
class Derived : public Base {};
void func( void ) {
Derived d;
// производный -> базовый
Base *bp = static_cast<Base*>(&d);
Base b;
// базовый -> производный
Derived &dr = static_cast<Derived&>(b);
// ...
}
```

dynamic_cast и static_cast

Для преобразований между указателями и ссылками на объекты в пределах иерархии может применяться как *dynamic_cast*, так и *static_cast*. Применение *dynamic_cast* предпочтительнее, так как

- *dynamic_cast* надежнее, особенно при нисходящих приведениях типа
- *dynamic_cast* не приводит к дополнительным расходам, если преобразование может быть безопасно произведено во время компиляции.

Другими словами, *dynamic_cast* будет генерировать тот же код, что и *static_cast*, если имеется такая возможность;

- *dynamic_cast* допускает нисходящее приведение виртуального базового класса;
- *dynamic_cast* проверяет и производит перекрестные приведения типа

Однако вы должны применять *static_cast*, если преобразование относится к указателям или ссылкам на не-полиморфные объекты.

Использование const_cast

Операция *const_cast* может применяться для того, чтобы удалить или добавить квалификаторы типа *const* или *volatile.*

Синтаксис: const_cast<T>(v) где:

Г является типом.

v сводится к представителю типа, отличающегося от Г модификаторами *const* и *volatile.*

Результат операции имеет тип *T.*

Следующий пример иллюстрирует операцию приведения *const_cast.*

```
//////////////////////////////////////////////////////////////////////////////////////
// CNST_CST.CPP: Применение операции const_cast...      //
//////////////////////////////////////////////////////////////////////////////////////
const int TblSize = 0x100; const int *iTbl;
void CreateTbl( void ) {
int *ip = new int[TblSize];
// ... iTbl = ip;
 }


void DestroyTbl( void )
{
delete  []  const_cast<int*>(iTbl);
}
```

Использование reinterpret_cast

Операция *reinterpret_cast* может использоваться для того, чтобы изменить точку зрения компилятора на тип объекта; однако операция не модифицирует сам объект.

Синтаксис: reinterpret_cast<T>(v) где

T может быть типом указателя, ссылки, целым (char, short, int, long, enum) или вещественным (float, double, long double) типом.

v может быть представителем типа указателя/ссылки, целой или вещественной переменной, если *T* является типом указателя/ссылки, v может быть указателем/ссылкой, если T является целым или вещественным типом.

Результат операции имеет тип *T.*

Можно применять *reinterpret_cast* для преобразования целого в указатель и обратно. Например:

```
#include <windows.h>
void func() {
DWORD dwData;
LPSTR lpszStr;
dwData = reinterpret_cast<DWORD>(lpszStr); lpszStr = reinterpret_cast<LPSTR>(dwData);
// ...
}
```

Вы можете также применять *reinterpret_cast* для преобразования указателей одного типа в другой, при условии, что настройки указателя не требуется. Таким образом, *reinterpret_cast* нельзя использовать для преобразования между указателями или объектами внутри иерархии

Примечание: Компилятор НЕ будет генерировать сообщение об ошибке, если вы преобразуете с помощью reinterpret_cast указатель/ссылку базового класса в производный. Результат, однако, будет неверным. Например:

```
// Законно, но неправильно!
Derived *dp = reinterpret_cast<Dreived*>(a_base_ptr);
```

Заключение

Как и управление исключениями, RTTI является сравнительно новым добавлением к языку C++, позволяющим строить более надежные и "выносливые" приложения. Новые операции приведения типа помогают проверить действительность явных преобразований типа и проясняют их цель. Конечно, реализация этих нововведений связана с некоторыми издержками, но если они применяются правильно, выгоды от их использования перевешивают возможные недостатки.

Глава 11
Borland C++: дополнительные возможности и расширения

Книга эта посвящена, в основном, собственно языку C++ и его реализации в Borland C++. Особенности этого пакета Борланда затрагивались в предыдущих главах лишь постольку, поскольку нужно было заставить его компилировать приведенные примеры программ. Но вряд ли можно написать серьезное приложение на "чистом" C++; современные приложения настолько сложны, что программисту просто необходимо использовать, например, готовые библиотеки, реализующие функции достаточно высокого уровня, и вспомогательные инструменты, такие как редактор ресурсов. *Мало* иметь хороший компилятор. Должна еще существовать определенная "инфраструктура" или окружение. Поэтому нам показалось целесообразным дать в этой главе хотя бы краткую сводку тех возможностей (помимо языковых), которые предоставляет программисту Borland C++ версий 4.0 и 4.5 для разработки реальных приложений (особенно для Windows) и которые не были освещены в книге. Это не более чем ознакомительный материал, но он может быть полезен в том плане, что программист, в том числе новичок в C++ и в Windows, будет знать, какую дополнительную информацию ему стоит поискать.

Тот расширенный сервис, который предлагает Borland C++, можно разбить на две категории: стандартные библиотеки и вспомогательные инструменты.

Библиотеки

В секции Standard Libraries (Стандартные библиотеки) окна Target Expert вы, конечно, обратили внимание на опции OWL и BWCC. В версии 4.5 там есть еще три: OCF, OLE и VBX.

OWL

Эта аббревиатура означает ObjectWindows Library, т.е. "объектная библиотека Windows". Библиотека содержит определения классов C++, реализующих интерфейс с Windows. Понятно, что классы C++ гораздо лучше подходят для моделирования архитектуры Windows, чем средства обычного C. Графическое окно является, можно сказать, классическим предметом для применения объектных методов программирования, почти готовым объектом. Помимо того, что классы OWL инкапсулируют функции API Windows, OWL предоставляет программисту некий каркас приложения, на основе которого он может разрабатывать свои производные классы в соответствии со своими потребностями.

Ниже дается краткая сводка категорий, на которые можно подразделить имеющиеся классы OWL:

• Окна. Все классы окон производятся от TWindow, который обеспечивает базовый интерфейс объектов-окон.

• Окна с обрамлением. Производный от TWindow класс TFrameWindow обычно используется для главного окна программы.

• Окна MDI. Имеются три класса, полностью поддерживающих интерфейс сложных документов (MDI): TMDIFrame, TMDICHent и TMDIChild.

• Графические классы. Они инкапсулируют весь интерфейс графического устройства Windows (GDI). Это классы TClientDC, TPaintDC и другие, служащие для конструирования объектов графического контекста устройства.

• Классы объектов GDI. С этими классами ассоциируются все объекты GDI: перья, кисти, шрифты, палитры и прочее. Например, перо Windows представляется как объект класса TPen.
• Декорированные окна. Эти классы упрощают программирование инструментальных линеек, строк индикации состояния и тому подобных вещей.

• Панели диалога. Класс TDialog обеспечивает интерфейс с диалогами Windows, созданными с помощью утилиты Resource Workshop или файлов сценария ресурсов.
• Стандартные диалоги. Классы служат для использования всех стандартных диалогов Windows: подтверждений, поиска/замены, опций печати, выбора шрифта, выбора цвета.
• Классы органов управления. Класс TControl позволяет использовать в окнах три вида управляющих элементов: стандартные, Widgets и декорирующие. Также поддерживаются специальные элементы управления Борланда (BWCC, см. следующий раздел) и 3D-элементы Microsoft.
• Классы модулей и приложений. Класс TApplication, производный от TModule, инкапсулирует такие функции приложения, как процедуры при запуске, циклы приема сообщений и обработку ошибок. Класс TD11 обеспечивает поддержку сходных функций для динамических библиотек.
• Классы документов и их представлений. Сюда относятся классы TDocManager, TDocument и TView, реализующие модель документ/вид при вводе и выводе. Эта модель позволяет работать с документами и их представлениями независимо. Можно ассоциировать с некоторым документом различные представления, или "виды" — например, графику можно отображать визуально, а можно в виде списка координат, который может редактироваться как текст.
• Прочие классы, реализующие разнообразные функции. Класс TClipboard, например, упрощает обмен информацией с буфером Clipboard. Классы проверки действительности служат для контроля корректности вводимых форматированных данных. Классы меню помогают при создании динамических меню.

BWCC

Как уже упоминалось, BWCC означает "специальные управляющие элементы Борланда" (Borland Windows Custom Controls). Это те красивые кнопки и прочие атрибуты, которые вы видите в диалоговых панелях IDE.
Библиотека эта может использоваться независимо, а может применяться совместно с OWL, которая обеспечивает в этом случае объектно-ориентиро-5анный программный интерфейс.

VBX

VBX это расширенный набор органов управления, используемый в Visual BASIC. Это высоко функциональные управляющие элементы, которые при программировании в VB вы просто выбираете из предлагаемой палитры и вставляете куда нужно. Существуют десятки различных элементов, поставляемых как Microsoft, так и другими производителями.
До недавнего времени почти невозможно было применять элементы VBX в программах на C или C++, так как они чрезвычайно тесно связаны с исполняющей системой Visual BASIC. Библиотеки Борланда VBX и OWL подставляют себя на

место исполняющей системы и с помощью низкоуровневого кода и некоторых классов делают элементы VBX непосредственно доступными для программ C++. В Borland C++ 4.0 можно было использовать VBX-элементы только в 16-битных программах. В версии 4.5 эта библиотека стала стандартной, так как появилась и 32-битная ее версия. 16-битную программу, применяющую VBX, можно теперь перевести в Win32 путем простой перекомпиляции.

OLE 2 и OCF

Эти опции имеются только в версии 4.5.
OLE расшифровывается как Object Linking and Embedding, т.е. "присоединение и внедрение объектов". Это расширение Windows, разработанное Microsoft, предоставляет приложениям следующие возможности:

• Присоединение объектов. Под объектом здесь понимаются данные, переданные одной программой, которая называется сервером, другой программе, называемой контейнером. Данными могут являться, например, ячейки электронной таблицы, созданной Quattro Pro. Передается здесь, собственно, не сам объект, а ссылка на него. Присоединенный объект остается собственностью сервера и сохраняется как документ сервера.

• Внедрение (встраивание) объектов. Этот термин также связан с передачей данных от сервера к контейнеру, но в отличие от предыдущего случая объект становится собственностью контейнера и сохраняется им как компонент составного документа. В обоих случаях (присоединения и внедрения) все выглядит так, как будто переданный объект органично входит в основной документ контейнера. Это,

кстати, означает, что сервер вместе с объектом должен передавать и его представление, или вид, так как контейнер, вообще говоря, не знает, как должны отображаться на экране полученные им данные.

• Редактирование на месте. Внедренный объект может редактироваться прямо в окне контейнера, хотя за редактирование отвечает, конечно, сервер. Присоединенный объект тоже может активироваться на месте, но для редактирования открывается отдельное окно.

• Автоматизация. Под этим словом понимается ситуация, когда одна программа (автоматный контроллер) управляет другой программой (автоматным сервером). Примером может служить калькулятор, на котором управляющая программа "нажимает кнопки". То же самое мог бы делать пользователь.

Мы перечислили только некоторые, наиболее характерные, особенности OLE 2. Использование OLE 2 требует от программиста разработки различных интерфейсов в зависимости от поставленной перед приложением задачи. Борландом разработана машина OLE, уже использованная в некоторых коммерческих приложениях, которая упрощает работу программиста, реализуя некоторое подмножество интерфейсов высокого уровня поверх OLE. Программирующие на C++ могут использовать эту поддержку OLE через набор классов, обобщенно названных OCF (Object Components Framework, буквально "каркас объектных компонент"). Вместо того чтобы создавать интерфейсы в стиле OLE, программист может конструировать объекты OCF и вызывать их методы. На основе этих классов вы можете производить собственные классы, наследующие возможности работы с OLE.

Для того, чтобы ввести в уже написанное приложение поддержку протокола OLE, не нужно существенно менять его архитектуру. OCF, например, транслирует сигналы и события, поступающие от OLE, в обычные сообщения

Windows. Вам нужно предусмотреть только соответствующие обработчики событий. Для многих сообщений в OCF имеются обработчики по умолчанию.

Инструменты

В этом разделе даются краткие сведения о трех инструментах Borland C++: Resource Workshop, AppExpert и ClassExpert, которые до известной степени позволяют автоматизировать процесс разработки приложений.

Resource Workshop

Эта утилита Борланда, название которой означает "мастерская ресурсов", предназначена для создания и редактирования всех типов ресурсов Windows. RW известны следующие ресурсы:

- Ускорители — "горячие клавиши", обычно ассоциируемые с командами меню.
- Битовые карты — матрицы растровых изображений, хранящиеся либо непосредственно в файле ресурсов, либо в отдельных файлах .BMP.
- Курсоры — битовые карты для изображения курсоров.
- Диалоги — панели диалогов и их органы управления.
- Шрифты — растровые матрицы шрифтов.
- Пиктограммы — битовые карты, используемые в качестве символов.
- Меню — описывают меню и их команды.
- Строковые таблицы — сообщения об ошибках и т.п.
- Информация о версии — авторские права и идентифицирующая информация, используемая обычно процедурами автоматической установки приложений.
- Ресурсы, определяемые пользователем.

Resource Workshop объединяет все ресурсы программы в проекте ресурсов, который может представлять собой один файл (в простых приложениях), а может иметь разветвленную структуру.

Ресурсы программы RW может хранить как в форме сценария (файлы .RC), так и в двоичной форме файлов .RES, непосредственно воспринимаемых компоновщиком ресурсов.

AppExpert и ClassExpert

AppExpert — автоматический генератор приложений, вернее, их оболочек. Он генерирует исходный код программы, использующий OWL, который может служить основой для разрабатываемого вами приложения Windows. Например, AppExpert генерирует исходный код (а также файлы ресурсов) для главного и дочерних окон, инструментальных линеек, меню и оперативной системы Help, обеспечивает стандартные возможности редактирования документов. Можно ввести в окна программы и дополнительные элементы, например, ползунки для пролистывания или кнопки Max/Min на линейке заголовка окна.

В версии 4.5 AppExpert может генерировать приложения, которые являются сервером или контейнером OLE 2, или тем и другим сразу. Можно построить либо .EXE-, либо DLL-сервер. Пользователь может также добавить к приложению автоматную поддержку.

После того, как AppExpert создаст оболочку приложения, вы можете с помощью утилит ClassExpert и Resource Workshop уточнять и развивать свой проект. Но, естественно, все то, что касается манипуляций со специфическими для вашей программы данными, вам придется написать самостоятельно.

ClassExpert работает с приложениями, генерированными AppExpert, и позволяет более-менее автоматически разрабатывать классы вашей программы. Можно вводить новые классы, редактировать классы, а также просматривать код уже существующих классов. Можно использовать ClassExpert совместно с Resource Workshop для ассоциации классов с ресурсами. Кроме того, ClassExpert показывает события для существующих классов, для которых вы можете предусмотреть собственные обработчики, и проверяет виртуальные функции, реализованные в вашей программе.
ClassExpert в версии 4.5 поддерживает автоматные возможности OLE 2. С его помощью можно автоматизировать любое приложение, которое генерировал AppExpert, в том числе построенное в Borland C++ 4.0.

Компиляция и компоновка

В версии 4.5 усовершенствованы 32-битный компилятор и 16-битный компоновщик.
Компилятор для 32-битных программ содержит новый оптимизатор и генератор кода. Генерация кода занимает примерно в два раза меньше времени, чем в версии 4.0. Полученный неоптимизированный код примерно на 25% быстрее, чем неоптимизированный код, генерированный старым компилятором. Для оптимизированного кода разница составляет примерно 7-10%.
Старый компилятор также имеется в пакете на тот случай, если приложение каким-либо образом использует его особенности. Он переименован в BCC32A.EXE. Можно вызывать его из командной строки по этому имени либо, чтобы можно было использовать его из IDE, переименовать в BCC32.EXE.

Компоновщик 16-битных приложений поддерживает дополнительный набор опций для устранения избыточных данных и сжатия кода .EXE- и .DLL-файлов. Эти опции реализованы как после-компоновочные операции. Сначала происходит обычная компоновка, а затем, если в командной строке присутствуют ключи новых опций, полученный файл снова открывается и производится оптимизация с записью нового файла. Этот файл обычно на 15% меньше и соответственно загружается примерно на 10% быстрее.
Вот перечень новых ключей:
/Oc Удаляет ненужные или дублированные настроечные данные, а также последовательности нулей в конце сегментов данных.
/Oi Ищет повторяющиеся последовательности данных (например, блоки одних нулей) и заменяет их сжатым эквивалентом (значением и счетчиком байт).
/Oa Минимизирует значение для выравнивания сегментов. /Or Оптимизирует выравнивание ресурсов.

Заключение

В этой главе мы постарались дать некоторое представление о том, какие средства имеются в Borland C++ 4.0 и 4.5 для создания современных, развернутых приложений Windows. Самым важным нововведением в версии 4.5 является, наверное, поддержка OLE 2, которая будет иметь большое значение при переходе к операционной системе Windows 95.

Глава 12
Borland C++ Development Suite Version 5.0

Во время подготовки 3-го русского издания этой книги появился новый продукт серии Borland C++ — Development Suite версии 5.0. Поэтому мы решили дать читателям хотя бы поверхностное представление о тех его новых возможностях, которых не было в предыдущих версиях пакета Borland C++ 4.0 и 4.5.

Общие сведения

Собственно, уже из названия — Suite ("комплект" или, не так прозаически, "сюита") — ясно, что речь идет о законченном наборе инструментальных средств, обслуживающих все этапы производства программного обеспечения. В пакет входят такие компоненты, как компилятор Borland C++ 5.0, CodeGuard 32/16, PVCS Version Manager и InstallShield Express, которые обеспечивают написание и отладку кода, сопровождение версии, а также поддержку установки (инсталляции) готового продукта. Все это составляет единую среду разработки, значительно сокращающую время, необходимое для полного цикла создания программ (которые, кстати, получаются при этом и более совершенными, и надежными).

В чем же состоят наиболее существенные отличия от предыдущих выпусков Borland C++? Среди важных, на наш взгляд, моментов можно перечислить следующие:

- Полный набор 32-битных инструментов разработки, позволяющих создавать приложения для всех целевых платформ — от DOS и Windows 3.x до Windows 95 и NT. При этом остается возможность разработки и в 16-битной среде с помощью входящих в Development Suite компонентов Borland C++ and Database Tools 4.5.
- Полностью настраиваемое окружение, управляемое на основе нового принципа — ObjectScripting.
- Введение новых элементов языка, соответствующих последним документам по ANSI C++.
- Новая версия библиотеки ObjectWindows 5.0.
- Совместимость с библиотекой MFC (Microsoft Foundation Classes).
- Более полная поддержка "утилизируемых" компонентов стандартов VBX и OCX (управляющих компонентов OLE).
- Лучшая интеграция пакета, с возможностями его расширения на основе OLE-интерфейсов.
- Поддержка разработок на основе технологии Java.

Ниже мы попытаемся коротко проиллюстрировать некоторые из этих особенностей Borland C++ Development Suite версии 5.0.

32-битная среда разработки

Среда разработки Borland C++ теперь — "натуральная" 32-битная. Понятно, что в 32-битных Windows 95 и NT 32-битные инструменты работают эффективнее 16-битных (по идее, они должны даже иметь меньший размер), а также предполагают лучший уровень системной защиты от ошибок, поскольку каждая из задач запускается в своем собственном пространстве процесса. Но, как упомянуто выше, в пакете оставлены и 16-битные версии инструментов из Borland C++ 4.5, чтобы облегчить программистам процесс перехода на новые системы разработки.

(Напомним, что и 16-битный инструментарий позволяет строить приложения для 32-битных сред выполнения.)

Помимо компилятора, написанного самой фирмой Borland, в 32-битной интегрированной среде имеется альтернативный — "родной" оптимизирующий компилятор фирмы Intel. Это может оказаться важным, если разработчик в целях повышения эффективности генерируемого кода захочет воспользоваться какими-то тонкими нюансами организации конвейеров и кэш-памяти в самых новых микропроцессорах Intel (какими в настоящее время являются Pentium и Pentium Pro).
Благодаря 32-битной среде разработки (использующей "мультилинейные" свойства Windows 95 и NT) можно выполнять компиляцию и построение приложений в фоновом режиме (параллельно, например, с редактированием кода или просмотром символов), задавая при этом наиболее оптимальное распределение приоритетов различных задач.
Наконец, следует упомянуть, что IDE в Borland C++ 5.0 обладает т.н. "логотипной" совместимостью с Windows 95. Сюда относится новый стиль оформления пользовательского интерфейса, возможность применения длинных имен файлов, а также поддержка электронной почты, имеющая большое значение при организации разработок в рабочих группах.

Что такое ObjectScript?

Эта новая черта IDE для многих разработчиков окажется, возможно, наиболее привлекательной. Программисты (особенно работавшие с "классическими" инструментами вроде редактора vi в UNIX и утилиты Make) часто пренебрежительно относятся к средствам типа IDE, упрекая их в недостаточной гибкости предоставляемых программисту возможностей.
В IDE Borland C++ 5.0 все поведение окружения задается *сценариями* (scripts). Каждая команда, каждое нажатие командных клавиш автоматически отображается "машиной сценариев" IDE на соответствующий сценарий, в котором программист может как угодно модифицировать или расширять функции нужного ему инструмента.
Естественно, в пакете предусмотрен набор сценариев, реализующих стандартное поведение окружения, которые вы далее можете усовершенствовать по своему вкусу.
С помощью соответствующих сценариев вы можете, например, автоматизировать форматирование кода, осуществлять его "синтаксически

ориентированное" редактирование, применять аббревиатуры для часто используемых операторов или библиотечных функций, автоматически вставлять строки для комментариев и делать многое другое. Кстати, можно создавать и сценарии для отладчика, которые будут описывать набор необходимых проверок, которые необходимо выполнить, так сказать, "по списку".
В IDE имеется машина сценариев, которые описаны с помощью специфического языка, называемого cScript. Это язык, напоминающий C++ (хотя и с некоторыми особенностями), так что знакомый с C или C++ программист не встретится с какими-либо трудностями при его освоении.
Это объектно-ориентированный язык, но в нем нет указателей, что делает его в известном смысле даже проще и надежнее. Другим отличием от C++ является то, что в последнем используется, по существу, "раннее связывание" классов (времени компиляции), а в cScript — "позднее" (времени выполнения). В C++ все объекты одного класса в принципе одинаковы, в то время как в cScript не нужно выводить новый класс, чтобы добавить нужный вам метод к какому-то конкретному представителю класса. В общем, такое поведение классов cScript компенсирует

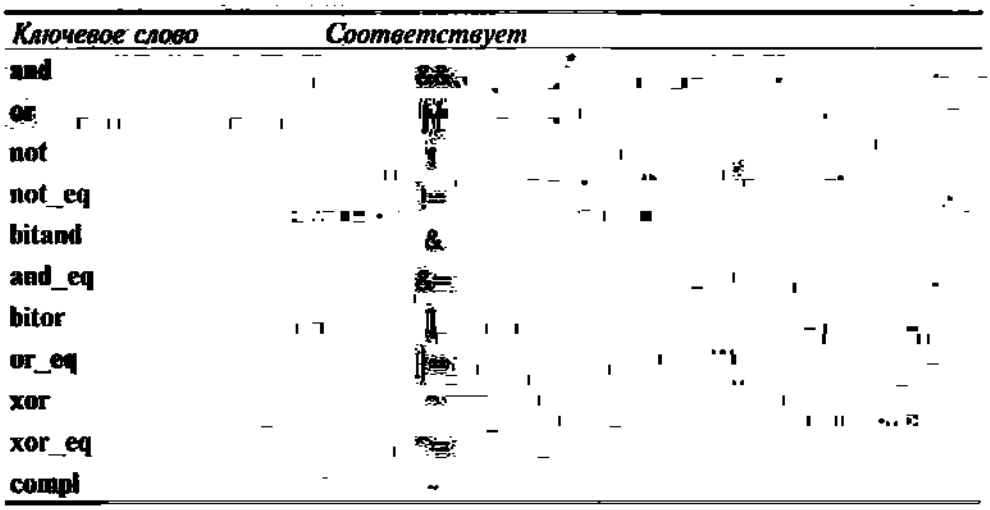
отсутствие в этом языке указателей, обеспечивающих в C++ полиморфизм и позднее связывание объектов.
Различные подсистемы IDE — редактор кода, компилятор, отладчик, менеджер проекта и другие — представлены в виде объектов cScript. Весь набор таких объектов составляет "Библиотеку классов IDE". Объекты экспонируют различные характеристики (properties), методы и события, — подобно тому, что можно увидеть в любой современной объектно-ориентированной схеме классов.

Нововведения в C++

C++ должен стать (кажется, в марте 1996 г.) ANSI-стандартизованным языком. В язык введены некоторые новые ключевые слова:
• **bool** — представляет тип, значениями которого могут быть только true либо false (предопределенные булевы литералы). True имеет численное значение 1, false — 0.
• **mutable** — позволяет модифицировать переменную, даже если она входит в состав выражения, квалифицированного как константа.

Может применяться только к элементам-данным класса. Смысл данного ключа в том, чтобы отменить действие спецификатора **const,** примененного к объекту класса.
• **explicit** — запрещает неявное преобразование типа при присваивании классов. Обычно если у класса есть конструктор с одним параметром, присваивание классу типа, соответствующего параметру, будет интерпретироваться как неявное приведение. При спецификации типа класса как **explicit** допустимы будут присваивания только значений того же самого типа.
• **namespace и using** — ключевые слова для управления *пространством имен*. Большинство нетривиальных приложений состоят из нескольких, а то и многих файлов исходного кода, компилируемых по отдельности и затем объединяемых компоновщиком. Такая файловая организация приводит к тому, что имена, не инкапсулированные ни в каком классе или функции (то есть ни в каком определенном пространстве имен) попадают в одно общее глобальное именное пространство. В результате модули, имеющие разное происхождение, могут оказаться несовместимыми при компоновке из-за дублирования имен.
Решение, предлагаемое C++, состоит в явном определении различных пространств, позволяющих разграничить используемые имена. Приложение разбивается на подсистемы, поручаемые различным программистам. Каждый из разработчиков может использовать какие угодно имена, не беспокоясь о том, что он, может быть, дублирует имена, введенные его коллегой. Но его "личное" пространство имен в программе везде обозначается уникальным идентификатором. Чтобы использовать разделение пространств, нужны два шага. Первым является идентификация именного пространства с помощью ключа **namespace.** Вторым — указание доступа к определенному пространству посредством ключевого слова **using.**
Появились в C++ и новые обозначения логических и побитовых операций — в стиле Паскаля, например, **and** вместо && или **bitor** вместо | (Табл. 11.1). Понятно, что такие обозначения лучше читаются и часто могут снизить вероятность ошибки при набивке кода, но в основном они встроены в язык из-за того, что на некоторых национальных клавиатурах затруднен ввод символов вроде & или .

Таблица 12.1. Обозначения логических и побитовых операций



Ключевое слово	Соответствует
and	&&
or	||
not	!
not_eq	!=
bitand	&
and_eq	&=
bitor	|
or_eq	|=
xor	^
xor_eq	^=
compl	~

Новая версия библиотеки OWL

Библиотека ObjectWindows 5.0 содержит инкапсуляции для многих стандартных управляющих компонентов Windows 95, таких, как закладки, страницы свойств, подсказки инструментальных кнопок и прочее. И поскольку OWL ориентируется на совместимость 16- и 32-битных приложений, в 16-битную ее версию добавлены эмуляторы этих элементов. Предусматривается возможность переноса OWL-приложений на системы с другой архитектурой, и уже существуют OWL Борланда для OS/2 и независимый вариант с названием WM_MOTIF для UNIX.
Ниже перечисляются некоторые новые классы OWL.

Классы для поддержки Windows 95

• **TMailer**
Предоставляет простой метод посылки документа через интерфейс MAPI.
• **TRegKey, TRegValue, TRegKeyIterator, TRegValueIterator, TRegItem, TRegList**
Эти классы инкапсулируют доступ к системному реестру.

OWL в полном объеме инкапсулирует стандартные органы управления интерфейса пользователя Windows 95. Чтобы облегчить разработчикам переход на 32-битные системы, OWL предлагает и 16-битные версии наиболее распространенных компонентов. Далее перечислены классы, их инкапсулирующие.

Стандартные управляющие компоненты

• **TCoIumnHeader**
Инкапсулирует Column Header интерфейса Win95.
• TImageList
Инкапсулирует ImageList.
• **TListWindow**
Инкапсулирует компонент ListView. Класс не назван TListView, поскольку в OWL уже есть класс с таким именем.
• **TTreeWindow**
Инкапсулирует TreeView. Класс не назван TTreeView, чтобы сохранить единообразие с TListWindow.
• **TPropertySheet, TPropertyPage**
Классы позволяют вам создавать многостраничные диалоги. "Sheet" представляет внешний диалог, содержащий одну или несколько страниц ("Page").
• **TRichEdit**

Инкапсулирует компонент Win95 RichEdit — окно редактирования насыщенного текста (RTF-формата).

- **TToolTip**

Позволяет легко реализовать выдачу всплывающего окна сообщения при входе курсора мыши в специфицированную область экрана.

В OWL есть и много других новых классов, в основном таких, которые предоставляют программисту более простые, чем раньше, способы реализации различных распространенных операций, таких, как рисование кнопок, пиктограмм, битовых матриц, приспособлений (gadgets) и прочего. Имеются, например, вспомогательные классы для управления контекстом Help'a и для поддержки списка недавно использовавшихся файлов (MRU List).

OWL в полном объеме инкапсулирует стандартные органы управления интерфейса пользователя Windows 95. Чтобы облегчить разработчикам переход на 32-битные системы, OWL предлагает и 16-битные версии наиболее распространенных компонентов. Далее перечислены классы, их инкапсулирующие.

Совместимость с MFC (Microsoft Foundation Classes)

Говорят, что OWL организована лучше, чем MFC, потому что более последовательно проводит принципы объектно-ориентированного программирования. Но, может быть, благодаря тому, что архитектура Windows тоже не очень-то соответствует идеям ООП, использование библиотеки MFC иногда оказывается предпочтительнее? Не беремся ответить на этот вопрос, однако возможность работы с MFC во всяком случае необходима, например, если речь идет о сопровождении имеющегося кода с MFC. По этой причине в Borland C++ 5.0 предусмотрена совместимость с MFC, что позволяет легко переносить уже существующие приложения в среду Borland C++.

Отладка

Отладка 32-битных приложений для Windows 95 и NT может производиться полностью в пределах интегрированной среды разработки. Встроенный 32-битный отладчик IDE предлагает вам теперь ряд возможностей, которые ранее в интегрированном отладчике отсутствовали.

Вообще отладчик IDE версии 4.5 кажется довольно примитивным инструментом в сравнении с имеющимся в Borland C++ 5.0. А с другой стороны, более развитые автономные отладчики для Windows (TDW и TDW32), которые использовались в предыдущих версиях, были по сути программами с текстовым дисплеем, отображавшимся в фиксированном окне, которое нельзя было даже перемещать по экрану. Новый же отладчик совмещает в себе истинно графический интерфейс пользователя с богатством функциональных возможностей.

Интегрированный отладчик предоставляет вам окно CPU, в котором одновременно отображаются пять различных представлений исследуемого кода. Он имеет чрезвычайно развитые средства управления точками останова и полностью поддерживает отладку параллельных процессов (мультилинейного кода) и исключений — как исключений языка C++, так и исключений 32-битной операционной системы.

Некоторые вспомогательные средства Development Suite

Коротко расскажем теперь о двух новых компонентах, вошедших в состав пакета Borland C++ Development Suite 5.0, которые окажутся чрезвычайно полезны, если не совершенно необходимы, при развертывании действительно больших проектов.

PVCS Version Manager

PVCS — это система управления версиями продукта, помогающая следить за изменениями, вносимыми в исходный код. Тем самым вы можете быстро понять, как развивается процесс разработки и модификации вашего продукта. Version Manager хранит все вносимые изменения с специальном файле *архива.* Архив может регистрировать самые различные файлы, не только исходный код, — точнее будет назвать их просто *исходными файлами проекта;* это могут быть и исполняемые модули, утилиты, библиотеки, документация. *Сдавая* файл в архив, вы тем самым регистрируете его; чтобы внести в файл изменения, вы должны сначала *взять* (как бы под расписку) в архиве его копию. После этого вы снова сдаете файл, и он регистрируется в качестве пересмотренной версии.
В принципе все очень просто, но часто лень бывает изучить какую-то из существующих систем контроля исходных файлов; в конце концов, это точный эквивалент того, что называют бюрократическими штучками. Входящий в Development Suite продукт хорош тем, что он тесно интегрирован в IDE, и вам не придется учить что-то новое или менять свой стиль работы. Вы получаете доступ к Version Manager через меню IDE Project.

InstallShield Express

Процесс разработки не заканчивается в тот момент, когда вы локализуете и устраните последний дефект вашей программы. После этого нужно еще подумать о том, как она будет устанавливаться на машине пользователя. Тут мало решить, какие именно файлы нужно поставлять. Нужно еще, чтобы все они попали в нужные каталоги; необходимо также внести изменения в системный реестр. А чтобы продукт соответствовал стандартам Windows 95, нужно еще уметь удалять его из системы.

Конечно, вы захотите иметь изящную программу инсталляции, позволяющую пользователю выбирать различные варианты. Тут окажется очень полезным InstallShield Express — инструмент, позволяющий визуально построить утилиту установки вашего продукта. Не будем здесь подробно вдаваться в те возможности и удобства, которые он вам предоставляет. Отметим только, что он может автоматически анализировать файлы проекта Borland C++ 5.0 и тем самым определять дополнительные файлы, которые необходимо устанавливать.
Работать с InstallShield Express довольно просто — он показывает вам своего рода "контрольный лист", который вы проходите пункт за пунктом, вычеркивая уже сделанное или пропуская ненужное. Последними из этих пунктов будут построение дисков дистрибутива (сжатие и разбиение на дискеты) и тестовый запуск полученной программы установки.

Технология Java. Что это такое?

Несмотря на постоянное совершенствование рабочих инструментов и технологий, перед разработчиками программ неизменно встает ряд трудноразрешимых проблем. Сюда относится рост стоимости разработок, трудность отладки сложных приложений и проблемы переноса программ между различными платформами.

Настоящее время, кроме того, выдвигает требование, чтобы приложения нового поколения, в особенности системы типа клиент/сервер, обеспечивали бы соединимость в рамках Internet/Intranet. В этом моменте особенно часто разработка становится трудной, долгой и дорогой.
Чтобы помочь в решении подобных проблем, фирма Sun Microsystems создала новый язык программирования, называемый Java (Ява). Язык был задуман как простой, объектно-ориентированный, мультилинейный (multithreaded) и не зависимый от платформы, с расширенной поддержкой для разработок Internet/Intranet. При помощи Явы разработчики могут создавать программы не менее сложные, чем те, что пишутся на C или C++ и которые, кроме того, могут быть развернуты при посредстве Web на различных аппаратных архитектурах и операционных системах.
Пока основной ажиотаж в отношении Явы фокусировался на превращении Web в истинно интерактивное окружение. На Яве программисты могут

создавать исполняемые модули, загружаемые затем через Internet на машину пользователя и запускаемые на ней подобно любым другим программам. Эти приложения могут быть электронными таблицами, графическими редакторами, играми. На самом деле приложения могут быть сколь угодно сложными.
Помимо этого у Явы имеется достаточный потенциал для того, чтобы объединить обособленные в настоящий момент рынки Internet и приложений клиент/сервер. На Яве, едином языке, могут писаться как серверная, так и клиентская части приложения, без учета различий операционных систем и аппаратных платформ.
Поскольку Java происходит от C++, это язык компактный и объектно-ориентированный. Ява отличается от C++ отсутствием сложного (множественного) наследования, шаблонов и перегрузки операций. К тому же Ява обходится без препроцессора и заголовочных файлов, что значительно упрощает управление исходными файлами и контроль их зависимостей. Наконец, отсутствует тип указателя, благодаря чему язык защищает программистов от ошибок в адресах и, соответственно, разрушения находящихся в памяти данных.
Отсутствие упомянутых выше механизмов возмещается структурированной обработкой исключений, языковой поддержкой мультилинейного программирования и возможностями динамического обновления функциональных свойств приложений. Последний момент представляется особенно привлекательным. Ява позволяет разработчикам развертывать приложения, которые автоматически подгружают модифицированные компоненты с удаленной машины.
Компилятор Явы не генерирует машинных инструкций. Вместо этого он выдает файл байтового кода, который работает на любой машине, имеющей исполнительную систему Java. Таким образом, становится реальностью истинное приложение типа "код один — платформ много". В настоящее время Ява работает на Windows 95/NT и Sun Solaris и находится в процессе реализации на 11 -ти других операционных системах, в том числе Macintosh и OS/2. Имеется возможность трансляции байтового кода в машинный для конкретной платформы, в результате чего производительность его становится сравнима с той, что имеет место для компилированного кода на C++.

Разработки на Java в Borland C++

IDE Borland C++ обеспечивает полную поддержку разработки приложений (особенно небольших) на Яве. Сюда входит поддержка в менеджере проекта, доступ к параметрам компилятора и отладчика Явы посредством

многостраничных панелей диалога в IDE, цветовое выделение синтаксических элементов языка в исходном коде, а также визуальный отладчик. IDE перехватывает и отображает ошибки, обнаруженные компилятором Явы. При появлении синтаксической ошибки вы можете просто щелкнуть на сообщении кнопкой мыши и сразу перейти к соответствующей строке исходного кода.
Для эффективности обучения и перехода к работе на новом языке в Development Suite имеется генератор приложений Java AppExpert. С его помощью вы можете быстро и легко построить компактный машинно-независимый код, который будет работать сразу на нескольких популярных системах.
Визуальный отладчик Java сам написан на Яве, что демонстрирует возможности этого языка как средства для разработок. Следующие версии отладчика будут сертифицированы и для других операционных систем, таких как Sun Solaris и Apple Macintosh, так что у вас будет единый отладочный инструмент, работающий везде, где есть Ява.
Наконец, в Borland C++ Development Suite входит ускоритель для Явы — AppAccelerator, который дает 5 — 10-кратный выигрыш в производительности при запуске кода Java на вашей технологической машине.

Требования к системе

Для полноценной установки Borland C++ 5.0 ваша система должна обеспечивать следующие характеристики:
• процессор Intel 486 или выше;
• операционная система Windows 95 или NT 3.51 (входящий в комплект поставки Borland C++ 4.5 может работать и на Windows 3.1);
• 16 Мбайт системной памяти (или больше);
• дисковод CD-ROM.
Необходимое пространство на жестком диске (может отличаться при разных размерах кластеров):
• 25 Мбайт в конфигурации для работы с CD-ROM (все инструменты запускаются с CD-ROM);

• 100 или более Мбайт для типичной конфигурации установки;
• 175 Мбайт для полной установки.

Заключение

В этом небольшом обзоре мы совершенно не коснулись очень многих аспектов нового пакета Borland C++, таких, как визуальные средства проектирования приложений для баз данных или программирование с OLE. По возможности мы старались создать у читателя какое-то, пусть поверхностное и тем самым даже искаженное, представление о новых и, по-видимому, интересных возможностях, которые предоставляет программисту новый продукт, только выходящий на рынок — Borland C++ Development Suite 5.0.

Приложение А
Схема декорирования имен в компиляторе Borland C++

Декорирование имен лежит в основе *перегрузки функций* и безопасного по отношению к типам *редактирования связей*. Компилятор C++ декорирует (попросту говоря, уродует) имена функций и элементов-функций класса, чтобы они отражали следующие моменты:

- Класс, к которому принадлежит функция (в случае функций-элементов).
- Имя функции или ее тип в случае конструкторов, деструкторов, перегруженных операций или функций преобразования.
- Тип аргументов, ожидаемых функцией.

Данный раздел описывает способы декорирования имен, принятые в Borland C++. Понимание схемы декорирования полезно при выяснении причин, по которым при компиляции приложений на C++ появляются сообщения об ошибках типа "Undefined symbol xxxx in module fname.ext". Объяснение таких ошибок часто оказывается связанным с декорированием имени. Четкое понимание особенностей реализации также будет полезным, если вы собираетесь писать код перегруженных операций или методов класса на языке ассемблера.

Общий обзор схемы

Обобщенный формат декорированного имени имеет вид @[classname@] *EncodedFuncName$qEncodedArgType,*
где

- *dossname* является именем класса (в случае функций-элементов класса);

- *EncodedFuncName* является именем функции. Если функция представляет собой конструктор, деструктор, перегруженную операцию или функцию преобразования, используется специальное кодированное имя;
- *EncodedArgType* является последовательностью, кодирующей тип воспринимаемых функцией аргументов.

@[classname@]

Любое декорированное имя начинается с символа @. Для элементов класса за ним следует имя класса и еще один символ @.

EncodedFuncName

EncodedFuncName в типичном случае представляет собой имя функции, если только функция не является конструктором, деструктором, перегруженной операцией или функцией преобразования. В следующей таблице описывается схема кодирования, применяющаяся для таких функций.

Таблица А.1. Кодирование функций

Тип функции	*Кодировка*	*Комментарии*
Конструктор	$bctr	
Деструктор	$bdtr	
Функция преобразования	$o	
Перегруженная операция	$bxxx	где xxx обозначает перегруженную операцию в соответствии с таблицей кодировки операций, приведенной ниже.

Таблица А.2. Кодировка перегруженных операций

Перегруженная операция	*Кодировка*
	add
&	adr
&	and
->	arow
->*	arwm
=	asg
()	call
~	cmp
, coma	
--	dec
delete	dele
/	div
==	eql
>=	geq
>	gtr
+	inc
*	ind
&&	land
||	lor
<=	leq
<<	lsh
<	lss
%	mod
*	mul

Таблица А.2. Кодировка перегруженных операций

Перегруженная операция	*Кодировка*
!=	neq
new	new
!	not
||	on
&=	rand
/=	rdiv
<<=	rlsh
-=	rmin
%=	rmod
*=	rmul
|=	ror
+=	rpls
>>=	rrsh
^=	rxor
-	sub
[]	subs
	xor

$qEncodedArgType

$qEncodedArgType является последовательностью символов, представляющей аргументы, воспринимаемые функцией. Кодировка включает в себя тип и модификаторы аргументов. В следующей таблице показаны символы, применяемые для различных модификаторов.

Таблица А.3. Кодировка модификаторов типа

Модификатор	*Кодировка*
Дальняя ссылка	m
Дальний указатель	n
Ближняя ссылка	r
Ближний указатель	p
Указатель huge	up
Указатель сегмента	ur
unsigned	u
signed	z
const	x
volatile	w

Основной объем последовательности *EncodedArgType* составляют типы аргументов, принимаемых функцией. В следующей таблице показаны кодировки типов.

Таблица А.4. Кодировка типов

Тип	*Кодировка*
char	c
double	d
многоточие	e
float	f
long double	g
int	i
long	l
short	s
void	v

Массивы и типы, определяемые пользователем

Определенные пользователем типы, такие, как перечисления или классы, кодируются десятичным числом, за которым следует имя типа. Число показывает длину имени типа. Типы массивов представляются буквой *a,* за которой следует размер массива и имя типа.

Приложение B
Вспомогательные функции RTL

Содержание исполнительной библиотеки Borland C++ не ограничивается набором функций, доступных для пользователя. RTL содержит определенное множество процедур, обычно называемых *вспомогательными функциями,* которые компилятор использует при обработке ваших модулей на языке C или C++. Встретив какую-либо конструкцию, компилятор Borland C++ может решить, что следует вызвать предопределенную процедуру вместо того, чтобы генерировать встроенную (in-line) последовательность инструкций. Например, для следующего фрагмента компилятор генерирует вызов специальной процедуры деления длинных целых:

```
#include <time.h> #include <iostream.h>
int main()
{
time_t secsPassed = time(NULL); time_t minsPassed = secsPassed/60;
cout << "Число минут, прошедших с 1/1/1970: " << minsPassed <<endl;
return ); }
```

Если установлена опция проверки переполнения стека (Check Stack Overflow), компилятор тоже генерирует вызов вспомогательной процедуры, которая проверяет состояние стека.

В этом приложении освещаются некоторые области, в которых компилятор Borland C++ использует преимущества, которые дает применение вспомогательных функций. Важно понять, что эти функции существуют для того, чтобы ими пользовался компилятор! Вы не должны вызывать их в явном

виде. Однако общее понимание того, как и когда они применяются, может оказаться весьма полезным. Вы можете столкнуться со вспомогательными функциями на этапе отладки, изучая окно CPU или исследуя стек вызовов. Вы также можете улучшить эффективность ваших приложений, избегая таких конструкций, для которых использование вспомогательных функций не будет оптимальным.

Генерирование выходных файлов на языке ассемблера

Одним из лучших методов исследования кода, генерируемого компилятором, является разрешение опции Generate Assembler Source (Генерировать ассемблерный текст). Выходные файлы на языке ассемблера содержат строки вашего исходного текста на C или C++ в качестве комментариев. Ниже приводится выдержка из выходного файла на языке ассемблера, полученного при компиляции предыдущего фрагмента кода:

```
;
; {
; time_t secsPassed = time(NULL);
push 0
```

```
push 0
call far ptr _time
add  sp,4
mov   word ptr [bp-4],dx .
mov   word ptr [bp-6],ax
time_t minsPassed = secsPassed/60;
push 0
push 60
push word ptr [bp-4]
push word ptr [bp-6]
call far ptr F_LDIV@
```

Ассемблерный код открывает нам, что для деления компилятор использовал функцию *F_LDIV@.*

Размещение массива из объектов класса

Если вы размещаете в памяти массив класса с конструктором, компилятор вызывает вспомогательную функцию, которая сначала выделяет память под массив, а затем инициализирует его содержимое, вызывая конструктор для каждого его элемента.

```
static int defaultX = - 1 ; static int defaultY = - 1 ;
class Point {
public:
Point( int _x=defaultX,    int _y=defaultY )
:x(_x),  y(_y)
{}
// ... private:
int x, y; }
const int numPoints = 100;
int main( void )
{
Point *ppt = new Point[numPoints]:
Л
*
*/
delete []ppt; return 0;
}
```

Динамическое размещение массива из Point, применяемое в данном примере, приводит к вызову вспомогательной функции vector_new_, как показывает листинг файла на языке ассемблера:

```
; {
; Point *ppt = new Point[numPoints;
push  seg  @Point@bctr$qv
push offset §Point@bctr$qv
push 5
push 0
push 100
```

```
push 4
push 0
push 0
call far ptr @_vector_new_$qnvuivluie
add sp,16
```

Копирование структур

При обработке присваивания структур Borland C++ может применять вспомогательную функцию @SCOPY. Эта функция ведет себя так же, как функция ANSI C memcpy — она копирует указанное число байт исходной переменной в принимающую.

Следующий пример иллюстрирует случай, когда понимание механизма вспомогательных функций может помочь вам в тонкой настройке вашей программы:

```
struct structA
{
int accountn;
char code;
}
struct structB
{
int accountn;
int code;
}
structA a1 , a2={ 100, 'x' };
structB b1 , b2={ 100, 0 };



int main( void ) {
a1 = a2; . •  b1 = b2;
return 0;
}
```

Первое из присваиваний приводит к вызову вспомогательной функции: ;

```
int main( void )
;
assume  cs:SCOPY_TXT,ds:DGROUP  jnain    proc  far
; {
; a1 = a2;
push ds
push  offset  DGROUP:_a1
push ds
push  offset  DGROUP:_a2
mov cx,3
call far ptr F_SCOPY@
```

Однако присвоение значения b2 переменной b1 производится реализуется в виде встроенного кода:

```
; b1 = b2;
mov dx,word ptr DGROUP:_b2+2 mov ax,word ptr DGROUP:_b2 mov word ptr DGROUP:b1+2,dx mov word ptr DGROUP:b1,ax
```

Сравнив *structA* и *structB,* вы заметите, что из размеры различаются на один байт. Таким образом, вызова вспомогательной функции можно избежать, дополнив *structA* до четного числа байт. Можно достичь этого и другим способом, задав компилятору опцию оптимизации кода по скорости.

Проверка переполнения стека

Опция компилятора *Test Stack Overflow* предписывает компилятору производить при входе в каждую компилируемую функцию проверку переполнения стека. На самом деле компилятор генерирует вызов вспомогательной функции, которая и выполняет эту задачу. Приведенные ниже ассемблерные листинги демонстрируют эффект применения данной опции при компиляции простой функции, возвращающей значение константы.

```
double pie()
{
   return 22/7;
}
```

```
; Check Stack Overflow: OFF
;
; double pie()
;
@pie$qv proa far
inc  bp
push bp
raov bp,sp
;
;
;  return 22/7;
;
fid dword ptr D.GH0UP:a§
jmp snort &1&58
<Bie58.:
```

```
; Check Stack Overflow:  ON
;
; double ple()
;
ipiejqv proc far
inc  bp
push bp
mov  bp.sp
xor  ax. ax
call far ptr F_CHSTCK§
;
;
;    return 22/7;
;
fid dword ptr 0GR0UP:s§
jmp short @1@58
9LШ5B:
```

Заключение

При компиляции вашего кода на языках C и C++ Borland C++ пользуется и многими другими вспомогательными функциями. Исходные тексты большинства из них прилагаются к исходным текстам исполнительной библиотеки Borland C++. Вы не должны явным образом вызывать эти функции, однако знание того, как они используются, поможет вам лучше понимать код, генерируемый компилятором.

Просто и ясно о BORLAND C++

Краткие и точные сведения, нацеленные на скорейшее получение результатов в работе с Borland C++.

- Ясные, подробные инструкции по выполнению рутинных операций.
- Описание синтаксиса ANSI C и расширений в пакете Borland C++.
- Примеры программ и подсказки наилучшего применения "идиом" C и C++.
- Уникальные возможности языка Borland C++: шаблоны, управление исключительными ситуациями, информация о типе во время исполнения.
- Знакомство с особенностями Borland C++ новой версии 5.0.

Эта книга станет вашим личным наставником в освоении пакета Borland C++.

Здесь сосредоточена самая сущность языка C++ без избыточной или повторяющейся информации, которая могла бы замедлить изучение предмета.

Вы сможете быстро достичь практических результатов и получить самую свежую информацию об особенностях языка Borland C++ и объектно-ориентированного подхода к программированию.

В этой книге - вся мощь, все подробности, все выразительные средства Borland C++.