

В.А.Скляр

ПРОГРАММИРОВАНИЕ НА ЯЗЫКАХ

Си и Си++



ВЫСШАЯ ШКОЛА

В.А.Скляр

ПРОГРАММИРОВАНИЕ НА ЯЗЫКАХ Си и Си⁺⁺

Издание второе,
переработанное и дополненное

Рекомендовано
Министерством общего
и профессионального образования
Российской Федерации
в качестве учебного пособия
для студентов вузов



Москва
«Высшая школа»
1999

УДК 681.3.06
ББК 32.973-01
С 43

Рецензенты:

кафедра вычислительной техники Санкт-Петербургского государственного электротехнического университета (зав. кафедрой проф. Д.В. Пузанков); чл.-корр. Академии наук Белоруссии, д-р техн. наук, проф. А.Д. Закровский (Академия наук Белоруссии)

Скляров В.А.

С 43 Программирование на языках Си и Си++: Учеб. пособие. — 2-е изд., перераб. и доп. — М.: Высш. шк., 1999. — 288 с.: ил.

ISBN 5-06-003486-0

В книге рассмотрены современные концепции объектно-ориентированного программирования (ООП) и описаны популярные алгоритмические языки Си и Си++ для персональных ЭВМ. Описаны основные конструкции этих языков и приведено большое число простых программ. Подобраны примеры, демонстрирующие главные преимущества технологии ООП. Многие языковые конструкции поясняются на простых рисунках. В результате сокращается время изучения языка и упрощается восприятие материала.

Второе издание (1-е — 1996 г.) переработано и дополнено описанием новых концепций в развитии ОПП (на примере системы программирования Visual C++, поддерживаемой библиотекой классов MFC).

Для студентов вузов. Может быть полезно учащимся старших классов школ, колледжей, лицеев, техникумов, а также всем интересующимся языками программирования.

Учебное издание

Скляров Валерий Анатольевич

ПРОГРАММИРОВАНИЕ НА ЯЗЫКАХ Си и Си ++

Ведущий редактор **Н.Е. Овчеренко**

Художник **В.А. Дмитриев**

Художественный редактор **Ю.Э. Иванова**

Технический редактор **Л.А. Овчинникова**

Компьютерная верстка **С.Н. Луговая**

Корректор **Г.Н. Петрова**

ЛР № 010146 от 25.12.96. Изд. ВТИ—45. Сдано в набор 28.07.98.
Подп. в печать 21.01.99. Формат 60х88¹/₁₆ Бум. газетн. Гарнитура Журн. Руб.
Печать офсетная. Объем 17,64 усл. печ. л. 17,89 усл.кр.-отт. 16,97 уч.-изд. л.
Тираж 4000 экз. **Заказ № 269**

Издательство «Высшая школа», 101430, Москва, ГСП-4, Неглинная ул., д. 29/14.

Набрано на персональном компьютере издательства.

Отпечатано в ОАО «Оригинал», 101898, Москва, Центр, Хохловский пер., д. 7.

ISBN 5-06-003486-0

© Издательство «Высшая школа», 1999

ВВЕДЕНИЕ

В представленной книге отражены современные концепции объектно-ориентированного программирования (ООП) и дано описание алгоритмических языков Си и Си++.¹

Язык Си — один из наиболее популярных современных языков программирования. С момента появления (начало 70-х годов) до настоящего времени он прошел несколько этапов своего развития и совершенствования. К 1980 г. сформировался новый подход к решению задач на ЭВМ, названный объектно-ориентированным программированием. Он позволил получить следующие преимущества:

- использование специального стиля написания программ, принятого в ООП, повышает доступность их восприятия и упрощает внесение возможных изменений (модификацию программы);

- существенное упрощение процедуры поиска ошибок;

- представление процесса проектирования программы в виде иерархии изолированных подзадач, что позволяет создавать большие системы коллективом параллельно работающих программистов;

- повышение надежности программного обеспечения.

Язык Си++ соединяет в себе все достоинства языка Си с новыми возможностями ООП.

Книга включает шесть глав и приложение.

Гл. 1 знакомит с инструментальными системами программирования Visual C++ и Borland C++*. Поскольку книга ориентирована на широкий круг пользователей, главное внимание уделено средствам интегрированного окружения. При этом описана современная интегрированная среда Developer

¹ Turbo C++, Borland C++ — торговые марки Borland International, Inc.; MS-DOS, Windows, Developer Studio, Visual C++, MFC — торговые марки Microsoft Corporation; IBM, IBM PC — торговые марки International Business Machine Corporation.

Studio для написания программ на языке Visual C++ 5.0 и сохранена информация из предыдущего издания об интегрированной среде Borland C++ 3.1. Это позволяет использовать как мощные современные компьютеры, так и компьютеры с ограниченными ресурсами. В действительности обе системы позволяют изучить языки программирования Си и Си++, что и является основной целью настоящей книги. Материал данной главы носит преимущественно справочный характер.

Гл. 2 представляет общие конструкции языков Си и Си++. При этом отражаются основные концепции современного стиля программирования. Представленный здесь материал будет полезен пользователям, знакомым с основами программирования, но не знающим языка Си.

Гл. 3 полностью посвящена новым возможностям языка Си++. Сначала излагаются базовые принципы ООП, затем детально поясняется, иллюстрируется рисунками и простыми примерами каждая новая конструкция.

В гл. 4 описан интерфейс языков Си и Си++ с языком Ассемблера.

В гл. 5 описаны приемы подготовки программ с использованием библиотеки MFC (Microsoft Foundation Classes). Эта глава заменяет главу с названием «Модели памяти и оверлейные программы» из предыдущего издания, которая вынесена в приложение.

Гл. 6 включает более сложные примеры программ для управления аппаратурой персонального компьютера. Программы написаны на языках Си++ и Ассемблера.

В приложениях приведена дополнительная справочная информация.

Чтобы диалог пользователя и ЭВМ был для читателя более наглядным, символы, которые человек вводит в машину, в книге подчеркнуты, а выводимые машиной, например на экран дисплея,— нет. Обозначение <ВВОД> говорит о том, что пользователь должен нажать клавишу ВВОД.

Книга будет полезна всем, кто работает на персональном компьютере семейства IBM и использует языки программирования Си и Си++.

ГЛАВА 1

ПОДГОТОВКА, КОМПИЛЯЦИЯ, КОМПОНОВКА, ОТЛАДКА И ВЫПОЛНЕНИЕ ПРОГРАММ

1.1. Введение в интегрированную среду Developer Studio

Developer Studio — это комплекс программных средств для работы с сегодняшними и будущими языками программирования, поставляемыми фирмой Microsoft. Мы рассмотрим, как эти средства используются для подготовки, компиляции, компоновки, отладки и выполнения программ в системе программирования Visual C++ 5.0.

Visual C++ поддерживает работу с консольными программами (консольными приложениями) и программами **Windows** (приложениями **Windows**). В первом случае взаимодействие с выполняемой программой осуществляется так же, как в среде ДОС. Это позволяет передать в программу все разрешенные средства ввода и вывода, какими, например, являются клавиатура и дисплей. Символьная информация может быть введена в программу с клавиатуры и выведена из программы на экран дисплея так же, как это делалось в среде ДОС. Программа для работы с приложениями **Windows** будет много сложнее, но для ее разработки могут быть применены специальные средства системы, такие как AppWizard и ClassWizard, которые используют библиотеку классов MFC. В этом случае структура программы будет существенно изменена. Для приложений **Windows** невозможно установить прямую связь между физическими и программными средствами ввода и вывода. Например, ввод в программу данных с клавиатуры и вывод из программы данных на экран дисплея можно произвести только с помощью средств операционной системы (а ею является **Windows**), то есть функций этой системы, обеспечивающих взаимодействие с программами пользователя. Множество таких функций называется API (Application Programming Interface). Любое обращение к системе со стороны пользовательской программы регистрируется как событие. Система обрабатывает события, предоставляя одновременно работающим пользовательским программам распределенные во времени ресурсы

компьютера, такие как клавиатура, дисплей, манипулятор «мышь» и т.п. Аналогичные механизмы могут использоваться и для взаимодействия программ друг с другом.

Интегрированная система Developer Studio включает следующие базовые компоненты.

Редактор, который используется для написания и редактирования текстов различных программ. Ключевые слова языка Си++ автоматически распознаются редактором и выделяются цветом.

Компилятор, который преобразует исходный текст (исходный код) программы в объектный код, сохраняемый в файлах типа .obj. В процессе преобразования исходный код проверяется на наличие ошибок.

Компоновщик, который соединяет необходимые объектные коды программ и библиотечных компонентов в единый выполняемый модуль, сохраняемый в файле типа .exe. Компоновщик также выявляет возможные ошибки, такие как отсутствие некоторого компонента будущей программы и т.п.

Библиотеки, которые упрощают использование языка для решения различных задач и делятся на два типа: стандартные библиотеки языка Си++, содержащие общие для разных компиляторов функции, такие как возведение в степень, извлечение квадратного корня, сравнение строк и т.п.; библиотека MFC, упрощающая создание программ в среде **Windows**.

Компонент AppWizard. Wizard в переводе на русский язык означает волшебник или маг, App — первые три буквы из слова Application (приложение). AppWizard автоматически строит все основные компоненты (скелет) будущей программы пользователя. В действительности мы можем построить таким образом программу со всеми желаемыми элементами внешнего интерфейса. В результате остается добавить только те компоненты, которые определяют специфику задач (не интерфейса), решаемых этой программой.

Компонент ClassWizard предоставляет средства расширения фундаментальных строительных блоков объектно-ориентированной программы, какими являются классы. Это позволяет расширять и совершенствовать как элементы MFC, так и классы, построенные программой **AppWizard**.

1.2. Использование базовых средств интегрированной среды

После загрузки интегрированной системы Developer Studio на экране появится несколько окон. Окно в левой части экрана содержит рабочее пространство будущего проекта. В правой

части экрана расположено окно редактора и в нижней части экрана — окно сообщений.

Левое окно позволяет работать с будущим проектом и содержит ссылки на базовые документы, которые могут быть просмотрены во время работы и полезны для получения информации о различных средствах Visual C++. Правое окно позволяет создавать и редактировать исходный текст различных компонентов будущей программы. Окно сообщений показывает результаты компиляции и компоновки программы. Компоненты интегрированной среды можно изменять путем активизации (включения) соответствующих пунктов меню, которое появляется при нажатии правой кнопки манипулятора «мышь». Выше были использованы следующие установки: Output, Workspace, Standard, Build MiniBar, InfoViewer, WizardBar.

Любая программа, которая разрабатывается в Visual C++, оформляется в виде рабочего проекта. Такой проект может включать консольную программу, программу **Windows** и т.п. При создании проекта автоматически строится его рабочее пространство (project workspace), которым является каталог (директорий), содержащий всю информацию (все файлы), имеющую отношение к проекту. В рабочем пространстве проекта можно создавать подпроекты, т. е. другие проекты, которые являются компонентами проекта более высокого уровня. В результате можно работать с любыми файлами, включенными в подпроект.

Для создания нового проекта необходимо выбрать из главного меню опции (режимы) File — New. Далее в появившемся окне диалога необходимо указать тип, имя и местоположение файла проекта. После этого автоматически создается каталог, который имеет имя проекта. Он предназначен для хранения следующей информации:

- исходный текст программы на языке Си++ (файл .cpp);

- базовое описание проекта (файл .dsp) с информацией о том, как будущая программа должна строиться из файлов, включенных в проект;

- файл установок (.opt) для отображения компонентов проекта;

- описание рабочего пространства (файл .dsw), содержащее информацию о том, что содержит проект;

- описание местоположения разных файлов проекта (файл .ncb), говорящее о том, в каком каталоге находится каждый файл и где он используется в проекте.

Для проекта можно установить различные режимы его использования с помощью опций Project — Settings. При создании проекта можно задать одну из двух типовых конфигураций — отладочную (Debug) и рабочую (Release). Это можно сделать с помощью опций Build — Configurations. В первом случае в прог-

рамму включается много дополнительной информации, необходимой для отладки. В результате можно просматривать промежуточные значения разных переменных программы, выполнять ее по шагам и т.п. Во втором случае программа оптимизируется для ее эффективного использования как конечного рабочего продукта. Это приводит к повышению быстродействия программы и к снижению объема занимаемой памяти. Выполняемая программа для отладочного режима создается в подкаталоге Debug основного каталога, а для рабочего режима — в подкаталоге Release основного каталога.

Рассмотрим пример построения проекта для простого консольного приложения. Рассмотрим следующую программу:

```
#include <iostream.h>
void main (void)
{    cout << "Наша первая программа" << endl;    }
```

При этом необходимо иметь установленный шрифт с русскими буквами, который делается доступным для использования в программах с помощью опций Tools — Options — Format и последующих установках шрифта с русскими буквами. При возникновении проблем русский текст в программе можно заменить английским, например, «Our first program». Все, что делает программа, это вывод текста в двойных кавычках на экран дисплея.

Для построения выполняемого модуля рассмотренной программы необходимо выполнить следующие действия:

1) Выбрать опции File — New. В результате появится окно диалога. В прямоугольное окно Project name: необходимо записать имя программы (запишем, например, имя First). Местоположение каталога проекта в окне Location: можно либо оставить без изменения, либо задать любое действительное имя (то есть имя существующего каталога, в котором можно создать подкаталог проекта, и путь к этому существующему каталогу). В левой части окна диалога необходимо выбрать опцию Win32 Console Application. После всех этих действий в окне диалога выбирается кнопка OK и проект будет автоматически создан;

2) Выбрать опции File — New. Теперь система знает о том, что проект уже создан, и выводит диалоговое окно, где предлагаются различные типы программ, которые можно создать и включить в проект. Поскольку наша цель заключается в разработке простейшей программы на языке Си++, то необходимо выбрать опцию C++ Source File (исходный текст программы на языке Си++) и имя файла программы (File name) на языке Си++ (например, то же

First). Теперь после нажатия кнопки ОК, открывается окно редактора для подготовки исходного текста программы;

3) Ввести с клавиатуры в окно редактора текст приведенной выше программы;

4) Сохранить файл с программой с использованием опций File — Save. Файл можно сохранить в любом каталоге, хотя предпочтительнее использовать каталог нашего проекта;

5) По умолчанию файл First.cpp (тип cpp характеризует исходные тексты программ на языке Си++) будет автоматически включен в наш проект. Если в проект необходимо включить новый файл, то надо либо выбрать опции Project — Add to Project — Files, либо нажать правую кнопку манипулятора «мышь» с курсором в окне редактора и в появившемся меню выбрать опцию Insert File into Project. Отметим, что файл с исходным текстом программы можно было бы создать сразу в проекте (опции Project — Add to Project — New);

6) Построить рабочую программу, то есть выполнить этапы компиляции, компоновки и загрузки. Для этого можно использовать различные способы, например:

выбрать опции Build — Build First.exe (эти же действия задаются нажатием клавиши F7);

выбрать опцию Build из меню, которое появится после нажатия правой кнопки манипулятора «мышь» с курсором, указывающим на проект (First files) в левом окне (это окно должно быть установлено в режим FileView);

выбрать соответствующую иконку для построения выполняемой программы в меню Build MiniBar.

При построении выполняемой программы все исходные файлы автоматически сохраняются;

7) Выполнить программу. Здесь тоже можно использовать несколько способов. Например, выбрать опции Build — Execute First.exe, выбрать иконку ! в меню Build MiniBar или просто нажать клавиши Ctrl-F5. При отсутствии ошибок на экране появляется строка: «Наша первая программа» или «Our first program» (если был использован английский текст).

Если при компиляции или при компоновке обнаруживаются ошибки, то процесс построения выполняемой программы прерывается и информация об ошибках выводится в окне сообщений, которое расположено в нижней части экрана. Для того чтобы найти строку программы с ошибкой, необходимо переместить курсор манипулятора «мышь» на строку с соответствующим сообщением об ошибке и дважды нажать левую кнопку «мыши». При необходимости можно получить дополнительную информацию об ошибке. Для этого надо переместить курсор в окне сообщений на строку, где содержится код ошибки (типа error

C2297) и нажать клавишу F1. В результате появляется окно с соответствующей информацией.

При успешном построении выполняемой программы в отладочном режиме, в каталоге с файлом проекта появится подкаталог Debug с новыми файлами, которые кратко характеризуются ниже:

First.exe — выполняемый файл для нашей программы;

First.obj — объектный файл, построенный после компоновки нашей программы;

First.ilc — этот файл используется компоновщиком при построении нового выполняемого модуля. В результате только модифицированные файлы будут компоноваться повторно;

First.pch — файл, содержащий информацию о предварительно откомпилированных заголовках (header files). Такие файлы, которые, в частности, поставляются вместе с системой программирования Visual C++, можно компилировать один раз и далее использовать результаты этой компиляции. Это позволяет существенно сократить время, необходимое для построения программы;

First.pdb — этот файл содержит информацию, которая используется для выполнения программы в отладочном режиме;

.idb — содержит дополнительную информацию для отладки.

Построенный файл First.exe может быть выполнен в среде Visual C++ с помощью опций Build — Execute First.exe (а также других опций, рассмотренных выше) или из среды **Windows** (например, из **Windows Explorer**) так же, как и любая другая выполняемая программа.

1.3. Использование дополнительных возможностей интегрированной среды

При построении нового проекта в левом окне (в окне проекта) можно отображать четыре разных типа данных: ResourceView, ClassView, FileView и InfoView. Нужный тип активизируется выбором соответствующей кнопки в нижней части окна проекта.

В режиме InfoView можно выбрать и отобразить на экране различные документы, которые содержат полезную для текущей работы информацию.

В режиме FileView можно получить информацию о файлах, включенных в проект, и вывести любой файл проекта в окно редактора. Информация о файлах отображается в виде привычной иерархической структуры (в виде дерева). Для вывода содержимого файла в окно редактора надо переместить курсор «мыши» на соответствующее имя и дважды нажать левую кнопку.

В режиме ClassView можно получить информацию о классах, используемых в программах проекта.

Режим ResourceView используется только для программ, работающих под управлением **Windows** (мы рассмотрим одну из таких программ ниже в этом разделе). Этот режим обеспечивает вывод информации о различных ресурсах **Windows**, используемых в программе, таких как меню, диалоговые окна, иконки и т.п.

По аналогии с другими окнами интегрированной системы, в окне проекта можно вывести меню с полезными опциями, имеющими отношение к проекту. Для этого необходимо переместить курсор «мыши» в зону окна проекта и нажать правую кнопку.

Покажем теперь, как установить разные полезные режимы работы системы программирования Visual C++. Все режимы делятся на две категории, к которым относятся:

1) общие режимы для всех проектов, выполняемых в интегрированной среде;

2) специфические режимы для индивидуального проекта.

Для установки общих режимов необходимо выбрать разделы меню Tools — Options. В результате появляется диалоговое окно, в верхней части которого перечислены возможные режимы (Options), поделенные на следующие функциональные группы: Editor (редактор), Tabs (табуляция), Debug (отладка), Compatibility (совместимость), Build (построение выполняемой программы), Directories (каталоги), Workspace (рабочее пространство проекта), DataView (представление данных), Macros (макросы), Format (формат), InfoViewer (предоставление информации). Дополнительную информацию о любом режиме можно получить, если выбрать знак вопроса в правом верхнем углу окна диалога и переместить этот знак вопроса на слово или раздел диалогового окна, по которым требуется дополнительная информация. Конкретные опции включаются и выключаются в соответствующих маленьких квадратных окнах (check box) для каждого активного режима.

Разделы меню Tools — Customize позволяют изменить установки всей интегрированной системы.

Индивидуальные режимы, которые будут действительно только для конкретного проекта, выбираются с помощью опций Project — Settings. В верхней части появившегося диалогового окна тоже можно выбрать разные режимы, такие как General (общие), Debug (отладка), C/C++ (Си и Си++), Link (компоновка), Resources (ресурсы), Browse Info (информация о поиске компонентов), Custom Build (специфическое построение выполняемой программы) и т.п. Многие из этих режимов следует оставить такими, какими они заданы по умолчанию. Первоначально следу-

ет установить опции Link — Shared MFC и C/C++ — Generate browse info. Первая опция позволяет исключить некоторые ошибки, которые возможны при компоновке. Вторая оказывается полезной при работе с исходными текстами программ. После ее установки можно выбрать курсором «мыши» любой элемент программы и нажать правую кнопку. С помощью появившегося меню можно узнать, где используется выбранный элемент и где он был определен. Информация такого типа хранится в файлах .bsc.

В заключение рассмотрим пример создания программы, работающей под управлением **Windows**. Для этого необходимо выполнить следующие действия:

1) создать проект так же, как и ранее, и заменить опцию Win32 Console Application на опцию MFC AppWizard (exe). Далее необходимо ввести имя проекта, например: FirstWin. Если старый проект не был закрыт, то надо выбрать опцию Create new workspace (в противном случае новый проект будет создан в рабочем пространстве старого проекта). После выбора кнопки OK появляется окно диалога для AppWizard;

2) AppWizard позволяет автоматически построить заказные средства пользовательского интерфейса для прикладной программы и выполняет много других полезных функций, которые задаются в последовательно появляющихся новых окнах диалога. Мы не будем рассматривать здесь детали установки разных функций (они будут рассмотрены в гл. 5) и поэтому в первом же диалоговом окне выберем иконку Finish (закончить). В новом появившемся диалоговом окне содержится информация о создаваемом проекте. Примем эту информацию и выберем кнопку OK. В результате автоматически строится проект программы, работающий под управлением **Windows**, и информация об этом проекте выводится в левом окне экрана;

3) компиляция, компоновка и выполнение программы осуществляются так же, как и раньше. Для этого достаточно выбрать опции Build — Execute FirstWin.exe или нажать клавиши Ctrl-F5. В результате на экране появится стандартное окно **Windows** с заголовком FirstWin. Оно совершенно бесполезное, но содержит все необходимые компоненты и атрибуты, принятые в системе **Windows**, такие как меню и панель управления с иконками в верхней части, строка состояния в нижней части и т.п. Вы можете изменить размеры окна и использовать средства меню (опции File — New), чтобы создать новые окна. Все эти средства были автоматически построены системой программирования Visual C++. Задача прикладного программиста заключается в наполнении построенной программы полезным содержимым. Основные сведения о том, как это сделать, будут даны в гл. 5.

1.4. Отладка программ

Отладка позволяет произвести проверку правильности работы выполняемых программ. При этом можно производить следующие основные операции:

выполнять программу по шагам, т. е. приостанавливать выполнение программы в каждой очередной строке. В этом случае каждый шаг связывается со строкой исходного текста программы. При необходимости пошаговое выполнение можно произвести и на более низком уровне, например, деасемблировать программу и связать каждый шаг с командой низкого уровня на языке Ассемблера;

выполнять программу между заранее установленными точками в программе, которые называются точками останова. Такой точкой может быть и текущее местоположение курсора в программе;

просматривать значение внутренних переменных в программе в любой точке останова;

просматривать значение внутренних переменных, которые хранятся во внутренних регистрах микропроцессора и в оперативной памяти;

выполнять много других полезных операций, которые выходят за рамки материала, рассмотренного в настоящей книге.

Работа отладчика в интегрированной системе Developer Studio будет рассматриваться ниже на простых примерах.

Пусть задана следующая программа:

```
#include<iostream.h>
void main(void)
{
    int a=1,b=2,c=3,d;    // объявление переменных a, b, c, d целого
                          // типа (int) и присваивание первым трем
                          // переменным начальных значений 1, 2 и 3
                          // соответственно
    d = a+b;              // вычисление значения d
    a = a+b+c+d;          // вычисление значения a
    a--;                  // уменьшение значения a на 1: операция
                          // декремент (--)
                          // вывод значений a, b, c, d
    cout << "a = " << a << "; b = " << b <<
        " c = " << c << "; d = " << d << endl;
}
```

Эта программа очень простая и все ее действия пояснены в комментариях, записанных в каждой строке после символа //. Результаты работы программы (оформленной как консольное приложение) представятся в виде:

`a = 8; b = 2; c = 3; d = 3.`

Для того чтобы выполнить эту программу в режиме отладки, можно производить разные действия. Например, для ее выполнения в пошаговом режиме можно нажать клавишу F10. После этого курсор (утолщенная стрелка) будет установлен на первую строку программы:

`int a = 1, b = 2, c = 3, d;`

В нижней части экрана появляются два окна. При необходимости эти окна можно вывести в любое место. Для этого надо переместить курсор манипулятора «мышь» на двойную линию в левой части выбранного окна и потянуть его в нужную точку экрана. Если раскрыть окно Variables, то в верхней его части имеется раскрывающийся список Context, который позволяет проследить последовательность вызова разных функций в программе. Если выбрать курсором любую функцию и нажать дважды левую кнопку манипулятора «мышь», то отображается текст этой функции. В результате можно просмотреть значения локальных переменных и получить другую необходимую информацию.

Поскольку в нашей программе лишь одна функция с именем main, то значения ее переменных (a, b, c, d) мы и будем просматривать в окне Variables.

Второе окно с именем Watch позволяет ввести любое имя и просматривать значение соответствующей переменной. Это значение можно изменять в окне, задать для его вычисления некоторое выражение и т.п.

Предположим, что мы хотим просмотреть значения всех наших переменных. Для этого необходимо ввести имена всех переменных в правое нижнее окно. После нажатия клавиши F10 курсор перемещается к следующей строке

`d = a+b;`

и начальные значения переменных появляются в обоих нижних окнах. Последующие нажатия клавиши F10 позволяют видеть в этих окнах все изменения просматриваемых переменных при активизации каждой новой строки программы (изменяемые на текущем шаге переменные выделяются цветом). Если необходимо, то программу можно запустить повторно с самого начала. Для этого необходимо выбрать опцию Debug — Restart или просто нажать клавиши Ctrl — Shift — F5.

В программе можно установить точки останова. Для этого надо переместить курсор на строку программы, перед которой необходимо остановить ее выполнение, и нажать клавишу F9 (как и выше можно использовать и другие способы задания точек останова). Точка останова индицируется красным кружком в пра-

вой точке экрана. Убрать точку останова можно повторным нажатием той же клавиши F9 в этой же строке. Теперь для выполнения программы до точки останова надо просто нажать клавишу F5 или выбрать опции Debug — Go или использовать другие методы. Аналогичный прием используется для приостановки выполнения программы в точке, где находится курсор. Для этого, например, курсор перемещается в нужную строку и нажимается клавиша F5.

Если в программе поместить курсор манипулятора «мышь» на имя переменной, то рядом в маленьком прямоугольном окне появится значение этой переменной. Любое значение отображается в состоянии приостановки, т. е. если значение некоторой переменной менялось от начала выполнения до точки останова, то выводиться будет последнее значение.

Для просмотра значений переменных можно использовать и другой способ. Для этого надо выбрать переменную, переместив на нее курсор и нажав левую кнопку «мыши», и затем выбрать опции Debug — QuickWatch (можно просто нажать клавиши Shift-F9). С помощью появившегося окна можно просмотреть и изменить значение выбранной переменной. Его можно использовать и для вычисления и ввода новых значений. Например, в окне Expression можно записать выражение $d = a + b + 12$ и выбрать кнопку Recalculate. В результате значение d в нижнем окне будет изменено на 15. Теперь это новое значение и будет использоваться в последующих вычислениях. Добавим простейшую функцию f в наш пример:

```
#include<iostream.h>
int f(int i)           // в функцию f передается целое значение (i) и эта
                      // функция возвращает целое значение
{
    i += 2;           // переданное в функцию целое значение
                      // увеличивается на 2
    return ++i; }     // новое значение увеличивается на 1 (инкремент)
                      // тируется) и возвращается из функции в вызывающую программу

void main(void)
{
    int a=1,b=2,c=3,d;
    d = a+b;
    a = a+b+c+d;
    a--;
    cout << "a = " << a << "; b = " << b <<
        " ; c = " << c << "; d = " << d << endl;
    // значение, вычисленное функцией f,
    // выводится на экран дисплея
    cout << "value = " << f(a) << endl;
}
```

Теперь результаты работы программы будут такими:

```
a = 8; b = 2; c = 3; d = 3  
value = 11
```

Для того чтобы войти в тело новой функции `f`, надо нажимать в пошаговом режиме клавишу `F11` вместо клавиши `F10`, т. е. сначала надо нажимать клавишу `F10`, а когда курсор переместится в строку с вызываемой функцией

```
cout<<"value ="<<f(a)<<endl;
```

нажать клавишу `F11`. После того, когда курсор переместится, например, в строку

```
i += 2;
```

можно выбрать функцию `f(int)` в прямоугольнике `Context` окна `Variables`. В результате можно видеть разные вычисляемые значения локальной переменной `i` и то, что функция `f` была вызвана из функции `main`.

Для просмотра последовательности вызываемых функций можно также использовать окно `Call Stack` (опции `View — Debug Window — Call Stack`). Здесь можно видеть как имена вызываемых функций, так и значения параметров, которые им передаются (например, при вызове функции `f` значение `i` равно 8).

Три нерассмотренные опции в подменю `View — Debug Windows`, которыми являются `Memory`, `Registers` и `Disassembly`, позволяют анализировать программу на низком уровне, что требует дополнительных знаний внутренних ресурсов компьютера и языка Ассемблера.

Окно `Memory` позволяет увидеть содержимое фрагмента оперативной памяти по заданному в верхней части окна адресу.

Окно `Registers` позволяет просмотреть значения, записанные в регистрах микропроцессора. Это полезно при отладке программы на уровне ассемблерных команд. Рассмотрим пример:

```
#include <iostream.h>  
void main(void)  
{    unsigned a; // объявляется беззнаковая  
      // переменная a целого типа  
    __asm {    mov ax,4 ; в регистр AX записывается значение 4  
               push ax ; содержимое регистра AX записывается  
                       ; в стековую память  
               add ax,2 ; к значению регистра AX прибавляется  
                       ; число 2  
               pop bx  ; содержимое регистра BX записывается  
                       ; в стековую память
```

```

sub ax,bx ; от значение в регистре AX вычитается
           ; значение в регистре BX
mov word ptr a,ax ; значение из регистра AX
               ; записывается в переменную a
    }
cout << "a = " << a << endl; // вывод на экран значения
                               // переменной a
}

```

Здесь в части программы, написанной на языке Ассемблера (она начинается с ключевого слова `_asm`), комментарии задаются после точки с запятой. Результат работы программы представляется в виде:

```
a = 2
```

Если в процессе отладки программы вывести окно `Registers`, то можно видеть, как изменяются значения регистров процессора. Используемые имена `ax` и `bx` задают правые 16-битные половины соответствующих расширенных 32-битных регистров `eax` и `ebx`. Эти регистры можно задать в приведенной выше программе вместо регистров `ax` и `bx` и тогда будет меняться 32-битное значение. Такая измененная программа приведена ниже:

```

#include <iostream.h>
void main(void)
{
    unsigned long a;
    __asm {
        mov eax,4
        push eax
        add eax,2
        pop ebx
        sub eax,ebx
        mov dword ptr a,eax
    }
    cout << "a = " << a << endl;
}

```

Если выполнить эту программу при активном окне `Memory`, то можно увидеть значение 4, которое заносится в стек командой `push eax`. Для этого необходимо указать адрес памяти, взятый из регистра указателя стека, — `ESP`. Записанное 32-битное значение (четырёхразрядное шестнадцатеричное значение) читается справа налево (40 00) и оно будет выделено цветом. Значения регистров микропроцессора при необходимости можно изменять в процессе выполнения программы, используя, например, опции `Debug` — `QuickWatch`.

Программу, написанную на языках Си/Си++, тоже можно деассемблировать, т. е. автоматически получить ее ассемблерный код. Например, начало нашей первой программы после деассемблирования будет выглядеть на экране дисплея примерно так:

```

5:  void main(void)
6:  {  int a=1,b=2,c=3,d;
0040E49A  push     ebp
0040E49B  mov      ebp,esp
0040E49D  sub      esp,10h
0040E4A0  mov      dword ptr [a],1
0040E4A7  mov      dword ptr [b],2
0040E4AE  mov      dword ptr [c],3
7:      d = a+b;
0040E4B5  mov      eax,dword ptr [a]
0040E4B8  add      eax,dword ptr [b]
0040E4BB  mov      dword ptr [d],eax
8:      a = a+b+c+d;
0040E4BE  mov      ecx,dword ptr [a]
0040E4C1  add      ecx,dword ptr [b]
0040E4C4  add      ecx,dword ptr [c]
0040E4C7  add      ecx,dword ptr [d]
0040E4CA  mov      dword ptr [a],ecx
9:      a--;
0040E4CD  mov      edx,dword ptr [a]
0040E4D0  sub      edx,1
0040E4D3  mov      dword ptr [a],edx
.....

```

Здесь указаны номера строк и строки нашей исходной программы и соответствующие ассемблерные команды, записанные под ними. Здесь можно анализировать выполнение программы на низком уровне. Например, локальные для функции `main` переменные `a`, `b` и `c` хранятся в области стека и, если указать адрес стека в окне Мемори, взятый из регистра ESP, то можно видеть, как происходит инициализация (установка начальных значений) этих переменных.

В трех следующих главах книги будут рассмотрены языки программирования Си и Си++. Примеры программ в своем большинстве будут строиться как консольные приложения. В пятой главе будет показано, как создавать несложные программы, работающие под управлением **Windows**. *

1.5. Введение в интегрированное окружение Borland C++

Работа с главным меню Borland C++. После загрузки Borland C++ на экране появится главное меню, показанное на рис. 1.1. Оно включает три компонента: строку главного меню в верхней части экрана, область для отображения окон в средней части и строку состояния в нижней части экрана.

Главное меню предоставляет возможность доступа ко всем командам Borland C++. Чтобы сделать активной верхнюю строку экрана, необходимо нажать клавишу «F10». В это время один из заголовков (например, File) будет выделен прямоугольником обратной засветки (световым маркером). Его можно перемещать на другие заголовки с помощью клавиш «←» и «→». Предположим, что выбран заголовок File. Тогда после нажатия клавиши «Enter» на экране появится подменю, включающее набор команд для манипуляций с файлами.

Предположим, что нужно завершить работу с подменю File. Для этого необходимо нажать клавишу «ESC». Эта клавиша также может отменить любое другое действие. Некоторые команды подменю могут быть помечены символами «...» (например, save as... в подменю File) или «▶» (например, Compiler в подменю Options). Первый из них говорит о том, что будет отображен блок диалога, а второй о том, что это подменю вызывает еще одно новое подменю.

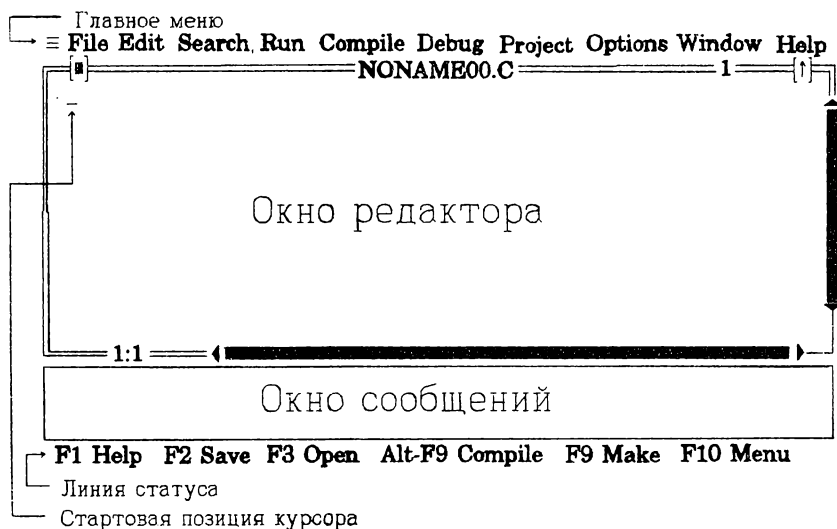


Рис. 1.1. Меню системы программирования Borland C++

Рядом с некоторыми командами записаны имена клавиш или их комбинаций (например, «Save» «F2» или «Quit Alt—X» в подменю File). Их нажатие приводит к вызову соответствующей команды. Например, если после загрузки Borland C++ нажать клавиши «Alt—X», то сразу осуществляется выход из системы в среду ОС.

К любому подменю, показанному на рис. 1.1, можно обратиться и по-другому. Для этого нажимается клавиша «Alt» и клавиша, соответствующая выделенной букве (цветом или яркостью) нужного подменю. Например, подменю File можно вызвать нажатием клавиш «Alt—F».

Меню Borland C++ позволяет использовать не только клавиатуру, но и манипулятор «мышь». Предварительно необходимо загрузить соответствующий драйвер (например, GMOUSE.COM). По умолчанию для выполнения тех или иных действий используется только левая кнопка манипулятора. Чтобы выбрать тот или иной пункт меню, надо установить на него маркер мыши и нажать эту кнопку. Так же выбирается требуемая команда.

Работа с окнами. Важное место в меню Borland C++ занимают окна. Их можно создавать (открывать) и уничтожать (закрывать). В любой момент времени допускается создание многих окон, однако активным является лишь одно из них. Активное окно всегда располагается на переднем плане экрана и заключается в прямоугольник из двойных линий. К нему, как правило, относятся выбираемые команды или вводимый текст. Существует несколько типов окон, которые имеют некоторые или все элементы, показанные на рис. 1.2.

Поясним действия, которые можно выполнить с помощью манипулятора «мышь»:

чтобы быстро закрыть окно, необходимо подвести маркер мыши к элементу в левом верхнем углу (см. рис. 1.2) и нажать левую кнопку;

чтобы сделать окно активным, маркер мыши устанавливается на его номер и нажимается левая кнопка. То же самое произойдет при нажатии клавиши «Alt — номер»;

элемент [▲] используется для увеличения размеров окна на весь экран. Здесь можно воспользоваться манипулятором «мышь» либо нажать клавишу «F5». После расширения окна на весь экран рассматриваемый элемент примет вид [◆]. Преобразование окна к прежнему виду осуществляется путем нажатия клавиши «F5» либо с помощью манипулятора «мышь»;

элементы ►, ▲, ▼, ◀ используются для перемещения текста в окне соответственно вправо, вверх, вниз и влево. Для этого на нужный элемент помещается маркер мыши и нажимает-

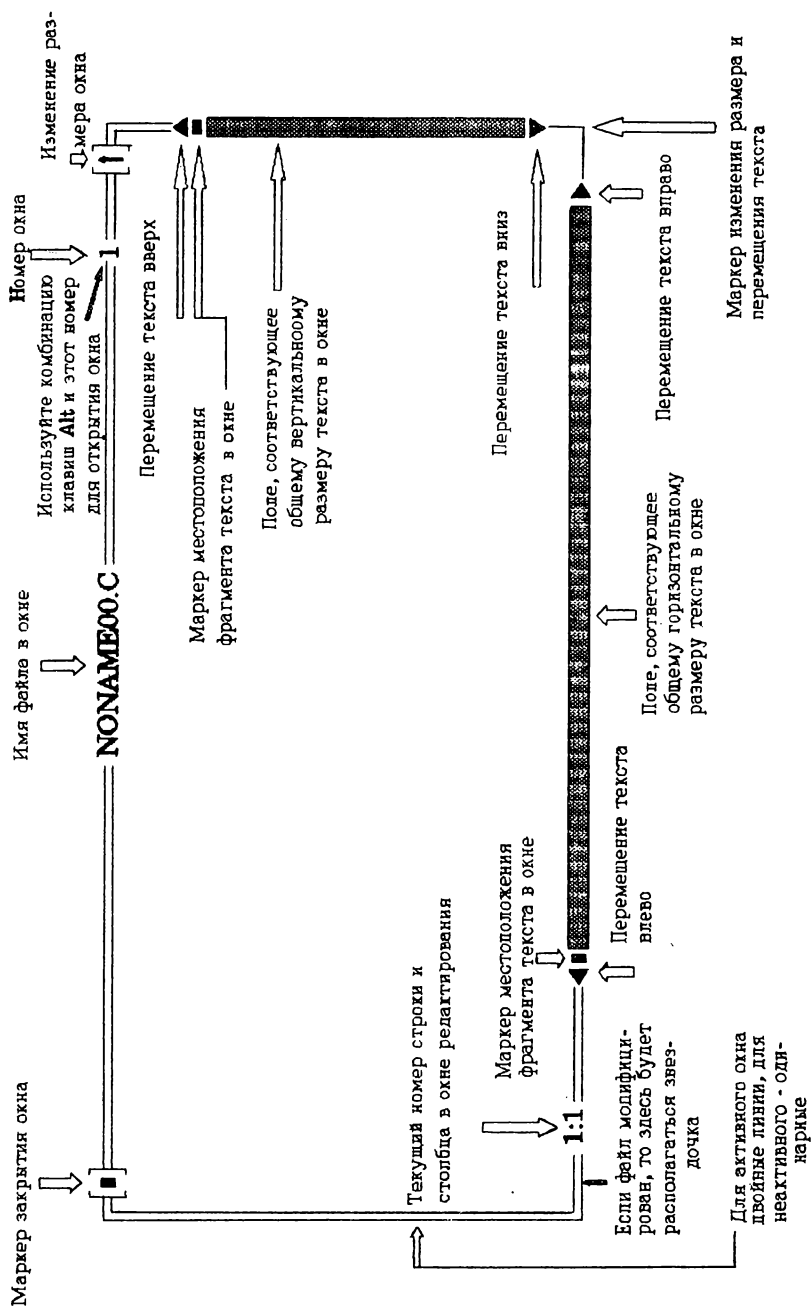


Рис. 1.2. Основные компоненты окна

ся левая кнопка. Когда текст переместится на требуемую величину, левая кнопка отпускается. Маркер местоположения текста перемещается по вертикальному или горизонтальному полю и показывает относительное местоположение выбранного фрагмента в общем тексте;

разрешается перемещать окно по экрану и изменять его размеры. Для этого можно воспользоваться меню Window. Его команды довольно простые. Многие из них будут пояснены в ходе последующего изложения. Изменить размеры окна можно так: установить маркер мыши на одинарную линию в правом нижнем углу; нажать левую кнопку; перемещением «мыши» установить нужный размер; отпустить левую кнопку.

Для чего нужна строка состояния. Строка состояния (линия статуса), расположенная в нижней части экрана (см. рис. 1.1), выполняет следующие функции:

напоминает о клавишах, которые в данный момент времени можно использовать в активном окне. Например, на рис. 1.1 клавиша «F1» дает возможность получить помощь по системе, «F2» — сохранить на текущем диске файл из активного окна редактора и т. п.;

позволяет выполнять требуемые действия с помощью манипулятора «мышь». Для этого маркер мыши устанавливается на нужное имя и нажимается ее левая кнопка;

информирует пользователя о действии, которое выполняет система в текущий момент времени (например, о сохранении файла);

предлагает советы и рекомендации по выбранной команде меню и элементам блока диалога.

Строка состояния изменяется при переключении от одного экрана к другому.

Добавим несколько слов о возможности получения подсказок, помощи и разнообразной справочной информации. Ко всему этому есть доступ из подменю Help. Оно позволяет получать все необходимые сведения по различным режимам работы системы и языку Си++. Обратим внимание на одну из команд Topic Search. Достаточно в окне редактора установить курсор на элемент языка или библиотечную функцию и нажать клавиши «Ctrl—F1», как в окне экрана появится полная информация по выбранному элементу.

Установка параметров рабочего окружения. При работе с меню Borland C++ используются некоторые стандартные файлы, определенные в системе программирования. Например, файлы типа h содержат описание библиотечных функций, глобальных переменных и т. п. Директива вида:

```
#include > файл_типа h >
```


приводит к включению в программу соответствующего файла, причем он должен выбираться из стандартно заданного поддиректория (об этом, в частности, говорят используемые угловые скобки <i>). Имя поддиректория (например, INCLUDE) нужно сообщить системе. Для этого последовательно выбираются меню Options и его команда Directories. В результате на экране появляется блок диалога. В нем есть три стандартные кнопки: выполнить (OK), отменить (Cancel) и подсказка (Help). Выбор нужной кнопки можно осуществить путем перемещения на нее прямоугольника обратной засветки с помощью клавиш «Tab» или «Shift—Tab». Затем необходимо нажать клавишу «Enter». Те же самые действия можно выполнить с помощью манипулятора «мышь». Для этого маркер мыши перемещается в соответствующую позицию экрана и нажимается ее левая кнопка. Кроме того, можно воспользоваться комбинацией клавиш. В этом случае нажимается клавиша «Alt» в сочетании с выделенной буквой нужного пункта. Например, комбинация клавиш «Alt—K» приведет к активизации меню OK.

Если выбрана кнопка OK, то Borland C++ активизирует те варианты, которые указаны в текущем блоке диалога. Кнопка Cancel не изменяет параметров и не выполняет никаких действий. Кнопка Help выводит на экран справочную информацию по выбранному блоку. Клавиша ESC всегда приводит к выполнению тех же действий, что и кнопка Cancel.

Для того чтобы задать поддиректории с файлами типа h, необходимо выбрать прямоугольник «Include Directories» и ввести нужную строку, например C:\T\INCLUDE.

Еще одну важную группу составляют библиотечные файлы типа LIB. Они включают набор стандартных программ для выполнения часто встречающихся функций (например, вычисление квадратного корня, сравнение двух строк и т. п.). Эти файлы могут быть помещены в поддиректорий LIB. Система должна знать об этом. Поэтому в прямоугольник «Library Directories» следует ввести нужную строку, например C:\T\LIB. Заметим, что в этом же поддиректории (C:\T\LIB) должны содержаться файлы Borland C++ CO?. OBJ. Здесь можно задавать имена нескольких поддиректориев, которые должны разделяться точкой с запятой (;), например: C:\T\LIB; C:\T\MYLIB.

В прямоугольнике «Output Directory» указывается поддиректорий, в который будут помещаться полученные объектные (типа OBJ) и выполняемые (типа EXE) файлы. Последний прямоугольник «Source Directories» определяет директории для исходных файлов.

Все рассмотренные параметры окружения будут установлены в процессе инсталляции Borland C++. Поэтому пояснения, кото-

рые даны в этом подразделе, нацелены лишь на то, чтобы показать, как их можно изменить (если возникнет такая необходимость).

В других блоках диалога могут присутствовать так называемые триггерные или селективные кнопки. При выборе триггерной кнопки она помечается символом X, например [X] Whole words only. Тогда соответствующий режим будет включен. Пустая триггерная кнопка (например, [] Whole words only) говорит о том, что соответствующий режим выключен. Включение/выключение режимов осуществляется одним из следующих двух способов:

перемещением на нужное поле маркера мыши и нажатием ее левой кнопки;

выделением нужной кнопки с помощью клавиш «Tab» (Shift—Tab) и нажатием клавиши пробела. Можно также нажать клавишу «Alt» в комбинации с выделенной буквой.

Селективные кнопки имеют то отличие, что они позволяют выбрать только один вариант из заданной группы. Включение режима индицируется звездочкой (*). Например, в группе

(*) From cursor

() Entire scope

можно выбрать первую строку или вторую, но не обе вместе. Включение/выключение режимов, связанных с селективными кнопками, осуществляется выбором нужной группы и использованием клавиш со стрелками.

Другие команды подменю Options позволяют просматривать и модифицировать параметры, определяющие функционирование Borland C++. Если вы только начинаете работать с Borland C++, то эти параметры не следует изменять. Тогда все они будут иметь значения, заданные по умолчанию при инсталляции системы.

1.6. Выполнение программ на языке Си ++

Подготовка исходного текста программы. Подготовка и корректировка исходных текстов программ осуществляются с помощью встроенного экранного редактора текста. Начать работу с новым файлом можно с помощью команд подменю File. Выберем сначала команду New. После этого создается новое активное окно редактора и в нем открывается файл с именем NONAMExx.C. На место символов xx будут подставляться цифры от 00 до 99.

Теперь можно набрать текст какой-нибудь программы. Если нажать клавишу «F2» для сохранения файла на текущем диске, то на экране появится блок диалога. Он позволяет изменить имя

(NONAMExx.C.) и поддиректорий для файла. Здесь же можно увидеть, что содержится в выбранном поддиректории.

Создать новый файл можно также с помощью команды Open подменю File. Она же позволяет загрузить в редактор текст одной из ранее подготовленных программ. Блок диалога показывает имя текущего поддиректория (например, C:\EXAMPLES) и все содержащиеся в нем файлы (например, EX1_1.CPP и EX1_2.CPP). Можно выбрать любой файл из списка. Это производится перемещением прямоугольника обратной засветки на нужное имя (EX1_1.CPP или EX1_2.CPP) и активизацией соответствующей кнопки. Выбор нужного поля блока диалога осуществляется клавишами «Tab» и «Shift—Tab», а выбор файла — клавишами со стрелками. Кроме того, можно воспользоваться манипулятором «мышь». Кнопка Open применяется для активизации нового окна с выбранным файлом. Кнопка Replace позволяет заменить файл в активном окне на файл, выбранный в блоке диалога.

Если в поле Name команды Open указать имя файла, которого нет в поддиректории, то открывается новый файл. С ним можно работать так же, как и по команде New.

Кроме рассмотренных выше команд можно выполнять следующие действия:

- набрать в поле Name имя файла с символами * и ? (например, MY*.CPP). Это приводит к тому, что в окне Files останутся только те файлы, которые соответствуют заданному шаблону;

- нажать клавишу «↓» (когда курсор находится в поле Name) либо выбрать мышью соответствующий элемент блока диалога. После этого на экран выводится список предыстории, содержащий имена файлов, с которыми работали раньше;

- просмотреть содержимое других директориев. Для этого необходимо выбрать в поле Files нужное имя.

Рассмотрим вопросы, связанные с редактированием исходного текста программы. Команды редактора Borland C++ помещены в табл. 1.1. Многие приемы редактирования текста предполагают работу с блоками. **Блок** — это любой фрагмент текста от одного знака до сотен строк. Одновременно можно выделить только один блок. Для этого отмечаются его начало и конец. Далее блок можно записать на диск, прочитать с диска и поместить в любое место программы, сохранить в специальном текстовом буфере, переместить, создать копию в программе и т. п. Например, можно выполнять такие действия:

1. Отметить начало некоторого блока. Для этого курсор перемещается в нужную точку и нажимаются клавиши «Ctrl—K—B». Эти же действия можно выполнить, нажимая клавиши управления перемещением курсора при нажатой клавише «Shift».

2. Отметить конец некоторого блока. Для этого курсор перемещается в нужную точку и нажимаются клавиши «Ctrl—K—K». Эти же действия можно выполнить, нажимая клавиши управления перемещением курсора при нажатой клавише «Shift». Помеченный фрагмент будет выделен цветом или яркостью.

3. Сохранить выделенный фрагмент в текстовом буфере (Clipboard). Для этого нажимаются клавиши «Ctrl—Ins».

4. Копировать выделенный текст в новое место экрана. Для этого курсор устанавливается в нужное место и нажимаются клавиши «Shift—Ins». В результате сохраненный фрагмент будет перенесен из текстового буфера на экран. Скопировать текст можно было сразу же в п. 3, переместив курсор и нажав клавиши «Ctrl—K—C» (см. табл. 1.1). Однако использование текстового буфера позволяет переносить блок из одного окна в другое.

Выделить требуемый фрагмент текста можно также с помощью манипулятора «мышь». Для этого маркер мыши перемещается в нужную точку экрана, нажимается левая кнопка манипулятора, маркер перемещается в конечную точку и левая кнопка отпускается.

При редактировании текста можно воспользоваться командами подменю Edit. Для этого, чтобы лучше понять их действие, необходимо произвести манипуляции с некоторым текстом. Например, попытаться выделить, сохранить в текстовом буфере, удалить и восстановить фрагменты любой программы. При этом желательно воспользоваться другими командами редактора, рассмотренными в табл. 1.1.

Т а б л и ц а 1.1

Команда	Назначение
Команды управления курсором	
← или Ctrl—S	Перемещение курсора на позицию влево
→ или Ctrl—D	Перемещение курсора на позицию вправо
Ctrl— ← или Ctrl—A	Перемещение курсора на слово влево
Ctrl— → или Ctrl—F	Перемещение курсора на слово вправо
↑ или Ctrl—E	Перемещение курсора на строку вверх
↓ или Ctrl—X	Перемещение курсора на строку вниз
Ctrl—W	Перемещение текста на строку вниз
Ctrl—Z	Перемещение текста на строку вверх
PgUp или Ctrl—R	Перемещение текста на страницу вверх
PgDn или Ctrl—C	Перемещение текста на страницу вниз
Home или Ctrl—Q—S	Перемещение курсора в начало строки
End или Ctrl—Q—D	Перемещение курсора в конец строки
Ctrl—Home или Ctrl—Q—E	Перемещение курсора в начало страницы
Ctrl—End или Ctrl—Q—X	Перемещение курсора в конец страницы

Команда	Назначение
Ctrl—PgUp или Ctrl—Q—R	Перемещение курсора в начало файла
Ctrl—PgDn или Ctrl—Q—C	Перемещение курсора в конец файла
Ctrl—Q—B	Перемещение курсора в начало блока
Ctrl—Q—K	Перемещение курсора в конец блока
Ctrl—Q—P	Перемещение курсора в позицию, где он находился перед последней командой

Команды вставки и удаления

Ins или Ctrl—V	Включить/выключить режим вставки
Ctrl—N	Вставить строку
Ctrl—Y	Удалить строку
Ctrl—Q—Y	Удалить символы до конца строки
Backspace или Ctrl—H	Удалить символ слева от курсора
Del или Ctrl—G	Удалить символ под курсором
Ctrl—T	Удалить слово справа от курсора

Команды для работы с блоками (фрагментами) текста

Shift ↓, ↑, ←, → или Ctrl—K—B	Отметить начало блока
Shift ↓, ↑, ←, → или Ctrl—K—K	Отметить конец блока
Ctrl—K—T	Отметить единственное слово как блок
Ctrl—Ins, Shift—Ins или Ctrl—K—C	Копировать блок
Shift—Del, Shift—Ins или Ctrl—K—V	Переместить блок
Ctrl—Del или Ctrl—K—Y	Удалить блок
Ctrl—K—H	Включить/выключить индикацию блока
Ctrl—K—W	Записать блок на диск
Ctrl—K—R	Прочитать блок с диска
Ctrl—K—P	Вывести блок на принтер
Ctrl—K—I	Сместить блок на шаг вправо (Indent block)
Ctrl—K—U	Сместить блок на шаг влево (Unindent block)
Ctrl—K—D	Выйти в главное меню
Ctrl—K—L	Отменить строку

Смешанные команды

Ctrl—U	Прекратить выполнение команды
Ctrl—O—I	Включить/выключить режим автоабзаца
Ctrl—P	Вставить управляющий символ
Ctrl—Q—F	Найти
Ctrl—Q—A	Найти и заменить
Ctrl—Q—n	Найти место метки в тексте (n — цифра от 0 до 9)

Команда	Назначение
Ctrl—K—D или Ctrl—K—Q	Выход из режима редактирования без сохранения текста
Ctrl—L	Повторить последний поиск
Ctrl—Q—L	Восстановить строку
F2 или Ctrl—K—S	Сохранить текст
Ctrl—K—n	Установить метку (n — цифра от 0 до 9)
Tab или Ctrl—I	Табуляция
Ctrl—O—T	Включение/выключение табуляции
Ctrl—Q—[Нахождение предыдущей скобки
Ctrl—Q—]	Нахождение следующей скобки

При работе с текстом важное место занимают команды поиска некоторых его фрагментов (слов, предложений и т. п.) и, возможно, их замены. Эти действия можно выполнить с помощью команд редактора либо из подменю Search. Первая команда Find выводит на экран блок диалога, который позволяет ввести образец для поиска и задать параметры, влияющие на поиск. Точно такой же блок диалога появится и по команде Ctrl—Q—F. Поясним некоторые параметры настройки:

если установить режим «Case sensitive», то различаются символы верхнего и нижнего регистров;

если установить режим «Whole words only», то осуществляется поиск только целых слов (с обеих сторон они должны окружаться символами пунктуации или пробела);

если установить режим «Forward», то поиск осуществляется по тексту вперед;

если установить режим «Backward», то поиск осуществляется по тексту назад;

если установить режим «Global», то поиск выполняется во всем тексте;

если установить режим «Selected text», то поиск выполняется в выделенном фрагменте текста;

если установить режим «From cursor», то действие команд Forward и Backward распространяется на текст от позиции курсора;

если установить режим «Entire scope», то действие команд Forward и Backward распространяется на весь текст.

Если выбрать команду Find, когда курсор в окне редактирования установлен на какое-нибудь слово, то можно осуществлять поиск этого слова. При нажатии клавиши «→» из текста выбираются дополнительные символы.

Команда Replace выводит на экран похожий блок диалога. Главное отличие заключается в том, что наряду с искомой строкой задается строка для замены. Новая кнопка «Change All» задает все возможные замены. Дополнительная триггерная кнопка «Prompt to Replace» позволяет требовать запрос на каждое изменение с целью подтверждения намерений пользователя.

Компиляция, компоновка и выполнение программ. Предположим, что необходимо разработать программу с именем MYPROG. Все, что необходимо сделать для этого, можно представить в виде следующей простой схемы:

MYPROG.CPP → MYPROG.OBJ → MYPROG.EXE

Сначала нужно создать файл с исходным текстом программы на языке Си++ MYPROG.CPP, затем в окне редактора набрать текст соответствующей программы и записать его в этот файл. Далее выполняется компиляция программы и образуется объектный файл (MYPROG.OBJ). На последней стадии после подключения компоновщика строится выполняемый файл (MYPROG.EXE). Его уже можно загружать и запускать в персональном компьютере.

Для компиляции, компоновки и выполнения программы используются подменю Compile и Run (см. рис. 1.1). Предположим, что в активном окне подготовлен файл MYPROG.CPP. Тогда выбор команды Compile to OBJ (можно просто нажать клавиши «Alt—F9») приведет к тому, что будет получен файл MYPROG.OBJ. В процессе работы в центре экрана появится окно, которое информирует о прохождении процесса компиляции и его результатах. При обнаружении ошибок в программе на экран выведется мигающая строка «Press any key». После нажатия любой клавиши появится новое окно с сообщениями об ошибках. Просмотреть их можно с помощью клавиш «↑» и «↓». Для каждого сообщения об ошибке выделяется предполагаемая строка программы, которая могла ее вызвать. Точно так же выдается информация о предупреждениях (Warning). По предположению системы каждое предупреждение — это тоже ошибка (однако она в этом не уверена). Не все предупреждения относятся к категории ошибок. В связи с этим в большинстве случаев (в частности, если нет специальных указаний пользователя) процесс компиляции не прерывается и строится выполняемый файл. Однако, особенно для начинающего пользователя, почти всегда все предупреждения будут относиться к категории ошибок. Поэтому следует прекратить компиляцию и выяснить их причину.

Команда Make EXE file используется при создании мультифайловых проектов. Пока отметим лишь то, что она осуще-

ствляет избирательную перекомпиляцию только тех файлов, исходный код которых был обновлен после создания для них объектных файлов:

Команда Link EXE file использует текущий объектный (типа OBJ) и библиотечные (типа LIB) файлы и строит выполняемый (типа EXE) файл. Команда Build all осуществляет полную рекомпиляцию всех файлов без учета даты и времени их создания.

Подменю Run (см. рис. 1.1) используется для того, чтобы задать выполнение программы, а также для инициализации и завершения сеанса отладки. Команда Run «Ctrl—F9» запускает на выполнение текущую программу и может передать ей аргументы, заданные в соответствующем блоке диалога (см. меню Run, команда Arguments). Если с момента последней компиляции исходный код был модифицирован, то будет проведена повторная компиляция и компоновка.

Приведем некоторые сведения о подменю Project (см. рис. 1.1), которое используется для создания мультифайловых проектов. В целом оно позволяет объединить некоторое подмножество файлов в единый файл проекта и управлять этим проектом.

Команда Open project отображает блок диалога Open Project File, который позволяет выбрать и загрузить либо создать новый проект (файл типа PRJ) по заданному имени. Здесь выполняются примерно те же действия, что и в команде Open подменю File.

Команды Add item и Delete item выбираются, когда в проект необходимо добавить либо удалить из него какой-нибудь файл. На экране появляется блок диалога, в который вписывается требуемое имя (добавляемого или удаляемого файла). Если задан файл проекта, то система использует его имя при создании выполняемого файла с расширением EXE.

Отладка программ. В процессе отладки можно:

1. Осуществлять пошаговое выполнение программы. После прохода каждой ее строки производится приостановка, позволяющая проанализировать промежуточные результаты, содержимое программно-доступных регистров микропроцессора и т. п. Допускается задание приостановок в каждой строке программы, в заранее указанных строках программы либо в строках, в которых в ходе отладки перемещается курсор.

2. Проверять значение и местоположение (адрес) некоторой переменной в ходе выполнения программы.

3. Анализировать значения переменных и выражений и, возможно, изменять их в ходе выполнения программы.

4. Просматривать последовательность вызова функций в программе.

5. Удалять и добавлять просматриваемые переменные и выражения в ходе выполнения программы.

Управление возможностями интегрированного отладчика осуществляется с помощью подменю Debug, показанного на рис. 1.1. Дадим краткую характеристику некоторым его командам. Первая из них Inspect позволяет проанализировать и при необходимости модифицировать значение элемента данных. Существует два способа открытия окна Inspecting:

можно установить курсор на выбранный элемент данных и нажать клавиши «Alt—F4»;

вызвать подменю Debug и далее блок диалога Data Inspect, в который ввести требуемый идентификатор. Можно также установить курсор на нужное имя и выбрать команду Inspect, а затем, находясь в блоке диалога, нажать клавишу «→», чтобы вывести из текста дополнительные символы.

Когда элемент окна Inspecting может быть изменен, в нижней части экрана появляется строка Alt—M. Теперь если нажать клавиши «Alt—M», то в блоке диалога будет запрошено новое значение инспектируемой переменной.

Значение переменной в блоке Inspecting обычно отображается в виде десятичного числа, за которым в скобках следует соответствующее шестнадцатеричное число со стандартным для языков Си и Си++ префиксом 0x. Если переменная имеет символьный тип, то отображается также символ, который соответствует заданному числовому значению.


Команда Evaluate/modify вычисляет и отображает в блоке диалога значение переменной или выражения. В некоторых случаях вычисленное значение может быть оперативно модифицировано. Поле Expression показывает слово, содержащее курсор в окне редактирования. Можно вычислить его значение после нажатия клавиши «Enter» либо предварительно изменить его. Как и выше, в случае нажатия клавиши «→» будут выводиться дополнительные символы из окна редактирования. Если отладчик может вычислить значение выражения, то оно отображается в поле Result. В поле New Value можно задать новое значение элемента, которое далее используется в программе. Кнопка Evaluate выбирается для вычисления значения выражения. Кнопка Modify используется в случае изменения значения элемента данных.

Команда Watches открывает новое подменю. Оно позволяет добавлять, удалять и редактировать в окне просмотра переменные и выражения. Команда Toggle breakpoint позволяет установить или отменить безусловную точку останова в строке с курсором.

Команда Breakpoints открывает блок диалога, который позволяет управлять использованием точек приостанова. Все такие точки будут показаны на экране. Для них задаются номера соответствующих строк, а также условия, при которых произойдет приостановка работы программы.

Как используются активные клавиши. Активные клавиши вызывают те же действия, что и многие команды меню. Если запомнить некоторые из них, то эффективность работы в среде Borland C++ будет выше. Теперь уже не надо последовательно проходить по системе меню для поиска нужной команды. Достаточно нажать одну либо две клавиши. Ниже приводится табл. 1.2 с информацией об основных активных клавишах (hot keys) системы программирования Borland C++ и действиях, которые они вызывают.

Таблица 1.2

Клавиша	Выполняемое действие
Активные клавиши общего назначения	
F1	Отображает окно с подсказкой
F2	Сохраняет на текущем диске файл из активного окна редактора
F3	Отображает блок диалога для открытия файла
F4	Выполняет программу до строки, содержащей курсор, либо до точки останова
F5	Расширяет активное окно на весь экран
F6	Выбирает следующее открытое окно
F7	Выполняет очередной шаг программы на этапе отладки (с вхождением в вызываемые функции)
F8	Выполняет очередной шаг программы на этапе отладки (без вхождения в вызываемые функции)
F9	Компилирует файл (или проект) в активном окне
F10	Активизирует главное меню Borland C++
Активные клавиши для выбора разделов главного меню	
Alt — пробел	Выбирает меню 
Alt — C	Выбирает меню Compile
Alt — D	Выбирает меню Debug
Alt — E	Выбирает меню Edit
Alt — F	Выбирает меню File
Alt — H	Выбирает меню Help
Alt — O	Выбирает меню Options
Alt — P	Выбирает меню Project
Alt — R	Выбирает меню Run

Клавиша	Выполняемое действие
Alt — S	Выбирает меню Search
Alt — W	Выбирает меню Window
Alt — X	Осуществляет выход из Borland C++ в среду ОС

Активные клавиши для управления окнами

Alt — номер	Активизирует окно с заданным номером
Alt — 0	Отображает на экране список окон. Вместе с Alt нажимается клавиша с цифрой ноль
Alt — F3	Закрывает активное окно
Alt — F4	Открывает окно для проверки значений переменных
Alt — F5	Отображает результаты работы программы пользователя
F5	Расширяет активное окно на весь экран
F6	Выбирает следующее открытое окно
Ctrl—F5	Изменяет размер или перемещает активное окно

Активные клавиши оперативной подсказки

F1	Отображает окно с подсказкой
Fi F1	Выдает информацию о том, как пользоваться подсказкой (если система помощи уже вызвана, то необходимо нажать только клавишу F1)
Shift—F1	Отображает перечень ключевых слов, по которым можно получить подсказку
Alt—F1	Отображает предыдущее окно подсказки
Ctrl—F1	Дает справку по выбранному (курсором) элементу языка

Активные клавиши для режимов выполнения и отладки

Alt—F4	Открывает окно для проверки значений переменных
Alt—F7	Осуществляет переход к предыдущей ошибке
Alt—F8	Осуществляет переход к следующей ошибке
Alt—F9	Компилирует программу и строит файл типа OBJ
Ctrl—F2	Осуществляет повторную инициализацию выполняемой программы
Ctrl—F3	Отображает последовательность функций, вызываемых в программе
Ctrl—F4	Вычисляет значение переменной или выражения и позволяет при наличии возможности его модифицировать
Ctrl—F7	Добавляет в окно просмотра переменную или выражение
Ctrl—F8	Позволяет установить или отменить точку останова в строке с курсором
Ctrl—F9	Компилирует, компоует, загружает и выполняет программу
Shift—F3	Компилирует ассемблерную программу

Клавиша	Выполняемое действие
Shift—F4	Подключает автономный отладчик (Turbo debugger)
Shift—F5	Подключает анализирующую программу Turbo Profiler
F4	Выполняет программу до строки, содержащей курсор
F7	Выполняет очередной шаг программы на этапе отладки (с вхождением в вызываемые функции)
F8	Выполняет очередной шаг программы на этапе отладки (без вхождения в вызываемые функции)
F9	Компилирует файл (или проект) в активном окне

ГЛАВА 2

ОБЩИЕ КОНСТРУКЦИИ ЯЗЫКОВ Си И Си++

2.1. Программы и данные

Основные понятия языка. В § 2.1 + 2.5 рассматриваются основные конструкции языка Си. Примеры программ на языке Си, демонстрирующие использование этих конструкций, вынесены в § 2.6.

Тело программы, написанной на языках Си и Си++, состоит из инструкций. Каждая инструкция вызывает некоторые действия на соответствующем шаге выполнения программы. При написании инструкций применяются определенные символы, составляющие алфавит языка. Алфавит включает латинские прописные и строчные буквы, цифры и специальные знаки. К таким знакам, например, относятся: точка (.), запятая (,), двоеточие (:), точка с запятой (;) и др. Внутреннее представление символов алфавита в персональном компьютере осуществляется на основе определенной системы кодирования. Соответствие между каждым символом и его кодом задается в виде специальной кодовой таблицы. На нее разработан стандарт ASCII и поэтому коды символов называют ASCII-кодами.

Различают видимые и управляющие символы. Первые могут быть отображены на экране дисплея либо отпечатаны на принтере. Вторые вызывают определенные действия в машине, например: звуковой сигнал — код 7₁₀; возврат курсора на один шаг — код 8₁₀, горизонтальная табуляция — код 9₁₀, перевод курсора на новую строку — код 10₁₀, перемещение курсора в начало строки — код 13₁₀ и т. п. Такие управляющие символы имеют десятичные номера 0—31, 127. Оговорим формы записи чисел в различных системах счисления. Во-первых, будет использована запись основания системы счисления в виде нижнего индекса (например, 13₁₀, D₁₆, 1101₂ — это одно и то же число). Во-вторых, будут использоваться формы записи чисел в различных системах счисления, принятые в языках Си и Си++ (после того как они будут рассмотрены).

Для представления каждого символа в персональном компьютере используется один байт, поэтому их общее число равно $2^8=256$. Напомним, что каждый байт состоит из восьми битов. Кодовая таблица, которая устанавливает соответствие между символом и его кодом, имеет 256 строк вида:

код_символа_в_заданной_системе_счисления — символ

Рассмотрим пример: 4810—0. Эта строка говорит о том, что символ 0 закодирован кодом 4810. Первая половина кодовой таблицы является стандартной, а вторая используется для представления символов национальных алфавитов, псевдографических элементов и т. п.

Важным понятием языка является идентификатор, который используется в качестве имени объекта (функции, переменной, константы и т. п.). Идентификаторы должны выбираться с учетом следующих правил:

1. Они должны начинаться с буквы латинского алфавита (a,..., z, A,..., Z) или с символа подчеркивания (_).

2. В них могут использоваться буквы латинского алфавита, символ подчеркивания и цифры (0,...,9). Использование других символов в идентификаторах запрещено.

3. В языках Си и Си++ буквы нижнего регистра (a,...,z), применяемые в идентификаторах, отличаются от букв верхнего регистра (A,...,Z). Это означает, что следующие идентификаторы считаются разными: name, NaMe, NAME и т. п.

4. Идентификаторы могут иметь любую длину, но воспринимаются и используются для различения объектов (функций, переменных, констант и т. п.) только определенные символы. Их число меняется для разных систем программирования (в Borland C++ оно может устанавливаться при настройке среды и имеет максимальное значение—32). Предположим, что это число установлено равным 5. Тогда идентификаторы count и counter будут идентичны, поскольку у них первые пять символов совпадают.

5. Идентификаторы для новых объектов не должны совпадать с ключевыми словами языка и именами стандартных функций из библиотеки.

В программах на языках Си и Си++ важная роль отводится комментариям. Они повышают наглядность и удобство чтения программ. Комментарии обрамляются символами /* и */. Их можно записывать в любом месте программы.

Язык Си++ вводит еще одну форму записи комментариев. Все, что записано после знака // до конца текущей строки, будет тоже рассматриваться как комментарий. Рассмотрим пример: // это комментарий.

Далее для пояснения инструкций языка комментарии будут использоваться очень широко. Тексты примеров программ могут набираться как с комментариями, так и без них (комментарии никак не отражаются на выполнении программы).

В программах на языках Си и Си++ пробелы, символы табуляции и перехода на новую строку игнорируются. Это позволяет записывать различные выражения в хорошо читаемом виде. Кроме того, строки программы можно начинать с любой позиции, что дает возможность выделять группы инструкций языка.

Объявления и типы данных. Программы оперируют с различными данными, которые могут быть простыми и структурированными. *Простые данные* — это целые и вещественные числа, символы и указатели (адреса объектов в памяти). Целые числа не имеют, а вещественные имеют дробную часть. Структурированные данные — это массивы, записи и файлы; они будут рассмотрены позже.

В языке различают понятия описания переменной и ее определения (объявления). Описание устанавливает свойства объекта: его тип (например, целый), размер (например, 4 байт) и т. п. Объявление наряду с этим вызывает выделение памяти.

В языке различают понятия «тип данных» и «модификатор типа». Тип данных — это, например, целый, а модификатор — со знаком или без знака. Целое со знаком будет иметь как положительные, так и отрицательные значения, а целое без знака — только положительные значения. В языках Си и Си++ можно выделить пять базовых типов, которые задаются следующими ключевыми словами: `char` — символьный; `int` — целый; `float` — вещественный; `double` — вещественный двойной точности; `void` — не имеющий значения. Дадим им краткую характеристику:

1. Переменная типа `char` имеет размер 1 байт. Ее значениями являются различные символы из кодовой таблицы, например: 'ф', '.', 'j' (при записи в программе они заключаются в одинарные кавычки).

2. Переменная типа `int` имеет размер 2 байт. Знаковые значения этой переменной могут лежать в диапазоне от —32768 до 32767.

3. Ключевое слово `float` позволяет объявить переменные вещественного типа. Их значения имеют дробную часть, отделяемую точкой, например: —5.6, 31.28 и т. п. Вещественные числа могут быть записаны также в форме с

плавающей точкой, например: $-1.09e+4$ (что равно $-1.09 \cdot 10^4$). Число перед символом *e* называется мантиссой, а после *e* — порядком. Переменная типа *float* занимает в памяти 32 бита. Она может принимать значения в диапазоне от $3.4e-38$ до $3.4e+38$.

4. Ключевое слово *double* позволяет объявить вещественную переменную двойной точности. Она занимает в памяти в два раза больше места, чем переменная типа *float* (т. е. ее размер 64 бита). Переменная типа *double* может принимать значения в диапазоне от $1.7e-308$ до $1.7e+308$.

5. Ключевое слово *void* используется для нейтрализации значения объекта, например для объявления функции, не возвращающей никаких значений.

Объект некоторого базового типа может быть модифицирован. С этой целью используются специальные ключевые слова, называемые *модификаторами*. Пока рассмотрим только четыре из них: *unsigned*, *signed*, *short*, *long*. Модификаторы записываются перед спецификаторами типа, например: *unsigned char*. Если после модификатора опущен спецификатор, то компилятор предполагает, что этим спецификатором является *int*. Таким образом, следующие строки:

```
long a;  
long int a;
```

являются идентичными и объявляют объект *a* как длинный целый. Табл. 2.1 иллюстрирует возможные сочетания модификаторов (*unsigned*, *signed*, *short*, *long*) со спецификаторами (*char*, *int*, *float* и *double*), а также показывает размер и диапазон значений объявляемого объекта.

Покажем теперь, где в тексте программы объявляются данные. В языке определены глобальные и локальные объекты. Первые объявляются вне функций и, следовательно, доступны для любой из них. Локальные объекты по отношению к функциям являются внутренними. Они начинают существовать при входе в функцию и уничтожаются после выхода из нее. Рис. 2.1 показывает место в программе, где объявляются глобальные переменные. Здесь же помечена область их действия. На рис. 2.2 показаны возможные места в программе, где объявляются локальные объекты.

В языке Си все объявления должны следовать перед инструкциями, составляющими тело функции. В языке Си++ это ограничение снято и объявления могут следовать в любом месте программы. Если они сделаны в функции, то соответствующие объекты будут *локальными*, а если вне функции, то *глобальными*.

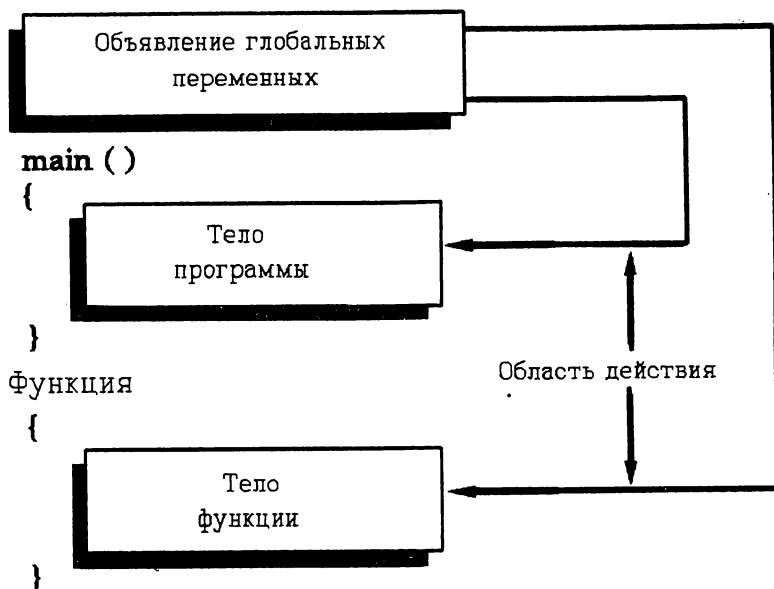


Рис. 2.1. Объявления глобальных переменных

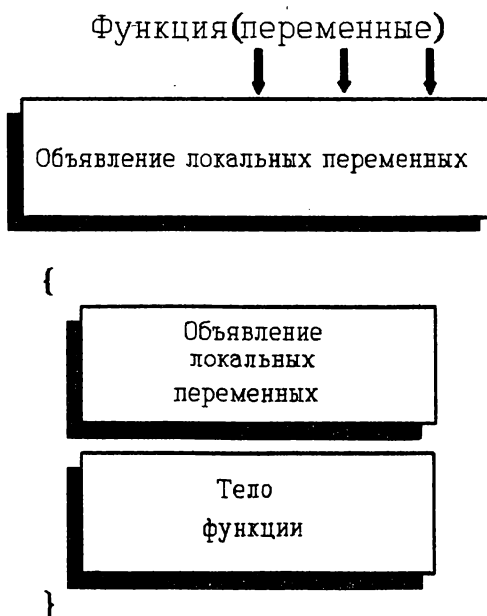


Рис. 2.2. Объявления локальных переменных

Таблица 2.1

Тип	Размер	Диапазон
char		
signed char	8	—128—127
unsigned char	8	0—255
int		
signed int		
short int		
signed short int	16	—32768—32767
unsigned int		
unsigned short int	16	0—65535
long int		
signed long int	32	—2147483648—2147483647
unsigned long int	32	0—4294967295
float	32	$3.4e-38$ — $3.4e+38$
double	64	$1.7e-308$ — $1.7e+308$
long double	80	$3.4e-4932$ — $1.1e+4932$

Рассмотрим примеры объявления данных:

```
int a, b;
long c, d;
unsigned e, f;
char g, h;
float i, j;
double k, l;
```

Здесь объявлены следующие переменные: целые a и b, длинные целые c и d, беззнаковые целые e и f, символьные g и h, вещественные i и j, вещественные двойной точности k и l.

Остановимся еще раз на типе char. В приведенном выше объявлении g и h — переменные типа char. Это значит, что они могут принимать значения, совпадающие с символами кодовой таблицы. Эти значения заключаются в одинарные кавычки.

Например, можно записать выражения:

```
g = 'Q';
h = 'h';
```

здесь g и h — переменные, которые могут принимать разные значения, а 'Q' и 'h' — конкретные значения, присваиваемые этим переменным. После этого в ячейке памяти персонального компьютера, зарезервированной для переменной g, будет записан код символа Q, а для переменной h — код символа h. Переменным g и h можно присвоить новые значения, например:

```
g = '3';
h = 'l';
```

Переменные в языке Си могут быть инициализированы при объявлении.

Делается это, например, так:

```
int a = 10, b = 20;
char c = 'R', ch = '*';
float f = 1.87;
```

Как вводить и выводить информацию. В любых программах функции ввода и вывода играют важную роль. В языках Си и Си++ их очень много. Сначала рассмотрим только те функции, которые встречаются в большинстве программ этого раздела.

Самый простой механизм ввода — чтение по одному символу из стандартного входного потока (с клавиатуры) с помощью функции `getchar`. Она имеет следующий прототип:

```
int getchar (void);
```

Здесь определен тип единственного аргумента (`void`) и тип возвращаемого функцией значения (`int`).

Инструкция вида:

```
x = getchar ();
```

присваивает переменной `x` очередной вводимый символ. Переменная `x` должна иметь символьный или целый тип. Другая функция — `putchar (x)`; выдает значение переменной `x` в стандартный выходной поток (на экран дисплея). Функция `putchar` имеет прототип:

```
int putchar (int);
```

Объявления `getchar` и `putchar` сделаны в стандартном включаемом файле `stdio.h`.

Заметим, что для функции `getchar` после выбора символа необходимо нажать клавишу «Enter». Иногда это создает определенные неудобства. Функции `getch` и `getche` устраняют их. Они имеют следующие прототипы:

```
int getch (void);  
int getche (void);
```

Обе эти функции вводят символ сразу же после нажатия соответствующей клавиши (здесь не надо дополнительно нажимать клавишу «Enter»). Отличие между ними заключается в том, что `getche` отображает вводимый символ на экране дисплея, а `getch` — нет. Прототипы этих функций содержатся в файле `conio.h`.

Форматный вывод данных. Функция `printf` обеспечивает форматный вывод. Ее можно записать в следующем формальном виде:

```
printf («управляющая строка», аргумент _1, аргумент _2,...);
```

Управляющая строка содержит компоненты трех типов: обычные символы, которые просто копируются в стандартный выходной поток (выводятся на экран дисплея); спецификации преобразования, каждая из которых вызывает вывод на экран

очередного аргумента из последующего списка; управляющие символьные константы.

Каждая спецификация преобразования начинается со знака % и заканчивается некоторым символом, задающим преобразование. Между знаком % и символом преобразования могут встречаться другие знаки в соответствии со следующим форматом:

`%(признаки) [ширина_поля] [.точность] [F; N; h; l; L] c_n`

Все параметры в квадратных скобках являются не обязательными.

На месте параметра `c_n` (символ преобразования) могут быть записаны:

- `c` — значением аргумента является символ;
- `d` или `l` — значением аргумента является десятичное целое число;
- `e` — значением аргумента является вещественное десятичное число в экспоненциальной форме вида `1.23e + 2`;
- `E` — значением аргумента является вещественное десятичное число в экспоненциальной форме вида `1.23E + 2`;
- `f` — значением аргумента является вещественное десятичное число с плавающей точкой;
- `g` (или `G`) — используется, как `e` или `f`, и исключает вывод незначащих нулей;
- `o` — значением аргумента является восьмеричное целое число;
- `s` — значением аргумента является строка символов (символы строки выводятся до тех пор, пока не встретится символ конца строки или же не будет выведено число символов, заданное точностью);
- `u` — значением аргумента является беззнаковое целое число;
- `x` — значением аргумента является шестнадцатеричное целое число с цифрами `0, ..., 9, a, b, c, d, e, f`;
- `X` — значением аргумента является шестнадцатеричное целое число с цифрами `0, ..., 9, A, B, C, D, E, F`;
- `r` — значением аргумента является указатель;
- `p` — применяется в операциях форматирования. Аргумент, соответствующий этому символу спецификации, должен быть указателем на целое. В него возвращается номер позиции строки (отображаемой на экране), в которой записана спецификация `%p`.

Рассмотрим необязательные параметры в спецификации преобразования:

признак минус (`—`) указывает, что преобразованный параметр должен быть выровнен влево в своем поле;

признак плюс (`+`) требует вывод результата со знаком;

строка цифр, задающая минимальный размер поля (`ширина_поля`). Здесь может так же использоваться символ `*`, который тоже позволяет задать минимальную ширину поля и точность представления выводимого числа.

точка (`.`), отделяющая размер поля от последующей строки цифр;

строка цифр, задающая максимальное число символов, которое нужно вывести, или же количество цифр, которое нужно вывести справа от десятичной точки в значениях типов `float` или `double` (`.точность`);

символ `F`, определяющий указатель типа `far`;

символ N, определяющий указатель типа near (указатели типов far и near будут рассмотрены ниже);

символ h, определяющий аргумент типа short int (используется вместе с символами преобразования d, i, o, u, x, X);

символ l, указывающий, что соответствующий аргумент имеет тип long (в случае символов преобразования d, l, o, u, x, X) или double (в случае символов преобразования e, E, f, g, G);

символ L, указывающий, что соответствующий аргумент имеет тип long double (используется вместе с символами преобразований e, E, f, g, G);

символ #, который может встречаться перед символами преобразования g, f, e (все они описываются ниже) и перед символом x. В первом случае всегда будет выводиться десятичная точка, а во втором — префикс 0X перед соответствующим шестнадцатеричным числом.

Если после знака % записан не символ преобразования, то он выводится на экран. Таким образом, строка %% приводит к выводу на экран знака %.

Функция printf использует управляющую строку, чтобы определить, сколько всего аргументов и каковы их типы. Аргументами могут быть переменные, константы, выражения, вызовы функций; главное, чтобы их значения соответствовали заданной спецификации.

При наличии ошибок, например, в числе аргументов или типе преобразования результаты будут неверными.

Среди управляющих символьных констант наиболее часто используются следующие:

- \a — для кратковременной подачи звукового сигнала;
- \b — для перевода курсора влево на одну позицию;
- \f — для подачи формата;
- \n — для перехода на новую строку;
- \r — для возврата каретки;
- \t — горизонтальная табуляция;
- \v — вертикальная табуляция;
- \\ — вывод символа \;
- \' — вывод символа ';
- \" — вывод символа ";
- \? — вывод символа ?.

Например, в результате записи инструкции вызова функции:

```
printf("\tЭВМ\n%d\n",i);
```

сначала выполняется горизонтальная табуляция (\t), т. е. курсор сместится от края экрана, затем на экран будет выведено слово ЭВМ, после этого курсор переместится в начало следующей строки (\n), затем будет выведено целое число i по формату %d, и окончательно курсор перейдет в начало новой строки (\n).

Форматный ввод данных. Функция scanf обеспечивает форматный ввод. Ее можно записать в следующем формальном виде:

scanf («управляющая строка», аргумент _1, аргумент _2,...);

Аргументы scanf должны быть указателями на соответствующие значения. Для этого перед именем переменной

записывается символ &. Назначение указателей будет подробно рассмотрено далее.

Управляющая строка содержит спецификации преобразования и используется для установления количества и типов аргументов. В нее могут включаться:

пробелы, символы табуляции и перехода на новую строку (все они игнорируются);

спецификации преобразования, состоящие из знака %, возможно, символа * (запрещение присваивания), возможно, числа, задающего максимальный размер поля, и самого символа преобразования;

обычные символы, кроме % (считается, что они должны совпадать с очередными неизвестными символами во входном потоке).

Рассмотрим символы преобразования функции scanf:

c — на входе ожидается появление одиночного символа;

d или i — на входе ожидается десятичное целое число и аргумент является указателем на переменную типа int;

D или I — на входе ожидается десятичное целое число и аргумент является указателем на переменную типа long;

e или E — на входе ожидается вещественное число с плавающей точкой;

f — на входе ожидается вещественное число с плавающей точкой;

g или G — на входе ожидается вещественное число с плавающей точкой;

o — на входе ожидается восьмеричное целое число и аргумент является указателем на переменную типа int;

O — на входе ожидается восьмеричное целое число и аргумент является указателем на переменную типа long;

s — на входе ожидается появление строки символов;

x — на входе ожидается шестнадцатеричное целое число и аргумент является указателем на переменную типа int;

X — на входе ожидается шестнадцатеричное целое число и аргумент является указателем на переменную типа long;

p — на входе ожидается появление указателя в виде шестнадцатеричного числа;

n — применяется в операциях форматирования. Аргумент, соответствующий этому символу спецификации, должен быть указателем на целое. В него возвращается номер позиции (после ввода), в которой записана спецификация %n;

u — на входе ожидается беззнаковое целое число и аргумент является указателем на переменную типа unsigned int;

U — на входе ожидается беззнаковое целое число и аргумент является указателем на переменную типа unsigned long;

[] — сканирует входную строку для получения символов.

Перед некоторыми символами преобразования могут записываться следующие модификаторы;

F — изменяет указатель, заданный по умолчанию, на указатель типа far;

N — изменяет указатель, заданный по умолчанию, на указатель типа near;

h — преобразует аргумент к типу short int (может записываться перед символами d, i, o, u, x);

l — преобразует аргумент к типу long int (может записываться перед символами d, i, o, u, x);

L — преобразует аргумент к типу long double (может записываться перед символами e, f, g).

2.2. Операторы и выражения

Переменные и константы. Выражения широко используются в программах на языке Си. Они состоят из операндов (переменные, константы и др.), соединенных знаками операций (сложение, вычитание, умножение и др.). Порядок выполнения операторов при вычислении значения выражения определяется их приоритетами и регулируется при помощи круглых скобок.

Все переменные до их использования должны быть объявлены. При этом задается тип, а затем идет список из одной или более переменных этого типа, разделенных запятыми. Например:

```
int a, b, c, d;  
char x, y;
```

Переменные можно разделять по строкам произвольным образом, например:

```
char x;  
char y;
```

Наряду с переменными в языке существуют следующие виды констант:

вещественные, например 123.456, 5.61e—8. Они могут снабжаться суффиксом F, например 123.456F, 5.61e—8F;

целые, например 125;

длинные целые, в конце записи которых добавляется буква (суффикс) L, например 361327L;

беззнаковые, в конце записи которых добавляется буква U, например 62125U;

восьмеричные, в которых перед первой значащей цифрой записывается ноль (0), например 071;

шестнадцатеричные, в которых перед первой значащей цифрой записывается пара символов ноль-икс (0X), например 0X1F2;

символьные — единственный символ, заключенный в одинарные кавычки, например 'g', '2', '.' и т. п. Символы, не имеющие графического представления, можно записывать, используя специальные комбинации, например \n (для новой строки), \0 (код которого 00000000). Эти комбинации выглядят как два символа, хотя фактически это один символ. Так же можно представить любой двоичный образ одного байта: '\NNN', где NNN — от одной до трех восьмеричных цифр.

Допускается и шестнадцатеричное задание кодов символов, которое представляется в виде '\x2B', '\x36' и т. п.;

строковые — последовательность из нуля символов или более, заключенная в двойные кавычки, например: «Это строковая константа». Кавычки не входят в строку, а лишь ограничивают ее. Строка представляет массив из перечисленных элементов, в конце которого помещается байт с символом \0. Таким образом, число байт, необходимых для хранения строки, на единицу превышает число символов между двойными кавычками;

константное выражение, состоящее из одних констант, которое вычисляется во время трансляции (например: $a=60+30$);

типа long double, в конце записи которых добавляется буква L, например: 1234567.89L.

Операции языка Си. Рассмотрим операции языка Си. В следующей главе будут дополнительно охарактеризованы новые операции языка Си++.

Выражения обычно содержат многие операции, выполняемые в строгой последовательности. Величина, определяющая преимущественное право на выполнение той или иной операции, называется *приоритетом*. В табл. 2.2 перечислены различные операции языка Си. Их приоритеты для каждой группы одинаковы (группа расположена между двумя соседними горизонтальными линиями). Чем большим преимуществом пользуется соответствующая группа операций, тем выше она расположена в таблице. Порядок выполнения определяет последовательность применения операций (слева направо или справа налево), если они относятся к одной группе и отсутствуют круглые скобки.

Охарактеризуем основные операции языка Си. Для задания каждой из них используются определенные знаки (см. табл. 2.2) или операторы. Сначала рассмотрим один из них — оператор присваивания (=). Выражение вида

$x = y;$

присваивает переменной x значение переменной y . Оператор «=» разрешается использовать многократно в одном выражении, например:

$x = y = z = 100;$

Здесь всем трем переменным (x , y , z) будет присвоено значение 100.

Различают унарные и бинарные операции. У первых из них один операнд, а у вторых — два. Начнем их рассмотрение

с операций, отнесенных к одной из следующих традиционных групп:

1. Арифметические операции.
2. Логические операции и операции отношения.
3. Операции с битами.

Арифметические операции задаются следующими операторами (см. табл. 2.2): $+$, $-$, $*$, $/$, $\%$. Последнюю из них нельзя применять к переменным вещественного типа. Примеры выражений с этими операторами: $a = b + c$; $x = y - z$; $r = t * v$; $s = k / l$; $p = q \% w$;

Логические операции отношения задаются следующими операторами (см. табл. 2.2): $\&$, $\|$, $!$, $>$, \geq , $<$, $\|$, $=$, $!=$. Традиционно эти операции должны давать одно из двух значений: истину и ложь. В языке Си принято следующее правило: истина — это любое ненулевое значение; ложь — это нулевое значение. Выражения, использующие логические операции и операции отношения, возвращают 0 для ложного значения и 1 — для истинного. Ниже приводится таблица истинности для логических операций:

x	y	x&y	x&y	!x
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Таблица 2.2

Знак операции	Назначение операции	Порядок выполнения
()	Вызов функции	Слева направо
[]	Выделение элемента массива	
.	Выделение элемента записи	
—>	Выделение элемента записи	
!	Логическое отрицание	
~	Поразрядное отрицание	Справа налево
-	Изменение знака	
++	Увеличение на единицу	
-	Уменьшение на единицу	
&	Взятие адреса	
*	Обращение по адресу	Слева направо
(тип)	Преобразование типа	
sizeof	Определение размера в байтах	
*	Умножение	
/	Деление	

Знак операции	Назначение операции	Порядок выполнения
%	Определение остатка от деления	Слева направо
+	Сложение	»
-	Вычитание	»
«	Сдвиг влево	»
»	Сдвиг вправо	»
<	Меньше, чем	»
<=	Меньше или равно	»
>	Больше, чем	»
>=	Больше или равно	»
=	Равно	»
!=	Не равно	»
&	Поразрядное логическое «И»	»
^	Поразрядное исключающее «ИЛИ»	»
	Поразрядное логическое «ИЛИ»	»
&	Логическое «И»	»
	Логическое «ИЛИ»	»
?:	Условная (тернарная) операция	»
=	Присваивание	»
zn=	Здесь zn — любая бинарная операция, например a *= b;	»
,	Операция запятая	»

Битовые операции можно применять к переменным, имеющим типы `int`, `char`, а также их варианты (например, `long`). Их нельзя применять к переменным типов `float`, `double`, `void` (или более сложных типов). Эти операции задаются следующими операторами: `~`, `<`, `>`, `&`, `^`, `||`. Ниже приводится таблица истинности для битовых операций `~`, `&`, `||`, `^`:

x	y	~x	x & y	x y	x ^ y
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Рассмотрим примеры операций сдвига. В выражении

`a = b << c;`

производится сдвиг значения `b` влево на `c` бит; в освободившиеся (справа) биты `b` заносятся нули. Если `b = 310 = 00112`, `c = 2`, то после выполнения операции `a = 11002 = 1210`. Это эквивалентно умножению числа 3 сначала на 2 и потом еще раз на 2. В выражении

`a = b >> c;`

производится сдвиг значения b вправо на c бит. Если $b = 1210 = 11002$, $c = 2$, то после выполнения операции $a = 00112 = 310$.

В языке предусмотрены две нетрадиционные операции инкремента ($++$) и декремента ($--$). Они соответственно предназначены для увеличения и уменьшения на единицу значения операнда. Операции $++$ и $--$ можно записывать как перед операндом, так и после него. В первом случае ($++n$ или $--n$) значение операнда (n) изменяется перед его использованием в соответствующем выражении, а во втором ($n++$ или $n--$) — после его использования. Рассмотрим две следующие строки программы:

```
a = b + c++;  
a1 = b1 + (++c1);
```

Скобки во второй из них поставлены только для повышения наглядности (операция $++$ имеет более высокий приоритет, чем $+$). Предположим, что $b = b1 = 2$, $c = c1 = 4$. Тогда после выполнения операций: $a = 6$, $b = 2$, $c = 5$, $a1 = 7$, $b1 = 2$, $c1 = 5$.

Широкое распространение находят также выражения с еще одной нетрадиционной тернарной или условной операцией $?:$ В формуле $y = x ? a : b$; $y = a$, если x не равно нулю (истинно), и $y = b$, если x равно нулю (ложно). Следующее выражение: $y = (a > b) ? a : b$; позволяет присвоить переменной y значение большей переменной (a или b), т. е. $y = \max(a, b)$.

Еще одним отличием языка является то, что выражение вида $a = a + 5$; можно записывать в другой форме: $a += 5$. Вместо знака $+$ можно использовать и символы других бинарных операций (см. табл. 2.2).

Другие операции из табл. 2.2 будут описаны позже в последующих параграфах.

Преобразование типов. Если в выражении появляются операнды различных типов, то они преобразуются к некоторому общему типу, при этом к каждому арифметическому операнду применяется такая последовательность правил:

1. Если один из операндов в выражении имеет тип `long double`, то остальные тоже преобразуются к типу `long double`.

2. В противном случае, если один из операндов в выражении имеет тип `double`, то остальные тоже преобразуются к типу `double`.

3. В противном случае, если один из операндов в выражении имеет тип `float`, то остальные тоже преобразуются к типу `float`.

4. В противном случае, если один из операндов в выражении имеет тип `unsigned long`, то остальные тоже преобразуются к типу `unsigned long`.

5. В противном случае, если один из операндов в выражении имеет тип `long`, то остальные тоже преобразуются к типу `long`.

6. В противном случае, если один из операндов в выражении имеет тип `unsigned`, то остальные тоже преобразуются к типу `unsigned`.

7. В противном случае все операнды преобразуются к типу `int`. При этом тип `char` преобразуется в `int` со знаком; тип `unsigned char` в `int`, у которого старший байт всегда нулевой; тип `signed char` в `int`, у которого в знаковый разряд передается знак из `char`; тип `short` в `int` (знаковый или беззнаковый).

Предположим, что вычислено значение некоторого выражения в правой части оператора присваивания. В левой части оператора присваивания записана некоторая переменная, причем ее тип отличается от типа результата в правой части. Здесь правила преобразования очень простые: значение справа от оператора присваивания преобразуется к типу переменной слева от оператора присваивания. Если размер результата в правой части больше размера операнда в левой части, то старшая часть этого результата будет потеряна.

В языке Си можно явно указать тип любого выражения. Для этого используется оператор преобразования типа. Он задается в следующей общей форме:

(тип) выражение

Здесь можно указать любой допустимый тип в языке Си.

Рассмотрим пример:

```
int x;  
float y, z;  
...  
z = y + (float) x;
```

Здесь переменная `x` целого типа явно преобразована к типу `float`.

Указатели и операции с ними. *Указатели* – это переменные, показывающие место или адрес памяти, где расположены другие объекты (переменные, функции и т. п.). Так как указатель содержит адрес некоторого объекта, то через него можно обращаться к этому объекту.

Унарная операция `&` дает адрес объекта, поэтому оператор `y = &x;` присваивает адрес `x` переменной `y`. Операцию `&` нельзя применять к константам и выражениям; конструкции вида `&(x + 7)` или `&28` недопустимы.

Унарная операция `*` воспринимает свой операнд как адрес некоторого объекта и использует этот адрес для выборки содержимого, поэтому оператор `z = *y;` присваивает `z` значение переменной, записанной по адресу `y`. Если `y = &x;`; `z = *y;` то `z = x`.

Объекты, состоящие из знака `*` и адреса (например, `*a`), необходимо объявлять. Делается это, например, так:

```
int *a, *b, *c;
char *d;
```

Объявление вида `char *d;` говорит о том, что значение, записанное по адресу `d`, имеет тип `char`.

Указатели могут встречаться и в выражениях. Если `y` — указатель на целое, т. е. имело место объявление `int *y;`, то `*y` может появиться там же, где и любая другая переменная, не являющаяся указателем. Таким образом, следующие выражения вполне допустимы:

```
*y = 7;
*x * = 5;
(*z)++;
```

Первое из них заносит число 7 в ячейку памяти по адресу `y`, второе увеличивает значение по адресу `x` в пять раз, третье добавляет единицу к содержимому ячейки памяти с адресом `z`. В последнем случае круглые скобки необходимы, так как операции `*` и `++` с одинаковым приоритетом выполняются справа налево (см. табл. 2.2). В результате если, например, `*z = 5`, то `(*z)++` приведет к тому, что `*z = 6`, а `*z++` всего лишь изменит сам адрес `z` (операция `++` выполняется над адресом `z`, а не над значением `*z` по этому адресу).

Указатели можно использовать как операнды в арифметических операциях. Если `y` — указатель, то унарная операция `y++`; увеличивает его значение; теперь оно является адресом следующего элемента. Указатели и целые числа можно складывать. Конструкция `y + n` (`y` — указатель, `n` — целое число) задает адрес `n`-го объекта, на который указывает `y`. Это справедливо для любых объектов (`int`, `char`, `float` и т. п.); транслятор будет масштабировать приращение адреса в соответствии с типом, определенным из соответствующего объявления.

Любой адрес можно проверить на равенство (`==`) или неравенство (`!=`) со специальным значением `NULL`, которое позволяет определить ничего не адресующий указатель.

Операторы организации циклов. Циклы организуются, чтобы повторить некоторую инструкцию или группу инструкций определенное число раз. В языке Си три оператора цикла:

for, while и do-while. Первый из них формально записывается в виде

```
for (выражение_1; выражение_2; выражение_3) тело цикла
```

Тело цикла составляет либо одна инструкция, либо любое подмножество инструкций, заключенных в фигурные скобки. В выражениях 1, 2, 3 фигурирует специальная переменная, называемая управляющей. По ее значению устанавливается необходимость повторения цикла либо выхода из него. Каждое из трех выражений в скобках предназначено для:

1) присвоения начального значения управляющей переменной;

2) проверки выполнения условия продолжения цикла;

3) изменения значения управляющей переменной.

Таким образом, выражение 1 присваивает начальное значение управляющей переменной, выражение 3 изменяет его на каждом шаге, а выражение 2 — проверяет, не достигло ли оно граничного значения, устанавливающего необходимость выхода из цикла. Любое из трех выражений в цикле for может отсутствовать, однако точка с запятой должна оставаться. Таким образом,

```
for (;;) {...}
```

это бесконечный цикл, из которого выходят другими способами.

В языке Си, принято следующее правило. Любое выражение с оператором присваивания, заключенное в круглые скобки, имеет значение, равное присваиваемому. Например, выражение $(a = 7 + 2)$ имеет значение 9. После этого можно записать другое выражение, например: $((a = 7 + 2) < 10)$, которое в данном случае будет всегда давать истинное значение. Следующая конструкция:

```
((c = getch()) != 'I')
```

позволяет вводить значение переменной c и давать истинный результат только тогда, когда введенным значением является буква I. В скобках можно записывать и несколько формул, составляющих сложное выражение. Для этих целей используется операция запятая. Формулы будут вычисляться слева направо, и все выражение примет значение последней вычисленной формулы. Например, если имеются две переменные типа char, то выражение:

```
z = (x = y, y = getch())
```

определяет следующие действия: значение переменной y присваивается переменной x; вводится символ с клавиатуры

52

и присваивается переменной *y*; *z* получает значение переменной *y*. Скобки здесь необходимы, поскольку операция запятая имеет более низкий приоритет, чем оператор присваивания, записанный после переменной *z* (см. табл. 2.2). Операция запятая находит широкое применение для построения выражений цикла *for* и позволяет параллельно изменять значения нескольких управляющих переменных.

Допускаются вложенные конструкции, т. е. в теле некоторого цикла могут встречаться другие операторы *for*.

Оператор *while* формально записывается в виде

while (выражение) тело цикла

Выражение в скобках может принимать ненулевое (истинное) или нулевое (ложное) значение. Если оно истинно, то выполняется тело цикла и выражение вычисляется снова. Если выражение ложно, то цикл *while* заканчивается.

Основным отличием между циклами *while* и *do-while* является то, что тело в цикле *do-while* выполняется по крайней мере один раз. Оператор *do-while* формально записывается в виде

do тело цикла *while* (выражение);

Тело цикла будет выполняться до тех пор, пока выражение в скобках не примет ложное значение. Если оно ложно при входе в цикл, то его тело выполняется ровно один раз.

Допускается вложенность одних циклов в другие, т. е. в теле любого цикла могут появляться операторы *for*, *while* и *do-while*.

В теле цикла могут использоваться новые операторы: *break* и *continue*. Первый из них обеспечивает немедленный выход из цикла. Оператор *continue* вызывает прекращение очередной и начало следующей итерации.

Операторы организации условных и безусловных переходов. Для организации условных и безусловных переходов в программе на языке Си используются операторы: *if* — *else*, *switch* и *goto*. Первый из них записывается в следующем формальном виде:

if (проверка_условия) инструкция_1; *else* инструкция_2;

Если условие в скобках принимает истинное значение, то выполняется инструкция_1, а если ложное — то инструкция_2. Если вместо одной необходимо использовать несколько инструкций, то они заключаются в фигурные скобки. В операторе *if* слово *else* может и отсутствовать. В этом случае,

если условие в скобках принимает истинное значение, то выполняется инструкция_1, а если ложное, то инструкция_1 пропускается. Таким образом, инструкция_2 будет выполнена всегда.

В операторе if-else непосредственно после ключевых слов if и else должны следовать другие инструкции. Если хотя бы одна из них является инструкцией If, ее называют *вложенной*. Согласно принятому в языке Си соглашению слово else всегда относится к ближайшему предшествующему ему if.

Оператор switch позволяет выбрать одну из нескольких альтернатив. Он записывается в следующем формальном виде:

```
switch (выражение)
{case константа 1: вариант 1; break;
. . . . .
case константа n-1: вариант n-1; break;
default: вариант n;}
```

Здесь вычисляется значение целого выражения в скобках (его иногда называют селектором) и оно сравнивается со всеми константами (константными выражениями). Все константы должны быть различными. При совпадении выполняется соответствующий вариант (одна или несколько инструкций). Вариант с ключевым словом default реализуется, если ни один другой не подошел (слово default может и отсутствовать). Если default отсутствует, а все результаты сравнения отрицательны, то ни один вариант не выполняется. Для прекращения последующих проверок после успешного выбора некоторого варианта используется оператор break, обеспечивающий немедленный выход из переключателя switch.

Допускаются вложенные конструкции switch.

Рассмотрим правила выполнения безусловного перехода; который можно представить в следующей форме:

```
goto метка;
```

Метка — это любой идентификатор. Оператор goto указывает, что выполнение программы необходимо продолжить начиная с инструкции, перед которой записана метка. В программе обязательно должна быть строка, где указана метка, поставлено двоеточие и записана инструкция, в которой должен выполняться переход. Метку можно поставить перед любой инструкцией в той функции, где находится соответствующий ей оператор goto. Ее не надо объявлять.

2.3. Структурированные типы данных

Массивы. В программе на языке Си можно использовать структурированные типы данных. К ним будем относить массивы, структуры и файлы.

Массив состоит из многих элементов одного и того же типа. Ко всему массиву целиком можно обращаться по имени. Кроме того, можно выбирать любой элемент массива. Для этого необходимо задать индекс, который указывает его относительную позицию. Число элементов массива назначается при его объявлении и в дальнейшем не меняется. Если массив объявлен, то к любому его элементу можно обратиться следующим образом: указать имя массива и индекс элемента в квадратных скобках. Массивы объявляются так же, как и переменные:

```
int a[100];  
char b[30];  
float c[42];
```

В первой строке объявлен массив *a* из 100 элементов целого типа: *a*[0], *a*[1], ..., *a*[99] (индексация всегда начинается с нуля). Во второй строке элементы массива *b* имеют тип *char*, а в третьей — *float*.

Двумерный массив представляется как одномерный, элементы которого тоже массивы. Например, объявление *char a*[10][20]; задает такой массив. По аналогии можно установить и большее число измерений. Элементы двумерного массива хранятся по строкам, т. е. если проходить по ним в порядке их расположения в памяти, то быстрее всего изменяется самый правый индекс. Например, обращение к девятому элементу пятой строки запишется так: *a*[5][9]. Пусть задано объявление:

```
int a[2][3];
```

Тогда элементы массива *a* будут размещаться в памяти следующим образом: *a*[0][0], *a*[0][1], *a*[0][2], *a*[1][0], *a*[1][1], *a*[1][2]. Имя массива *a* — это константа, которая содержит адрес его первого элемента (для нашего примера — *a*[0][0]). Предположим, что *a* = 1000. Тогда адрес элемента *a*[0][1] будет равен 1002 (элемент типа *int* занимает в памяти 2 байт), адрес следующего элемента *a*[0][2] — 1004 и т. п. Что же произойдет, если выбрать элемент, для которого не выделена память. К сожалению, компилятор не следит за этим. В результате возникнет ошибка и программа будет работать неверно.

В языке Си существует «сильная» взаимосвязь между указателями и массивами. Любое действие, которое достига-

ется индексированием массива, можно выполнить и с помощью указателей, причем последний вариант будет быстрее. Объявление

```
int a[5];
```

определяет массив из пяти элементов `a[0]`, `a[1]`, `a[2]`, `a[3]`, `a[4]`. Если объект `*у` объявлен как

```
int *у;
```

то оператор `у = &a[0]`; присваивает переменной `у` адрес элемента `a[0]`. Если переменная `у` указывает на очередной элемент массива `а`, то `у + 1` указывает на следующий элемент, причем здесь выполняется соответствующее масштабирование для приращения адреса с учетом длины объекта (для типа `int` — 2 байт, `long` — 4 байт, `double` — 8 байт и т. п.). Так как само имя массива есть адрес его нулевого элемента, то инструкцию `у = &a[0]`; можно записать и в другом виде: `у = а`; Тогда элемент `a[i]` можно представить как `*(а + i)`. С другой стороны, если `у` — указатель, то следующие две записи: `у[i]` и `*(у + i)` эквивалентны. Между именем массива и соответствующим указателем есть одно важное различие: Указатель — это переменная и `у = а`; или `у++`; — допустимые операции. Имя же массива — константа, поэтому конструкции вида `а = у`; `а++`; использовать нельзя, так как значение константы постоянно и не может быть изменено.

Переменные с адресами могут образовывать некоторую иерархическую структуру (могут быть многоуровневыми), типа указатель на указатель (т. е. он содержит адрес другого указателя), указатель на указатель на указатель и т. п. Их использование будет рассмотрено ниже на примере.

Если указатели адресуют элементы одного массива, то их можно сравнивать (отношения вида `<`, `>`, `==`, `!=` и другие работают правильно). В то же время нельзя сравнивать либо использовать в арифметических операциях указатели на разные массивы (соответствующие выражения не приводят к ошибкам при компиляции, но в большинстве случаев не имеют смысла). Как и выше, любой адрес можно проверять на равенство или неравенство со значением `NULL`. Указатели на элементы одного массива можно также вычитать. Тогда результатом будет число элементов массива, расположенных между уменьшаемым и вычитаемым объектами.

Язык Си позволяет инициализировать массив при объявлении. Для этого используется следующая форма:

```
тип имя_массива[...][...] = {список значений};
```

Рассмотрим примеры:

```
int a[5] = {0, 1, 2, 3, 4};  
char c[7] = {'a', 'b', 'c', 'd', 'e', 'f', 'g'};  
int b[2][3] = {1, 2, 3, 4, 5, 6};  
В последнем случае: b[0][0] = 1, b[0][1] = 2, b[0][2] = 3, b[1][0] = 4, b[1][1] = 5, b[1][2] = 6.
```

В языке допускаются массивы указателей, которые объявляются, например, следующим образом: `char *m[5];`. Здесь `m[5]` — массив, содержащий адреса элементов типа `char`.

Строки символов. Язык Си не поддерживает отдельный строковый тип данных, но он позволяет определить строки двумя различными способами. В первом используется массив символов, а во втором — указатель на первый символ массива. Объявление `char a[10];` указывает компилятору на необходимость резервирования места для максимум 10 символов. Константа `a` содержит адрес ячейки памяти, в которой помещено значение первого из десяти объектов типа `char`. Процедуры, связанные с занесением конкретной строки в массив `a`, копируют ее по одному символу в область памяти, на которую указывает константа `a`, до тех пор, пока не будет скопирован нулевой символ, оканчивающий строку. Когда выполняется функция типа `printf («%s», a);`, ей передается значение `a`, т. е. адрес первого символа, на который указывает `a`. Если первый символ нулевой, то работа функции `printf` заканчивается, а если нет, то она выводит его на экран, прибавляет к адресу единицу и снова начинает проверку на нулевой символ. Такая обработка позволяет снять ограничения на длину строки (конечно, в пределах объявленной размерности): строка может быть любой длины, до тех пор, пока есть место в памяти, куда ее можно поместить.

Второй способ определения строки — это использование указателя на символ. Объявление `char *b;` задает переменную `b`, которая может содержать адрес некоторого объекта. Однако в данном случае компилятор не резервирует место для хранения символов и не инициализирует переменную `b` конкретным значением. Когда компилятор встречает инструкцию вида `b = «Москва»;`, он производит следующие действия. Во-первых, как и в предыдущем случае, он создает в каком-либо месте объектного модуля строку **Москва**, за которой следует нулевой символ. Во-вторых, он присваивает значение начального адреса этой строки (адрес символа **М**) переменной `b`. Функция `printf ("%s", b);` работает так же, как и в предыдущем случае, осуществляя вывод символов до тех пор, пока не встретится заключительный ноль.

Массив указателей можно инициализировать, т. е. назначать его элементам конкретные адреса некоторых заданных строк при объявлении.

Структуры. *Структура* — это объединение одного или более объектов (переменных, массивов, указателей, других структур и т. п.). Как и массив, она представляет собой совокупность данных. Отличием является то, что к ее элементам необходимо обращаться по имени и что различные элементы структуры не обязательно должны принадлежать одному типу.

Объявление структуры осуществляется с помощью ключевого слова `struct`, за которым идет ее тип и далее список элементов, заключенных в фигурные скобки:

```
struct тип {тип элемента 1 имя элемента 1;  
            .  
            .  
            .  
            тип элемента n имя элемента n};
```

Именем элемента может быть любой идентификатор. Как и выше, в одной строке можно записывать через запятую несколько идентификаторов одного типа. Рассмотрим пример:

```
struct date {int day;  
            int month;  
            int year; };
```

Следом за фигурной скобкой, заканчивающей список элементов, могут записываться переменные данного типа, например: `struct date {...} a, b, c;` (при этом выделяется соответствующая память). Описание без последующего списка не выделяет никакой памяти; оно просто задает форму структуры. Введенное имя типа позже можно использовать для объявления структуры, например: `struct date days;`. Теперь переменная `days` имеет тип `date`. При необходимости структуры можно инициализировать, помещая за объявлением список начальных значений элементов. Разрешается вкладывать структуры одна в другую, например:

```
struct man {char name[30], fam[20];  
            struct date bd;  
            int voz; };
```

Определенный выше тип `date` включает три элемента: `day`, `month`, `year`, содержащий целые значения (`int`). Структура `man` включает элементы `name[30]`, `fam[20]`, `bd` и `voz`. Первые два — `name[30]` и `fam[20]` — это символьные массивы из 30 и 20 элементов каждый. Переменная `bd` представлена составным элементом (вложенной структурой) типа `date`. Элемент `voz` содержит значения целого типа (`int`). Теперь разрешается объявить переменные, значения которых принадлежат введенному типу:

```
struct man _man_ [100];
```

Здесь определен массив `_man_`, состоящий из 100 структур типа `man`. В языке Си разрешено использовать

массивы структур. Структуры могут состоять из массивов и других структур.

Чтобы обратиться к отдельному элементу структуры, необходимо указать его имя, поставить точку и сразу за ней написать имя нужного элемента, например:

```
_man [i].voz = 16;  
_man [j].bd.day = 22;  
_man [j].bd.year = 1976;
```

При работе со структурами необходимо помнить, что тип элемента определяется соответствующей строкой объявления в фигурных скобках. Например, `_man` имеет тип `man`, `year` — является целым числом и т. п. Поскольку каждый элемент структуры относится к определенному типу, его имя может появляться везде, где разрешено использовать значения этого типа. Допускаются конструкции вида `_man [i] = _man [j];` где `_man [i]` и `_man [j]` — объекты, соответствующие единому описанию структуры. Другими словами, разрешается присваивать одну структуру другой по их именам.

Унарная операция `&` позволяет взять адрес структуры. Предположим, что задано объявление:

```
struct date {int d, m, y;} day;
```

Здесь `day` — это структура типа `date`, включающая три элемента: `d`, `m`, `y`. Другое объявление `struct date *db;` устанавливает тот факт, что `db` — это указатель на структуру типа `date`. Запишем выражение: `db = &day;`. Теперь для выбора элементов `d`, `m`, `y` структуры необходимо использовать конструкции: `(*db).d`, `(*db).m`, `(*db).y`. Действительно, `db` — это адрес структуры, `*db` — сама структура. Круглые скобки здесь необходимы, так как точка имеет более высокий, чем звездочка, приоритет (см. табл. 2.2). Для аналогичных целей в языке Си предусмотрена специальная операция `—>`. Эта операция выбирает элемент структуры и позволяет представить рассмотренные выше конструкции в более простом виде: `db—> d`, `db —> m`, `db —> y`.

Оператор `typedef`. Рассмотрим описание структуры:

```
struct data {int d, m, y};
```

Фактически вводится новый тип данных — `data`. Теперь его можно использовать для объявления конкретных экземпляров структуры, например:

```
struct data a, b, c;
```

В язык Си введено специальное средство, позволяющее назначать имена новым типам данных. Таким средством

является оператор `typedef`. Он записывается в следующем виде:

```
typedef тип имя;
```

Здесь «тип» — любой разрешенный тип данных и «имя» — любой разрешенный идентификатор.

Рассмотрим пример:

```
typedef int INTEGER;
```

После этого можно сделать объявление:

```
INTEGER a, b;
```

Оно будет выполнять то же самое, что и привычное объявление:

```
int a, b;
```

Другими словами, `INTEGER` можно использовать как синоним ключевого слова `int`. При этом можно комбинировать объявления со словами `int` и `INTEGER`, например:

```
INTEGER a, b;
```

```
int c, d;
```

Здесь объявлены четыре переменные (`a`, `b`, `c`, `d`) целого типа.

Битовые поля. Особую разновидность структур представляют поля. *Поле* — это последовательность соседних битов внутри одного целого значения. Оно может иметь тип `signed int` либо `unsigned int` и занимать от 1 до 16 битов. Поля размещаются в машинном слове в направлении от младших к старшим разрядам. Например, структура:

```
struct prlm {int a:2; unsigned b:3; int:5;  
int c:1; unsigned d:5;} l, j;
```

обеспечивает размещение данных в двух байтах (в одном слове). Если бы последнее поле было задано так: `unsigned d:6;`, то оно размещалось бы не в первом слове, а в разрядах $0 + 5$ второго слова.

В полях типа `signed` крайний левый бит является знаковым. Например, такое поле шириной 1 бит может только хранить значения -1 и 0 , так как любая ненулевая величина будет интерпретироваться как -1 .

Поля используются для упаковки значений нескольких переменных в одно машинное слово с целью экономии памяти. Они не могут быть массивами и не имеют адресов, поэтому к ним нельзя применять унарную операцию `&`.

Смеси. *Смесь* — это некоторая переменная, которая может хранить (в разное время) объекты различного типа и размера. В результате появляется возможность работы в одной и той

же области памяти с данными различного вида. Для описания смеси используется ключевое слово `union`, а соответствующий синтаксис аналогичен записям. Пусть задано объявление:

```
union r {int ir; float fr; char cr;} z;
```

Здесь `ir` имеет размер 2 байт, `fr` — 4 байт и `cr` — 1 байт. Переменная `z` будет достаточно велика, чтобы сохранять самый большой из трех приведенных типов. Таким образом, размер `z` будет 4 байт. В один и тот же момент времени `z` может иметь значение только одной из указанных переменных (`ir`, `fr`, `cr`).

Файлы. В файлах размещаются данные, предназначенные для длительного хранения. Каждому файлу присваивается используемое при обращении к нему уникальное имя.

В языке Си отсутствуют инструкции для работы с файлами. Все необходимые действия выполняются через функции, включенные в стандартную библиотеку. Они позволяют работать с различными устройствами, такими, как диски, принтер, коммуникационные каналы и т. п. Эти устройства сильно отличаются друг от друга. Однако файловая система позволяет преобразовывать их в единое абстрактное логическое устройство, называемое *поток*ом. Существует два типа потоков: текстовые и двоичные.

Прежде чем читать или записывать информацию в файл, он должен быть открыт. Это можно сделать с помощью библиотечной функции `fopen`. Она берет внешнее представление файла (например, `C:MY_FILE.TXT`) и связывает его с внутренним логическим именем, которое используется далее в программах. *Логическое имя* — это указатель на требуемый файл. Его необходимо объявлять; делается это, например, так:

```
FILE *lst;
```

Здесь `FILE` — имя типа, описанное в стандартном определении `stdio.h`, `lst` — указатель на файл. Обращение к функции `fopen` в программе производится выражением:

```
lst = fopen (спецификация файла, вид использования файла);
```

Спецификация файла может, например, быть: `C:MY_FYLE.TXT` — для файла `MY_FILE.TXT` на диске `C:`; `A:/MY_DIR/EX2_3.CPP` — для файла `EX2_3.CPP` в поддиректории `A:/MY_DIR` и т. п. Вид использования файла может иметь вид:

`r` — открыть существующий файл для чтения;

`w` — создать новый файл для записи (если файл с указанным именем существует, то он будет переписан);

a — дополнить файл (открыть существующий файл для записи информации, начиная с конца файла, либо создать файл, если он не существует);
rb — открыть двоичный файл для чтения;
wb — создать двоичный файл для записи;
ab — дополнить двоичный файл;
rt — открыть текстовый файл для чтения;
wt — создать текстовый файл для записи;
at — дополнить текстовый файл;
r+ — открыть существующий файл для записи и чтения;
w+ — создать новый файл для записи и чтения;
a+ — дополнить или создать файл с возможностью записи и чтения;
r + b — открыть двоичный файл для записи и чтения;
w + b — создать двоичный файл для записи и чтения;
a + b — дополнить двоичный файл с предоставлением возможности записи и чтения.

Если режим t или b не задан (например, r, w или a), то он определяется значением глобальной переменной `_fmode`. Если `_fmode = O_BINARY`, то файлы открываются в двоичном режиме, а если `_fmode = O_TEXT` — в текстовом режиме. Константы `O_BINARY` и `O_TEXT` определены в файле `fcntl.h`.

Строки вида `r + b` можно записывать и в другой форме: `rb +`. Если в результате обращения к функции `fopen` возникает ошибка, то она возвращает указатель на константу `NULL`. После окончания работы с файлом он должен быть закрыт. Это делается с помощью библиотечной функции `fclose`. Она имеет следующий прототип:

```
int fclose (FILE *lst);
```

При успешном завершении функции `fclose` возвращает значение нуль.

Любое другое значение говорит об ошибке.

Рассмотрим другие библиотечные функции, используемые для работы с файлами (все они описаны в файле `stdio.h`):

1. Функция `putc` записывает символ в файл и имеет следующий прототип:

```
int putc (int c, FILE *lst);
```

здесь `lst` — указатель на файл, возвращенный функцией `fopen`, `c` — символ для записи (переменная `c` имеет тип `int`, но используется только младший байт). При успешном завершении `putc` возвращает записанный символ, в противном случае возвращается константа `EOF`. Она определена в файле `stdio.h` и имеет значение `-1`.

2. Функция `getc` читает символ из файла и имеет следующий прототип:

```
int getc (FILE *lst);
```

здесь `lst` — указатель на файл, возвращенный функцией `fopen`. Эта функция возвращает прочитанный символ. Соот-

ветствующее значение определяется типом `int`, но старший байт равен нулю. Если достигнут конец файла, то `getc` возвращает значение `EOF`.

3. Функция `feof` определяет конец файла при чтении двоичных данных и имеет следующий прототип:

```
int feof (FILE *lst);
```

здесь `lst` — указатель на файл, возвращенный функцией `fopen`. При достижении конца файла возвращается ненулевое значение, в противном случае возвращается 0.

4. Функция `fputs` записывает строку символов в файл. Она отличается от функции `puts` только тем, что в качестве второго параметра должен быть записан указатель на переменную файлового типа. Рассмотрим пример: `fputs («Example», lst);`. При возникновении ошибки возвращается значение `EOF`.

5. Функция `fgets` читает строку символов из файла. Она отличается от функции `gets` только тем, что в качестве второго параметра должен быть записан указатель на переменную файлового типа. Рассмотрим пример: `fgets (str, lst);`. Функция возвращает указатель на строку при успешном завершении и константу `NULL` в случае ошибки либо достижения конца файла.

6. Функция `fprintf` выполняет те же действия, что и функция `printf`, но работает с файлом. Ее отличием является то, что в качестве первого параметра задается указатель на переменную файлового типа. Рассмотрим пример: `fprintf (lst, "%x", a)`.

7. Функция `fscanf` выполняет те же действия, что и функция `scanf`, но работает с файлом. Ее отличием является то, что в качестве первого параметра задается указатель на переменную файлового типа. Рассмотрим пример: `fscanf (lst, "%x", &a);`. При достижении конца файла возвращается значение `EOF`.

8. Функция `fseek` позволяет выполнять чтение и запись с произвольным доступом и имеет следующий прототип:

```
int fseek (FILE *lst, long count, int access);
```

Здесь `lst` — указатель на файл, возвращенный функцией `fopen`, `count` — номер байта относительно заданной начальной позиции, начиная с которого будет выполняться операция, `access` — способ задания начальной позиции. Переменная `access` может принимать следующие значения:

- 0 — начальная позиция задана в начале файла;
- 1 — начальная позиция считается текущей;
- 2 — начальная позиция задана в конце файла.

При успешном завершении возвращается нуль, при ошибке — ненулевое значение.

9. Функция `ferror` позволяет проверить правильность выполнения последней операции при работе с файлом и имеет следующий прототип:

```
int ferror (FILLE *lst);
```

В случае ошибки возвращается ненулевое значение, в противном случае возвращается нуль.

10. Функция `remove` удаляет файл и имеет следующий прототип:

```
int remove (char *file_name);
```

Здесь `file_name` — указатель на строку со спецификацией файла. При успешном завершении возвращается нуль, в противном случае возвращается ненулевое значение.

11. Функция `rewind` устанавливает указатель текущей позиции в начало файла и имеет следующий прототип:

```
void rewind (FILE *lst);
```

В языке Си открываются пять стандартных файлов со следующими логическими именами:

`stdin` — для ввода данных из стандартного входного потока (по умолчанию с клавиатуры);

`stdout` — для вывода данных в стандартный выходной поток (по умолчанию на экран дисплея);

`stderr` — файл для вывода сообщений об ошибках (всегда связан с экраном дисплея);

`stdprn` — для вывода данных на принтер;

`stdaus` — для ввода и вывода данных в коммуникационный канал.

Перечисляемый тип данных. Перечисляемый тип данных предназначен для описания объектов из некоторого заданного множества. Он определяется ключевым словом `enum`. Рассмотрим пример:

```
enum seasons {spring, aummer, autumn, winter};
```

Здесь введен новый тип данных `seasons`. Теперь можно объявить переменные этого типа:

```
enum seasins a, b, c;
```

Каждая из них (`a`, `b`, `c`) может принимать одно из четырех значений: `spring`, `summer`, `autumn` и `winter`. Эти переменные можно было объявить сразу при описании типа:

```
enum seasons {spring, summer, autumn, winter} a, b, c;
```

Рассмотрим другое объявление:

```
enum days {mon, tues, wed, thur, fri, sat, sun} my_week;
```

Имена, занесенные в `days`, представляют собой константы целого типа. Первая из них (`mon`) автоматически устанавливается в нуль, и каждая следующая имеет значение на единицу больше, чем предыдущая (`tues = 1`, `wed = 2` и т. п.). Можно присвоить константам определенные значения целого типа (именам, не имеющим их, будут, как и раньше, назначены значения предыдущих констант, увеличенные на единицу). Например:

```
enum days {mon = 5, tues = 8, wed = 10, thur, fri, sat, sun} my_week;
```

После этого `mon = 5`, `tues = 8`, `wed = 10`, `thur = 11`, `fri = 12`, `sat = 13`, `sun = 14`.

Тип `enum` можно использовать для задания констант `true = 1` и `false = 0`, например:

```
enum t_f {false, true} a, b;
```

2.4. Функции

Общие сведения. Программы на языке Си обычно состоят из большого числа отдельных функций (подпрограмм). Как правило, эти функции имеют небольшие размеры и могут находиться как в одном, так и в нескольких файлах. Все функции являются глобальными. В языке запрещено определять одну функцию внутри другой. Связь между функциями осуществляется через аргументы, возвращаемые значения и внешние переменные. Передача значения из вызванной функции в вызвавшую происходит с помощью оператора возврата, который записывается в следующем формальном виде:

```
return выражение;
```

Таких операторов в подпрограмме может быть несколько и тогда они фиксируют соответствующие точки выхода. Вызвавшая функция может при необходимости игнорировать возвращаемое значение. После слова `return` можно ничего не записывать; в этом случае вызвавшей функции никакого значения не передается. Управление передается вызвавшей функции и в случае выхода «по концу» (последняя закрывающаяся фигурная скобка).

В языке Си аргументы функции передаются по значению, т. е. вызванная функция получает свою временную копию каждого аргумента, а не его адрес. Это означает, что функция не может изменять сам оригинальный аргумент в вызвавшей ее программе. Ниже будет показано, как убрать это ограничение. Если же в качестве аргумента функции используется имя массива, то передается начало массива (адрес

начала массива), а сами элементы не копируются. Функция может изменять элементы массива, сдвигаясь (индексированием) от его начала.

Рассмотрим, как соответствуют друг другу параметры в вызове и в списке функции. Пусть вызывающая программа обращается к функции следующим образом:

```
a = fun (b, c);
```

Здесь *b* и *c* — аргументы, значения которых передаются в вызываемую подпрограмму. Если описание функции начинается так:

```
void fun (int b, int c)
```

то имена передаваемых аргументов в вызове и в программе *fun* будут одинаковыми. Если же описание функции начинается, например, строкой

```
void fun (int i, int j)
```

то вместо имени *b* в вызвавшей функции для того же аргумента в функции *fun* будет использовано имя *i*, а вместо *c* — *j*. Пусть обращение к функции имеет вид

```
a = fun ( &b, &c);
```

Здесь подпрограмме передаются адреса переменных *b* и *c*. Поэтому прототип функции должен быть, например, таким:

```
void fun (int *k, int *c)
```

Теперь *k* получает адрес передаваемой переменной *b*, а *c* — адрес передаваемой переменной *c*. В результате в вызвавшей программе *c* — это переменная целого типа, а в вызванной программе *c* — это указатель на переменную целого типа. Если в вызове записаны те же имена, что и в списке параметров, но они записаны в другом порядке, то все равно устанавливается соответствие между *i*-м именем в списке и *i*-м именем в вызове.

Выше уже указывалось, что переменные передаются функции по значению, поэтому нет прямого способа в вызванной функции изменить некоторую переменную в вызвавшей функции. Однако это легко сделать, если передавать в функцию не переменные, а их адреса.

Рассмотрим, как функции можно передать массив в виде параметра. Здесь возможны три варианта:

1. Параметр задается как массив (например: `int m[100]`).
2. Параметр задается как массив без указания его размерности (например: `int m[]`).
3. Параметр задается как указатель (например: `int *m`). Этот вариант используется наиболее часто.

Независимо от выбранного варианта вызванной функции передается указатель на начало массива. Сами же элементы массива не копируются.

Если некоторые переменные, константы, массивы, структуры объявлены как глобальные, то их не надо включать в список параметров вызванной функции. Она все равно получит к ним доступ.

Функции в языке Си необходимо объявлять. В книге будем использовать уже рассмотренные ранее конструкции, называемые прототипом. В соответствующем объявлении будет дана информация о параметрах. Она представляется в следующем виде:

тип функция (параметр_1, параметр_2...);

Для каждого параметра можно указать только его тип (например: тип функция (int, float, ...);), а можно дать и его имя (например: тип функция (int a, float b, ...);). В языке Си разрешается создавать функции с переменным числом параметров. Тогда при задании прототипа вместо последнего из них указывается многоточие.

Классы памяти. В языке Си различают четыре основных класса памяти: внешнюю (глобальную), автоматическую (локальную), статическую и регистровую.

Внешние переменные определены вне функций и, следовательно, доступны для любой из них. Они могут объявляться только один раз. Выше уже говорилось, что сами функции всегда глобальные. Язык не позволяет определять одни функции внутри других. Область действия внешней переменной простирается от точки во входном файле, где она объявлена, до конца файла. Если на внешнюю переменную нужно сослаться до ее объявления или она определена в другом входном файле, то вступает в силу описание `extern`.

Автоматические переменные по отношению к функциям являются внутренними или локальными. Они начинают существовать при входе в функцию и уничтожаются при выходе из нее (для них можно использовать либо нет ключевое слово `auto`).

Статические переменные объявляются с помощью ключевого слова `static`. Они могут быть внутренними (локальными) или внешними (глобальными). Внутренние статические переменные, как и автоматические, локальны по отношению к отдельной функции. Однако они продолжают существовать, а не возникают и уничтожаются при каждом ее вызове. Другими словами, они являются собственной постоянной памятью для функции. Внешние статические переменные

доступны внутри оставшейся части файла после того, как они в нем объявлены, однако в других файлах они неизвестны. Это, в частности, позволяет скрыть данные одного файла от другого файла.

Регистровые переменные относятся к последнему классу. Ключевое слово `register` говорит о том, что переменная, о которой идет речь, будет интенсивно использоваться. Если возможно, значения таких переменных помещаются во внутренние регистры микропроцессора (Borland C++ использует для этих целей регистры SI и DI), что может привести к более быстрой и короткой программе. Для регистровых переменных нельзя взять адрес; они могут быть только автоматическими с допустимыми типами `int` или `char`.

Таким образом, можно выделить четыре спецификатора класса памяти: `extern`, `auto`, `static`, `register`. Они используются в следующей общей форме:

```
спецификатор_класса_памяти тип список_переменных;
```

Выше уже говорилось об инициализации, т. е. о присвоении различным объектам начальных значений. Если явная инициализация отсутствует, гарантируется, что внешние и статические переменные будут иметь значение нуль, а автоматическое и регистровые — неопределенное значение.

Указатели на функции. В языке Си сама функция не может быть значением переменной, но можно определить указатель на функцию. С ним уже можно обращаться как с переменной: передавать его другим функциям, помещать в массивы и т. п.

Код функции в персональном компьютере занимает физическую память. В этой памяти есть точка входа, которая используется для того, чтобы войти в функцию и запустить ее на выполнение. Указатель на функцию как раз и адресует эту точку входа. Это уже будет обычная переменная и с ним можно делать все, что можно делать с переменной. Однако через указатель можно войти в функцию, т. е. запустить ее на выполнение. Объявление вида:

```
int (*f) ();
```

говорит о том, что `f` — это указатель на функцию, возвращающую целое значение. Первая пара скобок необходима, без них `int *f();` означало бы, что `f` — функция, возвращающая указатель на целое значение. После объявления указателя на функцию в программе можно использовать объекты: `*f` — сама функция; `f` — указатель на функцию. Здесь имеется некоторая аналогия с массивами. Для любой функции

ее имя (без скобок и аргументов) является указателем на эту функцию.

Внешние аргументы функции main. В программы на языке Си можно передавать некоторые аргументы. Когда в начале вычислений производится обращение к main, ей передаются три параметра. Первый из них определяет число командных аргументов при обращении к программе. Второй представляет собой массив указателей на символьные строки, содержащие эти аргументы (в одной строке — один аргумент). Третий тоже является массивом указателей на символьные строки. Любая такая строка представляется в виде

переменная = значение\0

Последнюю строку можно найти по двум заключительным нулям.

Назовем аргументы функции main соответственно: argc, argv и env (возможны и любые другие имена). Тогда допустимы следующие описания:

```
main()  
main (int argc)  
main (int argc, char *argv[ ])  
main (int argc, char *argv[ ], char *env[ ])
```

Предположим, что на диске В: есть некоторая программа PROG.EXE. Обратимся к ней следующим образом:

В > PROG Брест Минск Гродно < ВВОД >

Тогда argv[0] — это указатель на PROG, argv[1] — на строку Брест и т. п. На первый фактический аргумент указывает argv[1], а на последний — argv[3]. Если argc = 1, то после имени программы в командной строке параметров нет. В нашем примере argc = 4.

Рекурсия. Рекурсией называется такой способ вызова, когда функция обращается к самой себе. Важным моментом при составлении рекурсивной программы является организация выхода. Здесь легко допустить ошибку, заключающуюся в том, что функция будет последовательно вызывать саму себя бесконечно долго. Поэтому рекурсивный процесс должен шаг за шагом так упрощать задачу, чтобы в конце концов для нее появилось нерекурсивное решение.

Библиотечные функции. В системах программирования подпрограммы для решения часто встречающихся задач объединяются в библиотеки. К числу таких задач относятся вычисление математических функций, ввод-вывод данных, обработка строк, взаимодействие со средствами операционной системы и т. п. Использование библиотечных подпрограмм избавляет пользователя от необходимости

разработки соответствующих средств и предоставляет ему дополнительный сервис.

Включенные в библиотеки функции на языке Си поставляются вместе с системой программирования. Их объявления даны в файлах *.h. Поэтому в начале программы с библиотечными функциями должны встречаться строки вида:

```
#include <включаемый_файл_типа_h>
```

Существуют также средства для расширения и создания новых библиотек с программами пользователя.

2.5. Другие возможности языка Си

Препроцессор. Транслятор Си имеет средство, расширяющее возможности языка, называемое препроцессором. Он выполняет подстановки для макровыводов, подключает заданные файлы и выполняет другие полезные функции. Для препроцессора предназначены строки программы, начинающиеся с символа # (в одной строке разрешается записывать только одну команду).

Директива:

```
#define идентификатор подстановка
```

вызывает замену в последующем тексте названного идентификатора на текст подстановка (обратите внимание на отсутствие точки с запятой в конце этой команды). Если директива имеет вид:

```
#define идентификатор (идентификатор, ..., идентификатор) подстановка,
```

причем между первым идентификатором и открывающейся круглой скобкой нет пробела, то это определение макроподстановки с аргументами. При наличии длинных определений в подстановке, продолжающихся в следующей строке, в конце очередной строки с продолжением ставится символ \.

Опишем другие директивы препроцессора. Первая из них include уже встречалась ранее. Ее можно использовать в двух формах:

```
#include «имя файла»
```

```
#include <имя файла>
```

Действие обеих команд сводится к включению в программу файлов с указанным именем. Первая из них загружает файл из текущего либо заданного в качестве префикса директория. Вторая команда осуществляет поиск файла в стандартных местах, определенных в системе программирования (см. команду Directories меню Options Borland C++). Если файл, записанный в двойных кавычках, не найден

в указанном директории, то поиск будет продолжен в поддиректориях, заданных для команды `#include <...>`. Директивы `#include` могут вкладываться одна в другую.

Директива `#error` записывается в следующей форме:

```
#error сообщение_об_ошибке
```

Если она встречается в тексте программы, то компиляция прекращается и на экран дисплея выводится сообщение об ошибке. Эта команда в основном применяется на этапе отладки. Заметим, что сообщение об ошибке не надо заключать в двойные кавычки.

Следующая группа директив позволяет избирательно компилировать части программы. Этот процесс называется *условной компиляцией*. В нее входят директивы `#if`, `#else`, `#elif`, `#endif`, `#ifdef`, `#ifndef`. Основная форма записи команды `#if` представляется в виде

```
#if константное_выражение
    последовательность_инструкций
#endif
```

Здесь проверяется значение константного выражения. Если оно истинно, то выполняется заданная последовательность инструкций, а если ложно, то эта последовательность инструкций пропускается.

Действие директивы `#else` подобно действию команды `else` в языке Си, например:

```
#if константное_выражение
    последовательность_инструкций_1
#else
    последовательность_инструкций_2
#endif
```

Здесь если константное выражение истинно, то выполняется последовательность инструкций 1, а если ложно — последовательность инструкций 2.

Директива `#elif` означает действие типа «else if». Основная форма ее использования представляется в виде

```
#if константное_выражение
    последовательность_инструкций
#elif константное_выражение_1
    последовательность_инструкций_1
. . . . .
#elif константное_выражение_n
    последовательность_инструкций_n
#endif
```

Эта форма подобна конструкции языка Си вида: `if...else if...else if...`

Директива

`#ifdef` идентификатор

устанавливает, определен ли в данный момент указанный идентификатор, т. е. входил ли он в команду вида `#define`. Строка вида

`#ifndef` идентификатор

проверяет, не определен ли в данный момент указанный идентификатор. За любой из этих команд может следовать произвольное число строк текста, возможно, содержащих инструкцию `#else` (`#elif` использовать нельзя) и заканчивающихся строкой `#endif`. Если проверяемое условие истинно, то игнорируются все строки между `#else` и `#endif`, а если ложно, то строки между проверкой и `#else` (если слово `#else` нет, то `#endif`). Приведенные инструкции (`#if`, `#ifdef`, `#ifndef`) могут «вкладываться» одна в другую.

Директива вида

`#undef` идентификатор

приводит к тому, что указанный идентификатор начинает считаться неопределенным, т. е. не подлежащим замене.

Директива `#line` предназначена для изменения значений переменных `_LINE_` и `_FILE_`, определенных в системе программирования Си. Переменная `_LINE_` содержит номер строки программы, выполняемой в текущий момент времени. Идентификатор `_FILE_` является указателем на строку с именем компилируемой программы. Основная форма записи директивы `#line` представляется в виде

`#line` номер «имя_файла»

Здесь номер — это любое положительное целое число, которое будет назначено переменной `_LINE_`, имя_файла — это необязательный параметр, который переопределяет значение `_FILE_`.

В макроопределение можно заносить два объекта вместе, разделенные знаками `##`, например:

```
#define pr(x,y) (x##y)
```

После этого `pr(a, 3)` вызовет подстановку `a3`.

Символ `#`, помещаемый перед макроаргументом, указывает о преобразовании его в строку; например после директивы:

```
#define pr1m (var) printf("#var=\"%d", var)
```

следующий фрагмент текста программы

```
year = 1993;  
pr1m(year);
```

преобразуется так:

```
year = 1993;  
printf("year"="%d", year);
```

Директива `#pragma` позволяет передать компилятору некоторые указания. Например, строка

```
#pragma inline
```

говорит о том, что в программе на языке Си находятся встроенные строки на языке Ассемблера.

Рассмотрим некоторые глобальные идентификаторы или макроимена. Стандартно определены пять таких имен: `_LINE_`, `_FILE_`, `_DATE_`, `_TIME_`, `_STDC_`. Два из них (`_LINE_` и `_FILE_`) уже описывались выше.

Идентификатор `_DATE_` определяет строку, в которой сохраняется дата трансляции исходного файла в объектный код.

Идентификатор `_TIME_` задает строку, сохраняющую время трансляции исходного файла в объектный код.

Макро `_STDC_` имеет значение 1, если используются стандартно определенные макроимена. В противном случае эта переменная будет не определена.

Рассмотрим примеры. Строка

```
#define begin {
```

дает возможность последующей замены в тексте программы открывающейся фигурной скобки (`{`) на слово `begin`.

После появления строки:

```
#define read (valr) scanf ("%d", &valr)
```

инструкция `read(i)`; воспринимается так же, как `scanf ("%d", &i)`; Здесь `valr` — аргумент и выполнена макроподстановка с аргументом.

Три следующие команды:

```
#ifdef write  
#undef write  
#endif
```

проверяют, определен ли идентификатор `write` (т. е. была ли команда вида `#define write...`), и если это так, то имя `write` начинает считаться неопределенным, т. е. не подлежащим замене. Директивы:

```
#ifndef write  
#define write fprintf  
#endif
```

проверяют, не определен ли идентификатор write, и если это так, то определяется идентификатор write вместо имени fprintf.

Использование программно-доступных элементов микропроцессора. На рис. 2.3 показаны все программно-доступные регистры микропроцессора Intel 8086. Адресуемая область памяти микропроцессора равна 1 Мбайт, и, следовательно, формат адреса равен 20 бит. Несмотря на генерацию 20-разрядных кодов, сам микропроцессор манипулирует логическими адресами, содержащими 16-разрядный сегментный (базовый) адрес и 16-разрядное внутрисегментное смещение. Механизм сегментации предполагает разбиение всего адресуемого пространства на области (сегменты) по 64 Кбайт каждая. Начальный адрес такой области (20 бит) имеет в четырех младших разрядах нули — $XXXX0_{16}$, т. е. сегменты могут начинаться на границах блоков по 16 байт (на границах параграфов). Здесь на место X может быть записана любая шестнадцатеричная цифра от 0 до F.

Начальный адрес сегмента хранится в 16-разрядном сегментном регистре, а обращение к ячейкам памяти внутри сегмента осуществляется с использованием 16-разрядного смещения. Если содержимое сегментного регистра равно нулю, физический адрес равен смещению. Большинство команд микропроцессора оперирует только 16-разрядными смещениями, а сегментные адреса находятся в одном из четырех регистров: CS — код, DS — данные; SS — стек; ES — дополнительные данные.

В состав блока регистров общего назначения входят четыре 16-разрядных элемента (AX, BX, CX, DX), допускающих независимую адресацию старших (H) и младших (L) половин. Все регистры этого блока участвуют в выполнении арифметических и логических операций, представляя операнды и фиксируя результат. Наряду с этим имеется множество команд, которые специализируют некоторые регистры (см. рис. 2.3).

Четыре 16-разрядных указательных и индексных регистра (SP, BP, SI, DI) предназначены для хранения внутрисегментных смещений. Эти же регистры могут участвовать в выполнении арифметических и логических операций над двухбайтными словами. Регистры SP (стека) и BP (базы) предназначены для доступа к данным в текущем сегменте стека, в SI и DI — в текущем сегменте данных. Некоторые

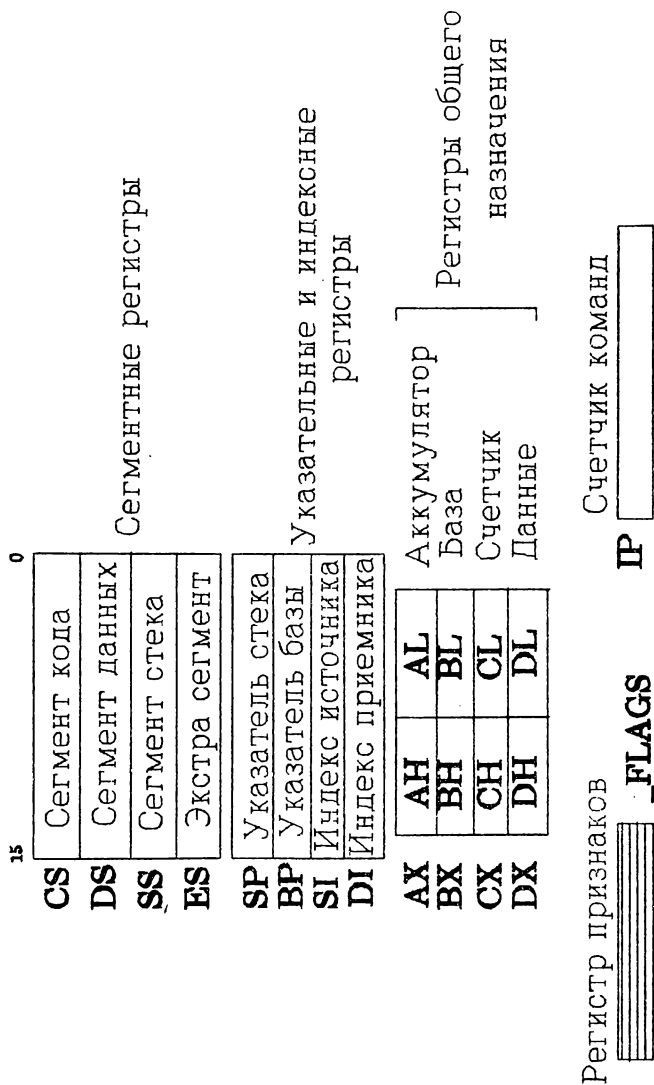


Рис 2.3

Рис. 2.3. Регистры микропроцессора 8086

команды микропроцессора специализируют эти четыре регистра.

В языке Си применены специальные объекты, называемые псевдопеременными. Они используются для обращения к ресурсам микропроцессора Intel 8086. Полный список псевдопеременных включает 21 элемент: `_AX`, `_BX`, `_CX`, `_DX`, `_CS`, `_DS`, `_SS`, `_ES`, `_SP`, `_BP`, `_DI`, `_SI`, `_AL`, `_AH`, `_BL`, `_BH`, `_CL`, `_CH`, `_DL`, `_DH`, `_FLAGS`. Первые двенадцать и последний имеют тип `unsigned int`, а оставшиеся восемь — `unsigned char`. Они определенным образом связаны с регистрами микропроцессора, показанными на рис. 2.3. Их имена образуются из имен регистров с префиксом — (переменная `_FLAGS` связана с регистром флагов). Присвоение значения какой-то переменной, например `_AX`, вызывает занесение этого значения в регистр `AX`. Получение значения переменной, например `_BX`, эквивалентно получению значения из регистра `BX`. Таким образом, псевдопеременные — это идентификаторы, соответствующие данным регистрам. Они могут трактоваться как глобальные объекты указанного типа.

2.6. Примеры

Рассмотрим примеры программ, в которых используются различные конструкции языка Си. Первый из них демонстрирует использование символов преобразования `n` в функциях `printf` и `scanf`.

```
/* пример EX2_1 */
#include <stdio.h>
void main(void)
{
    int x,n1,n2;
    printf("Введите целое число от -32768 до 32767\n");
    scanf("%d\n",&x,&n1);
    printf("x = %d\n\n",x,&n2);
    printf("n1 = %d, n2 = %d\n",n1,n2);
}
```

Результаты работы этой программы представляются в виде

```
Введите целое число от -32768 до 32767
234<BBOД>
x=234
n1=3, n2=7
```

Значение `n1` определяет число введенных цифр, а `n2` — число выведенных символов в строке `x=234`.

Следующий пример показывает использование спецификаций `%[]`, а также символов `*` и `#`.

```

/* пример EX2_2 */
#include <stdio.h>
void main(void)
{   char str_b[51],str_e[51];
    int x,n1,n2;
    float y;
    printf("Введите строку до 50 символов\n");
    scanf("%[Минск]%s",str_b,str_e);
    printf("str_b = %s, str_e = %s\n",str_b,str_e);
    y = 12.34567;
    n1 = 8;
    n2 = 3;
    x = 0x100;
    printf("y = %*.*f\n",n1,n2,y);
    printf("x(16) = %#x, x(16) = %x, x(10) = %i\n",x,x,x);
}

```

Результаты работы программы могут быть представлены в таком виде:

```

Введите строку до 50 символов
Мин—1—2—3—4—5<ВВОД>
str_b = Мин, str_e = -1—2—3—4—5
y = 12.346
x(16) = 0x100, x(16) = 100, x(10) = 256

```

Здесь пользователем введена строка Мин—1—2—3—4—5. Из нее только первые три символа (Мин) совпадают с первыми символами, заданными в квадратных скобках рассматриваемой спецификации [Минск]. Поэтому только эти три символа попадут в первую строку, а оставшиеся символы (—1—2—3—4—5) попадут во вторую строку. Число $n1 = 8$ определяет минимальную ширину поля для вывода, а число $n2 = 3$ — количество цифр после запятой. В результате выводимое число 12.34567 будет смещено относительно левой границы и после запятой будут выведены три цифры. Шестнадцатеричное число 0x100 выведено функцией printf с префиксом 0x, без префикса 0x и в десятичной форме.

Третья программа показывает использование условных операторов (if...else) и оператора for для организации цикла. Функцию clrscr() очистки экрана можно использовать только для компилятора Borland C++.

```

/* пример EX2_3 */
#include <conio.h>
#define SYM 'X' /* выводимый символ */
#define SPACE ' ' /* определение пробела */
#define LF 10 /* перевод строки */
#define CR 13 /* возврат каретки */
#define LEFT 24 /* левая граница символа */
#define RIGHT 51 /* правая граница символа */
#define BOTTOM 25 /* нижняя граница символа */
void main(void)
{   int col,line; /* col - номер колонки для вывода символа */

```

```

        /* line - номер линии для вывода символа */
clrscr(); /* очистка экрана */
for(line=1; line<=BOTTOM; line++)
{ /* вывод пробелов до левой границы символа */
  for(col=1; col<LEFT; col++)  putchar(SPACE);
  /* вывод символа X на весь экран */
  for(col=LEFT+1; col<RIGHT; col++)
  if((col==(LEFT+line)) || (col==(RIGHT-line)))
    putchar(SYM);
  else  putchar(SPACE);
  putchar(LF); /* возврат каретки и перевод строки после */
  putchar(CR); /* вывода каждой линии символа X */
}
}

```

После ее запуска на выполнение на весь экран будет выведен символ X. Новая библиотечная функция `clrscr` имеет следующий прототип:

```
void clrscr (void);
```

Она выполняет очистку экрана и описана во включаемом файле `conio.h`.

Четвертая программа демонстрирует использование рекурсивной функции для вычисления факториала. Для того чтобы выполнить ее как консольное приложение, в Visual C++ необходимо заменить строку `#include<values.h>` на строку `#include<limits.h>` и константу `MAXLONG` на константу `LONGMAX`.

```

/* пример EX2_4 */
#include <stdio.h>
#include <values.h>
#include <process.h>
/* рекурсивная функция factorial */
long factorial(int value)
{ long result = 1;
  if(value != 0)
  { result = factorial(value-1);
    /** проверка возможности вычисления факториала **/
    if(result > MAXLONG / (value+1))
    { fprintf(stderr, "Очень большое число\n"); exit(1); }
    result *= value; }
  return(result);
}
/* рекурсивное вычисление факториала числа value */
void main(void)
{ int value; /* факториал этого значения вычисляется */
  long result; /* переменная для результата */
  puts("Факториал какого числа?");
  scanf("%d", &value);
  result=factorial(value);
  printf("Результат: %ld\n", result);
}

```

Результаты работы этой программы представляются в виде

```

Факториал какого числа?
6<ВВОД>

```


Результат: 720

Пятая программа позволяет подсчитывать число символов и слов во вводимых строках (новые числа введенных символов и слов суммируются с предыдущими; пробелы входят в число введенных символов).

```
/* пример EX2_5 */
#include <stdio.h>
#include <conio.h>
#define ESC 27 /* 27 - ASCII-код клавиши ESC */
void Count_of_Lines(void)
{ /* статические переменные будут сохранять старые значения
   при каждом новом вызове функции Count_of_Lines */
  static int words = 0, symbols = 0; /* words - число слов
                                     symbols - число символов */
  char temp, t = 0; /* временные переменные */
  ++symbols;
/* число символов и слов выдается после нажатия клавиши ВВОД */
  while((temp = getche()) != '\r')
  { ++symbols; /* подсчитывается каждый символ */
/* после одного либо нескольких пробелов подсчитывается слово */
    if((temp == ' ') && (t == 1)) continue;
    if(temp == ' ') { t=1; ++words; }
    else t=0;
  }
  if(t == 1) -- words;
  else ++words;
  printf("\nСлов: %d; символов: %d.\n", words, symbols);
}
void main(void)
{ puts("Для завершения программы введите ESC в начале строки");
  puts("Строка не должна начинаться с пробела и с нажатия "
        "клавиши ВВОД");
  puts("Строка не должна завершаться пробелом");
  while(getche() != ESC) Count_of_Lines();
  getch('\b'); getch(' '); getch('\b'); }
```

Результаты работы этой программы представляются в виде

```
Для завершения программы введите ESC в начале строки
Строка не должна начинаться с пробела и с нажатия клавиши ВВОД
Строка не должна завершаться пробелом
Минск Москва Киев<ВВОД>
Слов: 3; символов: 17
ESC<ВВОД>
```

Следующая группа программ демонстрирует работу с файлами. Она позволяет организовать в файле на диске телефонный справочник и выполняет следующие функции:

- занесение фамилии абонента и номера телефона в справочник;
- поиск в справочнике номера телефона по фамилии абонента;
- удаление из справочника фамилии абонента и номера его телефона.

В рассматриваемых программах (они работают только в среде Borland C++) используется много новых библиотечных функций языка Си++, которые описаны в приложении.

Ниже приведен текст головной программы MAIN.CPP:

```
/* программа MAIN.CPP */
/*****
/* Это головная программа для работы с телефонным справочником */
*****/
#include "a:\myh.h" /* файл с глобальными переменными
и символьными значениями */
#include "a:\findt.cpp" /* поиск строки str в файле */
#include "a:\choicet.cpp" /* проверка наличия строки в файле */
#include "a:\addt.cpp" /* добавление строки в файл */
#include "a:\subt.cpp" /* удаление строки из файла */
void main(int argc, char *argv[])
{ if(argc == 3)
    if(*argv[1] == '+') { /* добавить запись */
        if(Choice(argv[2]) == 0) /* проверка нет-ли та-
кой записи в файле */
            { puts("Эта фамилия есть в справочнике"); exit(1); }
        Add(argv[2]); /* добавление записи */
    }
    else if(*argv[1] == '-') Sub(argv[2]); /* удаление
записи */
    else puts("Ошибочное значение аргумента");
    else if(argc == 2) Find(argv[1]); /* поиск записи */
    else puts("Ошибочное число аргументов");
}
```

С помощью директив #include в головную программу включаются другие файлы: MYH.H, FINDT.CPP, CHOICE.CPP, ADDT.CPP, SUBT.CPP. Предположим, что все они находятся в корневом директории диска A:. Если это не так, то необходимо изменить соответствующие директивы #include. В файле MYH.H определены глобальные переменные и некоторые символьные значения.

```
/* программа MYH.H */
/*****
/* определение глобальных переменных и символьных значений */
*****/
#include <stdio.h>
#include <process.h>
#include <string.h>
#define MAX_NAME 20 /* максимальное число букв в фамилии */
#define MAX_NUMBER 10 /* максимальное число букв в телефон-
ном номере */
char Name[MAX_NAME]; /* массив для фамилии */
char Number[MAX_NUMBER]; /* массив для телефонного номера */
/* файл телефонного справочника имеет имя TEL_NUM.TXT и
находится в поддиректории TEL диска A: */
char File[] = "A:\\TEL\\TEL_NUM.TXT";
int Count; /* число фамилий в справочнике */
FILE *F_tel; /* логическое имя файла TEL_NUM.TXT */
int Distance = MAX_NAME + MAX_NUMBER; /* размер одной
записи в файле TEL_NUM.TXT */
```

Файл MYH.H, в частности, определяет, что телефонный справочник будет организован в поддиректории TEL диска A:. Поэтому необходимо перед запуском программы MAIN.CPP создать этот поддиректорий либо использовать другой поддиректорий. В последнем случае необходимо изменить строку:

```
char File[] = "A:\\TEL\\TEL_NUM.TXT";
```

в файле MYH.H. эта строка так же задает имя файла с телефонным справочником (TEL_NUM.TXT).

Программа FINDT.CPP, текст которой приведен ниже, позволяет найти заданную строку в файле TEL_NUM.TXT.

```
/* программа FINDT.CPP */
/*****
/* Эта программа позволяет найти строку str в файле */
/* TEL_NUM.TXT */
*****/
void Find(char *str)
{ int i; /* временная управляющая переменная */
  /** если файл невозможно открыть для чтения, то **/
  /** завершение работы программы **/
  if((F_tel = fopen(File,"r")) == NULL)
  { fprintf(stderr, "\\\"%s\\\" невозможно открыть\\n",File);
    exit(1); }
  /** чтение числа записей (Count) в файле **/
  if(fread(&Count,sizeof(int),1,F_tel) != 1)
  { fprintf(stderr, "\\\"%s\\\": ошибка чтения\\n",File);
    exit(1); }
  /** в цикле for осуществляется поиск нужной записи **/
  for(i=0;i<Count;i++)
  { fread(Name,1,MAX_NAME,F_tel); /* читается имя */
    fread(Number,1,MAX_NUMBER,F_tel); /* читается номер */
    if(ferror(F_tel)) /* проверяется отсутствие ошибки */
    { fprintf(stderr, "\\\"%s\\\": ошибка чтения\\n",File);
      exit(1); }
    if(strcmp(str,Name) == 0) /* сравнивается с нужной
      строкой и если результат положительный, то фамилия
      и найденный номер телефона выводятся на экран */
    { printf("Фамилия: %s\\n",Name);
      printf("Номер телефона: %s\\n",Number);
      fclose(F_tel); return; }
  } /* если результат поиска отрицательный, то выводится
      следующее сообщение */
  fprintf(stderr, "\\\"%s\\\": запись отсутствует"
    " в базе данных\\n",File);
  fclose(F_tel);
  return;
}
```

Программа CHOICET.CPP, текст которой приведен ниже, позволяет проверить, есть ли заданная строка в файле TEL_NUM.TXT.

```

/* программа CHOICET.CPP */
/*****
/* Эта программа позволяет проверить есть строка str */
/* в файле TEL_NUM.TXT или нет */
*****/
int Choice(char *str)
{
    int i;
    char temp[MAX_NAME+MAX_NUMBER];
    if((F_tel = fopen(File,"r")) == NULL)
        return(1); /* строки str нет в файле */
    if(fread(&Count,sizeof(int),1,F_tel) != 1)
    { fprintf(stderr,"%s\": ошибка чтения\n",File);
      exit(1); }
    for(i=0;i<Count;i++)
    { fread(temp,1,(MAX_NAME+MAX_NUMBER),F_tel);
      if(ferror(F_tel))
      { fprintf(stderr,"%s\": ошибка чтения\n",File);
        exit(1); }
      if(strcmp(str,temp) == 0)
      { fclose(F_tel);
        return(0); /* строка str есть в файле */}
    }
    fclose(F_tel);
    return(1); /* строки str нет в файле */
}

```

Программа ADDT.CPP, текст которой приведен ниже, позволяет добавить заданную строку в файл TEL_NUM.TXT.

```

/* программа ADDT.CPP */
/*****
/* Эта программа позволяет добавить строку str */
/* в файл TEL_NUM.TXT */
*****/
void Open(void) /* функция Open создает файл для
                записи, если он не существует */
{
    if((F_tel = fopen(File,"wb+")) == NULL)
    { fprintf(stderr,"%s\": невозможно открыть\n",File);
      exit(1); }
    Count=0; /* сначала в файле 0 записей */
    if(!fwrite(&Count,sizeof(Count),1,F_tel))
    { fprintf(stderr,"%s\": ошибка записи\n",File);
      exit(1); }
}

void Add(char *s) /* функция Add добавляет запись в файл */
{
    char str[MAX_NAME],sn[MAX_NUMBER]; /* временные массивы */
    int i; /* управляющая переменная */
    for(i=0;i<MAX_NAME;i++) str[i] = ' '; /* пробелы в str */
    strcpy(str,s); /* копирование строки s в str */
    if((F_tel = fopen(File,"rb+")) == NULL)
        Open(); /* создание файла, если он не существует */
    else if(fread(&Count,sizeof(Count),1,F_tel) != 1)
    { fprintf(stderr,"%s\": ошибка чтения\n",File);
      exit(1); }
    printf("Номер телефона: "); /* запрашивается и вводится
                                номер телефона */
}

```

```

    if(gets(Number) == NULL || *Number == '\0')
    { fclose(F_tel);
      return; } /* возврат, если номер не введен */
/** установка указателя в файле на первую свободную запись */
if(fseek(F_tel,(long)(Distance * Count),SEEK_CUR) != 0)
{ fprintf(stderr,"%s\n": ошибка поиска\n",File);
  exit(1); }
fwrite(str,1,MAX_NAME,F_tel); /* запись в файл фамилии */
for(i=0;i<MAX_NUMBER;i++) sn[i] = ' '; /* пробелы в sn */
strcpy(sn,Number); /* копирование строки Number в sn */
fwrite(sn,1,MAX_NUMBER,F_tel); /* запись в файл номера */
if(ferror(F_tel)) /* проверка наличия ошибки */
{ fprintf(stderr,"%s\n": ошибка записи\n",File);
  exit(1); }
/** установка указателя в файле на первый байт */
if(fseek(F_tel,0L,SEEK_SET) != 0)
{ fprintf(stderr,"%s\n": ошибка позиционирования\n",File);
  exit(1); }
++Count; /* увеличение числа записей на единицу */
if(fwrite(&Count,sizeof(int),1,F_tel) != 1) /* запись
                                             значения Count в файл */
{ fprintf(stderr,"%s\n": ошибка записи\n",File);
  exit(1); }
fclose(F_tel);
return;
}

```

Программа SUBT.CPP, текст которой приведен ниже, позволяет удалить заданную строку из файла TEL_NUM.TXT.

```

/* программа SUBT.CPP */
/*****
/* Эта программа позволяет удалить строку str */
/* из файла TEL_NUM.TXT */
*****/
void Sub(char *str)
{ int i,j;
  char temp[MAX_NAME+MAX_NUMBER]; /* временный массив */
  if((F_tel = fopen(File,"r+")) == NULL)
  { fprintf(stderr,"%s\n": невозможно открыть\n",File);
    exit(1); }
  if(fread(&Count,sizeof(int),1,F_tel) != 1)
  { fprintf(stderr,"%s\n": ошибка чтения\n",File);
    exit(1); }
  /* в цикле for осуществляется поиск удаляемой строки в файле */
  for(i=0;i<Count;i++)
  { fread(temp,1,MAX_NAME+MAX_NUMBER,F_tel);
    if(ferror(F_tel))
    { fprintf(stderr,"%s\n": ошибка чтения\n",File);
      exit(1); }
    if(strcmp(str,temp) == 0) /* если строка найдена,
                               то она удаляется */
    { for(j=i;j<Count;j++) /* здесь удаляется строка */
      { fread(temp,1,MAX_NAME+MAX_NUMBER,F_tel);
        fseek(F_tel,(long)(j*Distance+2L),SEEK_SET);
        fwrite(temp,1,MAX_NAME+MAX_NUMBER,F_tel);
        fseek(F_tel,(long)((j+2)*Distance+2L),SEEK_SET);
        if(ferror(F_tel))

```

```

        { fprintf(stderr, "\\\"%s\\\": ошибка чтения\\n", File);
          exit(1); }
    }
    --Count; /* при удалении строки декремент Count */
    fseek(F_tel, 0L, SEEK_SET); /* установка указателя */
    /* запись уменьшенного значения Count в файл */
    if(fwrite(&Count, sizeof(Count), 1, F_tel) != 1)
    { fprintf(stderr, "\\\"%s\\\": ошибка записи\\n", File);
      exit(1); }
    fclose(F_tel);
    puts("Запись удалена из файла");
    return;
}
}
fprintf(stderr, "\\\"%s\\\": отсутствует в базе данных\\n", File);
fclose(F_tel); return;
}

```

Ниже приводится возможный сценарий работы с программой MAIN.

```

main + Петров<ВВОД>
Номер телефона: 22—11—33<ВВОД>
main + Иванов<ВВОД>
Номер телефона: 45—46—47<ВВОД>
main + Козлов<ВВОД>
Номер телефона: 12—24—56<ВВОД>
main Иванов<ВВОД>
Фамилия: Иванов
Номер телефона: 45—46—47
main — Иванов<ВВОД>
Запись удалена из файла
main Иванов
"A:TEL\TEL_NUM.TXT": запись отсутствует в файле

```

Последняя программа SHOWT.CPP позволяет вывести на экран содержимое телефонного справочника.

```

/* программа SHOWT.CPP */
/*****
/* Эта программа позволяет вывести все записи из */
/* файла TEL_NUM.TXT */
*****/
#include "a:\myh.h"
void Show(void)
{ int i; /* временная управляющая переменная */
  /** если файл невозможно открыть для чтения, то **/
  /** завершение работы программы **/
  if((F_tel = fopen(File, "r")) == NULL)
  { fprintf(stderr, "\\\"%s\\\": невозможно открыть\\n", File);
    exit(1); }
  /** чтение числа записей (Count) в файле **/
  if(fread(&Count, sizeof(int), 1, F_tel) != 1)
  { fprintf(stderr, "\\\"%s\\\": ошибка чтения\\n", File);
    exit(1); }
  /** в цикле for осуществляется вывод всех записей **/
  for(i=0; i<Count; i++)

```

```

{ fread(Name,1,MAX_NAME,F_tel);      /* читается имя */
  fread(Number,1,MAX_NUMBER,F_tel);   /* читается номер */
  if(ferror(F_tel)) /* проверяется отсутствие ошибки */
  { fprintf(stderr,"\"%s\": ошибка чтения\n",File);
    exit(1);  }
    /** вывод на экран фамилии и номера телефона **/
    printf("Фамилия: %s; номер телефона: %s.\n",Name,Number);
  }
  fclose(F_tel);
}
void main(void)
{ Show(); }

```

Большое число других примеров программ на языке Си приведено в книгах [4, 5, 6, 8, 9, 10, 19, 20, 23, 24, 27, 28, 29].

ГЛАВА 3

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ Си ++

3.1. Введение в язык Си ++

Объектно-ориентированное программирование. Первым языком объектно-ориентированного программирования (ООП) был язык Simula, разработанный в 60-х годах Dahl, Myhrhang и Nygard. В ООП введено понятие объекта и реализованы механизмы вычислений, позволяющие:

- 1) описывать структуру объекта;
- 2) описывать действия с объектами;
- 3) использовать специальные правила наследования объектов;
- 4) устанавливать различную степень защиты компонентов объектов и определять различные права доступа к ним;
- 5) передавать сообщения между объектами.

Главная причина возникновения ООП связана с поиском простых путей для создания очень сложных программ. ООП наследует лучшие черты структурного программирования и комбинирует их с некоторыми новыми подходами.

Сначала рассмотрим содержательную характеристику основных элементов ООП и введем некоторые новые понятия. В последующих параграфах этой главы они будут определены более точно. Для описания компонентов объектов используются специальные конструкции, подобные структурам (struct) и смесям (union) в языке Си. Одна из таких конструкций называется классом (class). Компоненты класса можно отнести к одному из двух базовых типов: данные и методы или функции.

Рассмотрим простой пример. Предположим, что необходимо построить класс **Точка** (Point). Создадим следующее описание:

Класс **Точка**

Компоненты *x* и *y* целого типа;

Если задан объект типа **Точка**, с двумя параметрами *x* и *y*, то на экране по координатам *x* и *y* отобразить точку.

Конечно, это очень упрощенное описание. Заметим важную особенность. Как и в случае данных типа `struct` и `union` языка Си, тип класс задает некоторый шаблон. Он определяет подмножество компонентов — данных и функций. Ниже увидим, что некоторые компоненты можно скрыть и таким образом запретить их использование вне класса. Далее объявляются конкретные экземпляры класса, которые называются *объектами*.

На основании рассмотренного класса **Точка** можно построить другой класс, например **Прямоугольник** (`Rectangle`). Пусть его компонентами будут два объекта типа **Точка** и любые функции, задающие действия (перемещение, копирование и т. п.). Затем можно создать программу, которая будет манипулировать объектами типа **Точка** и **Прямоугольник** для построения некоторых графических изображений.

ООП базируется на трех ключевых понятиях:

1. Пакетирование (`Encapsulation`).
2. Наследование (`Inheritance`).
3. Полиморфизм (`Polymorphism`).

Дадим им краткую характеристику.

По своей сути объект содержит данные (`DATA`) и код (`CODE`), который манипулирует этими данными. Выполняемый код получен из некоторых определенных пользователем функций, включенных в конкретный класс. Часть данных и кода может быть защищена от воздействия на них любыми средствами вне класса. Такое связывание кода и данных называют *пакетированием*. Пакетирование предполагает структурирование данных внутри класса, определение возможных действий для манипуляции с ними и установление прав доступа к данным.

После описания класса можно объявить его конкретные экземпляры. Описанный класс представляет собой новый тип данных.

Один класс может быть получен из другого. При этом первый из них называется *производным*, а второй — *базовым*. Например, на основании базового класса **Прямоугольник** может быть образован производный класс **Окно** (`Window`). Из производного класса можно получить определенный доступ к данным и функциям базового класса. Таким образом, *наследование* — это возможность передачи некоторых свойств одного объекта другому объекту. Его важной особенностью является возможность поддержки концепции классификации.

Полиморфизм по своей сути означает то, что одно и то же имя может быть использовано для взаимосвязанных, но несколько отличающихся элементов программы. Например,

можно определить функцию, которая в зависимости от контекста будет получать целые или вещественные аргументы. При вызове функции анализируется тип аргументов и выполняется код, соответствующий этому типу. В языке Си++ полиморфизм поддерживается на этапе компиляции и на этапе выполнения программы.

Дадим определение ООП. *Объектно-ориентированное программирование* — это технология создания новых объектов (типов данных), которые наследуют определенные черты существующих (ранее созданных) объектов (типов данных). Важное место в ООП занимает структура программы. Здесь необходима большая работа на предварительных этапах с целью создания эффективных библиотек.

Использование подходов, принятых в ООП, дает следующие преимущества:

1. Предоставляется возможность создания более сложных программ при снижении трудоемкости программирования.
2. Повышается надежность программ.
3. Стиль написания программы повышает доступность ее восприятия и упрощает внесение возможных изменений (модификацию программы).
4. Существенно упрощается процедура поиска ошибок.
5. Сложные программы могут быть декомпозированы на изолированные подзадачи, что позволяет создавать большие системы коллективом одновременно работающих программистов.

Некоторые отличительные особенности языка Си++. Если охарактеризовать язык Си++ в целом, то можно сказать, что он расширяет существующие конструкции языка Си. Это, в частности, означает, что программы, написанные на языке Си, могут компилироваться и выполняться в среде Си++. Однако обратное утверждение неверно. Заметим также, что наилучшие результаты можно получить путем использования новых подходов, принятых в ООП.

В этом параграфе рассматриваются некоторые отличительные черты языка программирования Си++. Их более детальное обсуждение дается в последующих параграфах.

Рассмотрим простой пример.

```
/* пример EX3_1 */
#include <iostream.h>
void main(void)
{ int i; // так можно задавать комментарии в C++
  long j;
  unsigned char my_str[] = "пример вывода в C++\n";
  cout << "Так можно выводить данные:" << my_str;
```

```

    cout << "Введите целое и длинное целое число: ";
// так можно вводить данные в C++
    cin >> i >> j;
    cout << "целое = " << i << ", длинное целое = "
        << j << "\n";
    cout << "введите строку: ";
    cin >> my_str;
    cout << "введенная Вами строка: " << my_str;
}

```

Здесь демонстрируются новые возможности ввода/вывода по отношению к языку Си. Включенный файл `iostream.h` описывает функции, обеспечивающие соответствующую поддержку. Наличие в тексте программы строки `cout << "Так можно выводить данные">> << my_str;` приводит к тому, что текст **Так можно выводить данные: пример вывода в Си++** появится на экране дисплея и курсор переместится в начало следующей строки (так как текст заканчивается управляющим символом `\n`). Выводимыми данными могут быть символы, строки и числа. Наличие в тексте программы другой строки `cin >> i >> j;` позволяет ввести целое число и длинное целое число с клавиатуры и передать его переменным `i` и `j`. Вводимыми данными также могут быть символы, строки и числа. Строка `cout << "целое = " << i "длинное целое = " << j << "\n";` приводит к выводу текста **целое =**, введенного числа `i`, текста **длинное целое =**, введенного числа `j` и переводу курсора в начало следующей строки. Аналогичным образом оператор `<<` можно использовать любое число раз. Это еще будет показано ниже на примерах.

Заметим, что наряду с новыми возможностями можно использовать любые библиотечные функции ввода/вывода языка Си (`printf`, `scanf` и т. п.). Кроме того, в другом контексте операторы `>>` и `<<` задают операции сдвига.

Укажем еще одну новую возможность языка Си++. Для записи комментариев разрешается использовать две наклонные черты, например `//` это комментарий. Текст комментария начинается с любой позиции после знака `//` и оканчивается в конце текущей строки.

Одной из наиболее важных отличительных особенностей языка Си++ является введение новых типов данных, называемых *классами*. Ключевое слово `class` дает возможность пользователю создать новый тип данных, подобный записям (`struct`) и смесям (`union`) в языке Си. Выше уже говорилось о том, что компонентами класса могут быть данные и функции. Их можно задавать с атрибутами `private` (локальный),

public (глобальный) и protected (защищенный). Пока оставимся только на первых двух.

По умолчанию все компоненты класса имеют атрибут private. Это означает, что они будут недоступны вне класса, за исключением некоторых специальных ситуаций, которые рассматриваются ниже. Атрибут можно изменять путем записи перед компонентами нового ключевого слова (например, public) и двоеточия. Глобальные компоненты доступны в любом месте программы. В результате одна из упрощенных форм описания класса представляется в виде (в ходе последующего изложения материала она будет уточняться)

```
class имя_класса {  
    данные и функции с атрибутом private по умолчанию public:
```

```
    данные и функции с атрибутом public  
}; объекты этого класса через запятую;
```

Список объектов после закрывающейся фигурной скобки может быть и пустым (тогда задается только шаблон, который можно использовать в будущем).

Рассмотрим пример программы, на котором будем давать дальнейшие пояснения:

```
/* пример EX3_2 */  
#include <iostream.h>  
#define size 100  
/*****  
/* Описание шаблона для класса list_int */  
/*****  
class list_int {  
    int my_list[size]; // это компоненты-данные класса  
    int count; // list_int с атрибутом private  
public: // все остальные компоненты имеют атрибут public  
    void init(void); //  
    void add(int); // это компоненты-функции  
    void del(int); // класса list_int  
    void show(void); //  
};  
/*****  
/* Описание компонента-функции init класса list_int */  
/* (инициализация списка) */  
/*****  
void list_int::init(void)  
{ count = 0; }  
/*****  
/* Описание компонента-функции add класса list_int */  
/* (добавление элемента целого типа в список) */  
/*****  
void list_int::add(int new_el)  
{ if(count == size)  
    { cout << "список заполнен\n";  
      return; }  
  for(int i=0; i<count; i++)  
    if(my_list[i] == new_el)  
      return;  
  my_list[count++] = new_el; }  
}
```

```

/*****
/* Описание компонента-функции ins класса list_int */
/*   (удаление элемента целого типа из списка)   */
*****/
void list_int::del(int del_el)
{ for(int i=0; i<count; i++)
    if(my_list[i] == del_el)
    { for(int j=i; j<count; j++)
        my_list[j] = my_list[j+1];
      count--;
    }
  return; }
/*****
/* Описание компонента-функции show класса list_int */
/*   (вывод на экран дисплея элементов списка)   */
*****/
void list_int::show(void)
{ for(int i=0; i<count; i++)
    cout << my_list[i] << " ";
  cout << "\n";
}
void main(void)
{ list_int m_l; // объявление экземпляра m_l класса list_int
  m_l.init(); // инициализация списка
  m_l.add(135); // добавление числа 135 в список
  m_l.add(28); // добавление числа 28 в список
  m_l.add(234); // добавление числа 234 в список
  m_l.show(); // вывод на экран элементов списка
  m_l.del(28); // удаление числа 28 из списка
  m_l.show(); // вывод на экран элементов списка
}

```

В этом примере определен один класс с именем `list_int`. Сначала для него задан пустой список объектов (сразу после закрывающейся фигурной скобки следует точка с запятой). Четыре компонента класса (`init`, `add`, `ins`, `show`) являются функциями, и для них должен быть задан текст программы. Код функции может записываться сразу же после появления соответствующего имени в пределах класса. В этом случае он будет встроенный. Например, для первой функции `init` после слова `public`: можно было бы записать такой текст:

```
void init (void) { count=0; }
```

В примере избран другой возможный путь и все функции описаны отдельно. В результате появились строки:

```
void list_int::init(void)
{ count=0; }
```

Знак `::` (два рядом расположенных двоеточия) называется областью действия оператора (`scope resolution operator`). Он говорит компилятору о том, что данная версия функции `init` принадлежит классу `list_int` (функция `init` задана в контексте класса `list_int`). Далее будет показано, что различным классам разрешается использовать функции с одинаковыми именами.

В функции `main` задано объявление одного объекта `m_l` класса `list_int` (строка `list_int m_l;`). Теперь к компонентам этого класса можно обращаться так же, как к компонентам структур или смесей: записывается имя объекта, ставится точка и далее записывается имя нужного компонента объекта, например: `m_l.init();`. Обратим внимание на то, что для объявления объекта достаточно записать только имя класса, а ключевое слово `class` использовать не надо.

Компоненты `my_list` и `count` в классе `list_int` являются локальными. Это означает, что обращения в функции `main` вида `m_l.count;` или `m_l.my_list[i];` приведут к возникновению ошибки.

Рассмотрим описание любой функции — компоненты класса, например `add`. Во-первых, так как она является членом класса, то может обращаться к локальным компонентам, например `count`. Во-вторых, обращение к другой функции — компоненту класса производится без записи префикса имени класса и точки (в примере это `m_l`). Если же функция обращается к компоненту, не принадлежащему этому классу (возможно к глобальному компоненту другого класса), то указанный выше префикс (типа `m_l`) является обязательным.

Еще одной особенностью языка Си++ является то, что тип данных можно задать в любом месте функции (не обязательно в ее начале). Например, в функции `add` переменная `i` целого типа объявлена непосредственно в операторе цикла:

```
for (int i = 0; i < count; i++)
```

Функция `main` заносит три целых числа в список, затем выводит этот список на экран дисплея, удаляет из него второй элемент и повторно выводит новый список на экран дисплея. Максимальное число элементов (100) задается командой макропроцессора `#define` и может быть легко изменено.

Рассмотрим конструкции языка Си++, позволяющие выполнять перегрузку функций (`function overloading`). Две или более функции могут иметь одно и то же имя, а также число параметров и различаться только типами параметров и возвращаемых значений. Механизмы описания и использования таких функций демонстрируются в приведенной ниже программе:

```
/* пример EX3_3 */
#include <iostream.h>
/*****
/* Для функции add имеется возможность перегрузки */
*****/
// первое описание add
int add(int i, int j)
```

```

{ cout << "функция оперирует целыми числами\n";
  return i + j;
}
// второе описание add
double add(double i, double j)
{ cout << "функция оперирует вещественными числами\n";
  return i + j;
}
void main(void)
{ // вычисление суммы целых чисел с помощью add
  cout << "2 + 3 = " << add(2, 3) << "\n";
  // вычисление суммы вещественных чисел с помощью add
  cout << "2.5 + 2.0 = " << add(2.5, 2.0) << "\n";
}

```

В ней созданы две близкие, но в то же время различные функции `add`. Первая из них складывает два целых, а вторая — два вещественных числа двойной точности. Компилятор Си++ знает, какую функцию следует использовать в каждом конкретном случае, поскольку он анализирует типы ее аргументов.

Полезность повторной загрузки можно пояснить, рассмотрев некоторые библиотечные функции языка Си, например `itoa`, `ltoa`, `ultoa`. Все они преобразуют соответственно целое, длинное целое и беззнаковое длинное целое число в строку. Несмотря на то что они выполняют близкие действия, используемые имена различны. Язык Си++ позволяет ввести одно и то же имя для всех трех функций. Это безусловно дает преимущества (сокращает число библиотечных функций, повышает читаемость программы и т. п.).

Свойство полиморфизма распространяется и на другие объекты языка Си++. Важное место среди них занимает повторная загрузка операторов. Выше уже говорилось, что один и тот же знак `>>` (или `<<`) использовался для выполнения сдвига и ввода/вывода. Если некоторый оператор перегружен, то он позволяет реализовать новые действия для определенных классов. Наряду с этим сохраняется и его старое предназначение. Рассмотрим пример:

```

/* пример EX3_4 */
#include <iostream.h>
#include <string.h>
/*****
/* Описание класса String */
*****/
class String
{
  char str[51]; // локальный компонент
public:         // ниже идут глобальные компоненты
  void init(char *s); // функция инициализации

```

```

int operator - (String s_new);
} my_string1, my_string2; // здесь объявлены два объекта
// my_string1 и my_string2 класса String
/*****
/* Функция init обеспечивает копирование строки-аргумента */
/* (s) в строку-компонент (str) класса String */
*****/
void String::init(char *s)
{ strcpy(str,s); }
/*****
/* Теперь оператор (-) можно использовать для вычитания */
/* двух строк */
*****/
int String::operator - (String s_new)
{ for(int i=0; str[i] == s_new.str[i]; i++)
    if (str[i] == 0) return 0;
    return str[i] - s_new.str[i];
}
void main(void)
{ char s1[51], s2[51];
  cout << "Введите первую строку (до 50 символов): ";
  cin >> s1;
  cout << "\nВведите вторую строку (до 50 символов): ";
  cin >> s2;
  my_string1.init(s1); // инициализация объекта my_string1
  my_string2.init(s2); // инициализация объекта my_string2
  // вывод на экран разности двух строк
  cout << "\nСтрока 1 - Строка 2 = ";
  cout << my_string1 - my_string2 << "\n";
}

```

Здесь оператор — (минус) используется для вычитания двух строк. Результат вычитания — это разность ASCII-кодов первых двух несовпавших символов. Так, для двух строк **Минск** и **Минский** вторая из них будет больше первой (первые пять букв совпадают, а шестая буква и в слове **Минский** больше заключительного нуля в слове **Минск**). Строка вида

int String::operator — (String s_new)

говорит о том, что оператор находится в контексте класса String и его результатом является целое число (int). Заметим, что если оператор — это компонент класса String, то его первый операнд должен всегда иметь тип класса String. Другими словами, если задано выражение A — B, то переменная A должна быть объектом класса String. Тип второго операнда задан в круглых скобках (в примере он тоже String).

В последующих строках переменная str — это компонент первого операнда, s_new.str — компонент второго операнда. Ниже в выражении my_string1 — my_string2 заданы конкретные операнды (my_string1 и my_string2 — объекты класса String).

Обратим также внимание на то, что объекты my_string1 и my_string2 объявлены сразу же после шаблона класса

String. Область их действия распространяется на все последующие функции программы.

В предыдущих примерах сталкивались с вопросами инициализации, т. е. присвоения начальных значений компонентам объектов. С этой целью использовались специальные компоненты-функции — `init`. Язык Си++ предоставляет средства, позволяющие автоматически осуществлять инициализацию при создании объекта. Такие действия выполняются специальной функцией, называемой конструктором (`constructor`). Эта функция всегда имеет то же самое имя, что и класс, в котором она определена. Так, для примера `EX3_4` вместо функции `init` можно было бы записать:

```
String (char *s) { strcpy(str,s); }
```

Заметим, что конструктор никогда не должен возвращать значения. Он всегда вызывается при объявлении экземпляра некоторого класса (т. е. при создании объекта). Противоположные действия по отношению к конструктору вызывают функции деструкторы (`destructor`), или разрушители, которые уничтожают объект. Например, конструктор может выделить память, а деструктор — освободить ее. Деструктор имеет то же имя, что и конструктор, но перед ним записывается знак тильда (`~`). Приведем пример, в котором используются рассматриваемые новые компоненты. Заметим, что здесь введение некоторых операторов (например, вывода) фактически бессмысленно и приводится только для пояснений сущности конструкторов и деструкторов:

```
/* пример EX3_5 */
#include <iostream.h>
#include <string.h>
class String
{
    int i;
public:
    String(int j); // это конструктор
    ~String(void); // это деструктор
    void show_i(void);
};
String::String(int j)
{ i = j; cout << "работает конструктор\n"; }
void String::show_i(void) { cout << "i = " << i << '\n'; }
String::~String(void)
{ cout << "работает деструктор\n"; }
void main(void)
{ String my_ob1(25); // инициализация объекта my_ob1
  String my_ob2(36); // инициализация объекта my_ob2
  my_ob1.show_i();
  my_ob2.show_i();
}
```

Ниже приведены результаты выполнения этой программы:

```
работает конструктор
работает конструктор
i = 25
i = 36
работает деструктор
работает деструктор
```

В заключение подраздела перечислим новые ключевые слова, которые введены в язык Си++: `class`, `delete`, `friend`, `inline`, `new`, `operator`, `private`, `protected`, `public`, `template`, `this`, `virtual`.

3.2. Объекты и работа с ними

Описание объекта. В языке Си++ для описания объектов можно использовать не только ключевое слово `class`. Для этих же целей могут применяться ключевые слова `struct` и `union`. Различие между классом (`class`) и структурой (`struct`) состоит лишь в том, что компоненты класса имеют по умолчанию атрибут `private`, а компоненты структуры — `public`. Рассмотрим пример:

```
/* пример EX3_6 */
#include <iostream.h>
// компоненты структуры по умолчанию имеют атрибут public
struct example
{ void print(void) { cout << "использование структуры\n"; }
private:
    int a; // а будет иметь атрибут private
public: // остальные компоненты имеют атрибут public
    example(int A) { a = A; } // это конструктор
    void put_x(void) { cout <<
        "Вывод значения а для примера: " << a << "\n"; }
};
void main(void)
// объявление объекта типа example
{ example ex1(55);
  ex1.print();
  ex1.put_x();
}
```

Здесь все необходимые пояснения приведены в комментариях. Результаты выполнения программы представляются в виде

```
использование структуры
Вывод значения а для примера: 55
```

Смеси (`union`) тоже близки к классам (`class`). Фактически смесь — это та же структура (`struct`), в которой все компоненты-данные помещаются в одно и то же место памяти.

Смесь может включать конструктор и деструктор. Рассмотрим простой пример:

```
/* пример EX3_7 */
#include <iostream.h>
#include <string.h>
// компоненты смеси имеют атрибут public
union my_union
{   char str1[14];
    struct str1_2
    {   char str1[5];
        char str2[8];   } my_s;
    my_union(char *s); // это конструктор
    void print(void);
};
my_union::my_union(char *s)
{   strcpy(str,s);   }
void my_union::print(void)
{   cout << my_s.str2 << '\n';
    my_s.str2[0] = 0;
    cout << my_s.str1 << '\n';
}
void main(void)
{   my_union ob("МинскБеларусь");
    ob.print();
}
```

Результаты выполнения программы представляются в следующем виде:

```
Беларусь
Минск
```

Сначала функция print выводит байты из строки my_s.str2. Там будет записано: Беларусь\0. Затем в нулевой байт my_s.str2 записывается заключительный ноль и вывод строки my_s.str1 будет осуществляться до этого байта.

В заключение укажем, что компоненты смеси не могут иметь атрибуты private и protected.

Операции с объектами. Рассмотрим возможные операции с объектами. Они в основном такие же, как и с другими типами данных. Например, объект можно передать функции в качестве параметра. Другими словами, правомочны такие действия:

```
class my_ob {...}
...
void function (my_ob p) {...}
```

Разрешается создавать массивы объектов типа my_ob. Если задать объявление вида

```
my_ob a, *b;
b = &a;
```

то **b** будет указателем на объект **a**. Теперь обращение к его компонентам можно осуществлять с помощью знака **—>**. Пусть

k — компонент **a**, тогда обращение к нему будет производиться так: **b** → **k**. Ранее говорилось о том, что при увеличении либо уменьшении значения указателя на единицу автоматически прибавляется или вычитается масштабный коэффициент, учитывающий размер соответствующего типа данных. Все это сохраняет силу и при операциях с объектами. Например, если **b** — это указатель на очередной элемент в массиве объектов, то **(b + 1)** — указатель на следующий элемент (т. е. выполняется смещение на размер объекта).

Рассмотрим пример, обобщающий различные операции с объектами:

```
/* пример EX3_8 */
#include <iostream.h>
class My
{ int i;
public:
    My() { i = 0; } // это конструктор
    void put_i(void) { cout << "i = " << i << "\n"; }
    void set_i(int Y) { i = Y; }
};
// функции function передается объект x типа My
void function(My x)
{ My *y; // y - указатель на объект типа My
  y = &x; // теперь y указатель на объект x типа My
  x.put_i(); // обращение к объекту по имени
  y->put_i(); // обращение к объекту через указатель
};
void main(void)
{ My mas[4],*p; // объявление массива mas из четырех
                // объектов и указателя p на объект
  mas[0].set_i(20); // присвоение значений компоненту i
  mas[1].set_i(50); // объектов mas[0], mas[1] и mas[2]
  mas[2].set_i(100);
  p = &mas[1]; // теперь p указатель на mas[1]
  p->put_i(); // вывод значения 50
  (p+1)->put_i(); // вывод значения 100
  (p-1)->put_i(); // вывод значения 20
  function(mas[1]); // вызов функции и передача ей объекта
// mas[1] в качестве параметра (выведутся значения 50 и 50)
  function(mas[3]); // вызов функции и передача ей объекта
// mas[3] в качестве параметра (выведутся значения 0 и 0)
}
```

Все необходимые пояснения к примеру приведены в комментариях. Результаты выполнения программы представляются в следующем виде:

```
i = 50
i = 100
i = 20
i = 50
i = 50
i = 0
i = 0
```

3.3. Наследование и защита

Базовые и производные классы. Одной из важных новых черт языка Си++ является возможность или свойство наследования. Класс, на основе которого строится другой класс, называется *базовым*. Построенный класс называется *производным*. Для задания его шаблона используется следующий синтаксис:

```
class имя_производного_класса: необязательный_атрибут имя_базово-  
го_класса {тело_производного_класса} объекты_производного_класса (через  
запятую);
```

Рассмотрим пример:

```
class Point; public Location {...};
```

Здесь класс Location является базовым и имеет атрибут public. Класс Point является производным. Список объектов опущен (после закрывающейся фигурной скобки стоит точка с запятой). Двоеточие (:) отделяет производный класс от базового. Атрибут класса задается ключевыми словами private и public. Он тоже может опускаться; в этом случае принимается атрибут, заданный по умолчанию (напомним, что для классов он private, а для структур — public). Заметим, что смесь (union) не может быть базовым или производным классом.



Рис. 3.1. Использование атрибутов private, public, protected для защиты доступа к компонентам класса

Производный класс получает определенный доступ к компонентам базового класса. Предположим, что компонент *x* базового класса имеет атрибут public. Тогда доступ к нему

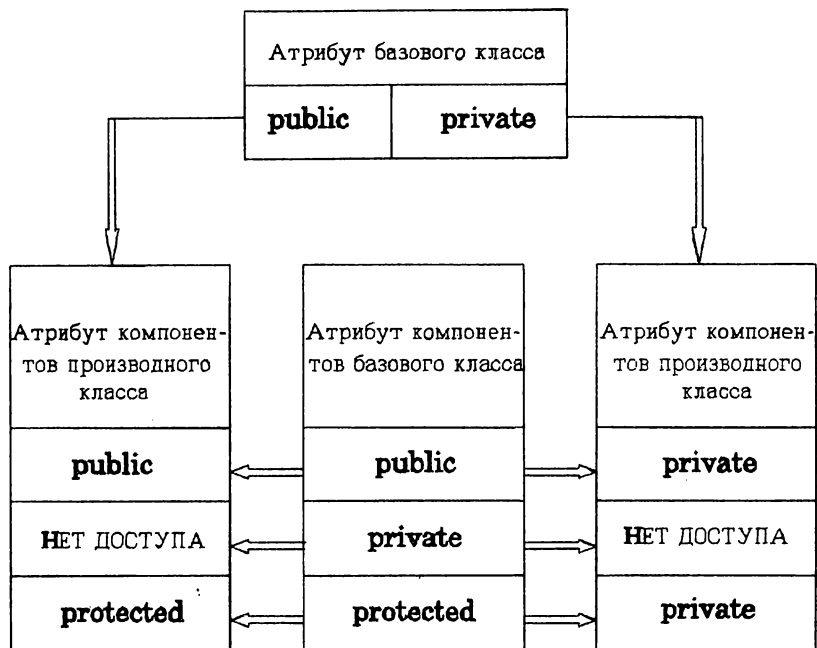


Рис. 3.2. Влияние атрибута базового класса на способ доступа к его компонентам в производных классах

в производном классе осуществляется по имени *x* (т. е. не надо использовать префикс из имени объекта базового класса и точки). Определим возможности языка Си++, связанные с защитой доступа. Для этого рассмотрим рис. 3.1, который он демонстрирует, каким образом атрибуты *public*, *private* и *protected* влияют на механизмы доступа к компонентам. Атрибут *public* не ограничивает доступ. Соответствующий компонент становится глобальным, и к нему можно обращаться везде. Атрибут *private* разрешает доступ к компоненту только из функций — компонентов этого класса. Исключением являются внешние функции, заданные со спецификатором *friend*. О них поговорим позже. Атрибут *protected* по отношению к *private* расширяет доступ возможностью обращения со стороны функций — компонентов производных классов.

Обратим внимание на то, что атрибут на рис. 3.1 записывается внутри шаблона класса, т. е. в описании, заключенном в фигурные скобки.

Рассмотрим рис. 3.2; здесь уже атрибут базового класса — это то, что записывается перед соответствующим именем

(например, `public Location {...}`). Атрибуты компонентов имеют тот же смысл, что и на рис. 3.1. Из рис. 3.2 видно, как атрибут класса изменяет способ доступа к компонентам, заданным в базовом классе.

Ключевые слова `public`, `private` и `protected` могут появляться в любом порядке и любое число раз в шаблоне класса или структуры.

Рассмотрим пример (рассматриваемая программа работает только в среде Borland C++):

```
/* пример EX3_9 */
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
class Location
{
protected:
    int x,y; // эти компоненты будут доступны в
              // производном классе
public:
    // конструктор для класса Location
    Location(int InitX,int InitY);
    void set(int a,int b) { x = a; y = b; }
    int getx() { cout << "x = " << x << "\n"; return x; }
    int gety() { cout << "y = " << y << "\n"; return y; }
    void show_mess() { cout << "Нажмите любую клавишу"; }
};
/*****
/* Класс Point является производным от базового класса */
/*
/* Location
*****/
class Point : public Location
{
    int color;
public:
    Point(int InitX,int InitY,int InitC);
    void putpixel() { ::putpixel(x,y,color); }
};
Location::Location(int InitX,int InitY)
{
    x = InitX; y = InitY;
    cout << "работает конструктор Location\n"; }
Point::Point(int InitX,int InitY,int InitC) :
    Location(InitX,InitY) {
    color = InitC;
    cout << "работает конструктор Point\n"; }
void main(void)
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,""); // инициализация графической системы
    Point my_point(300,200,4);
    my_point.putpixel();
    my_point.getx(); // вызов к функции getx базового класса
    my_point.gety(); // вызов к функции gety базового класса
    my_point.show_mess();
    getch(); // приостановка до нажатия любой клавиши
    closegraph(); // закрытие графической системы
}
```

Здесь созданы базовый класс `Location` и производный класс `Point`. Компоненты `x` и `y` у класса `Location` будут иметь

в Point атрибут protected, а остальные компоненты Location — атрибут public. Если начать описание класса Point другой строкой:

```
class Point: private Location
```

или

```
class Point : Location
```

то все компоненты Location будут иметь в Point атрибут private.

Оговорим теперь на том же примере правила использования конструкторов. Сначала отметим, что если базовый класс имеет конструктор с одним или более аргументами, то любой производный класс должен также иметь конструктор. В нашем примере конструктор для класса Location задан в виде

```
Location(int InitX, int InitY)
{ x = InitX, y = InitY;
  cout << "работает конструктор Location\n";}
```

Он содержит два аргумента целого типа InitX и InitY. Теперь объявление объекта в функции main (либо в другой функции) может осуществляться, например, так:

```
Location my_l(10, 20);
```

В результате после такого объявления компоненты x и y класса Location получают значения 10 и 20 и на экране появится строка: **работает конструктор Location\n**.

В соответствии со сделанными выше замечаниями производный класс Point тоже должен иметь конструктор. В нашей программе он задан следующим образом:

```
Point::Point(int InitX,int InitY,int InitC):
  Location(InitX, InitY) {color= InitC;
  cout << "работает конструктор Point\n";}
```

Значения InitX и InitY интерпретируются как горизонтальная и вертикальная координаты точки, а InitC — ее цвет на экране.

В языке Си++ допускается образовывать производный класс от нескольких базовых классов (об этом еще будет говориться ниже). Поэтому общий синтаксис описания конструктора производного класса представляется в виде

имя_конструктора_производного_класса (список_аргументов_для_производного_класса): имя_базового_класса_1 (список_аргументов_для_базового_класса_1)...имя_базового_класса_N(список_аргументов_для_базового_класса_N)

Список аргументов базовых классов может включать константы, глобальные параметры и/или параметры из списка конструктора производного класса.

Все остальное, что необходимо для пояснения рассмотренного примера, оговаривалось выше. Результаты выполнения программы на персональном компьютере с дисплейными адаптерами EGA/VGA представляются в виде

```
работает конструктор Location
работает конструктор Point
x = 300
y = 200
Нажмите любую клавишу
```

На экране по соответствующим координатам (300, 200) появится точка красного цвета.

В приведенном примере используются библиотечные функции для вывода на экран дисплея графического изображения. Все они описаны в приложении.

Поясним одну функцию:

```
void putpixel() {::putpixel(x, y, color); }
```

Обратим внимание на то, что эта функция класса Point имеет то же самое имя, что и библиотечная функция putpixel, к которой она обращается. В таких случаях для исключения рекурсивных вызовов (или ошибок в программе) необходимо записывать префикс :: перед библиотечной функцией. Действительно, в противном случае функция putpixel не будет знать, к чему ей обращаться (к самой себе или к библиотечной функции с тем же именем). Префикс :: устраняет возможные неоднозначности.

Оговорим еще одну возможность использования операции ::. Для этого рассмотрим простой пример:

```
int a; // это глобальная переменная a
.....
void my_func(void)
{ int a; // это локальная переменная a
  a = 10; // локальная переменная a получает значение 10
  a = 20; // глобальная переменная a получает значение 20
  cout << a << " " << ::a << "\n"; // выводятся значения 10 и 20
  ..... }
```

Здесь операция :: позволяет получить доступ к глобальному элементу (в случае, если локальный и глобальный элементы имеют одинаковые имена).

Множественное наследование. Рассмотрим вопросы, связанные с множественным наследованием, в результате которого производный класс образуется из нескольких базовых классов. Рассмотрим следующую программу:

```

/* пример EX3_10 */
#include <iostream.h>
class Base_1
{   int a;
protected:
    int b;
public:
    Base_1(int x,int y);
    ~Base_1();
    void show1(void)
    { cout << "a = " << a << "; b = " << b; }
};
class Base_2
{
protected:
    int c;
public:
    Base_2(int x);
    ~Base_2();
    void show2(void) { cout << "; c = " << c << "\n"; }
};
class Derive : public Base_1, public Base_2
{   int p;
public:
    Derive(int X,int Y,int Z);
    ~Derive();
    void show3(void) { cout << "a + b + c = " << p+b+c; }
};
Base_1::Base_1(int x,int y) { a=x; b=y;
    cout << "\nконструктор Base_1"; }
Base_1::~~Base_1() { cout << "\ндеструктор Base_1"; }
Base_2::Base_2(int x) { c=x;
    cout << "\nконструктор Base_2"; }
Base_2::~~Base_2() { cout << "\ндеструктор Base_2"; }
Derive::Derive(int X,int Y,int Z) : Base_1(X,Y), Base_2(Z)
{   p=X; cout << "\nконструктор Derive\n"; }
Derive::~~Derive() { cout << "\ндеструктор Derive"; }
void main(void)
{   Derive my_d(3,5,7);
    my_d.show1();
    my_d.show2();
    my_d.show3();
}

```

Каждый из классов Base_1 и Base_2 содержит свой конструктор (первый с двумя аргументами и второй с одним). В связи с этим производный класс тоже содержит свой конструктор. В функции main объявлен объект my_d (Derive my_d(3, 5, 7);). В результате будут вызваны конструкторы базовых и производного классов. Рассмотрим строку, начинающую конструктор производного класса:

```
Derive (int X,int Y,int Z) : Base_1(X,Y), Base_2(Z)
```

Согласно принятому в Си++ соглашению сначала последовательно вызываются конструкторы базовых классов в

соответствии с их появлением в списке. В примере сначала вызывается конструктор Base_1 и затем конструктор Base_2. После исчерпания списка вызывается конструктор производного класса. Выполнение деструкторов осуществляется в обратном порядке. На основании сказанного результаты работы программы представятся в следующем виде:

```
конструктор Base_1
конструктор Base_2
конструктор Derive
a = 3; b = 5; c = 7
a + b + c = 15
деструктор Derive
деструктор Base_2
деструктор Base_1
```

Спецификатор friend. В языке Си++ существует возможность доступа к компонентам класса с атрибутом private из функций, не являющихся компонентами этого класса. Однако эти функции должны иметь спецификатор friend. Рассмотрим пример. Предположим, что задано два класса Point и Circle. Функции putpixel и put_circle, первая из которых компонент Point, а вторая — Circle, обеспечивают отображение на экране дисплея точки и окружности. Допустим нужна еще одна функция (назовем ее equal_color), которая сравнивает цвета точки и окружности и выдает соответствующее сообщение (**цвета совпадают** или **цвета не совпадают**). Рассматриваемая функция не является компонентом классов Point и Circle и в то же время требует доступа к их компонентам с атрибутом private. Чтобы разрешить такой доступ, необходимо объявить функцию equal_color в классах Point и Circle со спецификатором friend. Рассмотрим пример программы, использующей функцию equal_color (программа работает только в среде Borland C++):

```
/* пример EX3_11 */
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
class Circle;
class Location
{
protected:
    int x,y;
public:
    Location(int InitX,int InitY) { x = InitX; y = InitY; }
    void set(int a,int b) { x = a; y = b; }
    int getx() { cout << "x = " << x << "\n"; return x; }
    int gety() { cout << "y = " << y << "\n"; return y; }
};
class Point : public Location
{ int color;
public:
```

```

    Point(int InitX,int InitY,int InitC);
    void putpixel() { ::putpixel(x,y,color); }
// объявление функции со спецификатором friend
    friend void equal_color(Point p,Circle c);
};
class Circle : public Location
{ int color,temp,radius;
public:
    Circle(int InitX,int InitY,int InitC,int InitR);
    void put_circle() { temp = getcolor();
        setcolor(color); circle(x,y,radius);
        setcolor(temp); }
// объявление функции со спецификатором friend
    friend void equal_color(Point p,Circle c);
};
// конструктор для класса Point
Point::Point(int InitX,int InitY,int InitC) :
    Location(InitX,InitY) { color = InitC; }
// конструктор для класса Circle
Circle::Circle(int InitX,int InitY,int InitC,int InitR) :
    Location(InitX,InitY) { color = InitC;
        radius = InitR; }
/*****
* Функция equal_color сравнивает цвета точки и окружности */
/* (она не является компонентом классов Point и Circle) */
*****/
void equal_color(Point p,Circle c)
{ if(p.color == c.color) cout << "\nцвета совпадают\n";
  else cout << "\nцвета не совпадают\n";
}
void main(void)
{ int gd = DETECT, gm;
  initgraph(&gd,&gm,""); // инициализация графической системы
  Point my_point(400,150,4);
  Circle my_circle1(500,200,4,80);
  Circle my_circle2(300,200,3,40);
  my_point.putpixel();
  my_circle1.put_circle();
  my_circle2.put_circle();
  equal_color(my_point,my_circle1);
  equal_color(my_point,my_circle2);
  getch(); // приостановка до нажатия любой клавиши
  closegraph(); // закрытие графической системы
}

```

В начале программы появилось пустое объявление:

```
class Circle;
```

Оно необходимо в связи с тем, что в классе Point есть обращение к еще не описанному классу Circle (equal_color(Point p, Circle c);). Поэтому в начале программы компилятору сообщается, что класс Circle будет описан позже (в противном случае будет выдано сообщение об ошибке). Подобные описания необходимы лишь при наличии функций со спецификатором friend. В примере класс Location является базовым для двух производных классов — Point и Circle. В результате работы программы на экран будет выведена точка, две окружности и два сообщения — «цвета совпадают» (для

точки и первой окружности) и «цвета не совпадают» (для точки и второй окружности).

Новые библиотечные функции, используемые в программе, описаны в приложении.

Строки программы:

```
temp = getcolor( );
setcolor(color);
cirlce(x,y,radius);
setcolor(temp);
```

сохраняют текущий цвет в переменной temp, изменяют цвет, отображают окружность и восстанавливают первоначальный цвет из переменной temp.

3.4. Перегрузка функций и операторов

Перегрузка на этапе компиляции и выполнения. Выше уже говорилось о том, что язык Си++ позволяет перегружать функции и операторы. В настоящем разделе эти вопросы рассматриваются более подробно.

В ООП находят применение термины раннего и позднего связывания. В первом случае операции перегрузки выполняются на этапе компиляции, а во втором — на этапе выполнения программы. Раннее связывание предполагает вызовы стандартных функций, а также перегрузку функций и операторов. Преимуществом использования такого подхода является высокая скорость выполнения программ и малые затраты памяти, недостатком — некоторая потеря гибкости программирования различных задач. Позднее связывание предполагает вызовы виртуальных функций (они будут рассмотрены ниже). Его преимуществом является достижение большей гибкости программирования при некотором снижении скорости выполнения программ. В большинстве случаев пользователи комбинируют эти подходы.

Перегрузка функций. Вопросы перегрузки функций уже обсуждались в § 3.1 (см. пример EX3_3). Поэтому ограничимся рассмотрением еще одного примера.

```
/* пример EX3_12 */
#include <iostream.h>
#include <stdlib.h>
char *toa(int num, char *s, int r)
{ return itoa(num,s,r); }
char *toa(long num, char *s, int r)
{ return ltoa(num,s,r); }
char *toa(unsigned long num, char *s, int r)
{ return ultoa(num,s,r); }
void main(void)
```

```

{   char str[81];
    int r;
    cout << "Задайте систему счисления от 2 до 36" << '\n';
    cin >> r;
    cout << "int = " << toa(3248, str, r) << '\n';
    cout << "long = " << toa(773681L, str, r) << '\n';
    cout << "unsigned long = " << toa(3456789000UL, str, r) << '\n';
}

```

В программе три библиотечные функции языка Си++ — itoa, ltoa, ultoa — заменены одной — toa. Прототипы библиотечных функций представляются в виде

```

char *ltoa(int num, char *str, int radix);
char *ltoa(long num, char *str, int radix);
char *ultoa(unsigned long num, char *str, int radix);

```

Все они описаны во включаемом файле stdlib.h. Каждая функция преобразует число num в строку, на которую указывает str. Преобразование осуществляется в системе счисления radix ($2 \leq \text{radix} \leq 36$). Число num является целым для функции itoa, длинным целым — для функции ltoa и беззнаковым длинным целым — для функции ultoa. Все функции возвращают указатель на полученную строку.

Результаты выполнения программы могут быть представлены в следующем виде:

```

Задайте систему счисления от 2 до 36
10<ВВОД>
int = 3248
long = 773681
unsigned long = 3456789000

```

Перегрузка конструкторов. Конструктор — это тоже функция, а значит он может быть перегружен. Рассмотрим пример:

```

/* пример EX3_13 */
#include <iostream.h>
class Over
{   int i;
    char *str;
public:
    Over() { str = "первый конструктор\n"; i = 0; }
    Over(char *S) { str = S; i = 50; }
    Over(char *S, int X) { str = S; i = X; }
    Over(int *Y) { str = "четвертый конструктор\n"; i = *Y; }
    void print(void) { cout << "i = " << i << " "; str = " << str; }
};
void main(void)
{   int a = 10, *b;
    b = &a;
}

```

```

' // объявление четырех объектов: my_over, my_over1 и my_over2
Over my_over,
    my_over1("для конструктора с одним параметром\n"),
    my_over2("для конструктора с двумя параметрами\n",100),
    my_over3(b);
my_over.print(); // активен конструктор Over()
my_over1.print(); // активен конструктор Over(char *S)
my_over2.print(); // активен конструктор Over(char *S,int X)
my_over3.print(); // активен конструктор Over(int *Y)
}

```

В нем задано четыре конструктора: первый без параметров — `Over()`; второй с одним параметром — `Over(char *S)`; третий с двумя параметрами — `Over(char *S,int X)`; четвертый с одним параметром — `Over(int *Y)`. Объявление `my_over` (без параметров) приводит к выполнению первого конструктора, `my_over1` — второго, `my_over2` — третьего и `my_over3` — четвертого. Результаты выполнения программы представляются в виде

```

l=0; str=первый конструктор
l=50; str=для конструктора с одним параметром
l=100; str=для конструктора с двумя параметрами
l=10; str=четвертый конструктор

```

Динамическая инициализация. В языке Си++ локальные и глобальные переменные могут быть инициализированы во время выполнения программы. Этот процесс иногда называют динамической инициализацией (dynamic initialization). Выше уже говорилось о том, что объявления локальных переменных могут появляться в любом месте тела соответствующей функции (а не только в ее начале, как в языке Си). В Си++ такие объявления разрешается задавать в виде

```
Int j = strlen(str);
```

Здесь после знака «=» можно записать другие выражения, которые вычисляются в процессе выполнения программы. Рассмотрим пример:

```

/* пример EX3_14 */
#include <iostream.h>
#include <string.h>
void main(void)
{
    int i,j;
    i = 4;
    char str[] = "0 0 1 2 3 4 5 6 7 8 9";
    cout << "Пример динамической инициализации\n";
    cout << "str = " << str << "\n";
    int z = strlen(str)-1;
    for(int k=i,l=z;k<=l;k++,l--)
        { j=str[k]; str[k]=str[l]; str[l]=j; }
    cout << "str = " << str << "\n";
}

```

Отметим, что цель этой и всех других, описанных в книге программ — демонстрация на простых примерах новых возможностей языка Си++. Если не учитывать этого требования, то текст программы можно было бы сделать короче. Из приведенного примера видно, что объявления появляются в любом месте тела программы. Остановимся на одном из них:

```
int z = strlen(str) — 1;
```

На этапе компиляции значение `strlen(str)` неизвестно, поскольку здесь используется библиотечная функция `strlen`, вычисляющая длину строки и подсоединяемая на этапе компоновки. В результате инициализация переменной `z` будет выполнена только в процессе выполнения программы. Выражение в цикле

```
int k=1, i=z;
```

задает переменные `k` и `i` целого типа и присваивает им значения `i` и `z`.

Результаты выполнения программы представляются в виде

```
Пример динамической инициализации
str=0 0 1 2 3 4 5 6 7 8 9
str=0 0 9 8 7 6 5 4 3 2 1
```

Механизмы динамической инициализации могут быть применены и к конструкторам. Модифицируем пример EX3_13:

```
/* пример EX3_15 */
#include <iostream.h>
class Over
{ int i;
  char *str;
public:
  Over() { str = "первый конструктор\n"; i = 0; }
          // это первый конструктор
  Over(char *S) { str = S; i = 50; }
          // это второй конструктор
  Over(char *S,int X) { str = S; i = X; }
          // это третий конструктор
  Over(int *Y) { str = "четвертый конструктор\n"; i = *Y; }
          // это четвертый конструктор
  void print(void) { cout << "i = " << i << "; str = " << str; }
};
void main(void)
{ int a = 10,*b;
  b = &a;
  // объявление четырех объектов: my_over, my_over1 и my_over2
  Over my_over,
    my_over1("для конструктора с одним параметром\n"),
    my_over2("для конструктора с двумя параметрами\n",100),
    my_over3(b);
  my_over.print(); // активен конструктор Over()
  my_over1.print(); // активен конструктор Over(char *S)
```



```

my_over2.print(); // активен конструктор Over(char *S,int X)
my_over3.print(); // активен конструктор Over(int *Y)
/***** ДОБАВЛЕНИЕ В ПРОГРАММУ *****/
char my_str[51];
cout << "Введите строку до 50 символов\n";
cin >> my_str; // ввод строки my_str
Over my_over4(my_str); // объявление объекта my_over3
my_over4.print(); // активен конструктор с одним параметром
cout << '\n';
}

```

Объявление объекта `my_over3` и его инициализация осуществлены на этапе выполнения программы. Результаты ее работы могут быть представлены в виде

```

l = 0; str = первый конструктор
l = 50; str = для конструктора с одним параметром
l = 100; str = для конструктора с двумя параметрами
l = 10; str = четвертый конструктор
Введите строку до 50 символов
1122334455<Enter>
l = 50; str = 1122334455

```

Шаблон, описывающий
класс `my_class`

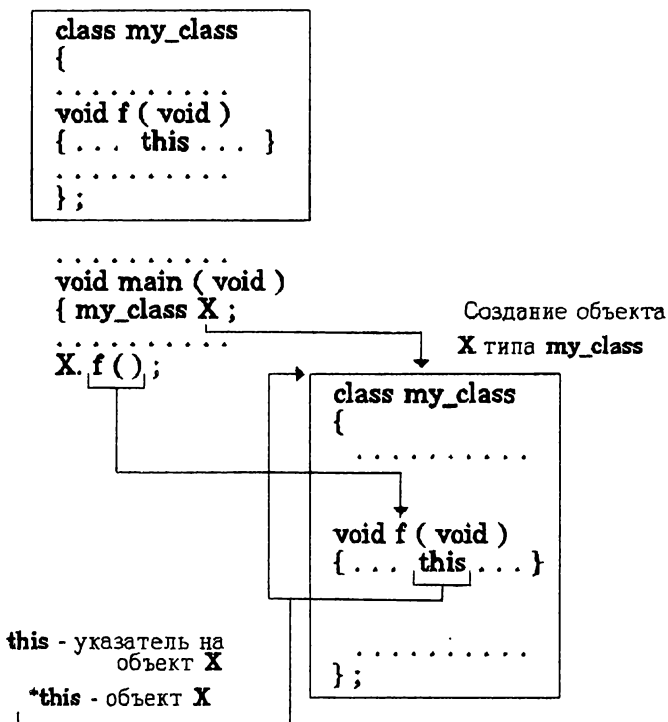


Рис. 3.3. Пример использования указателя `this`

Указатель this. Когда выполняется любая функция — компонент класса, она получает указатель на соответствующий объект. Например, для строки `my_over3.print()`; в примере EX3_15 функция `print()` получает указатель на объект `my_over3`. Доступ к нему в функции-компоненте (в примере `print()`) осуществляется через ключевое слово `this`. Рассмотрим рис. 3.3, который иллюстрирует сказанное. Сначала описан шаблон класса `my_class`, в котором есть функция `f` (другие компоненты нас не интересуют). Принято, что `f` не возвращает значений и не имеет параметров (это тоже не важно). В функции `f` встречается указатель `this`. Предположим, что в функции `main` (или в любой другой функции) объявлен объект `X` класса `my_class`. В результате создан показанный на рис. 3.3 экземпляр. Будем считать, что функция `f` имеет атрибут `public` и к ней происходит обращение вида `X.f()`; Тогда `this` будет указывать на объект `X` класса `my_class`.

Рассмотрим пример:

```
/* пример EX3_16 */
#include <stdio.h>
#include <string.h>
class String
{ char str[100];
  int k;
  int *r;
public:
  String() { k=123; r=&k; }
  void read() { gets(str); }
  void print() { puts(str); }
  void read_print();
};
void String::read_print()
{ char c;
  printf("k = %d\n",this->k);
  printf("*r = %d\n",*(this->r));
  puts("Введите строку");
  /******
  /* Ключевое слово this содержит скрытый указатель на рас- */
  /* матризуемый класс (то есть на класс String). Поэтому */
  /* конструкция this->read выбирает (через указатель) */
  /* функцию read этого класса */
  /******
  this->read();
  puts("Введенная строка");
  this->print(); // аналогичная конструкция для функции print
  /******
  /* В цикле for одинаково удаленные от середины строки */
  /* символы меняются местами */
  /******
  for(int i=0, j=strlen(str)-1; i<j; i++, j--)
  { c=str[i]; str[i]=str[j]; str[j]=c; }
  puts("Измененная строка");
  (*this).print(); // можно использовать и такую конструкцию
}
```

```

void main()
{
    String s;
    s.read_print(); // обращение к функции read_print
}

```

Здесь в функции `read_print()`, которая является компонентом класса `String`, используется указатель `this`. В этом случае конструкции `this` → компонент и `(*this)`, компонент позволяют обращаться к компоненту класса `String`. Результаты работы программы представляются в виде

```

k = 123
*r = 123
Введите строку
1234567890<ВВОД>
Введенная строка
1234567890
Измененная строка
0987654321

```

Перегрузка операторов. Охарактеризуем более детально перегрузку операторов. В этом случае используется синтаксис:

тип_возвращаемого_значени.. имя_класса::operator знак(список_аргументов)
(определение оператора по отношению к данному классу)

Оператор всегда определяется по отношению к компонентам некоторого класса. В результате его старое предназначение сохраняет силу. Компилятор автоматически распознает, какое действие необходимо выполнить путем анализа операндов.

Функция `operator` может быть только компонентом класса либо иметь спецификатор `friend`. Эти два пути ее определения имеют некоторые различия и рассматриваются ниже.

Пусть функция `operator` является компонентом класса. При этом в случае унарной операции `operator` не будет иметь аргументов, а в случае бинарной операции будет иметь один аргумент. В качестве отсутствующего аргумента используется указатель `this` на тот объект, в котором определен оператор. Так, в примере EX3_4, где рассматривается функция `operator`, `str[i]` — это то же самое, что и `this` → `str[i]`.

Рассмотрим простой пример (программа работает только в среде Borland C++; в § 5.7 показано, как ее можно выполнить в среде Windows с использованием MFC и Visual C++).

```

/* пример EX3_17 */
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
class Point
{ int x,y;
public:

```

```

Point(int InitX,int InitY) { x = InitX; y = InitY; }
Point operator++(void) { return Point(x++,y++); }
Point operator-(Point my_p)
    { return Point(x-my_p.x,y-my_p.y); }
void put_point(void) { putpixel(x,y,1); }
};
void main(void)
{
    int gd = DETECT, gm;
    initgraph(&gd,&gm,""); // инициализация графической системы
    Point my_point(10,200);
    for(int j=0;j<10;j++)
        { for(int i=0;i<100;i++)
            { ++my_point;
              my_point.put_point(); }
          my_point = my_point - Point(60,110);
        }
    cout << "Нажмите любую клавишу";
    getch(); // приостановка до нажатия любой клавиши
    closegraph(); // закрытие графической системы
}

```

Первая функция `operator` переопределяет унарную операцию `++`. Заметим, что невозможно переопределить операции `++` и `--` так, чтобы они по-разному воспринимались при записи до и после операнда. Вторая функция `operator` переопределяет бинарную операцию `-`. Если учесть тот факт, что выражение `my_point — Point(60, 110)` возвращает значение типа `Point`, то можно использовать знак «`-`» последовательно со многими операндами, например:

```
my_point = my_point — Point(60, 110) — Point(20, 5);
```

Результатом работы программы является вывод на экран десяти наклонных линий. Доступ из функции `operator` к компонентам `x` и `y` созданного объекта `my_point` осуществляется через указатель `this`. Это позволяет изменить значения `x` и `y` (увеличить их на единицу).

При использовании функции `operator` имеют место следующие ограничения:

- 1) язык Си++ не позволяет различать операции `++` и `--`, записанные до и после операнда;
- 2) нельзя изменять приоритет выполнения операций;
- 3) нельзя изменять число операндов, хотя функция `operator` может и проигнорировать некоторый операнд;
- 4) перегруженные операторы распространяются на любой производный класс, за исключением оператора «`=`» (при необходимости можно повторно перегрузить в производном классе уже перегруженный оператор);

5) следующие операторы не могут быть перегружены: `..`, `::`, `*`, `?`.

Рассмотрим, как задается функция operator со спецификатором friend. Главное отличие заключается в том, что теперь для унарных и бинарных операций требуются все необходимые аргументы (два для бинарных операций и один для унарных). Объясняется это тем, что функция со спецификатором friend не имеет доступа к указателю this. Кроме того, операторы =, (), [] и —> не могут быть перегружены функциями со спецификатором friend. Рассмотрим пример.

```
/* пример EX3_18 */
#include <stdio.h>
#include <string.h>
class My_str
{ char *str;
public:
/*****
/* Первый конструктор задает пустой список строк (с этой */
/* целью выделяется память в один байт, в который запи- */
/* сывается ноль) */
*****/
My_str() { str = new char; *str = 0; }
/*****
/* Второй конструктор выделяет память под очередную строку */
*****/
My_str(char *);
void print() { puts(str); }
/*****
/* Ключевое слово оператор позволяет по новому использовать */
/* знак > (для проверки того факта, что первая строка боль- */
/* ше второй). Ключевое слово friend позволяет внешней функ- */
/* ции использовать компоненты типа private объекта My_str */
*****/
friend int operator > (My_str str1, My_str str2);
/*****
/* Можно не использовать ключевое слово friend, но тогда */
/* первый операнд для оператора всегда имеет тип рассматри- */
/* ваемого класса. В результате эти строки перепишутся так: */
/* int operator > (My_str str2) */
/* { return strcmp(str, str2.str) > 0; } */
*****/
}; // КОНЕЦ ОПИСАНИЯ КЛАССА My_str
/*****
/* Здесь описан второй конструктор */
*****/
My_str::My_str(char *s)
{ str=new char[strlen(s)+1]; // выделяется память под строку
// str длиной strlen(s)+1
strcpy(str,s); } // строка s копируется в строку str
/*****
/* Описание функции оператор (со спецификатором friend) */
*****/
int operator > (My_str str1, My_str str2)
{ return strcmp(str1.str, str2.str) > 0; }
/*****
/* Функция input обеспечивает ввод не более 20 строк и */
/* возвращает их действительно введенное число count. */
```

```

/* Введенные строки попадают в область памяти, выделен- */
/* ную в классе My_str */
/*****
void input(My_str *s, int &count)
{ static char buf[100]; // буфер для очередной строки
  count=0;              // при завершении count - число
                        // действительно введенных строк
  puts("Введите строки для сортировки");
  while(count++ < 20)
    if(scanf("%s",buf) == EOF) break; // завершение по Ctrl-Z
    else s[count] = My_str(buf);
}
/*****
/* функция output выводит список из size строк на экран */
/*****
void output(My_str *s,int size)
{ for(int i=0;i<size;i++) s[i].print();
}
/*****
/* функция sort производит сортировку size строк */
/*****
void sort(My_str *s, int size)
{ int pr=1; // признак для проверки окончания сортировки
  while(pr) { pr=0;
    for(int i=0;i<size-1;i++)
      if(s[i]>s[i+1])
      { My_str temp=s[i];
        s[i]=s[i+1];
        s[i+1]=temp;
        pr=1; }
    }
}
void main(void)
{ int size;
  My_str list_str[20];
  input(list_str,size); // ввод строк
  sort(list_str,size);  // сортировка строк
  puts("\nСписок отсортированных строк");
  output(list_str,size); // вывод отсортированных строк
}

```

Здесь появилось новое ключевое слово `new`. В языке Си++ операторы `new` и `delete` используются для динамического выделения и освобождения памяти (подобно библиотечным функциям `malloc`, `free` и др.). Общий синтаксис их использования представляется в следующем виде:

```

указатель_на_выделенную_память = new имя_компонента;
delete указатель_на_выделенную_память;

```

Оператор `new` пытается выделить память под компонент с заданным именем. При успешном завершении возвращается указатель на начало соответствующей области. В случае неудачи возвращается константа `NULL`. Оператор `delete` освобождает выделенную память.

Обратим внимание на объявление функции `input`:

```
void input (My_str *s, int &count)
```

Напомним, что в языке Си аргументы передаются в функцию по значению. Вызванная функция не может изменить оригинальные значения в вызывающей программе. Для устранения этого ограничения вызываемой функции передаются не сами аргументы, а указатели на соответствующие значения. Эти механизмы подробно рассматривались в гл. 2. Язык Си++ позволяет автоматически сообщать компилятору, что функции передается указатель. Это осуществляется путем записи перед передаваемым компонентом знака `&`. В приведенном выше примере есть такое объявление `int &count`. Теперь компилятор знает, что работать с компонентом `count` необходимо через указатель. В результате после завершения `input` значение `count` в вызывающей функции будет изменено.

В программу можно последовательно вводить различные строки. Завершение ввода осуществляется нажатием клавиш «Ctrl—Z». После этого введенные строки будут отсортированы в алфавитном порядке.

В большинстве практических приложений можно обойтись функцией `operator`, являющейся компонентом класса. При этом есть одно неудобство. Пусть `Ob` — некоторый объект и оператор `+` перегружен так, что он работает с объектом `Ob` (через указатель `this`) и некоторым целым значением. В результате выражение

```
Ob = Ob + 10;
```

будет вполне корректным. Однако выражение

```
Ob = 10 + Ob;
```

уже является ошибочным. Причина заключается в том, что первым операндом является целое число, а не объект. Разрешить возникшие сложности можно использованием двух функций со спецификатором `friend`, что показано в следующем примере:

```
/* пример EX3_19 */
#include <iostream.h>
struct two_friends
{
    int i;
    two_friends operator=(int k) { i=k; return *this; }
    friend two_friends operator*(two_friends f1,int j);
    friend two_friends operator*(int j,two_friends f1);
};
// здесь первый параметр класс, а второй - целое
two_friends operator*(two_friends f1,int j)
```

```

{ two_friends t_f;
  t_f = j * F1.i;
  return t_f;
}
// здесь первый параметр целое, а второй - класс
two_friends operator*(int j,two_friends F1)
{ two_friends t_f;
  t_f = F1.i * j;
  return t_f;
}
void main(void)
{ two_friends X;
  X = 10;
  X = X * 2; // объект умножается на целое число
  cout << X.i << "\n"; // будет выведено число 20
  X = 4 * X; // целое число умножается на объект
  cout << X.i << "\n"; // будет выведено число 80
}

```

Заметим, что точно так же (путем использования нескольких функций `operator`, причем не обязательно со спецификатором `friend`) можно определять несколько допустимых типов возвращаемых значений. Например, один `operator` определяет возвращаемый тип как объект, а другой как строку символов.

Задание аргументов по умолчанию. В языке Си++ имеется возможность назначать значения аргументам функции, заданные по умолчанию. Рассмотрим описание:

```
void function(int x=0) {...}
```

После этого к функции (`function`) можно обратиться одним из двух путей: `function (целое_число);` (например, `function (123);`) и `function();`. В первом случае функция получит в качестве значения аргумента введенное целое число, а во втором — будет использовано значение по умолчанию (в примере — 0). Рассмотрим следующую программу:

```

/* пример EX3_20 */
#include <iostream.h>
void example(char *s,char s1[] = "нет строки 2, ",
             char s2[] = "нет строки 3")
{ cout << s << s1 << s2 << "\n"; }
void main(void)
{ example("Аргументы по умолчанию: ");
  example("Аргументы по умолчанию: ", "первый аргумент, ");
  example("Аргументы по умолчанию: ", "первый аргумент, ",
          "второй аргумент");
}

```

Здесь использовано три вызова функции `example` с одним, двумя и тремя аргументами. Результаты работы программы представляются в виде

Аргументы по умолчанию: нет строки 2, нет строки 3
Аргументы по умолчанию: первый аргумент, нет строки 3
Аргументы по умолчанию: первый аргумент, второй аргумент

Обратим внимание на следующее ошибочное объявление функции `example`:

```
void example(char *s, char s1[] = «нет строки», char *p);
```

Если встретился параметр по умолчанию (`char s1[] = «нет строки»`), то за ним могут идти только аналогичные параметры (т. е. тоже заданные по умолчанию).

Параметры по умолчанию можно указывать также в конструкторах, например:

```
Point::Point(int InitX, int InitY, int InitC = 1)
```

В этом случае при объявлении:

```
Point my_point(10, 30);
```

Третий параметр примет значение 1.

Виртуальные функции. Пусть задан базовый класс `Base` и производный от него класс `Derive`. Запишем следующие объявления:

```
Base *p //p — указатель на объект класса Base
```

```
Base B_ob; //B_ob — объект класса Base
```

```
Derive D_ob; //D_ob — объект класса Derive
```

В языке `C++` используется следующее правило: любая переменная, объявленная как указатель на объект базового класса, может применяться как указатель на объект производного класса. Для нашего примера будут правильными следующие выражения:

```
p # B_ob; //p — указатель на объект класса Base
```

```
p # D_ob; //p — указатель на объект класса Derive
```

Теперь путем использования указателя `p` можно получить доступ ко всем элементам объекта `D_ob`, наследованным из класса `Base`. Однако через него нельзя получить доступ к специфичным (собственным) компонентам класса `Derive`. Если задан указатель на объект производного класса, то через него нельзя получить доступ к компонентам базового класса. Сделаем важное замечание. Пусть задан указатель `p` на объект базового класса. Тогда при увеличении или уменьшении его значения на единицу используется масштабный коэффициент для базового класса. Если этот же указатель будет адресовать объект производного класса, то масштабирование (при увеличении либо уменьшении значения указателя) относительно производного класса будет выполняться неверно.

Виртуальные функции позволяют решать задачу перегрузки (свойство полиморфизма) во время выполнения программы. Они представляют собой функции, объявленные со спецификатором `virtual` в базовом классе и которые затем переопределены в одном или более производном классах. При этом их имена, тип возвращаемого значения, а также число и типы аргументов не меняются. Пусть задано выражение `p = #B_ob;` и виртуальная функция `my_virt`. Предположим, что имеем дело с классами `Base` и `Derive`, описанными в начале параграфа, и `my_virt` определена со спецификатором `virtual` в классе `Base` и без спецификатора `virtual` в классе `Derive`. Тогда `p -> my_virt` будет выбирать функцию базового класса. Запишем другое выражение: `p = #D_ob;`. Теперь `p -> my_virt` будет выбирать функцию производного класса. Таким образом, переопределение некоторого указателя позволяет выполнять различные версии виртуальных функций.

При использовании виртуальных функций имеют место следующие ограничения:

1) прототипы виртуальной функции в базовом классе и во всех производных классах должны соответствовать друг другу (тип возвращаемого значения, а также число и типы параметров должны совпадать). В противном случае функция будет рассматриваться как перегруженная, а не как виртуальная;

2) виртуальная функция должна быть компонентом класса (она не может иметь спецификатор `friend`). Допустимо, чтобы деструктор имел спецификатор `virtual`, однако для конструктора это запрещено.

Если функция объявлена виртуальной, то она сохраняет это свойство для любого производного класса (на любом уровне вложенности). Если в некотором производном классе виртуальная функция не описана (пропущена), то используется версия базового класса.

Абстрактные классы. Иногда возникают ситуации, при которых в случае пропуска виртуальной функции в производном классе нельзя использовать ее версию из базового класса. Тогда объявляется виртуальная функция, исключаящая обращения по умолчанию — `pure virtual function`. При этом используется такой синтаксис:

```
virtual тип_возвращаемого_значения имя_функции (список параметров) = 0;
```

Подобные объявления требуют от любого производного класса наличия собственной версии виртуальной функции. Если она пропущена, то будет выдано сообщение об ошибке.

Если некоторый класс имеет хотя бы одну `pure` виртуальную функцию, то он называется *абстрактным*. Для такого

класса невозможно создавать объекты. Его можно использовать только как базовый класс с целью наследования компонентов в производных классах.

В заключение укажем еще один возможный путь использования ключевого слова `virtual`. При множественном наследовании базовый класс не может быть задан в производном классе более одного раза. Например, второе из следующих двух объявлений является ошибочным:

```
class Base {...};  
class Derive: Base, Base {...};
```

В то же время базовый класс можно косвенно передать производному классу более одного раза:

```
class Derive1 : public Base {...};  
class Derive2 : public Base {...};  
class Derive1_2 : public Derive1, public Derive2 {...};
```

В этом случае каждый объект класса `Derive1_2` будет иметь два подобъекта класса `Base`. Чтобы подчеркнуть это, используется ключевое слово `virtual`, например:

```
class Derive1 : virtual public Base {...};  
class Derive2 : virtual public Base {...};  
class Derive1_2 : public Derive1, public Derive2 {...};
```

Теперь `Base` является виртуальным базовым классом, а класс `Derive1_2` имеет только один подобъект класса `Base`.

3.5. Другие возможности языка Си ++

Спецификатор `inline`. Спецификатор `inline` сообщает компилятору о возможности встраиваемых расширений. По аналогии со спецификатором `register` компилятор может учесть либо нет это пожелание. Встраиваемые расширения позволяют сократить затраты времени на вызов функции. С этой целью на место вызова вставляется непосредственно скомпилированный код. Подобные действия обоснованы для небольших по размеру и часто вызываемых функций.

Существуют два возможных пути создания функций типа `inline`. Первый из них использует спецификатор `inline` в следующей форме:

`inline` объявление_функции

Рассмотрим простой пример.

```
/* пример EX3_21 */  
#include <iostream.h>  
inline void function(char *str)  
{ cout << str; }  
void main(void)  
{ function("пример встраиваемой функции\n"); }
```

Результаты выполнения этой программы представляются в виде

пример встраиваемой функции

Второй путь создания функций типа `inline` — это включение описания тела функции в шаблон класса. Это неоднократно производилось (например, для функции `put_circle` в примере EX3_11). Любая функция, определенная внутри класса, автоматически получает спецификатор `inline`, причем ключевое слово `inline` записывать не надо. Подобные действия целесообразны лишь для очень простых функций, код которых занимает небольшую область памяти. В противном случае может существенно увеличиться объем программы.

Возникают ситуации, когда компилятор не генерирует встроенный код:

1) при наличии циклов, операторов `switch` и `goto` в функциях, возвращающих значения;

2) для функций, не возвращающих значения, в которых встречается оператор `return`;

3) для рекурсивных функций и функций, содержащих статические переменные.

Операторы `new` и `delete`. Рассмотрим подробнее операторы языка Си++ для динамического выделения (`new`) и освобождения (`delete`) памяти. Сначала уточним общую форму их использования:

```
указатель_на_выделенную_память = new имя (инициализирующие значения);
```

```
delete указатель_на_выделенную_память;
```

Имя может быть любого типа, кроме функции, однако указатели на функцию допустимы. Инициализирующие значения являются не обязательными.

Оператор `new` выделяет требуемую память и возвращает указатель на ее начало. При ошибке и в случае невозможности выделения памяти возвращается `NULL`. Оператор `delete` освобождает ранее выделенную память по заданному указателю на ее начало.

Операторы `new` и `delete` имеют следующие преимущества по сравнению с библиотечными функциями языка Си (типа `malloc` и `free`):

1. Оператор `new` автоматически вычисляет размер элемента, для которого выделяется память (здесь нет необходимости использовать оператор типа `sizeof`).

2. Отсутствует необходимость в преобразовании типов, поскольку `new` автоматически возвращает указатель на нужный элемент.

3. Допускается (при определенных ограничениях) инициализация выделенного блока памяти.

4. Возможна перегрузка операторов `new` и `delete` относительно заданного класса.

Оператор `template`. Слово `template` переводится на русский язык как шаблон (такое же значение имеет слово `templet`). Оператор `template` позволяет задавать шаблоны для классов (можно сказать, что он позволяет определять классы, содержащие компоненты различных типов). Такая возможность достигается параметризацией класса.

Предположим, что необходимо вычислить минимальное значение двух переменных `x` и `y` — $\min(x, y)$. Тип переменных может быть различный, например `int` и `double`. Для решения рассматриваемой задачи можно создать две перегружаемые функции вида

```
int min(int x, int y)
{ return (x < y)? x:y; }
double min(double x, double y)
{ return (x < y) ? x:y; }
```

Если переменные `x` и `y` могут иметь другие типы, необходимо создавать новые перегружаемые версии функции `min`. Это не всегда удобно.

Оператор `template` позволяет задать в виде параметра тип функции `min` и типы ее аргументов. Введем вместо типов `int` и `double` параметр `MY` (можно использовать так же любое другое имя). Тогда вместо двух рассмотренных версий можно записать одну:

```
MY min(MY x, MY y)
{ return (x < y) ? x:y; }
```

Задать параметр `MY` можно с помощью оператора `template` в следующем виде:

```
template < class MY >
MY min(MY x, MY y)
{ return (x < y) ? x:y; }
```

Теперь вместо слова `MY` можно задать любой допустимый тип данных, например:

```
int i = min(10, 20);
long j = min(123456L, 987654L);
double f = min(123.456, 34.1234);
```

Рассмотрим пример программы, в которой выполняются подобные действия.

```
/* пример EX3_22 */
#include <iostream.h>
template <class MY>
MY min(MY x, MY y)
```

```

    { return (x<y) ? x:y; };
void main(void)
{ int i = min(10,20);
  long j = min(123456L,987654L);
  double f = min(123.456,34.1234);
  cout << "int_i = " << i << "\nlong_j = " << j <<
    "\ndouble_f = " << f << "\n";
}

```

Результаты этой программы представляются в виде

```

int_i = 10
long_j = 123456
double_f = 34.1234

```

По аналогии можно параметризовать типы различных компонентов в классе. Предположим, что задан класс с именем `my_class`. В этом классе с атрибутом `private` определены компоненты `a`, `b` и `array[2]`. Типы этих компонентов могут быть разными. Тогда можно написать следующую программу:

```

/* пример EX3_23 */
#include <iostream.h>
template <class M>
class my_class
{ M a,b,array[2];
public:
    my_class(M A,M B,M C,M D) {
        a = A; b = B; array[0] = C, array[1] = D; }
    void show(void) {
        cout << "a = " << a << "\nb = " << b <<
            "\narray[0] = " << array[0] << "\narray[1] = "
            << array[1] << "\n"; }
};
void main(void)
{ my_class<int> int_var(10,20,30,40);
  my_class<double> double_var(100000.5,200000.6,
                              300000.7,400000.8);
  my_class<char> char_var('A','B','B','Γ');
  int_var.show();
  double_var.show();
  char_var.show();
}

```

Результаты работы этой программы представляются в виде

```

a = 10
b = 20
array[0] = 30
array[1] = 40
a = 100000.5
b = 200000.6
array[0] = 300000.7
array[1] = 400000.8
a = A
b = B
array[0] = B
array[1] = Γ

```

В класс можно также передать некоторые параметры, которые задаются в списке после оператора `template`. Рассмотрим пример:

```
/* пример EX3_24 */
#include <iostream.h>
template <class M, int x = 1, y = 50>
class my_class
{ M a,b,array[2];
public:
    my_class(M A,M B,M C,M D) {
        a = A; b = B; array[0] = C; array[1] = D; }
    void show(void) {
        cout << "a = " << a << "\nb = " << b <<
            "\narray[0] = " << array[0] << "\narray[1] = "
            << array[1] << "\nx = " << x <<
            "\ny = " << y << "\n"; }
};
void main(void)
{ my_class<int> int_var(10,20,30,40);
  my_class<double,5> double_var(100000.5,200000.6,
                                300000.7,400000.8);
  my_class<char,7,12> char_var('A','B','C','D');
  int_var.show();
  double_var.show();
  char_var.show();
}
```

В дополнение к предыдущим результатам после каждой выдачи значений `a`, `b`, `array[0]`, `array[1]` будут отображены следующие пары: `x = 1, y = 50`; `x = 5, y = 50`; `x = 7, y = 12`.

Новые возможности операции &. Рассмотрим новые возможности операции `&` в языке Си++. С этой целью приведем пример следующих объявлений:

```
int i; // объявление переменной i целого типа
int &j=i; // объявление нового имени j для i
j=10; // здесь достигается тот же эффект, как и для i=10
```

Здесь элемент `j` не является адресом. Существуют некоторые ограничения использования операции `&`:

- 1) если назвать операцию `&` как ссылка, то нельзя выполнять ссылку на ссылку;
- 2) нельзя задавать ссылки на битовые поля;
- 3) нельзя создавать массивы ссылок;
- 4) невозможно создать указатель на ссылку.

Ссылку можно использовать для передачи параметров в функцию (об этом уже говорилось выше). Рассмотрим пример:

```
void function(long &my_l);
```

Здесь аргумент `my_1` будет передан через ссылку. В результате он может быть изменен в самой функции (после передачи управления вызывающей программе `my_1` будет иметь измененное значение).

Организация ввода/вывода. Рассмотрим некоторые новые возможности языка Си++ по организации ввода / вывода. Заметим, что все старые функции аналогичного назначения, применяемые в языке Си, сохраняют силу.

Вспомним про использование операций `<<` и `>>`. Замечательным свойством языка Си++ является то, что пользователь может определить собственные типы данных, на которые будет распространяться ввод/вывод. Например, допускается определение некоторого класса и с помощью знаков `<<` и `>>` ввод и вывод всех требуемых компонентов соответствующего объекта.

Язык Си++, так же как и Си, работает с потоками данных. С началом выполнения каждой программы автоматически открываются четыре потока: `cin` (соответствует `stdin`), `cout` (соответствует `stdout`), `cerr` (соответствует `stderr`) и `clog`. Первые два из них связаны со стандартным вводом и выводом. Они уже неоднократно рассматривались выше. Потоки `cerr` и `clog` связаны со стандартным выводом. Разница между ними состоит в том, что `clog` предварительно буферизируется, а `cerr` — нет (для `clog` вывод произойдет лишь тогда, когда буфер будет заполнен).

Как и для языка Си, по умолчанию стандартные потоки связаны с клавиатурой и дисплеем. Типы операндов, пересылаемых в стандартный выходной поток и получаемых из стандартного входного потока, могут быть: `char` (signed и unsigned); `short` (signed и unsigned); `int` (signed и unsigned); `long` (signed и unsigned); `char*`; `float`; `double`; `long double`; `void*`. В последнем случае указатель выводится на экран дисплея в шестнадцатеричном формате.

Все определения для рассмотренных потоков помещены в файл `iostream.h`. Укажем некоторые полезные классы этого файла. Первый из них `istream` можно использовать для создания входных потоков; второй, `ostream` — для создания выходных потоков; третий, `iostream` — для создания одновременно входных и выходных потоков. Все они являются производными от базового класса `ios`, который включает функции для поддержки форматируемого ввода/вывода.

При использовании потоков ввода/вывода языка Си++ по умолчанию задаются некоторые параметры, такие, как формат, система счисления и т. п. Их можно изменить, используя специальные операторы, называемые *манипуляторами* (*manipulator*) (некоторые манипуляторы приведены в табл. 3.1). Соответствующие описания помещены в файл `omanip.h`,

который должен быть включен в программу директивой `#include`.
Рассмотрим пример использования манипуляторов.

```
/* пример EX3_25 */
#include <iostream.h>
#include <iomanip.h> // файл с описанием манипуляторов
void main(void)
{ /* вывод числа 20 в десятичной, шестнадцатеричной
   и восьмеричной системах счисления */
  cout << dec << 20 << endl << hex << 20 << endl <<
    oct << 20 << endl;
  /* использование новых символов заполнителей вместо
   пробела (++) и установка ширины поля вывода */
  cout << dec << setw(10) << setfill('+') << 10 << endl <<
    setw(20) << setfill('-') << 10 << endl;
  // вывод двух цифр после точки
  cout << setprecision(2) << 106.2865 << endl;
  // вывод трех цифр после точки
  cout << setprecision(3) << 372.3827 << endl;
}
```

Результаты выполнения программы представятся в виде

```
20
14
24
++++++10
-----10
106.29
372.383
```

Для получения того же результата в Visual C++ необходимо задать манипуляторы `setprecision(5)` и `setprecision(6)` соответственно.

Таблица 3.1

Манипулятор	Способ описания	Выполняемое действие
dec	outs << dec ins >> dec	Задаёт вывод чисел в десятичном формате
hex	outs << hex ins >> hex	Задаёт вывод чисел в шестнадцатеричном формате
oct	outs << oct ins >> oct	Задаёт вывод чисел в восьмеричном формате
endl	outs << endl	Задаёт перевод курсора в начало следующей строки
ends	outs << ends	Вставляет заключительный ноль в конец строки
setfill(int)	ins >> setfill(N) outs << setfill(N)	Устанавливает символ-заполнитель N
setprecision(int)	ins >> setprecision(N) outs << setprecision(N)	Устанавливает вывод N цифр после точки в числах с плавающей точкой
setw(int)	ins >> setw(N) outs << setw(N)	Устанавливает ширину поля ввода/вывода, равную N

Директивы препроцессора и макроимена. Рассмотрим новые директивы препроцессора и макроимена, которые могут использоваться в системе программирования Borland C++.

Оператор `defined` используется только с директивами `#if` и `#elif`. Он позволяет установить, определен или не определен некоторый идентификатор. Выражения вида

```
defined (идентификатор)
```

или

```
defined идентификатор
```

принимают единичное значение, если идентификатор определен (т. е. он встречался в командах `#define`), и нулевое значение в противном случае. Таким образом, директивы

```
#if defined AAA  
#ifdef AAA
```

дают один и тот же результат.

Преимуществом оператора `defined` является возможность его использования в выражениях, например:

```
#if !defined (AAA) ,:: defined (BBB)
```

Директива `#pragma` уже рассматривалась в гл. 2. Общая форма ее задания представляется в виде

```
#pragma имя_директивы
```

Borland C++ позволяет использовать следующие имена директив: `argsused`, `exit`, `hdrfile`, `hdrstop`, `inline`, `option`, `saveregs`, `startup`, `warn`, `intrinsic`. Дадим их краткую характеристику.

Директива `#pragma argsused` может быть помещена между функциями и действует только на следующую функцию. Она отменяет вывод предупреждения о том, что некоторая переменная никогда не используется в функции (Parameter name is never used in function `func_name`).

Директивы `#pragma exit` и `#pragma startup` задают функции, которые должны быть вызваны перед выполнением и после завершения программы. Эти директивы задаются в следующей форме:

```
#pragma startup function_name priority  
#pragma exit function_name priority
```

Вызываемая функция должна иметь следующий прототип:

```
void function_name(void);
```

Приоритет (`priority`) является целым числом от 64 до 255 (по умолчанию он равен 100). Меньший номер задает больший приоритет. Функция с именем `function_name` должна быть

описана до того, когда встретится соответствующая строка с директивой `#pragma`.

Директива `hdrfile` позволяет задавать имя специального файла с расширением `SYM` (по умолчанию этот файл имеет имя `TCDEF.SYM`). Директива `hdrstop` исключает использование этого файла, что приводит к уменьшению размера дискового пространства.

Директива `#pragma inline` сообщает компилятору, что программа содержит встроенные ассемблерные коды (см. гл. 4).

Директива `#pragma option` используется для включения опций из командной строки в код вашей программы. Она задается в следующей форме:

```
#pragma option опции
```

Директива `#pragma saveregs` имеет специальное предназначение и иногда бывает нужна для интерфейса с кодами на языке Ассемблера.

Директива `#pragma warn` позволяет управлять выдачей предупреждений компилятора. Для более подробного знакомства с этими директивами необходимо обратиться к фирменным руководствам.

Директива `intrinsic` используется для решения задач оптимизации программы. В этом случае код некоторых библиотечных функций будет встраиваться в программу. В результате размер программы увеличивается, но вызов библиотечных функций будет осуществляться быстрее.

Рассмотрим теперь новые глобальные идентификаторы Borland C++ (макроимена):

`_BCPLUSPLUS_` — определен при компиляции программ на языке Borland C++. Задает версию системы, например, для системы программирования Borland C++ 3.0 имеет значение `0x0300`, для системы программирования Borland C++ 3.1 имеет значение `0x0310`;

`_CDECL_` — определен, если используются соглашения, принятые в языке Си;

`_TINY_`, `_SMALL_`, `_COMPACT_`, `_MEDIUM_`, `_LARGE_`, `_HUGE_` — позволяют установить модель памяти, заданную пользователем в Borland C++. При конкретной компиляции определено только одно из этих макроимен (см. гл. 5);

`_cplusplus` — определен при использовании компилятора языка Си++ (принимает значение 1);

`_DLL_` — определен, если используются динамически связанные библиотеки;

`_MSDOS` — дает целочисленную константу для всех случаев компиляции;

`_OVERLAY` — определен, если включена поддержка оверлейных структур (см. гл. 5);

`_PASCAL` — определен, если используются соглашения, принятые в языке PASCAL;

`_TCPLUSPLUS` — принимает те же значения, что и `_BCPLUSPLUS`;

`_TEMPLATES` — определен, если поддерживается оператор `template`;

`_TURBOC` — дает номер текущей версии Borland C++ в виде шестнадцатеричной константы (для системы программирования Borland C++ 3.1 эта константа будет равна `0x0410`);

`_WINDOWS` — определен, если генерируется код, специфичный для системы Windows.

Ключевые слова языков Си и Си++. В языке Си могут быть использованы следующие ключевые слова: `asm`; `auto`; `break`; `case`; `cdecl`; `char`; `const`; `continue`; `_cs`; `default`; `do`; `double`; `_ds`; `else`; `enum`; `_es`; `_export`; `extern`; `far`; `_fastcall`; `float`; `for`; `goto`; `huge`; `if`; `int`; `interrupt`; `_loadds`; `long`; `near`; `pascal`; `register`; `return`; `_saveregs`; `_seg`; `short`; `signed`; `sizeof`; `_ss`; `static`; `struct`; `switch`; `typedef`; `union`; `unsigned`; `void`; `volatile`; `while`.

Слова `cdecl`, `_cs`, `_ds`, `_es`, `_export`, `far`, `_fastcall`, `huge`, `interrupt`, `_loadds`, `near`, `pascal`, `_saveregs`, `_seg`, `_ss`, отсутствуют в ANSI-стандарте на языке Си.

Кратко охарактеризуем предназначение тех ключевых слов, которые еще не рассмотрены.

Модификаторы переменных и функций `cdecl` и `pascal` устанавливают для определяемого объекта соглашения, соответственно принятые в языках Си и Паскаль (модификатор `pascal` используется в гл. 4).

Модификатор `const` запрещает изменение значения определяемого элемента. Указатель, заданный с модификатором `const`, не может быть изменен. Это не относится к объекту, на который он указывает. Рассмотрим примеры:

```
const float a = 26.2557;  
const var = 159;
```

Модификатор `_export` делает функцию переносимой из среды системы Windows.

Модификатор `_fastcall` предполагает возможность передачи параметров в функцию через внутренние регистры микропроцессора. Компилятор обрабатывает соответствующие вызовы функций не так, как это делается в большинстве языков

программирования. Это, в частности, говорит о том, что вместе с модификатором `_fastcall` нельзя использовать модификаторы `cdecl` и `pascal` (поскольку функции в языках Си и Паскаль используют стек для передачи параметров). Функции, которые определены с модификатором `_fastcall`, имеют другие имена по сравнению с обычными функциями. Компилятор расширяет эти имена префиксом `@`.

Модификатор функции `_lroadds` записывает в регистр DS указатель на текущий сегмент данных. Модификатор функции `_saveregs` позволяет сохранять значения всех регистров и восстанавливать их после завершения функции (кроме регистра с возвращаемым значением).

Модификатор переменной `volatile` противоположен `const`. Он указывает, что определяемый объект может быть изменен не только в программе, но и внешним воздействием, таким, например, как прерывание со стороны другой программы.

Использование ключевого слова `asm` подробно рассматривается в гл. 4. Использование ключевых слов `_cs`, `_ds`, `_es`, `_seg`, `far`, `huge`, `near` подробно описано в гл. 5.

Перечислим теперь ключевые слова языка Си++: `class`, `delete`, `friend`, `inline`, `new`, `operator`, `private`, `protected`, `public`, `template`, `this`, `virtual`.

Оговорим приоритет выполнения новых операций языка Си++:

`::` — выполняется слева направо и включается в группу 1 (высший приоритет);

`new`, `delete` — выполняются справа налево и включаются в группу 2;

`*, &, ->*` — выполняются слева направо и образуют новую группу 3.

В заключение укажем о наличии в системе программирования Borland C++ двух полезных стандартных включаемых файлов. Первый из них — `complex.h` предоставляет доступ к множеству библиотечных функций для работы с комплексными числами. Второй — `bcd.h` дает возможность использования двоично-десятичной арифметики при реализации различных вычислений.

ГЛАВА 4

ИНТЕРФЕЙС ЯЗЫКА Си++ С ЯЗЫКОМ АССЕМБЛЕРА

4.1. Вызов подпрограмм и передача параметров в языке Си++

Язык Borland C++ поддерживает два способа передачи параметров в вызываемую функцию:

1. Стандартный способ языка Си.
2. Передача параметров с модификатором `pascal`.

Рассмотрим программу на языке Си++, демонстрирующую первый способ.

```
/* пример EX4_1 */
#include <stdio.h>
extern "C" void asm_f1(int *i1, int *i2, unsigned long l1);
void main(void)
{   int i=5, j=7;
    unsigned long l=0x12345678;
    asm_f1(&i, &j, l);
}
```

Строка

```
extern "C" void asm_f1 (int *i1, int *i2, unsigned long l1);
```

определяет, что программа `asm_f1` описана в другом файле. Особенностью компилятора языка Си++ является то, что если не заданы специальные режимы, он добавляет перед своими идентификаторами символ подчеркивания (`_`). Таким образом, идентификатор `asm_f1` после компиляции будет выглядеть так: `_asm_f1`. Эта особенность будет учитываться далее при рассмотрении процедуры `asm_f1` на языке Ассемблера. Спецификатор "C" говорит о том, что используется интерфейс между программами на разных языках, принятый в языке Си (а не Си++). Строки

```
int i = 5, j = 7;
unsigned long l = 0x12345678;
```

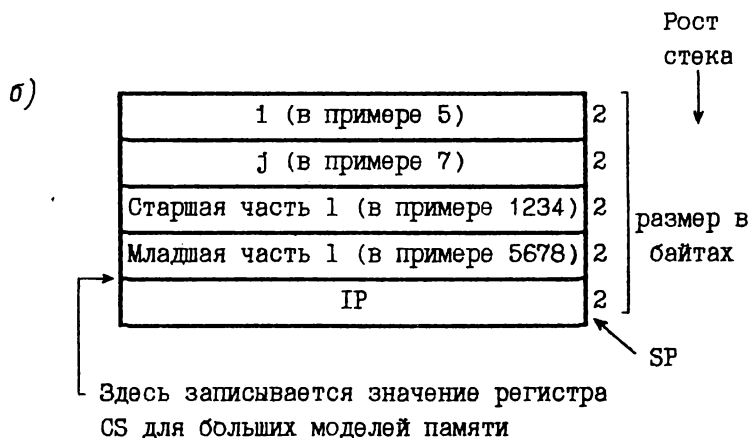
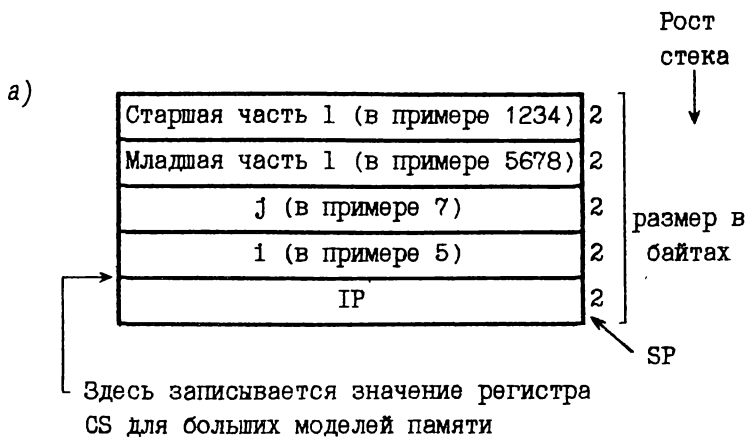


Рис. 4.1. Размещение в стеке параметров, передаваемых в программу. `asm_f1` для способа 1(а) и способа 2(б)

объявляют и инициализируют переменные `i` и `j` целого типа и переменную `i` беззнакового длинного целого типа. После этого `i = 5`, `j = 7`, `i = 1234567816`. Далее вызывается функция `asm_f1` на языке Ассемблера и ей передаются три параметра: `i`, `j` и `i`. Передача параметров из вызывающей в вызываемую программу осуществляется через стек. В соответствии с первым способом параметры передаются в стек справа налево, т. е. сначала `i`, потом `&j` и далее `&i`. Следом за параметрами в стек помещается адрес возврата, т. е. адрес команды в языке Си++, которая следует за командой `asm_f1`

(&i, ... &j, i);. Если для программы на языке Си++ заданы малые модели памяти (tiny, small, compact), то таким адресом возврата является значение IP (для следующей после asm_f1 команды). Если для программы на языке Си++ заданы большие модели памяти (medium, large, huge), то адресом возврата является значение CS:IP (для следующей после asm_f1 команды). Заполнение стека для малых моделей памяти показано на рис. 4.1, а, а для больших — на рис. 4.1, б. Здесь следует учесть, что для процессора 8086 данные в стек заносятся только словами (по 2 байт). Переменные i и j требуют 2 байт памяти. Их адреса &i и &j (именно они заданы в программе) для малых моделей данных (tiny small, medium) тоже требуют 2 байт. Поэтому каждое значение &i и &j будет занесено в стек одной командой push микропроцессора (МП) 8086. Переменная l требует 4 байт памяти и заносится в стек двумя командами push МП 8086.

З а м е ч а н и е: малыми моделями памяти для кода являются tiny, small и compact, а для данных — tiny, small и medium.

Проанализируем выполняемый код, который будет построен компилятором языка Си++ для этой программы. Для этого опишем полученный выполняемый код на языке Ассемблера (используемые здесь и далее команды языка Ассемблера описаны в приложении).

```

push bp; сохранение регистра bp в стеке
; язык Си++ сохраняет все локальные переменные в стеке (в нашем
; примере ими являются i, j и l). В результате после инструкций
; int i=5, j=7; unsigned long l=0x12345678; в область стека
; будут записаны следующие значения
mov bp, sp ; запись в регистр bp адреса вершины стека
sub sp,8 ; выделение в стеке места для локальных переменных
mov word ptr [bp-2], 5 ; сохранение в стеке значения i=5
mov word ptr [bp-4], 7 ; сохранение в стеке значения j=7
mov word ptr [bp-8], 5678H ; сохранение в стеке младшей части
; значения l=0x12345678 (т. е. 5678)
mov word ptr [bp-6], 1234H ; сохранение в стеке старшей части
; значения l=0x12345678 (т. е. 1234)
; после инструкции asm_f1 (&i, &j, i); командами push в стек будут
; занесены передаваемые параметры (это делается в следующем
; фрагменте программы)
push word ptr [bp-6] ; значение 1234 в стек
push word ptr [bp-8] ; значение 5678 в стек
lea ax, [bp-4] ; адрес [bp-4] в ax
push ax ; адрес [bp-4] в стек
lea ax, [bp-2] ; адрес [bp-2] в ax
push ax ; адрес [bp-2] в стек
call _asm_f1 ; вызов функции _asm_f1
add sp,8 ; удаление из стека переданных параметров
mov sp,bp ; удаление из стека локальных переменных
pop bp ; восстановление регистра bp из стека

```

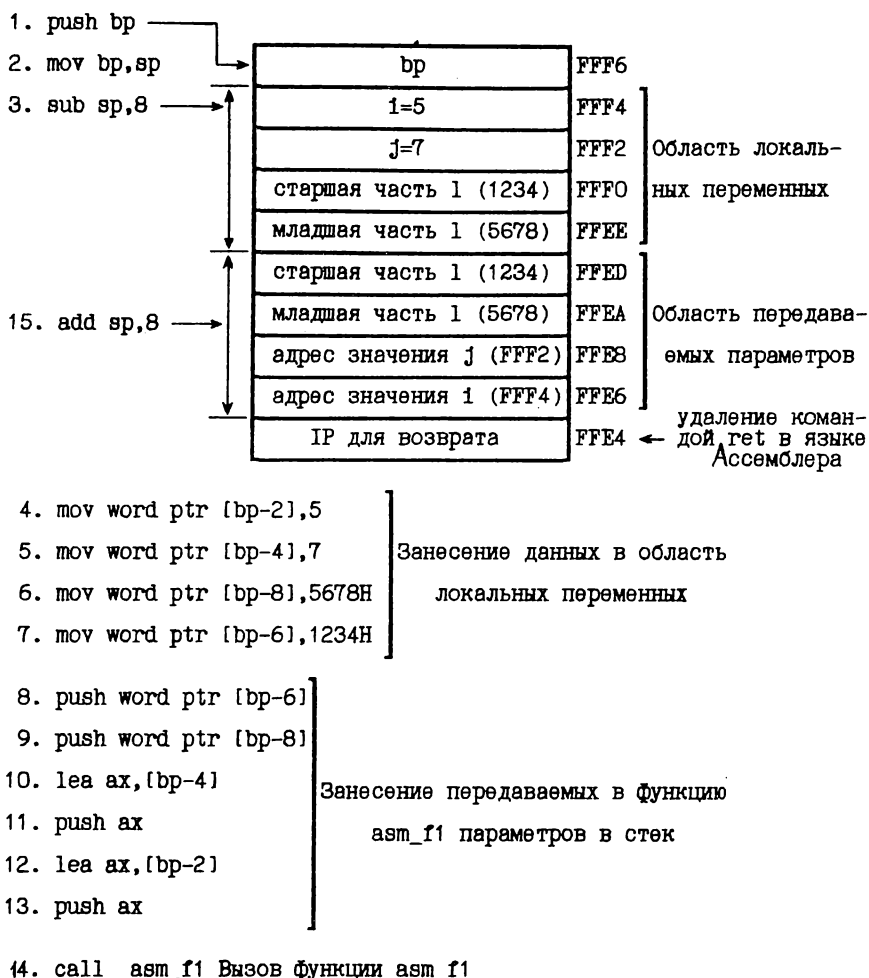



Рис. 4.2. Пример заполнения стека программой на языке Си++

Рис. 4.2 показывает заполнение стека рассматриваемой программой. В нем выделены области локальных переменных и передаваемых параметров. Предположим, что программа использует модель памяти small, а также то, что она скомпилировала и компоновала с соответствующей процедурой на языке Ассемблера (как это сделать, будет показано ниже). Допустим, что после запуска программы на выполнение в регистре SP будет значение FFF8. С учетом этого значения (Sp = FFF8) на рис. 4.2 указаны конкретные адреса слов стека

и цифрами помечена последовательность выполнения ассемблерных команд.

Рассмотрим второй способ передачи параметров, поддерживаемой языком Си++. При этом вызываемая программа объявляется со спецификатором `pascal`. Рассмотрим пример:

```
/* пример EX4_2 */
#include <stdio.h>
extern "C" void pascal asm_f3(int *i1,int *i2,unsigned long l1);
void main(void)
{   int i=5,j=7;
    unsigned long l=0x12345678;
    asm_f3(&i,&j,l);
}
```

Модификатор `pascal` задает паскалевскую передачу параметров в программу `asm_f3`. В результате параметры будут передаваться в стек слева направо, т. е. сначала `&i`, потом `&j` и далее `i` (см. рис. 4.1, б).

Проанализируем выполняемый код, который будет построен компилятором языка Си++ для этой программы. Для этого опишем полученный выполняемый код на языке Ассемблера.

```
push bp      ; сохранение регистра bp в стеке
mov bp,sp    ; запись в регистр bp адреса вершины стека
sib sp,8     ; выделение в стеке места для локальных переменных
mov word ptr [bp-2],5 ; сохранение в стеке значения i=5
mov word ptr [bp-8],7 ; сохранение в стеке значения j=7
mov word ptr [bp-8],5678H ; сохранение в стеке младшей части
                        ; значения l=0x12345678 (т. е. 5678)
mov word ptr [bp-6],1234H ; сохранение в стеке старшей части
                        ; значения l=0x12345678 (т. е. 1234)
lea ax,[bp-2] ; адрес [bp-2] в ax
push ax      ; адрес [bp-2] в стек
lea ax,[bp-4] ; адрес [bp-4] в ax
push ax      ; адрес [bp-4] в стек
push word ptr [bp-6]
push word ptr [bp-8]
call ASM_F3 ; вызов функции asm_f3
процедура ASM_F3 должна удалить параметры из стека командой ret 8
mov sp, bp ; удаление из стека локальных переменных
pop bp     ; восстановление регистра bp из стека
```

В примере без модификатора `pascal` вызываемая функция (`asm_f1`) не знает, сколько параметров она должна получить из стека, поскольку в языке Си++ допускается функция с переменным числом параметров. Например, для функции `printf` первым параметром является управляющая строка и она определяет, сколько всего параметров в функции. Об этом знает только вызывающая программа, поскольку она передает эти параметры. Вызываемая функция не знает,

сколько параметров передано, поскольку при разных вызовах число параметров может быть разным. В результате только вызывающая функция может удалить из стека нужное число параметров и в ней появляется строка вида

```
add sp,<число байт в стеке под параметры>
```

Функции с модификатором `pascal` не могут иметь переменного числа параметров. В результате вызываемая функция всегда знает, сколько параметров передается. Поэтому именно она отвечает за то, чтобы удалить их из стека командой вида

```
ret<число байт в стеке под параметры>
```

Если сравнить варианты ассемблерного кода для первой и второй программ, то можно увидеть имеющиеся отличия. Еще одним отличием является то, что перед идентификаторами, заданными с модификатором `pascal`, не добавляется символ подчеркивания (`_`). Об этом еще будет говориться ниже.

4.2. Вызов ассемблерных программ из программ на языке Си++

При соединении программ на языках Си++ и Ассемблера эти программы должны использовать одну и ту же модель памяти. Кроме того, для каждой программы необходимо указать, какие переменные она передает в другую программу и какие переменные она получает из другой программы. С этой целью необходимо выполнить следующие требования:

1. Если процедура `MY_ASM` на языке Ассемблера вызывается из программы на языке Си++, то такая процедура в языке Ассемблера должна быть описана как `PUBLIC` (должна появиться после директивы `PUBLIC`). Аналогичное описание необходимо и в том случае, когда две процедуры на языке Ассемблера компилируются по отдельности. В этом случае в вызываемой процедуре должна появиться директива `PUBLIC` с именем этой процедуры. Директива `PUBLIC` записывается в следующей форме:

```
PUBLIC [язык] идентификатор, [язык] идентификатор...
```

Здесь вместо слова «язык» можно записать: `C`, `PASCAL`, `BASIC`, `FORTTRAN`, `ASSEMBLER` или `PROLOG`. Этот параметр является не обязательным.

2. Предположим, что некоторые переменные должны быть глобальными, т. е. к ним должен быть доступ из объединя-

емых программ на языках Си++ и Ассемблера. Возможны такие случаи:

- глобальная переменная объявлена в программе на языке Ассемблера. В этом случае в программе на языке Ассемблера она должна иметь атрибут PUBLIC, а в программе на языке Си++ — extern;

- глобальная переменная объявлена в программе на языке Си++ (в этом случае в программе на языке Си++ она объявлена как внешняя). Тогда в программе на языке Ассемблера эта переменная должна иметь атрибут EXTRN. Директива EXTRN записывается в следующей форме:

EXTRN [язык] идентификатор:тип [:количество определений]...

Здесь вместо слова «язык» можно записать: C, PASCAL, BASIC, FORTRAN, ASSEMBLER или PROLOG. Этот параметр является не обязательным. Вместо слова «тип» можно записать: NEAR, FAR, PROC, BYTE, WORD, DWORD, DATAPTR, CODEPTR, FWORD, QWORD, TBYTE, ABS. Фактически здесь определяется тип данного или указателя (адреса). Последний параметр задает количество определений идентификатора и является не обязательным. Дадим краткую характеристику операторам определения типа:

ABS — данное имеет абсолютное значение;

BYTE — данное имеет размер 1 байт;

CODEPTR — определяет адрес процедуры, заданный по умолчанию;

DATAPTR — ближний или дальний указатель на данные в зависимости от текущей модели памяти;

DWORD — данное имеет размер 4 байт;

FAR — определяет дальний указатель на код;

FWORD — данное имеет размер 6 байт;

NEAR — определяет ближний указатель на код;

PROC — метка процедуры (ближняя или дальняя в зависимости от модели);

QWORD — данное имеет размер 8 байт;

TBYTE — данное имеет размер 10 байт;

WORD — данное имеет размер 2 байт.

Если некоторый идентификатор в языке Ассемблера должен одновременно иметь атрибуты PUBLIC и EXTRN, то он описывается как GLOBAL. Директива GLOBAL записывается в следующей форме:

GLOBAL [язык] идентификатор:тип [:количество определений] ...

Она имеет те же поля, что и директива EXTRN.

Рассмотренные выше правила определения в различных программах внешних переменных, функций и процедур поясняются дополнительно на рис. 4.3. На рис. 4.3 процедура my_asm на языке Ассемблера вызывается из программы на языке Си++. В языке Ассемблера перед идентификатором my_asm появился символ подчеркивания (_my_asm). Это

Вызывающая программа
на языке Си++

Вызываемая программа
на языке Ассемблера
Имя MY_ASM (может быть любым)

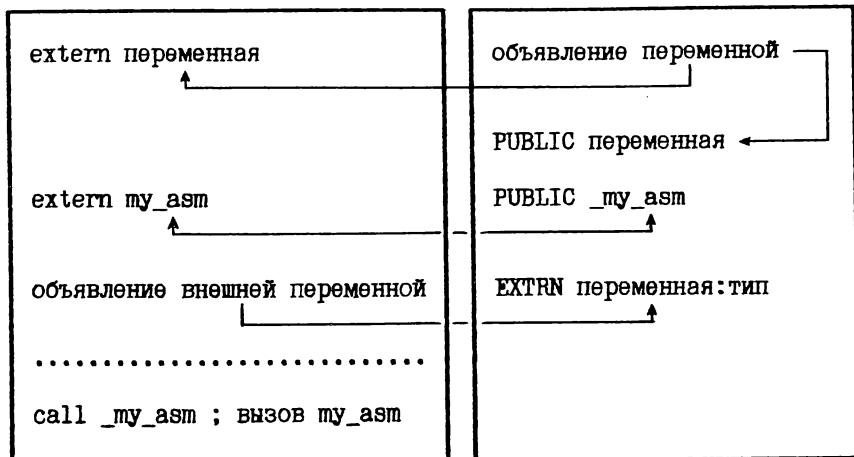


Рис. 4.3. Вызов программы на языке Ассемблера из программы на языке Си++

определяется тем, что компилятор Си++ автоматически добавляет символ подчеркивания перед всеми своими идентификаторами. Этот символ можно не добавлять, если в языке Ассемблера явно указать, что используется идентификатор языка Си++. Такие указания делаются в директивах PUBLIC и EXTRN, например:

```
PUBLIC C my_asm
```

После этого перед идентификатором my_asm символ подчеркивания добавлять не надо.

Ниже приводится пример двух программ. Первая из них написана на языке Си++ и находится в файле C_ASM1.CPP, а вторая написана на языке Ассемблера и находится в файле ASM1.ASM. Вторая программа вызывается из первой.

Рис. 4.4 поясняет чтение переданных параметров из стека в программе на языке Ассемблера (на рис. 4.4 показаны все

```

/* пример_CPP EX4_3 */
#include <stdio.h>
extern "C" void asm_f1(int *i1,int *i2,unsigned long l1);
void main(void)
{
    int i=5,j=7;
    unsigned long l=0x12345678;
    printf("i = %d; j = %d; l = %lx\n",i,j,l);
    asm_f1(&i,&j,l);
    printf("i = %d; j = %d; l = %lx\n",i,j,l);
}
  
```

```

; пример_ASM EX4_3
.MODEL SMALL
.CODE
PUBLIC C asm_f1
asm_f1 PROC near
    push bp
    mov bp,sp
        push si
        push di
    mov si,[bp+4]
    mov di,[bp+6]
    mov bx,[bp+8] ; здесь показано, как передается 4-
    mov ax,[bp+10] ; байтное значение. Далее оно не
                    ; используется
        mov cx,[si]
        xchg cx,[di]
        mov [si],cx
        pop di ; обратить внимание
        pop si ; обратить внимание
    pop bp
    ret
asm_f1 ENDP
end

```

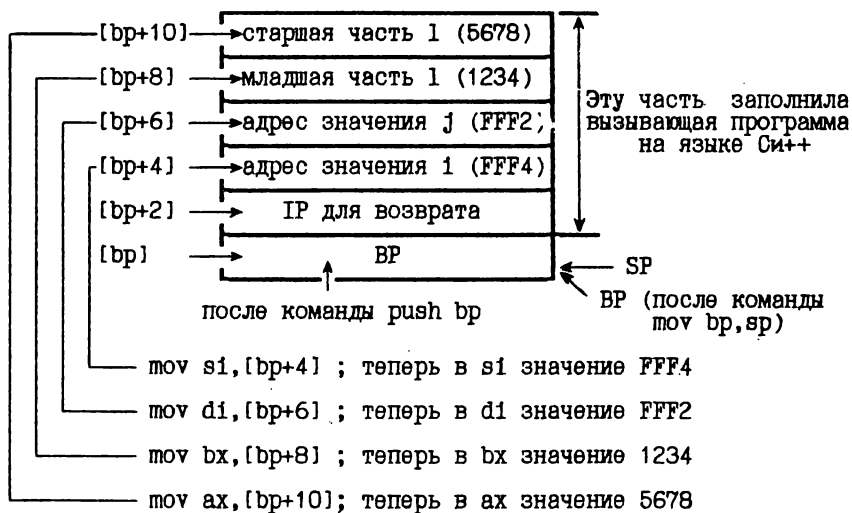


Рис. 4.4. Чтение переданных параметров в программе на языке Ассемблера

необходимые команды этой программы для чтения параметров из стека).

После значения регистра bp в стек дополнительно занесены значения регистров si и di. Это необходимо делать в связи с тем, что компилятор Си++ может использовать регистры si и di под переменные типа register (регистрового типа). При вызове ассемблерной процедуры в этих регистрах могут быть некоторые значения и они должны сохраниться

после завершения ассемблерной процедуры. Таким образом, при вызове ассемблерной процедуры необходимо сохранять значения в регистрах `bp`, `si` и `di`. Если некоторые из этих регистров не используются, то сохранять их в стеке не надо. Кроме того, можно компилировать программу на языке Си++ с опцией `-r`, запрещающей использование регистровых переменных. В этом случае сохранять регистры `si` и `di` не надо.

Рассмотрим теперь, как в системе программирования Borland C++ создать исполняемый файл, объединяющий программы на языке Си++ и Ассемблера. Для этого необходимо выполнить следующие действия:

1. С помощью встроенного редактора системы программирования Borland C++ (файл `TC.EXE`) создать файл `C_ASM.CPP`.

2. С помощью встроенного редактора системы программирования Borland C++ создать файл `ASM1.ASM`.

3. Откомпилировать файл `ASM1.ASM`. Для этого в системе программирования Borland C++ выполнить последовательность команд: \equiv — Turbo Assembler. Подменю \equiv выбирается нажатием клавиш «Alt» — <пробел>. В результате будет построен объектный файл `ASM1.OBJ`. Все перечисленные действия будут выполнены после нажатия клавиш «Shift—F3».

4. Создать файл проекта, например, с именем `MC1.PRJ`. Для этого в системе программирования Borland C++ выполнить последовательность команд: Project — Open project. В появившееся после этого окно необходимо вписать имя файла (`MC1.PRJ`). Затем включить в файл `MC1.PRJ` имена соединяемых программ (`C_ASM1.CPP` и `ASM1.OBJ`). Включение осуществляется нажатием клавиши «Ins» и запись соответствующего имени в появившееся окно. Заметим, что в файл проекта можно включить и имя исходного модуля `ASM1.ASM`. Однако в этом случае с помощью команды Local options для файла `ASM1.ASM` необходимо задать компилятор ассемблер (об этом будет говориться в гл. 5).

5. Выполнить компиляцию и компоновку файла `MC1.PRJ`. В результате будет получен исполняемый файл `MC1.EXE`, объединяющий программы на языках Си++ и Ассемблера. Для построения файла `MC1.EXE` и запуска его на исполнение достаточно нажать клавиши «Ctrl — F9».

В системе программирования Borland C++ можно производить отладку файла `MC1.EXE` с помощью встроенного или внешнего отладчика. Внешний отладчик `TD.EXE` (Turbo Debugger) включается последовательностью команд: \equiv — Turbo Debugger (можно просто нажать клавиши «Shift—F4»). Выход из отладчика будет осуществлен после нажатия клавиш «Alt—X».

Рассмотрим пример объединяемых программ на языках Си++ и Ассемблера, которые используют паскалевский способ передачи параметров. Первая из них написана на языке Си++ и имеет имя C_ASM3.CPP, а вторая написана на языке Ассемблера и имеет имя ASM3.ASM.

Здесь передаваемые параметры помещены в стек слева направо и это учитывается при их выборе в программе на

```

/* пример_CPP EX4_4 */
#include <stdio.h>
extern "C" void pascal asm_f3(int *i1,int *i2,unsigned long l1);
void main(void)
{   int i=5,j=7;
    unsigned long l=0x12345678;
    printf("i = %d; j = %d; l = %lx\n",i,j,l);
    asm_f3(&i,&j,l);
    printf("i = %d; j = %d; l = %lx\n",i,j,l);
}

; пример_ASM EX4_4
.MODEL SMALL
.CODE
PUBLIC  ASM_F3
ASM_F3  PROC near
        push bp
        mov bp,sp
                push si
                push di
        mov bx,[bp+4] ; здесь показано, как передается 4-
        mov ax,[bp+6] ; байтное значение. Далее оно не
                        ; используется
        mov si,[bp+8]
        mov di,[bp+10]
                mov cx,[si]
                xchg cx,[di]
                mov [si],cx
                pop di
                pop si

        pop bp
        ret 8
ASM_F3  ENDP
end

```

языке Ассемблера. Все остальное, что необходимо для получения выполняемого модуля, объединяющего эти программы, было описано выше.

Рассмотрим вызов ассемблерных процедур, возвращающих значения в программу на языке Си++. Здесь используются следующие правила, которые учитывают особенность получения программой на языке Си++ возвращаемых значений:

1. Если ассемблерная процедура возвращает в программу на языке Си++ 16-битовое значение, то оно должно быть помещено в регистр AX. Для языка Си++ это значения данных типов unsigned char, char, unsigned, short, short, unsigned int, int, enum и значения указателей типа near.

2. Если ассемблерная процедура возвращает в программу на языке Си++ 32-битовое значение, то оно должно быть помещено в регистры DX:AX. Для языка Си++ это значения данных типа unsigned long, long и значения указателей типа far.

3. Значения, которые в языке Си++ имеют типы float, double и long double, возвращаются на вершину стека сопроцессора 80 x 87 (регистр сопроцессора ST(0)). Вызывающая функция (программа на языке Си++) может перенести возвращенные значения в нужное место.

4. Структура длиной 1 байт возвращается в регистр AL. Структура длиной 2 байт возвращается в регистр AX. Структура длиной 4 байт возвращается в регистры AX:DX. Структура длиной 3 байт или больше, чем 5 байт, возвращается в виде области оперативной памяти (статической области оперативной памяти). Указатель на эту область передается через регистр AX для малых моделей данных и через регистры DX:AX для больших моделей данных.

Табл. 4.1 дает соответствие между типами данных в языках Ассемблера и Си++.

Таблица 4.1

Тип данных в языке Си++	Тип данных в языке Ассемблера	Тип данных в языке Си++	Тип данных в языке Ассемблера
unsigned char	byte	unsigned long	dword
char	byte	long	dword
enum	word	float	dword
unsigned short	word	double	qword
short	word	long double	tbyte
unsigned int	word	near*	word
int	word	far*	dword

Рассмотрим пример трех программ. Первая из них написана на языке Си++ и имеет имя C_ASM4.CPP. Две другие написаны на языке Ассемблера и имеют имена ASM4_1.ASM и ASM4_2.ASM. Программа на языке Си++ вызывает ассемблерные программы, которые соответственно возвращают значения типа int и типа long.

Первая функция printf в программе на языке Си++ вызывает процедуру asm_f4_1 на языке Ассемблера и

```
/* пример_CPP EX4_5 */
#include <stdio.h>
extern "C" asm_f4_1(int i1,int i2);
extern "C" long asm_f4_2();
extern a=1000,b=2000;
void main(void)
{ printf("i(10) + j(20) = %d\n",asm_f4_1(10,20));
```

```

// если возвращаемое значение int, то оно в AX
    printf("a(1000) * b(2000) = %ld\n",asm_f4_2());
// если возвращаемое значение long, то оно в DX:AX
}

; пример_ASM EX4_5_1
.MODEL SMALL
.CODE
PUBLIC C asm_f4_1
asm_f4_1 PROC near
    push bp
    mov bp,sp
    mov bx,[bp+4]
    mov ax,[bp+6]
    pop bp
    add ax,bx
    ret
asm_f4_1 ENDP
end

; пример_ASM EX4_5_2
.MODEL SMALL
.CODE
PUBLIC C asm_f4_2
EXTRN C a:near,C b:near
asm_f4_2 PROC near
    mov ax,word ptr a
    mul word ptr b
    ret
asm_f4_2 ENDP
end

```

передает ей два целых числа. Эта процедура складывает полученные числа и возвращает результат целого типа в регистр AX. Программа на языке Си++ получает этот результат и выводит его на экран. Вторая функция printf на языке Си++ вызывает процедуру asm_f4_2 на языке Ассемблера без параметров. Процедура asm_f4_2 перемножает глобальные переменные a, b и возвращает результат длинного целого типа в регистрах DX:AX. Программа на языке Си++ получает этот результат и выводит его на экран.

В заключение укажем еще одну особенность компиляции ассемблерных программ, которые объединяются с программами на языке Си++. Если используется компилятор Турбо ассемблера TASM или макроассемблера MASM, то необходимо использовать опцию /mx. В результате компилятор будет различать большие и маленькие буквы в идентификаторах для объектов типа PUBLIC и EXTERNAL, т. е. для переменных и функций, которые передаются из одной программы в другую. Например, компиляцию программы ASM4_1.ASM необходимо производить следующим образом:

Здесь предполагается, что компилятор TASM находится на диске C:.

4.3. Вызов программ на языке Си++ из программ на языке Ассемблера

Как и выше, для каждой программы необходимо указать, какие переменные она передает в другую программу и какие переменные она получает из другой программы. Для этого необходимо выполнить следующие требования:

1. Если функция MY_C на языке Си++ вызывается из программы на языке Ассемблера, то такая функция в языке Ассемблера должна быть описана как EXTRN (должна появиться после директивы EXTRN). Форма задания директивы EXTRN рассматривалась в § 4.2. Если используется модель памяти huge, то директива EXTRN должна быть записана за пределами сегментов.

2. Описание всех глобальных переменных должно быть выполнено в соответствии с правилами, рассмотренными в § 4.2.

3. Если функция на языке Си++ возвращает значение, то оно будет помещено в регистры, указанные в § 4.2.

Ниже приводится пример вызова функции fun на языке Си++ из процедуры на языке Ассемблера. Функции fun передаются два указателя на строки str1 и str2. Функция fun производит их сравнение и выдает результат в виде строки со значением «Строки равны» или «Строки не равны». Указатель на эту строку возвращается в процедуру на языке Ассемблера. Значение полученной строки с помощью функции 9 прерывания 21h выводится на экран дисплея.

Если компоновка программы производится в среде Borland C++, то необходимо создать файл проекта, в который включаются строки с именами объединяемых программ

```
; пример_ASM EX4_6
.MODEL SMALL
.DATA
str1 DB 'Минск',0
str2 DB 'Минск',0
str3 DB 'Киев',0
.CODE
PUBLIC C main
EXTRN C fun:near
main PROC near
    mov ax,DGROUP
    mov ds,ax
    lea ax,str2
    push ax
    lea ax,str1
```

```

        push ax
        call fun
        add sp,4
        mov dx,ax
        mov ah,9
        int 21h
        lea ax,str3
        push ax
        lea ax,str1
        push ax
        call fun
        add sp,4
        mov dx,ax
        mov ah,9
        int 21h
        mov ah,4ch
        int 21h
main     ENDP
        end

/* пример_CPP EX4_6 */
extern "C" unsigned char *fun(unsigned char *str1,
                               unsigned char *str2)
{
    unsigned char *str;
    for(;*str1 == *str2;str1++,str2++)
        if(*str1 == 0)
            return (unsigned char *)"\x0D\x0Aстроки равны$";
    return (unsigned char *)"\x0D\x0Aстроки не равны$";
}

```

(например, ASM7.OBJ и C ASM7.CPP). При этом в соответствии с правилами языка Си++ головная программа должна иметь имя main. Это сделано в процедуре на языке Ассемблера. Если объединение программ осуществляется с помощью автономного компоновщика (например, link или tlink), то головная программа может иметь любое имя. Компоновщик вызывается, например, так:

```
C > link asm7 c_asm7 < BВOD >
```

Предварительно должны быть получены файлы asm7.obj и c_asm7.obj. Второй объектный файл строится с помощью программ TC.EXE или TCC.EXE.

4.4. Вызов библиотечных функций языка Си++ из программ на языке Ассемблера

Из программ на языке Ассемблера можно вызывать стандартные библиотечные функции языка Си++. Все требования, которые необходимо выполнить, уже описывались выше, поэтому ограничимся рассмотрением примера.

Здесь из программы на языке Ассемблера вызываются библиотечные функции языка Си++ puts, printf и scanf. Предполагается, что программа выполняется в среде Borland

```

; пример_ASM EX4_7
.MODEL SMALL
.STACK 300h
.DATA
c DB 13,10,'%s',0
sstr DB 50 dup(?)
s DB 13,10,"Введите строку до 49 символов",0
.CODE
EXTERN _puts:near,_printf:near,_scanf:near
PUBLIC _main
_main PROC NEAR
    mov ax,DGROUP
    mov ds,ax
    lea ax,s
    push ax
    call _puts
    add sp,2
    lea ax,sstr
    push ax
    lea ax,c
    push ax
    call _scanf
    add sp,4
    lea ax,sstr
    push ax
    lea ax,c
    push ax
    call _printf
    add sp,4
    mov ah,4ch
    int 21h
_main ENDP
END

```

C++. Если она имеет имя MYASM.ASM, то предварительно необходимо создать объектный файл MYASM.OBJ. Затем создается и компонуется файл проекта со строкой MYASM.OBJ. После запуска программы на выполнение необходимо ввести строку до 49 символов, которая далее выводится на экран.

Эту же программу можно скомпилировать и скомпоновать с помощью команд следующего вида:

```

C> tasm myasm/mx <ВВОД>
C> tlink myasm\tc\lib\cs.lib\tc\lib\cOs <ВВОД>

```

Здесь указаны файл с вызываемыми библиотечными функциями cs.lib и файл загрузчика cOs.obj. Предполагается, что они находятся в поддиректории /tc/lib. Если используется другая модель памяти, то задаются другие файлы, например для модели MEDIUM — файлы sm.lib и cOm.obj.

4.5. Упрощенные конструкции для компилятора TASM

Турбо-ассемблер TASM позволяет использовать новые конструкции, упрощающие интерфейс между программами на языках Си++ и Ассемблера. В результате можно автоматически создать имена, поддерживающие стиль языка Си++, помещать параметры в стек и очищать стек после вызова функций языка Си++. Покажем, как можно модифицировать рассмотренный выше пример для сравнения строк.

```
; пример_ASM EX4_8
.MODEL SMALL
.DATA
str1 DB 'Минск',0
str2 DB 'Минск',0
str3 DB 'Киев',0
.CODE
PUBLIC C main
EXTRN C fun:near
main PROC near
    mov ax,DGROUP
    mov ds,ax
    lea bx,str2
    lea ax,str1
    call fun C,ax,bx
    mov dx,ax
    mov ah,9
    int 21h
    lea bx,str3
    lea ax,str1
    call fun C,ax,bx
    mov dx,ax
    mov ah,9
    int 21h
    mov ah,4ch
    int 21h
main ENDP
end
```

```
/* пример_CPP EX4_8 */
extern "C" unsigned char *fun(unsigned char *str1,
                               unsigned char *str2)
{
    unsigned char *str;
    for(;*str1 == *str2;str1++,str2++)
        if(*str1 == 0)
            return (unsigned char *)"\x0D\x0AСтроки равны$";
    return (unsigned char *)"\x0D\x0AСтроки не равны$";
}
```

Здесь используются конструкции вида

call функция [язык [,аргумент_1]..]

На место параметра «язык» можно записать C, PASCAL, BASIC, FORTRAN, PROLOC или NONLANGUAGE. На место параметра «аргумент_N» можно поместить любое значение,

которое помещается в стек (аргументы — это значения параметров для функции, передаваемые через стек). Турбо-ассемблер автоматически формирует команды для записи значений параметров в стек в соответствии с правилами, принятыми в языке Си++ (сначала в стек будет помещено значение последнего аргумента, а в конце — первого).

Аналогичные конструкции введены и для вызова программ на языке Ассемблера из программ на языке Си++. Рассмотрим пример:

```
/* пример_CPP EX4_9 */
#include <stdio.h>
extern "C" void asm_f10(int *i1,int *i2);
void main(void)
{   int i=5,j=7;
    printf("i = %d; j = %d\n",i,j);
    asm_f10(&i,&j);
    printf("i = %d; j = %d\n",i,j);
}

; пример_ASM EX4_9
.MODEL SMALL
.CODE
PUBLIC C asm_f10
asm_f10 PROC C near i1:dataptr, i2:dataptr
    push si
    push di
    mov si,i1
    mov cx,[si]
    mov di,i2
    xchg cx,[di]
    mov [si],cx
    pop di
    pop si
    ret
asm_f10 ENDP
end
```

Здесь задано, что в программу на языке Ассемблера передаются два указателя на данные i1 и i2. Далее эти указатели непосредственно используются в программе на языке Ассемблера.

Рассмотрим еще один пример:

```
/* пример_CPP EX4_10 */
#include <stdio.h>
extern "C" asm_f9(int i,int j);
void main(void)
{   int i=100,j=50;
    printf("i + j = %d\n",asm_f9(i,j));
}
```

```

; пример_ASM EX4_10
.MODEL SMALL
.CODE
PUBLIC C asm_f9
asm_f9 PROC C near i1:word, i2:word
        mov bx,i1
        mov ax,i2
        add ax,bx
        ret
asm_f9 ENDP
end

```

В заключение укажем, что если ассемблерная программа вызывает оверлейную процедуру или функцию, то такая ассемблерная программа должна быть типа FAR.

4.6. Встроенный ассемблер (режим inline в программах на языке Си ++)

Наряду с рассмотренными возможностями Borland C++ позволяет записывать ассемблерный код непосредственно в программе на языке Си++. Любую ассемблерную команду можно записать в следующем виде (при использовании Visual C++ перед символом asm надо записывать два символа подчеркивания: `_asm`):

`asm код_операции операнды; или новая строка`

Здесь «код_операции» задает команду языка Ассемблера (например, `mov`), «операнды» — это операнды команды (например, `ax, bx`). В конце записывается точка с запятой или команда завершается. Если с помощью одного слова `asm` необходимо задать много ассемблерных команд, то они заключаются в фигурные скобки. Это демонстрируется в приведенном ниже примере. После точки с запятой в конце команды нельзя записывать комментарии (как это делается в программе на языке Ассемблера). Комментарии можно записывать только в форме, принятой в языке Си++. Новую ассемблерную команду (начиная со слова `asm`) можно поместить в строку с предыдущей командой после точки с запятой.

В программе на языке Си++, использующей ассемблерные команды, необходимо задать директиву `#pragma inline` (вместо нее можно использовать опцию `-B` при компиляции программы). Заметим, что опция `-B` предполагает использование компилятора TASM. Если использовать другой компилятор, то необходимо задать опцию `-Exxx`, где `xxx` — другой ассемблер. Рассмотрим пример:


```

/* пример_CPP EX4_11 */
#include <iostream.h>
#pragma inline
void main(void)
{ int a=10,b=20,c;
  cout << "a = " << a << "; b = " << b << "\n";
  asm mov ax,10 ; // в ax значение 10
  asm mul a ; /* умножение ax * a */
  c = _AX;
  cout << "c = " << c << "\n";
  asm {
    mov ax,a
    mov bx,b
    xchg ax,bx
    mov a,ax
    mov b,bx
  }
  cout << "a = " << a << "; b = " << b << "\n";
}

```

Эту программу можно набрать во встроенном редакторе Borland C++. Для компиляции, компоновки и выполнения программы в среде Borland C++ достаточно нажать клавиши «Ctrl—F9».

Ассемблерные строки могут содержать выполняемые инструкции внутри функции или объявления переменных вне функции. Ассемблерные строки вне функции помещаются в сегмент DATA, а внутри функции — в сегмент CODE (см. гл. 5). Рассмотрим пример:

```

/* пример_CPP EX4_12 */
#include <stdio.h>
#pragma inline
asm v1 DW 10
asm v2 DW 90
void main(void)
{ unsigned i;
  again:
    asm { mov ax,v2
          add ax,v1
          push ax
          mov i,ax
        }
    printf("i = %d\n",i);
    if(i > 190) asm jmp _end_
    else if(i == 150) asm { mov ah,2
                          mov dl,7 // звуковой сигнал
                          int 21h }

    asm { pop ax
          mov v2,ax
          jmp again }
  _end_: puts("конец работы программы\n");
}

```

Эта программа выводит на экран целые значения от 100 до 200 через 10. Добавление 10 производится в ассемблерной команде `add ax, v1` (после выдачи значения 150 вырабатывается звуковой сигнал).

В заключение укажем, что ассемблерные команды могут свободно использовать регистры SI и DI (их значения не будут изменены командами языка Си++). Если ассемблерные команды используют регистры SI и DI, то компилятор Си++ не будет выбирать эти регистры под переменные типа `register`.

ГЛАВА 5

ПРОГРАММИРОВАНИЕ В СРЕДЕ WINDOWS С ИСПОЛЬЗОВАНИЕМ БИБЛИОТЕК КЛАССОВ

5.1. Построение простейшей программы

На рис. 5.1 показано, как прикладные программы (т. е. программы пользователя) взаимодействуют с системой **Windows**. Любая прикладная программа состоит из двух главных частей, которыми являются функция **WinMain()**, вызываемая один раз при запуске прикладной программы, и функция **WindowProc()**, которая ждет поступающих от **Windows** сообщений и обрабатывает их как с помощью собственных ресурсов, так и путем вызова сервисных функций API. Таким образом:

1) система **Windows** и прикладные программы взаимодействуют друг с другом. Взаимодействие осуществляется через сообщения. Предположим, что процедуры А и В являются компонентами некоторой программной системы. Когда мы говорим, что процедура А посылает сообщение процедуре В, то это означает, что процедура А вызывает подпрограмму в теле процедуры В. Каждое сообщение определяется именем вызываемой подпрограммы и аргументами, т. е. некоторыми значениями, которые должны быть переданы в вызываемую подпрограмму;

2) если прикладная программа «хочет» обратиться к системе **Windows**, она посылает соответствующее сообщение, содержащее имя нужной функции API и значения ее аргументов;

3) если **Windows** «хочет» обратиться к прикладной программе, также посылается сообщение. При этом управление передается некоторой специальной части программы, находящейся в теле функции **WindowProc()** (рис. 5.2), которая имеет структуру инструкций switch-case языков Си/Си++. В эту часть передается параметр, задающий действия, которые необходимо выполнить;

4) сообщение может быть направлено либо непосредственно, либо через очередь. В последнем случае сообщения будут извлекаться из очереди прикладной программой. В обоих случаях прикладная программа имеет одну центральную точку получения и обработки сообщений.

В целом функция **WinMain()** выполняет все, что необходимо для того, чтобы инициализировать внутренние переменные и

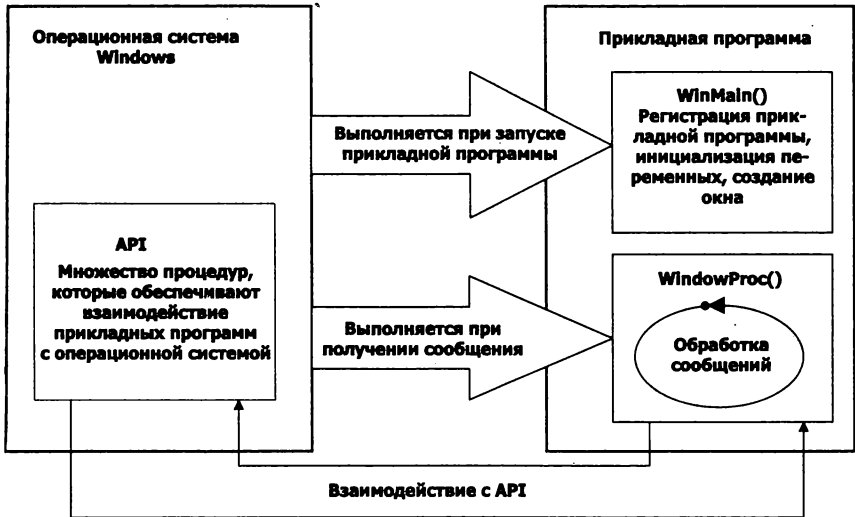


Рис. 5.1. Организация взаимодействия прикладной программы с системой Windows

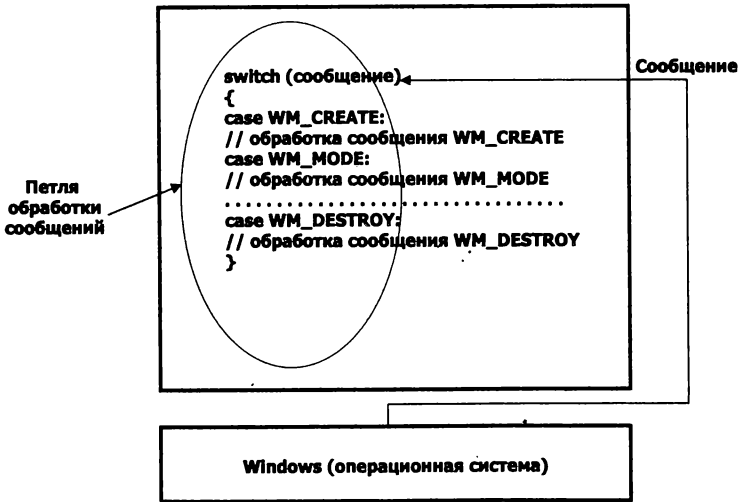


Рис. 5.2. Обработка полученных от системы Windows сообщений

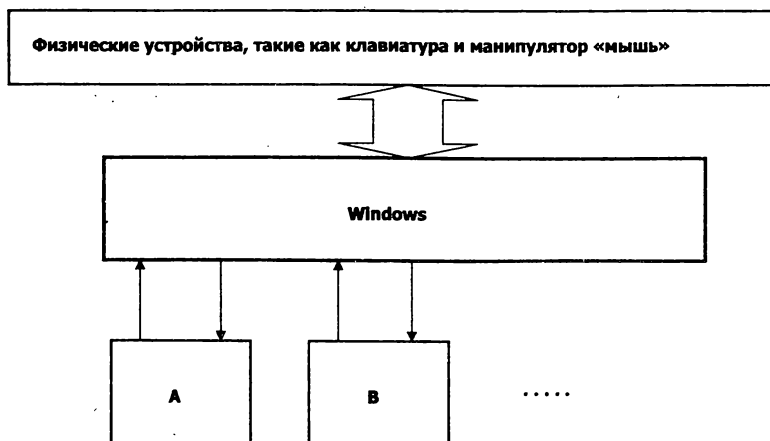


Рис. 5.3. Взаимодействие системы Windows с программами и устройствами

установить окна на экране дисплея для прикладной программы. Она также содержит средства для извлечения сообщений из очереди и передачи их в прикладную программу.

Функция **WindowProc()** обрабатывает все сообщения, которые не находятся в очереди.

Сообщения могут поступать в систему **Windows** как от программных, так и от физических устройств, таких как манипулятор «мышь» или клавиатура, и далее передаваться в прикладную программу. Источниками программных сообщений являются, например, средства интерфейса с пользователем, отображаемые на экране дисплея, разделы меню, различные кнопки, полосы горизонтальной и вертикальной прокрутки и т. п.

Взаимодействие между различными прикладными программами (такими как А и В на рис. 5.3.), а также между прикладными программами и физическими устройствами (см. рис. 5.3.) осуществляется через систему **Windows** (через систему функций, включенных в API). Существующие библиотеки классов, такие как MFC (Microsoft Foundation Classes) и ObjectWindows (Borland), группируют средства API в подсистемы, организованные в виде удобных для программирования иерархических структур. Такие подсистемы функций, каждая из которых предназначена для решения близких по сути задач, рассматриваются как абстрактные типы или классы. Последние позволяют использовать технологию объектно-ориентированного программирования, т. е. использовать такие ее базовые концепции, как пакетирование

(encapsulation), наследование (inheritance) и полиморфизм (polymorphism). Программа пользователя строится как система новых классов (пакетов), в основном, наследуемых из классов, включенных в библиотеки, такие как MFC или ObjectWindows. Взаимодействие между объектами этих классов организуется через сообщения. Специализация пользовательских (наследуемых) классов осуществляется путем добавления и использования новых компонентов-функций и компонентов-данных (см. гл. 3).

Ниже будут даны основные сведения по библиотеке MFC. Все классы, включенные в MFC, имеют префикс (первую букву) C, например, CView, CObject и т. п. Компоненты-данные имеют префикс m_. В целом, система MFC использует Венгерскую нотацию (соглашение) для различных имен. Так, указатели (pointers) имеют префикс p, целые числа — префикс n и т. п. Например, имя m_ и nCmdShow означает компонент-данное (префикс m) целого типа (составной префикс m_ и n).

Рассмотрим теперь структуру простейшей программы, основанной на использовании MFC. При этом мы не будем прибегать к услугам AppWizard (см. гл. 1). Выберем тип проекта Win32 Application. Поскольку мы будем использовать MFC, то надо установить соответствующую опцию, например: Project-Settings-General-Shared DLL.

Текст нашей простейшей программы представляется в следующем виде:

```
#include <afxwin.h> class CFirstApp : public CWinApp{ public: virtual BOOL
InitInstance(); };
    class CFirstWin : public CFrameWnd{ public: CFirstWin()
// конструктор { Create(0, "Our first program"); }
};
BOOL CFirstApp::InitInstance(void){ m_pMainWnd = new CFirstWin;
m_pMainWnd -> ShowWindow(m_nCmdShow); return TRUE;}
CFirstApp OurApplication;
```

Рассмотрим эту программу более подробно. Строка

```
#include <afxwin.h>
```

содержит описание многих классов, включенных в MFC. Для нашей программы мы использовали только два класса MFC:

CWinApp и CFrameWnd.

Первый класс CWinApp обеспечивает все, что необходимо для запуска, инициализации (начальной установки), выполнения и завершения прикладной программы. Поэтому первое, что мы сделали в нашей программе, это построили произвольный класс CFirstApp, который наследуется из класса MFC CWinApp:

```
class CFirstApp : public CWinApp
{
    public:
        virtual BOOL InitInstance();
};
```

В нашем примере мы хотим использовать все функции класса CWinApp без изменений, кроме одной из них, прототип (описание) которой

```
virtual BOOL InitInstance();
```

Здесь BOOL это тип возвращаемого значения. Таким значением может быть TRUE (истина) либо FALSE (ложь). Функцию InitInstance мы изменили таким образом, чтобы в заголовке будущего окна выволился наш текст: Our first program (наша первая программа). Переопределенная функция InitInstance выглядит следующим образом:

```
BOOL CFirstApp::InitInstance(void)
{
    m_pMainWnd = new CFirstWnd;
    m_pMainWnd->ShowWindow(m_nCmdShow);
    return TRUE;
}
```

Эта новая функция будет автоматически вызвана вместо функции InitInstance MFC. Используемые компоненты-данные (m_pMainWnd и m_nCmdShow) взяты из базового класса CWinApp. После инструкции

```
m_pMainWnd = new CFirstWnd;
```

в динамической памяти создается интерфейсный объект и указатель на него сохраняется в компоненте m_pMainWnd.

Интерфейсный объект содержит средства построения интерфейсного элемента, каким собственно и является окно. В примере использовано окно типа CFrameWnd, в котором мы изменили только заголовок:

```
class CFirstWin : public CFrameWnd
{
    public:
        CFirstWin() // конструктор
        {
            Create(0, "Our first program");
        }
};
```

Новый текст заголовка задается в переопределенном конструкторе (см. второй аргумент функции Create). Первый аргумент функции Create (эта функция наследуется из класса CFrameWnd) говорит о том, что мы хотим использовать без изменений атрибуты окна, заданные по умолчанию. Второй аргумент — это строка

текста, которая выведена в качестве заголовка окна. Функция `Create` имеет и другие аргументы, но мы их не задавали. В результате будут использованы аргументы, заданные по умолчанию.

Вернемся к функции `InitInstance`. Функция `ShowWindow()`, которая наследуется из базового класса, обеспечивает построение интерфейсного элемента (окна). Ей передается параметр `m_nCmdShow`, который инструктирует эту функцию и задает окна на экран.

Последняя строка:

```
CFirstApp OurApplication;
```

определяет объект `OurApplication`, который будет отвечать за все действия в нашей программе. Он должен быть описан в глобальной области перед вызовом главной функции `WinMain` (аналог функции `main` в программах на языках Си/Си++). Функция `WinMain` автоматически строится и вызывается с помощью MFC. Она, в свою очередь, вызывает функцию `InitInstance` для объекта класса `CFirstApp` (мы назвали его `OurApplication`).

Выше мы создали объект класса `CFirstWin` в динамической памяти с помощью оператора **new**. Однако нет необходимости освобождать эту память с помощью оператора **delete**. Об этом позаботится функция `WinMain`.

После выполнения нашей программы на экране появится окно, показанное на рис. 5.4.

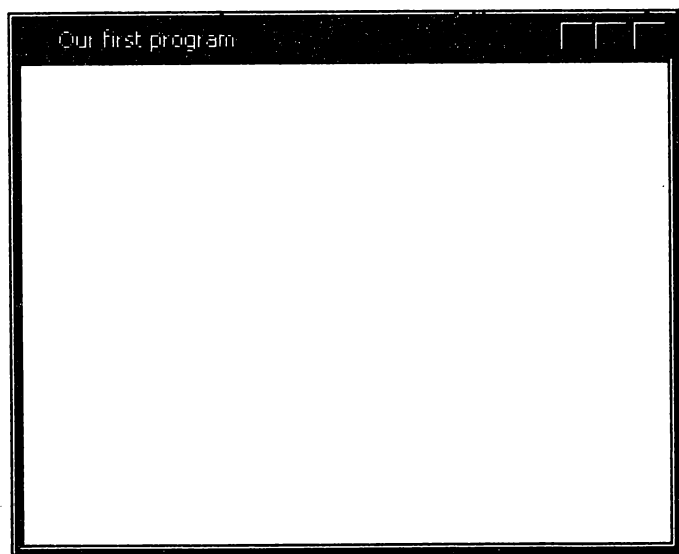


Рис. 5.4. Пример выполнения простейшей программы

В будущем описания классов мы будем помещать в файлы заголовков типа. h, а описания функций (которые не были описаны в классах) — в файлы типа. cpp.

В целом, наша программа очень простая и абсолютно «бесполезная», но она позволяет построить стандартное окно (стандартный интерфейсный элемент) со всеми необходимыми свойствами. То есть можно изменять с помощью стандартных средств системы **Windows** размер окна, минимизировать его и т. п.

5.2. Базовые компоненты программы

В программе, основанной на использовании MFC, можно выделить два базовых понятия — документ (Document) и вид документа (View).

Документ — это имя, которое используется для того, чтобы охарактеризовать те данные в программе, с которыми взаимодействует пользователь. Пользовательский класс документов наследуется из класса MFC **CDocument**.

Программа может использовать один или несколько документов. В первом случае она характеризуется как SDI (Single Document Interface), а во втором случае — MDI (Multiple Document Interface).

Вид (View) определяется объектом, который позволяет отобразить все или некоторые данные из соответствующего документа. Этот объект задает как способ отображения данных, так и механизм взаимодействия с ними со стороны пользователя. Для того чтобы обеспечить все эти действия, необходимо наследовать пользовательский класс из класса **CView** MFC.

В целом, такие понятия, как документ, вид и окно, на экране дисплея можно связать следующим образом:

- 1) документ содержит множество данных, которыми оперирует программа;
- 2) вид определяет часть данных из п. 1), хотя, в частном случае, может задавать и все эти данные;
- 3) окно рассматривается как некоторая заготовка или фрейм. Например, оно включает заголовок в верхней части и рабочую область (ниже заголовка), в которую и помещаются данные из п. 2).

С одним и тем же объектом документа можно связать разные объекты типа вид. В результате можно отображать различные данные документа в разных окнах.

Управление объектами документов в программе осуществляется классами шаблонов документов, которыми являются **CSingleDocTemplate** — для программы с единственным докумен-

том (SDI) и CMultiDocTemplate — для программы со многими документами (MDI).

MFC обеспечивает механизм взаимосвязи объектов рассмотренных типов. Так, объект документа автоматически обрабатывает список указателей на связанные с ним объекты типа «вид». Объект типа «вид» хранит в своих данных указатель на объект документа, с которым он связан. Объект типа «фрейм» имеет указатель на активный объект типа «вид». Объекты документа и типа «фрейм» создаются объектом шаблона документа. Объект типа «вид» создается объектом типа «фрейм». Объект шаблона документа создается объектом приложения (см. класс CWinApp в первой секции этой главы).

В целом, можно выделить четыре базовых класса MFC, которые будут включены практически во все прикладные программы:

- 1) класс приложения **CWinApp**;
- 2) класс типа фрейм **CFrameWnd**;
- 3) класс типа документ **CDocument**;
- 4) класс типа вид **CView**.

Пользовательские классы будут наследоваться из этих классов и использовать функции MFC вместе с новыми функциями, определяющими конкретную специализацию прикладной программы.

5.3. Построение прикладной программы с помощью инструментов AppWizard и ClassWizard

В гл. 1 кратко были описаны основные этапы построения прикладной программы с помощью инструмента AppWizard. Рассмотрим эти этапы более подробно. Здесь и в последующих разделах кроме AppWizard будем использовать два новых инструмента: ClassWizard и редактор ресурсов. Первый из них позволяет расширять и специализировать классы в прикладной программе. Второй используется для создания ресурсов, таких как меню, панели инструментов, диалоговые окна, средства контроля, иконки (пиктограммы) и т. п.

AppWizard позволяет автоматически строить прикладные программы типа SDI (с единственным документом) и MDI (со многими документами). Рассмотрим сначала этапы построения и структуру полученной программы типа SDI (назовем ее FirstSDI). Построение программы осуществляется поэтапно и предполагает выполнение следующих действий:

- 1) создается проект с опцией MFC AppWizard (exe) (см. секцию 1.3) и вводится его имя — FirstSDI. В появившемся на экране диалоговом окне можно выбрать одну из трех опций: Single document (для программы типа SDI), Multiple documents (для

программы типа MDI) и Dialog based. В последнем случае вообще не открываются документы. Результатом является простое диалоговое окно без меню. В нашем случае выбирается опция Single document, нажимается клавиша Next (с помощью манипулятора «мышь») и осуществляется переход к следующему этапу;

2) новое диалоговое окно позволяет выбрать уровень поддержки операций с базами данных. Поскольку мы не планируем использование баз данных, то выберем опцию None (т. е. работа с базами данных не поддерживается). Далее, как и выше, нажимается клавиша Next (перейти к следующему этапу);

3) новое диалоговое окно позволяет выбрать степень поддержки операций для работы с включенными элементами, которые используют технологию связывания OLE (Object Linking and Embedding). Технология OLE позволяет разрабатывать такие программы, в которые можно импортировать (вставлять) объекты из других документов (например, документы Word). Сегодня эта технология получила новое название — ActiveX. Мы не будем импортировать объекты из других программ. Поэтому выберем опцию None и клавишу Next;

4) новое диалоговое окно позволяет задать опции, определяющие внешнее представление компонентов пользовательского интерфейса. Примем опции, заданные по умолчанию, и перейдем к следующему этапу;

5) очередное диалоговое окно позволяет установить две опции. Первая позволяет задать включенные в текст прикладной программы комментарии (ответим Yes, т. е. будем включать комментарии). Вторая опция определяет, как библиотека MFC будет использоваться в программе. Не вдаваясь в детали, выберем опцию As shared DLL. При этом средства библиотеки не будут включены в программный код и присоединяются на этапе выполнения программы;

6) в очередном диалоговом окне появляется список имен классов, которые AppWizard построил для прикладной программы. Этими классами являются: CFirstSDIApp, CMainFrame, CFirstSDIDoc и CFirstSDIView. Выберем первое имя CFirstSDIApp. После этого в нижней части диалогового окна можно видеть имя класса (Class name), имя файла заголовка (Header file), имя базового класса (Base class) и имя файла на языке Си++ (Implementation file). Некоторые имена можно менять, но мы не будем это делать. Для класса CFirstSDIView можно изменять базовый класс для того, чтобы обеспечить выполнение желаемых действий. Предположим, что мы хотим обеспечить возможность редактирования текста в окне документа. Для этого заменим в окне Base class класс CView, заданный по умолчанию,

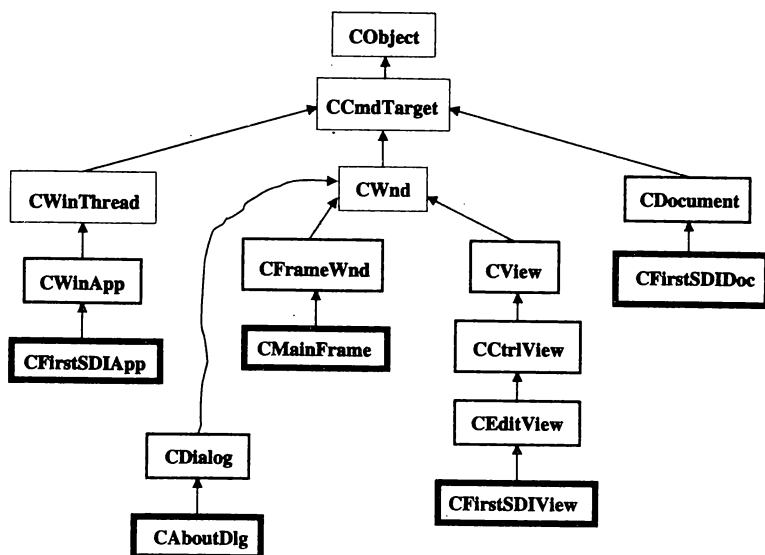


Рис. 5.5. Место классов программы в общей иерархической подструктуре классов MFC

на класс CEditView. После этого выберем кнопку Finish (закончить).

Построим выполняемый файл программы CFirstSDI.exe (см. гл. 1), которая будет представлять собой простейший текстовый редактор. Для того чтобы просмотреть классы программы, можно выбрать опцию ClassView в левом окне.

Наша программа содержит четыре основных класса, которые были кратко охарактеризованы выше: CFirstSDIApp — класс приложения, CMainFrame — класс фрейма, CFirstSDIDoc — класс документа и CFirstView — класс представления документа (класс вида документа).

Новый класс CAboutDlg позволяет вывести диалоговое окно с информацией о нашей программе. В разделе Globals (см. окно ClassView) содержится одно глобальное имя нашей программы — **theApp**. Для того чтобы отобразить в правом окне код любого класса и включенной в него функции, достаточно переместить курсор «мыши» на соответствующее имя в левом окне и дважды нажать левую клавишу «мыши». Если проанализировать все классы, то можно увидеть, что наша программа имеет иерархию классов, показанную на рис. 5.5. Классы нашей прикладной программы на этом рисунке заключены в прямоугольники с двойными линиями. Прямоугольники с одинарными сплошными линиями

показывают пути наследования наших классов из основных базовых классов, рассмотренных выше. Рис. 5.5 показывает также место классов программы в общей иерархической подструктуре классов MFC.

Рассмотрим теперь, как работает программа. Для этого в левом окне с установленным режимом ClassView выберем опцию Globals, где записано имя theApp главного объекта программы, расположенного в глобальной области. Если выбрать этот объект (переместить курсор «мыши» на имя theApp и дважды нажать ее левую кнопку), то в правом окне появится строка:

CFirstSDIApp the App;

которая содержится в файле FirstSDI.cpp. После создания объекта theApp автоматически вызывается функция WinMain(), которая в свою очередь вызывает функции CFirstSDIApp::InitInstance() и CFirstSDIApp::Run(). Функция InitInstance(), выполняет инициализацию программы, создает шаблон документа, фрейм главного окна, документ и его вид. Функция Run() обеспечивает начальную обработку сообщений **Windows**. Вызов рассмотренных выше функций можно просмотреть по шагам в отладчике (см. § 1.4).

Программа типа MDI создается с помощью инструмента AppWizard примерно так же. На первом этапе необходимо выбрать опцию Multiple documents, а все остальные этапы можно повторить без изменений. После запуска построенной программы на выполнение можно создать множество документов путем выбора опций File — New в главном меню нашей программы. Новое представление (вид) того же самого документа можно создать выбрав опции Window — New Window.

Рассмотрим теперь дополнительные возможности инструмента ClassWizard (см. опции View — ClassWizard). Если вызвать диалоговое окно ClassWizard для нашей программы типа MDI, то можно проанализировать и добавить, например, следующие компоненты:

1) сообщения. Для этого необходимо выбрать опцию Message Maps. В результате в классы можно добавить новые функции, которые будут автоматически вызываться при поступлении от системы **Windows** определенных сообщений (см. § 5.4). В тело этих новых функций можно включить код, который будет обрабатывать соответствующие сообщения. Имя текущего класса показывается в правом верхнем окне (Class name). В левом нижнем окне (Object IDS) можно выбрать идентификатор (имя) сообщения **Windows**. При этом в правом нижнем окне (Messages) появляются сообщения, которые автоматически генерируются для соответствующего идентификатора;

2) производные классы. Для этого надо выбрать кнопку AddClass;

3) новые переменные. Для этого надо выбрать опцию Member Variables.

Другие возможности этого инструмента будут рассмотрены в следующем параграфе.

Для добавления ресурсов в программу необходимо выбрать опции Insert — Resource. Мы рассмотрим использование инструмента Class Wizard и редактора ресурсов в последующих параграфах этой главы на примерах.

5.4. Анализ и обработка сообщений

Для того чтобы обработать некоторое сообщение, необходимо установить связь между этим сообщением и функцией, которая будет вызвана при получении этого сообщения. Такая связь задается при помощи карты сообщений (Message Map). Любой класс, который может обрабатывать сообщения, имеет такую карту, автоматически создаваемую инструментами AppWizard и ClassWizard.

Вернемся к программе CFirstSDI из § 5.3 и рассмотрим построенный класс CFirstApp:

```
class CFirstSDIApp : public CWinApp
{
    public:
        CFirstSDIApp();
        .....
    public:
        virtual BOOL InitInstance();
        .....
        afx_msg void OnAppAbout();
        .....
        DECLARE_MESSAGE_MAP()
};
```

На месте многоточий в программе, построенной с помощью AppWizard, находятся комментарии, которые нельзя удалять и редактировать, поскольку они используются системой программирования Visaul C++. Строка

DECLARE_MESSAGE_MAP()

это макро, указывающее, что класс CFirstSDIApp может содержать компоненты функции, которые обрабатывают поступающие от системы **Windows** сообщения.

Если в описание класса включено макро **DECLARE_MESSAGE_MAP()**, то необходимо задать код для соответствующей карты сообщений, который помещается между строками(макро)**BEGIN_MESSAGE_MAP()**и**END_MESSAGE_MAP()**. Этот

код был автоматически добавлен инструментом AppWizard в нашу программу и содержится в файле CFirstSDIApp:

```
BEGIN_MESSAGE_MAP(CFirstSDIApp, CWinApp)
.....
ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
.....
ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)

.....
ON_COMMAND(ID_FILE_PRINT_SETUP,
CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

Эта карта сообщений устанавливает соответствие между командами меню (такими, как new — создать новый файл, open — открыть существующий файл и т. п.) и функциями (такими, как CWinApp::OnFileNew, CWinApp::OnFileOpen и т. п.), которые будут вызываться при поступлении соответствующих сообщений. Например, если выбрана опция new, то с помощью макро ON_COMMAND вызывается функция OnFileNew, которая является компонентом класса MFC CWinApp. Первым аргументом макро ON_COMMAND является имя (например, ID_FILE_NEW), которое поставлено в соответствие опции new. При выборе этой опции автоматически генерируется сообщение, которое вызывает функцию OnFileNew, заданную в виде второго аргумента.

Макро BEGIN_MESSAGE_MAP() тоже имеет аргументы. Первый из них задает имя нашего класса CFirstSDIApp, для которого собственно и рассматривается карта сообщений. Второй аргумент задает имя базового класса CWinApp.

Различные сообщения, которые могут обрабатываться программой, делятся на три группы:

1) стандартные сообщения **Windows**, которые имеют префикс WM. Примерами таких сообщений являются WM_RBUTTONDOWN, которое возникает при нажатии правой кнопки манипулятора «мышь», WM_KEYDOWN, возникающее при нажатии клавиатуры, и т. п.

2) управляющие сообщения, которые посылаются от средств контроля;

3) командные сообщения, которые формируются от элементов, организующих интерфейс с пользователем (элементами меню и т. п.).

Рассмотрим пример построения программы, которая обрабатывает командные сообщения. Назовем новую программу **CComMes** и построим ее на основе CFirstSDIApp. В меню добавим два новых пункта. Первый пункт позволит вычертить на

экране прямоугольник (Rectangle) или окружность (Circle), а второй будет задавать либо их увеличение (Zoom in), либо уменьшение (Zoom out).

Добавление новых опций в меню можно произвести с помощью редактора ресурсов. Предварительно построим выполняемый модуль программы CComMes так же, как мы это делали для программы SFirstSDIApp (заменяем только класс **CEditView** на заданный по умолчанию класс **CView**).

Построенная программа уже имеет присоединенное меню, а значит и соответствующие ресурсы. Их можно увидеть, если выбрать в левом окне опцию Resource View и, далее, раздел Menu (Меню), в котором записан идентификатор IDR_MAINFRAME.

Заметим, что в случае программы типа MDI здесь будет записано два идентификатора IDR_MAINFRAME и IDR_COMMESTYPE. Первый задает меню, которое выводится, если в программе не открыт документ, а второй — если документ открыт. Здесь новые пункты (назовем их Figure и Zoom) целесообразно включать только во второе меню, поскольку они предполагают работу с документом.

Поскольку в нашем примере только один документ, то выберем меню IDR_MAINFRAME. Для вызова редактора ресурсов необходимо переместить указатель «мыши» на имя IDR_MAINFRAME и дважды нажать ее левую клавишу. На экране появится меню для нашей программы. Справа, после пункта Help, есть свободное место для нового пункта, которое необходимо выбрать «мышью» и ввести имя нового раздела меню. Если перед любой буквой ввести символ &, то эта буква позволит выбрать пункт меню с клавиатуры. Например, если ввести имя F&igure, то всплывающее подменю Figure можно выбрать с клавиатуры, нажав клавиши Alt-i. Активный символ (в нашем примере i) будет подчеркнут в соответствующем имени (в нашем примере Figure). В процессе ввода имени на экране появится диалоговое окно с именем Properties (свойства), которое можно использовать для изменения различных параметров подменю или пункта подменю. При необходимости пункт Figure можно поместить в любой позиции верхней строки. Для этого надо установить на него указатель «мыши», нажать левую кнопку и переместить подменю Figure в новое место.

Диалоговое окно Properties имеет иконку типа канцелярской кнопки в левом верхнем углу. Если выбрать эту иконку, то окно будет как бы прикреплено к экрану. Это удобно, поскольку оно не будет исчезать при выборе других окон.

Добавление подпунктов (опций) в новое подменю осуществляется примерно так же; т. е. выбирается первая свободная линия, которая появляется ниже имени Figure, и в ней записывается нужное имя (&Rectangle и &Circle для нашего примера).

Поскольку эти новые пункты не вызывают всплывающее меню, то в диалоговом окне Properties следует сбросить флажок Pop-up. Там же для одного нового пункта (например, Rectangle) следует установить флажок Checked.

В левом верхнем прямоугольнике (с именем ID) диалогового окна Properties (режим General) можно видеть имя идентификатора соответствующего раздела меню, который будет использоваться в качестве первого аргумента макро ON_COMMAND карты сообщений. Его (имя) можно изменить, но префикс ID_ надо оставить (а лучше всего вообще ничего не менять). В нижней части окна Properties (Prompt) можно записать текст, который будет появляться в нижней линии статуса окна, когда курсор «мыши» перемещается на имя соответствующего пункта меню.

По аналогии с рассмотренным выше необходимо добавить новое всплывающее меню &Zoom с подпунктами i&n и &out. Активные символы в именах выбираются так, чтобы они не повторялись для разных пунктов. Поскольку символ i использовался в подменю F&igure, то мы не можем выбрать его снова. Поэтому мы выбрали символ n в подпункте in. Заметим, что в нашем примере только один (активный) пункт подменю может иметь установленный флажок Checked.

Рассмотрим теперь, как использовать инструмент ClassWizard для построения функций, обрабатывающих сообщения от разных пунктов меню. Диалоговое окно ClassWizard можно вызвать несколькими способами, например, нажать клавиши Ctrl-w. Выберем в этом окне опцию Message Maps. При этом отображаются пять окон, в которых можно видеть:

- 1) имя текущего проекта (окно Project);
- 2) имя текущего класса (окно Class name);
- 3) список идентификаторов (имен) сообщений, для которых в текущий класс можно добавить функции, обрабатывающие эти сообщения (окно Object IDs);
- 4) типы сообщений, которые действительны для выбранного в предыдущем окне имени (окно Messages);
- 5) список функций, обрабатывающих сообщения; которые уже определены для текущего класса (окно Member functions). Здесь виртуальные функции помечены символом V, а сообщения **Windows** — символом W.

Все имена, которые мы присвоили новым пунктам меню, уже включены в список имен окна Object IDs. Соответствующие сообщения мы будем обрабатывать в классе CComMesDoc. Для этого необходимо выбрать имя класса CComMesDoc в окне Class name и соответствующий идентификатор в окне Object IDs. В окне Messages появится два типа возможных сообщений COMMAND и UPDATE_COMMAND_ID. Первое из них ge-

нерируется, когда выбирается соответствующий пункт меню, а второе, когда меню обновляется (например, изменяется флажок Checked). Сообщение этого типа генерируется перед тем, как появляется всплывающее (Pop-up) подменю, что позволяет при необходимости модифицировать атрибуты всех пунктов этого подменю.

Для добавления функции, обрабатывающей сообщение, необходимо выбрать кнопку Add Function (добавить функцию). В результате появляется диалоговое окно с именем функции, которое можно либо принять, либо изменить. Если выбрать кнопку OK, то новая функция будет добавлена в нижнее окно (Member functions).

Предположим, что мы выполнили рассмотренные выше действия для всех подпунктов меню. Теперь наш класс CComMesDoc будет выглядеть примерно так (комментарии поясняют назначение каждой функции, обрабатывающей сообщения);

```
class CComMesDoc : public CDocument
{
    .....
    // Generated message map functions
protected:
    //{AFX_MSG(CComMesDoc)
    afx_msg void OnFigureRectangle(); // вычерчивание
                                     // прямоугольника

    afx_msg void OnFigureCircle(); // вычерчивание
                                   // окружности
    afx_msg void OnZoomIn(); // увеличение размера
    afx_msg void OnZoomOut(); // уменьшение размера
    //}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

Каждая функция имеет префикс afx_msg, который указывает, что это обработчик сообщений. Карта сообщений будет выглядеть примерно так (см. файл CComMesDoc.cpp):

```
BEGIN_MESSAGE_MAP(CComMesDoc, CDocument)
    //{AFX_MSG_MAP(CComMesDoc)
    ON_COMMAND(ID_FIGURE_RECTANGLE, OnFigureRectangle)
    ON_COMMAND(ID_FIGURE_CIRCLE, OnFigureCircle)
    ON_COMMAND(ID_ZOOM_IN, OnZoomIn)
    ON_COMMAND(ID_ZOOM_OUT, OnZoomOut)
    //}AFX_MSG_MAP
    END_MESSAGE_MAP()
```

Все, что необходимо для этой карты, мы уже рассматривали. Например, строка:

```
ON_COMMAND(ID_FIGURE_RECTANGLE, OnFigureRectangle)
```

говорит о том, что если выбрать опции Figure — Rectangle, то посылается сообщение с именем ID_FIGURE_RECTANGLE (оно было автоматически выбрано системой) и в ответ на него вызывается функция OnFigureRectangle (ее имя было автоматически назначено системой).

Теперь нам необходимо модифицировать программу и сформировать код для всех обработчиков сообщений, в теле которых пока нет никаких инструкций. Прежде всего добавим следующие компоненты-данные:

m_xb, m_yb — координаты левого верхнего угла нашего прямоугольника. В случае рисования окружности мы будем вписывать ее в прямоугольник, местоположение которого задается этими же координатами;

m_xe, m_ye — координаты нижнего правого угла нашего прямоугольника (они же задаются и для окружности);

m_E — текущий тип элемента (т. е. либо прямоугольник, либо окружность).

Добавим соответствующие объявления в секцию protected классов CComMesView (первая строка) и CComMesDoc (вторая строка):

```
WORD m_xb, m_yb, m_xe, m_ye;  
WORD m_E;
```

Здесь тип WORD это то же самое, что и тип unsigned int. Для добавления переменной можно переместить указатель «мыши» на имя класса (CComMesDoc) в левом окне, нажать правую кнопку и выбрать опцию Add Member Variable.

Для того чтобы задать тип элемента, можно определить константы с неповторяющимися значениями, например,

```
const WORD CIRCLE = 101U;           // для окружности  
const WORD RECTANGLE = 102U;       // для прямоугольника
```

Эти объявления можно поместить в файл типа .h, например, с именем constants.h:

```
#if !defined (constants_h)  
#define constants_h  
    // константы для окружности и прямоугольника  
#endif
```

Для того чтобы присвоить начальные значения новым элементам данного класса CComMesDoc, необходимо модифицировать конструктор этого класса, т. е. добавить, например, в него следующее объявление:

```
m_E = RECTANGLE;
```

В результате по умолчанию на экране будет вычерчиваться прямоугольник (и это задано также флажком Checked меню). Для добавления приведенной выше строки необходимо раскрыть компоненты класса CComMesDoc в левом окне, переместить указатель «мыши» на имя конструктора (CComMesDoc) и дважды нажать левую кнопку. Затем в правом окне надо ввести новую строку.

Новые функции обработчики сообщений можно задать в следующем виде:

```
void CComMesDoc::OnFigureCircle()
{
    // TODO: Add your command handler code here
    m_E = CIRCLE;
}
void CComMesDoc::OnFigureRectangle()
{
    // TODO: Add your command handler code here
    m_E = RECTANGLE;
}
void CComMesDoc::OnZoomIn()
{
    // TODO: Add your command handler code here
    if (m_zoom < 11)
    {
        ++m_zoom; UpdateAllViews(NULL);
        m_out_dis = FALSE;
    }
    if (m_zoom == 10) m_in_dis = TRUE;
}
void CComMesDoc::OnZoomOut()
{
    // TODO: Add your command handler code here
    if (m_zoom > 0)
    {
        --m_zoom; UpdateAllViews(NULL);
        m_in_dis = FALSE;
    }
    if (m_zoom == 0) m_out_dis = TRUE;
}
```

Первая и вторая функции устанавливают тип текущей фигуры (соответственно окружность и прямоугольник). Третья функция увеличивает на единицу значение переменной m_zoom и может изменить две другие переменные m_out_dis и m_in_dis. Объявления этих трех переменных тоже надо добавить в секцию protected класса CComMesDoc. Первая имеет тип WORD, а две последние тип BOOL, возможными значениями для которого являются TRUE (истина) и FALSE (ложь). Переменную m_zoom мы будем использовать для того, чтобы установить, на сколько изменять размер фигуры. Изменение (увеличение или уменьшение) размера будем определять как 10m_zoom, причем зададим $1 \leq m_zoom \leq 10$. Константы типа 1 и 10 следовало бы заменить символическими именами, которые описываются в файле заголовка (по аналогии с тем, как мы это делали раньше).

Значения `m_out_dis` и `m_in_dis` будут позже использоваться для того, чтобы затенить пункт меню, если его выбор не будет приводить к каким-либо действиям в программе. Например, если `m_zoom=10`, то размер фигуры увеличивать нельзя ($1 \leq m_zoom \leq 10$) и пункт `in` меню `zoom` будет показан затененным. Функция `OnZoomOut()` построена по аналогии с третьей функцией, при этом:

1) начальные значения переменных `m_zoom`, `m_in_dis` и `m_out_dis` устанавливаются в конструкторе класса `CComMesDoc`, который теперь имеет вид:

```
CComMesDoc::CComMesDoc()
{
    // TODO: add one-time construction code here
    m_E = RECTANGLE;
    m_zoom = 0;
    m_out_dis = TRUE;
    m_in_dis = FALSE;
}
```

2) если `m_out_dis=TRUE`, то пункт `out` меню `Zoom` должен быть затенен;

3) если `m_in_dis=TRUE`, то пункт `in` меню `Zoom` должен быть затенен;

4) функция `UpdateAllViews()` обновляет все объекты типа вид, связанные с рассматриваемым объектом документа.

Добавим теперь функции обработчики сообщений `UPDATE_COMMAND_UI`. Для этого надо повторить рассмотренные выше действия с помощью инструмента `ClassWizard` и вместо типа `COMMAND` в окне `Messages` выбрать тип `UPDATE_COMMAND_UI`.

После этого добавленные в класс `CComMesDoc` компоненты функции и компоненты данные будут выглядеть примерно так:

```
protected:
    WORD m_in_dis;
    BOOL m_out_dis;
    WORD m_zoom;
    WORD m_E;
//{{AFX_MSG(CComMesDoc)
    afx_msg void OnFigureCircle();
    afx_msg void OnUpdateFigureCircle(CCmdUI* pCmdUI);
    afx_msg void OnFigureRectangle();
    afx_msg void OnUpdateFigureRectangle(CCmdUI* pCmdUI);
    afx_msg void OnZoomIn();
    afx_msg void OnZoomOut();
    afx_msg void OnUpdateZoomOut(CCmdUI* pCmdUI);
    afx_msg void OnUpdateZoomIn(CCmdUI* pCmdUI);
}}
```

Карта сообщений для класса CComMesDoc тоже будет расширена:

```
BEGIN_MESSAGE_MAP(CComMesDoc, CDocument)
//{{AFX_MSG_MAP(CComMesDoc)
    ON_COMMAND(ID_FIGURE_CIRCLE, OnFigureCircle)
    ON_UPDATE_COMMAND_UI(ID_FIGURE_CIRCLE,
        OnUpdateFigureCircle)
    ON_COMMAND(ID_FIGURE_RECTANGLE, OnFigureRectangle)
    ON_UPDATE_COMMAND_UI(ID_FIGURE_RECTANGLE,
        OnUpdateFigureRectangle)
    ON_COMMAND(ID_ZOOM_IN, OnZoomIn)
    ON_COMMAND(ID_ZOOM_OUT, OnZoomOut)
    ON_UPDATE_COMMAND_UI(ID_ZOOM_OUT, OnUpdateZoomOut)
    ON_UPDATE_COMMAND_UI(ID_ZOOM_IN, OnUpdateZoomIn)
//}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Новые функции представим в следующем виде:

```
void CComMesDoc::OnUpdateFigureCircle(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->SetCheck(m_E == CIRCLE);
    UpdateAllViews(NULL);
}
void CComMesDoc::OnUpdateFigureRectangle(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->SetCheck(m_E == RECTANGLE);
    UpdateAllViews(NULL);
}
void CComMesDoc::OnUpdateZoomOut(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->Enable(m_out_dis == FALSE);
}
void CComMesDoc::OnUpdateZoomIn(CCmdUI* pCmdUI)
{
    // TODO: Add your command update UI handler code here
    pCmdUI->Enable(m_in_dis == FALSE);
}
```

В них использованы две новые функции MFC SetCheck() и Enable(). Первая из них устанавливает маркер Checked около соответствующего пункта меню, если ее аргумент принимает значение TRUE. Символ Checked удаляется, если аргумент имеет значение FALSE. Вторая функция затеняет соответствующий элемент меню, если ее аргумент имеет значение FALSE. В действительности аргументы функций SetCheck() и Enable() имеют целый тип и единичное значение соответствует значению TRUE, а нулевое — значению FALSE.

Добавим в класс CComMesDoc еще две функции, которые будут использоваться позже:

```
WORD CComMesDoc::getElement()
{
    return m_E;
}
WORD CComMesDoc::getZoom()
{
    return m_zoom;
}
```

Первая из них возвращает значение текущего элемента (RECTANGLE или CIRCLE), а вторая — значение коэффициента m_zoom для изменения размера фигуры. Добавить эти функции очень просто. Для этого надо выбрать указателем «мыши» класс CComMesDoc (режим ClassView) в левом окне, нажать правую кнопку и выбрать опцию Add member function.

Теперь можно запустить на выполнение нашу программу. К сожалению, пока она еще ничего не рисует (основные сведения о выводе графических фигур будут даны в § 5.5). Однако новые разделы меню появятся на экране. Их можно выбирать. В подменю Figure можно установить пункты Rectangle и Circle в состоянии Checked (отмечается галочкой) и Unchecked. Сначала пункт out в подменю Zoom будет затенен. При первом же выборе пункта in пункт out становится активным. Если выбрать пункт in больше десяти раз, то он затеняется.

Рассмотренная программа очень простая, но уже позволяет выполнять простейшие операции с меню. Мы будем усложнять ее в последующих параграфах этой главы.

5.5. Графика

Когда все окно или его часть должны быть перерисованы, система **Windows** посылает сообщение **WM_PAINT**, которое может быть перехвачено прикладной программой. Точкой отсчета для координатной системы окна, используемой при рисовании фигур, является левый верхний угол с горизонтальной координатой $x=0$ и вертикальной координатой $y=0$. Наиболее часто положительные приращения координат x и y осуществляются вправо и вниз.

Все фигуры и текст отображаются на экране в графическом режиме с использованием средств графического интерфейса **GDI** (Graphical Device Interface). При выводе любой графической информации необходимо использовать специальную структуру данных, называемую контекст от устройства (device context). Она позволяет преобразовывать независимые от устройств (таких, как дисплей) вызовы графических функций GDI в последовательность

операций, которые необходимо приложить ко входу соответствующего физического устройства.

Если прикладная программа получает сообщение WM_PAINT, то автоматически вызывается функция CView::OnDraw(), которая для нашего примера из § 5.4 будет иметь следующий вид:

```
void CComMesView::OnDraw(CDC* pDC)
{
    CComMesDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: добавьте код для рисования
}
```

В функцию передается указатель pDC на объект класса CDC, который поддерживает контекст устройства и включает множество функций для вывода текста и рисования различных фигур.

Указатель pDoc содержит адрес (возвращаемый функцией GetDocument()) документа, связанного с текущим объектом типа вид. Строка ASSERT_VALID(pDoc) проверяет, содержит ли pDoc разрешенный (действительный) адрес.

Для того чтобы вычерчивать прямоугольник и окружность в рабочей области нашего окна на экране, функция OnDraw() может быть представлена в следующем виде:

```
void CComMesView::OnDraw(CDC* pDC)
{
    CComMesDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    if (pDoc->getElement() == RECTANGLE)
        pDC->Rectangle(10,10,
            20+pDoc->getZoom()*10,20+pDoc->getZoom()*10);
    if (pDoc->getElement() == CIRCLE)
        pDC->Arc(100,100,
            120+pDoc->getZoom()*10,120+pDoc->getZoom()*10,
            100,100,100,100 );
}
```

Здесь функция Rectangle позволяет вычертить прямоугольник, который задается координатами верхнего левого угла (в примере два аргумента со значениями 10 и 10) и нижнего правого угла (в примере третий и четвертый аргументы). Функция Arc позволяет вычертить окружность, которая вписывается в прямоугольник, заданный первыми четырьмя координатами. Поскольку эта же функция может использоваться для рисования дуги, то последующие координаты (в примере 100, 100, 100) задают точку начала (100, 100) и точку конца (100, 100) дуги. Поскольку мы хотим нарисовать полную окружность, то начальные и конечные координаты совпадают. Эта же функция Arc может использоваться и для рисования эллипса, если изменить квадрат, заданный пер-

выми четырьмя координатами, на прямоугольник, куда и будет вписываться эллипс (или его часть, если в координатах 5—8 записаны соответствующие значения).

Функция `getElement()` вызывается для того, чтобы определить тип фигуры (прямоугольник или окружность). Функция `getZoom` позволяет вычислить требуемый размер фигуры.

Если теперь выполнить нашу программу с измененной функцией `CComMesView::OnDraw()`, то она позволит рисовать окружности и прямоугольники на экране и изменять их размеры с помощью выбора соответствующих пунктов в разделах `Figure` и `Zoom` меню.

Приведем примеры некоторых других полезных функций для рисования:

`MoveTo` (горизонтальная координата, вертикальная координата) — перемещение текущей точки отсчета в заданную позицию. Например, вызов `pDC->MoveTo(10, 20);` — перемещает точку отсчета в позицию $x=10$, $y=20$. Здесь мы предполагаем, что по умолчанию задан режим вывода `MM_TEXT`, в котором координаты задаются в физических точках экрана. Например, при разрешении экрана 800×600 точек текущая позиция установится на 10 точек вправо и на 20 точек вниз относительно левого верхнего угла окна, которое является исходной точкой отсчета;

`LineTo` (горизонтальная координата, вертикальная координата) — рисование линии от текущей точки до заданной координатами этой функции точки и изменение текущей позиции на новую позицию. Например, вызов функции `pDC->LineTo(50, 80);` — рисует линию и устанавливает текущую позицию $x=50$, $y=80$.

Функции `MoveTo()` и `LineTo` имеют следующие формальные описания (прототипы);

```
CPoint MoveTo(int x, int y);  
CPoint MoveTo(POINT aPoint);  
BOOL LineTo(int x, int y);  
BOOL LineTo(POINT aPoint);
```

где тип `POINT` определяется следующим образом:

```
typedef struct tagPOINT  
{  
    LONG x;  
    LONG y;  
} POINT;
```

Объект класса `CPoint` содержит координаты старой позиции текущей точки отсчета в своих данных x и y типа `LONG`. Функция `LineTo` возвращает значение `TRUE` при успешном завершении (при рисовании линии) и значение `FALSE` в противном случае.

Покажем теперь, как менять цвет и толщину линий, выводимых на экран. Для этого изменим нашу функцию OnDraw() следующим образом:

```
void CComMesView::OnDraw(CDC* pDC)
{
    CComMesDoc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    CPen my_Pen;
    my_Pen.CreatePen(PS_DOT,1,RGB(0,255,0));
    CPen* pOld = pDC -> SelectObject(&my_Pen);
    CPoint r = pDC -> MoveTo(150,100);
    pDC -> LineTo(200,150);
    pDC -> LineTo(180,200);
    pDC -> LineTo(r.x,r.y);
    pDC -> SelectObject(pOld);
    if (pDoc->gelElement() == RECTANGLE)
        pDC -> Rectangle(10,10,
            20+pDoc->getZoom()*10,20+pDoc->getZoom()*10);
    if (pDoc->gelElement() == CIRCLE)
        pDC -> Arc(100,100,
            120+pDoc->getZoom()*10,120+pDoc->getZoom()*10,
            100,100,100,100);
}
```

Строка CPen my_Pen; объявляет объект my_Pen класса CPen, который и задает тип элемента для рисования. Функция SelectObject(&my_Pen) разрешает использование нового рисовального элемента, указатель на который и передается в функцию. Эта функция возвращает указатель на старый рисовальный элемент, который сохраняется в переменной pOld типа CPen*.

Функция CreatePen задает параметры нового рисовального элемента и имеет следующее описание (прототип):

```
BOOL CreatePen(int aPenStyle, int aWidth, COLORREF aColor);
```

где aPenStyle — стиль линии, aWidth — толщина линии, aColor — цвет линии. Функция возвращает значение TRUE при успешном завершении и значение FALSE — в противном случае. Параметр стиль может принимать значения PS_SOLID (непрерывная линия), PS_DASH (штриховая линия), PS_DOT (пунктирная линия), PS_DASHDOT (штрихпунктирная линия), PS_DASHDOTDOT (штрих пунктирная линия с двойным пунктиром между штрихами), PS_NULL (пустая линия, т. е. ее не видно на экране), PS_INSIDEFRAME (специальная линия). Стили 2—5 действительны только при aWidth=1, где aWidth задает толщину в минимальных графических единицах изображения (по умолчанию в физических точках экрана). Цвет можно задать с помощью макро RGB с тремя параметрами, определяющими соответственно

интенсивность красного, зеленого и синего цветов в интервале от 0 до 255. В нашем примере мы выбрали максимальную интенсивность зеленого цвета и нулевую интенсивность красного и синего цветов.

Строка `pDC -> SelectObject(pOld);` восстанавливает старое значение рисовального элемента вывода прямоугольника и окружности.

После запуска программы на выполнение на экране появится пунктирная ломаная линия зеленого цвета вместе с нашими предыдущими фигурами. Заметим, что прямоугольник перекрывает линию, а окружность является прозрачной. Это происходит потому, что функция `Rectangle()` предполагает использование заливки, а функция `Arc` — нет. Если необходимо использовать заливку и для окружности, то следует выбрать функцию `Ellipse()`. В этом случае соответствующая часть функции `OnDraw()` может быть изменена следующим образом:

```
COLORREF old_Color =  
    pDC -> SetTextColor( RGB(200,30,80) );  
pDC -> TextOut(150,50,"new ellipse: ",  
    strlen("new ellipse: "));  
CBrush my_Brush(HS_CROSS,RGB(0,0,255));  
pDC -> SelectObject(my_Brush);  
if (pDoc->getElement() == RECTANGLE)  
    pDC -> Rectangle(10,10,  
        20+pDoc->getZoom()*10,20+pDoc->getZoom()*10);  
if (pDoc->getElement() == CIRCLE)  
{  
    pDC -> Ellipse(150,190,190,120);  
    pDC -> Arc(100,100,  
        120+pDoc->getZoom()*10,120+pDoc->getZoom()*10,  
        100,100,100,100);  
}
```

Здесь добавлены несколько новых функций:

функция `SetTextColor()` устанавливает новый цвет в соответствии с заданным параметром и возвращает старый цвет в объекте типа `COLORREF`;

функция `TextOut()` выводит текст, заданный в ее третьем аргументе, с координатами (левый верхний угол текста), заданными в первых двух аргументах (последний аргумент определяет длину выводимой строки);

функция `Ellipse()` рисует эллипс, вписанный в прямоугольник, который задан ее четырьмя аргументами (первые два определяют координаты левого верхнего угла прямоугольника, а оставшиеся два — координаты его правого нижнего угла).

Новый объект `my_Brush` типа `CBrush` позволяет задать цвет заполнения фигур, которые могут иметь заливку. В нашем примере такими фигурами являются прямоугольник и эллипс.

Строка `CBrush my_Brush(RGB(0, 0, 255));` задает сплошной заливатель синего цвета. Значение, которое передается в кон-

структур объекта типа CBrush, имеет тип COLORREF и, как выше, можно использовать макро RGB с максимальной интенсивностью синего цвета для нашего примера. Другой конструктор с двумя параметрами позволяет задать тип заполнителя, например:

```
CBrush my_Brush(HS_CROSS, RGB(0,0,255));
```

Здесь аргумент HS_CROSS задает наполнитель из горизонтальных и вертикальных линий (другие типы можно найти в справочных материалах к системе программирования Visual C++), цвет которых определяется макро RGB (см. второй параметр).

Для использования заполнителя в программе необходимо его выбрать. Это делается в следующей строке:

```
pDC -> SelectObject(my_Brush);
```

После запуска программы на выполнение дополнительно выводится текст new ellipse и эллипс (при выборе опции Circle в подменю Figure). Прямоугольник и эллипс имеют заданный наполнитель. По аналогии можно использовать и другие графические функции, заданные в классе CDC. Для этого следует воспользоваться документацией, поставляемой вместе с системой программирования Visual C++. Вся эта документация доступна при работе с интегрированной средой Developer Studio.

5.6. Взаимодействие с устройствами

В этом параграфе мы покажем, как перехватить и обработать сообщения, поступающие от таких устройств, как манипулятор «мышь», клавиатура и таймер. Все эти сообщения можно найти в окне Message после активизации инструмента ClassWizard. Для этого в окне Class name необходимо задать имя класса, для которого мы будем обрабатывать сообщения (для нашего примера следует задать класс CComMesView). Все рассматриваемые сообщения являются стандартами и поэтому имеют префикс WM_.

Обработку сообщений, поступающих от устройств, рассмотрим на простом примере. Предположим, что надо выполнить определенные действия при нажатии клавиш клавиатуры, при нажатии кнопок и при перемещении «мыши», а также при поступлении определенных сигналов от таймера.

Для этого в окне Messages нам будут нужны следующие сообщения:

WM_KEYDOWN — которое возникает при нажатии клавиши клавиатуры;

WM_KEYUP — которое возникает при отпускании клавиши клавиатуры;

WM_LBUTTONDOWN — которое возникает при нажатии левой кнопки «мыши»;

WM_LBUTTONDOWNBLCLK — которое возникает при двойном нажатии левой кнопки «мыши»;

WM_RBUTTONDOWN — которое возникает при нажатии правой кнопки «мыши»;

WM_RBUTTONDOWNBLCLK — которое возникает при двойном нажатии правой кнопки «мыши»;

WM_MOUSEMOVE — которое возникает при перемещении «мыши»;

WM_TIMER — которое возникает по сигналам от таймера.

Для каждого из этих сообщений установим функцию-обработчик. Для этого надо выбрать сообщение указателем «мыши» и добавить соответствующую функцию (можно просто выбрать сообщение и дважды нажать левую кнопку «мыши»).

Теперь необходимо задать инструкции в теле новых функций. Предположим, что они следующие:

```
void CComMesView::OnKeyDown(UINT nChar, UINT nRepCnt,
                             UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
    CClientDC my_DC(this);
    char ss[40];
    if (nChar == 'K') KillTimer(1);
    if (nChar == 'C') { KillTimer(1); Invalidate(); }
    wsprintf(ss, "keyboard: key pressed: %c", nChar);
    my_DC.TextOut(200, 100, ss, strlen(ss));
    // AfxMessageBox(ss, MB_OK);
    CView::OnKeyDown(nChar, nRepCnt, nFlags);
}

void CComMesView::OnKeyUp(UINT nChar, UINT nRepCnt,
                           UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
    RECT my_r = {200, 100, 500, 130};
    InvalidateRect(&my_r);
    CView::OnKeyUp(nChar, nRepCnt, nFlags);
}

void CComMesView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CClientDC my_DC(this);
    char ss[15];
    wsprintf(ss, "x=%d, y=%d", point.x, point.y);
    my_DC.TextOut(point.x, point.y, ss, strlen(ss));
    CView::OnLButtonDown(nFlags, point);
}
```

```

void CComMesView::OnRButtonDbtClk(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    InvalidateRect(NULL);

    CView::OnRButtonDbtClk(nFlags, point);
}

void CComMesView::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    //
    gettimeofday(
        char ss[256], t[20];
        _strtime(t);
        sprintf(ss, "Timer On -> Time: %s ", t);
        CClientDC my_DC(this);
        my_DC.TextOut(10, 200, ss, strlen(ss));
        CView::OnTimer(nIDEvent);
    }

void CComMesView::OnLButtonDbtClk(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    char ss[256], t[20];
    _strdate(t);
    sprintf(ss, "Date: %s", t );
    CClientDC my_dc(this);
    my_dc.TextOut(50, 10, ss, strlen(ss));
    CView::OnLButtonDbtClk(nFlags, point);
}

void CComMesView::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    CClientDC my_DC(this);
    char ss[25];
    sprintf(ss, "x=%d, y=%d ", point.x, point.y);
    my_DC.TextOut(200, 10, ss, strlen(ss));
    CView::OnMouseMove(nFlags, point);
}

void CComMesView::OnRButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    SetTimer(1, 500, NULL);
    CView::OnRButtonDown(nFlags, point);
}

```

В приведенных выше функциях задаются следующие аргументы:

nChar — виртуальный код, присвоенный нажатой (отпущенной) клавише;

nRepCnt — число повторений, являющееся результатом нажатия клавиши и удерживания ее в таком состоянии в течение определенного времени;

nFlags — для функций клавиатуры позволяет определить тип нажатой клавиши и получаемый от нее код. Например, в битах 0—7 содержится код сканирования клавиши (scan code);

nFlags — для функций «мыши» содержит набор флажков (признаков) статуса, которыми могут быть: MK_CONTROL (если нажата клавиша Ctrl), MK_LBUTTON (если нажата левая кнопка «мыши»), MK_MBUTTON (если нажата средняя кнопка «мыши»), MK_RBUTTON (если нажата правая кнопка «мыши»), MK_SHIFT (если нажата клавиша Shift). Проверить установку конкретного бита в параметре nFlags можно, например, так: if (nFlags & MK_SHIFT)...;

point — объект класса CPoint, хранящий горизонтальную и вертикальную координаты указателя «мыши»;

nIDEvent — определяет идентификатор таймера.

В обработчиках сообщений использованы следующие новые функции и классы:

CClientDC — производный класс от CDC, который содержит контекст устройства, представляющего рабочую область пользовательского окна, т. е. ту часть окна, в которую мы обычно и выводим информацию. В конструктор этого класса передается указатель this на текущий интерфейсный объект;

SetTimer() — функция установки таймера. Ее первым аргументом является идентификатор таймера, который не может принимать нулевое значение. Идентификатор задается потому, что в программе можно использовать несколько таймеров. Вторым аргументом задает периодичность выдачи сообщений от таймера в миллисекундах. Третий аргумент — это указатель на функцию, которая должна быть вызвана (если аргумент равен NULL, то посылается сообщение WM_TIMER);

KillTimer() — функция сброса (остановки) таймера, идентификатор которого задается ее единственным аргументом;

Invalidate() — функция, очищающая рабочую область окна;

wsprintf() — функция записи в строку. Она аналогична функции printf, но выводит информацию не на экран, а в строку, заданную в виде первого аргумента;

AfxMessageBox() — функция, позволяющая выводить окно сообщений. Первый аргумент задает сообщение, которое будет выведено, а второй аргумент указывает на то, что окно сообщений должно иметь кнопку OK;

InvalidateRect() — функция очистки области окна, заключенной в прямоугольник, указатель на который передается в виде параметра. В нашем примере мы определили и инициализировали прямоугольник как объект класса RECT. Если задать в качестве параметра рассматриваемой функции NULL, то очищается вся рабочая область;

_strtime() и _strdate() — это функции, копирующие в буфер, указатель который задан в виде параметра, текущие время и дату.

Заметим, что в каждом обработчике сообщения наряду с новыми операциями вызывается и соответствующая системная функция, например:

```
CView::OnKeyDown(nChar, nRepCnt, nFlags);
```

Строки такого вида добавляются автоматически и позволяют выполнить операции, которые стандартно предусмотрены соответствующими классами MFC.

При нажатии любой клавиши клавиатуры автоматически вызывается функция `OnKeyDown`, которая выполняет следующие действия;

- если была нажата клавиша с символом K, то сбрасывается таймер с идентификатором 1 (все рассматриваемые символы являются латинскими);

- если была нажата клавиша с символом C, то сбрасывается таймер и очищается рабочая область;

- выводится текст «keyboard: key pressed:» и виртуальный символ, соответствующий нажатой клавише. Если убрать символы комментария (//) в строке `AfxMessageBox(ss, MB_OK)`, то аналогичный текст будет выводиться в окне сообщений.

При отпускании клавиши автоматически вызывается функция `OnKeyUp` и текст, выведенный предыдущей функцией, исчезает с экрана.

При перемещении манипулятора «мышь» автоматически вызывается функция `OnMouseMove()` и текущие координаты ее указателя выводятся в рабочую область.

При нажатии левой кнопки «мыши» автоматически вызывается функция `OnLButtonDown()` и текущие координаты указателя выводятся рядом с самим указателем.

При нажатии правой кнопки «мыши» автоматически вызывается функция `OnRButtonDown()` и устанавливается таймер, периодически посылающий сообщение `WM_TIMER` (через 500 миллисекунд). При этом периодически вызывается функция `OnTimer()`, которая выводит в рабочую область окна текущее время. Напомним, что таймер останавливается при нажатии клавиши K или C. В последнем случае дополнительно очищается экран.

При двойном нажатии левой кнопки «мыши» автоматически вызывается функция `OnLButtonDbClick()`, которая выводит текущую дату в рабочую область окна экрана.

При двойном нажатии правой кнопки «мыши» автоматически вызывается функция `OnRButtonDbClick()`, которая очищает рабочую область окна. Заметим, что поскольку функции `Invalidate()` и `InvalidateRect()` приводят ксылке сообщения `WM_PAINT`, то часть рабочей области окна, которая рисуется в функции `OnDraw()`, не очищается (перерисовывается).

По аналогии с рассмотренным выше можно строить и другие функции обработки сообщений, генерируемых как в программе, так и в результате взаимодействия с физическими устройствами.

5.7. Ресурсы

Ресурсами программы пользователя являются ускорители (Accelerator), диалоговые окна (Dialog), иконки (Icon), меню (Menu), таблицы строк (String Table), панели инструментов (Toolbar) и т. п. Некоторые из них, такие как ускорители (задание активных символов, типа *i* в подменю Figure), таблицы строк (добавление строки в нижнюю строку статуса) и меню, мы уже рассматривали в §5.4. В этом параграфе покажем, как добавить в программу панели инструментов, иконки и диалоговые окна.

Для того чтобы добавить панель инструментов, необходимо установить левое окно в режим ResourceView и раскрыть соответствующий ресурс (Toolbar). Если выбрать появившийся идентификатор IDR_MAINFRAME двойным нажатием левой кнопки «мыши», то в правом окне появится редактор соответствующего ресурса. Любая кнопка, включенная в панель инструментов, представляется графически в виде матрицы точек экрана, которая изображает картинку, соответствующую этой кнопке. При необходимости можно выбрать новую кнопку и нарисовать ее с помощью средств встроеного графического редактора (редактора битовых изображений). Если выбрать новую кнопку указателем «мыши», для чего, как обычно, следует дважды нажать ее левую клавишу, то на экране появляется диалоговое окно Properties (свойства), которое покажет назначенный кнопке идентификатор.

Если добавить кнопки всех новых пунктов меню, таких как Rectangle и Circle (в подменю Figure), а также in и out (в подменю Zoom), то после запуска программы на выполнение они появятся в верхней строке управления и их можно выбирать так же, как и соответствующие пункты меню. Для каждой из этих кнопок можно задать подсказки (tooltip), которые появляются в маленьком прямоугольнике около указателя «мыши» при помещении его на соответствующую кнопку. Делается это очень просто. Сначала в левом окне, установленном в режим ResourceView, необходимо выбрать ресурс String Table. В нем надо найти строки, поставленные в соответствие новым пунктам меню. Для нашего примера это: ID_FIGURE_RECTANGLE, ID_FIGURE_CIRCLE, ID_ZOOM_IN и ID_ZOOM_OUT. В столбце Caption для этих строк записывается подсказка, которая появляется в нижней линии статуса при выборе соответствующих пунктов меню курсором манипулятора «мышь», а также при выборе рассмотренных выше

новых кнопок панели управления. Для вывода подсказки надо продолжить нужную строку символами \n и дописать после них текст, который мы хотим использовать в виде подсказки. Предположим, что при увеличении изображения (режим ID_ZOOM_IN) мы хотим использовать подсказку Zoom in, а в линии статуса ранее уже выводилась строка Zoom in figure. Тогда в старую строку вида:

ID_ZOOM_IN Zoom in figure,

которая содержится в таблице строк, надо дописать подстроку \nZoom in. В результате мы получим следующую новую строку в таблице строк:

ID_ZOOM_IN Zoom in figure\nZoom in

После этого при перемещении указателя манипулятора «мышь» на кнопку панели инструментов, соответствующую режиму увеличения размера фигуры, рядом с этим указателем появится маленький прямоугольник с текстом Zoom in.

Для создания новой иконки, которая будет присоединена к нашей программе, надо раскрыть ресурс Icon в левом окне (режим ResourceView) и отредактировать изображение иконки (или иконок), заданное по умолчанию. Для этих целей опять же используется встроенный графический редактор для битовых матриц.

Рассмотрим теперь построение и присоединение к программе диалоговых окон.

Построение диалогового окна осуществляется с помощью редактора ресурсов. Для этого необходимо в режиме ResourceView левого окна переместить указатель «мыши» на имя Dialog и нажать ее правую кнопку. В результате появляется новое подменю. Если выбрать пункт Insert Dialog (вставить диалог) этого подменю, то на экране отображается новое диалоговое окно, которое можно редактировать, т. е. изменять его размер, добавлять средства управления и т. п. К средствам управления относятся кнопки различного типа, текстовые окна и т. п. Они могут быть выбраны «мышью» в появившемся меню Controls и помещены в нужное место диалогового окна. Расположение средств контроля в диалоговом окне можно выравнивать с помощью полезного всплывающего меню Layout. В остальном присоединение ресурса осуществляется по аналогии с тем, что уже было рассмотрено.

Различают два типа диалоговых окон: модальные (Modal) и немодальные (Modeless). Окно первого типа всегда выводится поверх остальных окон, при этом оно становится активным, а

работа с остальными окнами той же программы приостанавливается. По окончании работы с модальным окном его необходимо закрыть для перехода к другому окну той же самой программы. Немодальные диалоговые окна позволяют перейти в другие окна той же самой программы, не заканчивая работу с ними. Мы рассматриваем здесь только модальные диалоговые окна.

Для использования диалогового окна в программе необходимо создать новый класс, который будет поддерживать работу с этим окном. Это можно сделать с помощью инструмента ClassWizard.

Сначала надо построить новое диалоговое окно (например, с именем COur_Dialog) с помощью редактора ресурсов. Далее можно поместить на новое окно указатель «мыши», нажать ее правую кнопку и в появившемся меню выбрать опцию ClassWizard... . Далее, как и выше, надо отвечать на поставленные вопросы.

Покажем для примера, как создать диалоговое окно, содержащее группу из трех взаимоисключающих кнопок (radio button), которые будут задавать один из трех возможных цветов (Red — красный, Green — зеленый и Blue — синий) для текста new ellipse в предыдущей программе. Для этого выполним следующие действия:

1. Построим диалоговое окно, в которое поместим три взаимоисключающие кнопки с именами Red, Green и Blue. Эти кнопки поместим в одну группу, в которой активной может быть только одна из кнопок. При этом необходимо установить опцию Group в окне Properties только для первой кнопки, например, Red. Переключатель Group не надо устанавливать для остальных кнопок, поскольку они входят в ту же группу. В окне Properties зададим также идентификатор IDC_RED для первой кнопки с именем Red.

2. Установим все кнопки в вертикальный ряд и выровняем расстояние между ними. Для этого выделим их рамкой с помощью манипулятора «мышь» и выберем опции Layout->AlignControls->Left, Layout->Space Evenly->Down.

3. В наборе элементов управления Controls выберем элемент «группа» (Group) и очертим рамку вокруг наших кнопок. Назовем эту группу Color. Если теперь размер нашего диалогового окна будет недостаточен для расположения группы, то его можно изменить с помощью «мыши». При необходимости можно изменить порядок обхода кнопок с помощью клавиши табуляции. Для этого надо выбрать опции Layout->Tab Order.

4. С помощью инструмента ClassWizard добавим новую переменную m_color (текущий цвет) в наш диалоговый класс (COur_Dialog). Это делается с помощью опции Member Variables

для идентификатора нашей первой кнопки IDC_RED в окне ClassWizard.

5. Теперь наш класс COur_Dialog представится в следующем виде:

```
class COur_Dialog : public CDialog
{
public:
    COur_Dialog(CWnd* pParent = NULL);
    .....
enum { IDD = IDD_DIALOG1 };
int         m_color;
    .....
protected:
    virtual void DoDataExchange(CDataExchange* pDX);
    .....
};
```

Конструктор этого класса содержит единственную строку `m_color = -1;`

Функция `DoDataExchange()` готовится автоматически средствами ClassWizard и представляется в виде:

```
void COur_Dialog::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{AFX_DATA_MAP(COur_Dialog)
    DDX_Radio(pDX, IDC_RED, m_color);
    //}AFX_DATA_MAP
}
```

Все функции, имена которых имеют префикс DDX, производят обмен данными. Второй и третий аргументы любой такой функции являются соответственно идентификатором элемента управления (в примере IDC_RED) и именем переменной класса для этого элемента управления (в примере m_color). Значение m_color будет устанавливаться так, что для первой кнопки (Red) оно равно нулю, для второй кнопки (Green) — единице и для третьей кнопки (Blue) — двум. Другие элементы контроля имеют свою специфику, не совпадающую со взаимоисключающими кнопками. Поэтому для получения соответствующей информации следует воспользоваться справочными документами.

6. Для вывода диалогового окна на экран добавим опцию Color во всплывающее меню Figure (см. § 5.4). Текст новой функции, обрабатывающей соответствующее сообщение, будет примерно таким:

```
void CComMesDoc::OnFigureColor()
{
    COur_Dialog my_dlg;
```

```

        if (my_dlg.DoModal() == IDOK)
        {
            color = my_dlg.m_color;
            SetModifiedFlag();
            UpdateAllViews(NULL);
        }
    }
}

```

В ней создается интерфейсный объект `my_dlg` нашего диалогового класса `Cour_Dialog` и вызывается функция `DoModal()` для построения соответствующего интерфейсного элемента (для отображения на экране нашего диалогового окна). Если эта функция завершается успешно, то выполняются строки в фигурных скобках.

В класс `CComMesDoc` добавлены переменная `color` целого типа (с атрибутом `private`) и функция `GetColor()` с атрибутом `public`, представленная в следующем виде:

```

int CComMesDoc::GetColor()
{
    return color;
}

```

Функция `OnFigureColor()` сохраняет текущий цвет в переменной `color` и вызывает две следующие функции. Первая функция (`SetModifiedFlag()`) устанавливает признак того, что текущая информация в рабочем окне была изменена. Далее это приводит к выводу сообщения с предложением сохранить содержимое измененного документа. Вторая функция (`UpdateAllViews()`) рассматривалась выше.

7. Теперь осталось изменить функцию `OnDraw()`, которая может быть такой:

```

void CComMesView::OnDraw(CDC* pDC)
{
    .....
    COLORREF Old_Color;
    switch ( pDoc -> GetColor() )
    {
        case 0:
            Old_Color = pDC->SetTextColor(RGB(255,0,0));
            break;
        case 1:
            Old_Color = pDC->SetTextColor(RGB(0,255,0));
            break;
        case 2:
            Old_Color = pDC->SetTextColor(RGB(0,0,255));
    }
    pDC -> TextOut(150,50,"new ellipse: ",
        strlen("new ellipse: "));
    pDC -> SetTextColor(Old_Color);
    .....
}

```

Затем, после запуска программы на выполнение, можно выбрать новую опцию Color в подменю Figure. В результате появится наше диалоговое окно. В нем можно выбрать один из трех цветов. После нажатия клавиши OK в диалоговом окне это окно пропадает и изменяется цвет строки new_ellipse, отображаемой в рабочей области экрана.

Напомним, что для безошибочной компиляции необходимо добавить файлы заголовка в соответствующие описания. Так, файл ComMesDoc.cpp должен содержать строку # include «Our_Dialog.h».

По аналогии можно строить и другие программы, работающие с диалоговыми окнами.

В заключение покажем, как можно доработать примеры из главы 3, чтобы они выполнялись в среде **Windows**. Рассмотрим пример EX_17, который включает специфические графические функции системы программирования Borland и не может быть выполнен в среде Visual C++. Однако для того чтобы этот пример заработал, надо очень несущественно его изменить. Описание нового класса (назовем его сейчас PPoint) надо включить в файл заголовка типа .h (назовем его EX_17h). Этот файл будет примерно таким:

```
class PPoint
{ int x,y;
public:
    PPoint(int InitX,int InitY) { x = InitX; y = InitY; }
    PPoint operator++(void) { return PPoint(x++,y++); }
    PPoint operator-(PPoint my_p)
        { return PPoint(x-my_p.x,y-my_p.y); }
    int Getx() { return x; }
    int Gety() { return y; }
};
```

Он изменился очень несущественно.

Теперь необходимо добавить строки для рисования в функцию OnDraw(), которая будет выглядеть примерно так:

```
void CComMesView::OnDraw(CDC* pDC)
{
    .....
    PPoint my_point(10,200);
    CPoint old = pDC->MoveTo(0,0); // сохранение позиции
    for (int j=0; j<10; j++)
    {
        for (int i=0; i<100; i++)
        {
            ++my_point;
            pDC->MoveTo(my_point.Getx(),
                        my_point.Gety());
            pDC->SetPixel(my_point.Getx(),
                        my_point.Gety(),RGB(255,0,0) );
        }
    }
}
```

```

    }
    my_point = my_point-PPoint(60,110);
}
    pDC->MoveTo(old); // восстановление позиции
    .....
}

```

Здесь функция SetPixel() позволяет нарисовать точку с заданными в первых двух аргументах координатами и цветом, заданным в третьем аргументе. Теперь программа будет работать так же, как и в главе 3 для системы Borland, т. е. будет выводить в рабочей области нашего окна десять наклонных линий. Разница заключается только в том, что старая программа работала в среде ДОС, а новая программа работает в среде **Windows**. Класс PPoint может быть и таким:

```

class PPoint
{ int x,y;
public:
    PPoint(int InitX,int InitY) { x = InitX; y = InitY; }
    PPoint operator++(void) { return PPoint(x++,y++); }
    PPoint operator-(PPoint my_p)
        { return PPoint(x-my_p.x,y-my_p.y); }
    void put_point(CDC *p_CDC) {
        p_CDC->MoveTo(x,y);
        p_CDC->SetPixel(x, y,RGB(255,0,0)); }
};

```

Тогда по сравнению с программой в главе 3 почти ничего не меняется и в функции OnDraw() вместо строк

```

pDC->MoveTo(my_point.Getx(),my_point.Gety());
pDC->SetPixel(my_point.Getx(),my_point.Gety(),RGB(255,0,0));

```

будет записана строка

```

my_point.put_point(pDC);

```

По аналогии можно выполнить в среде **Windows** и другие программы, рассмотренные в настоящей книге.

ГЛАВА 6

ПРИМЕРЫ ПРОГРАММ НА ЯЗЫКАХ Си и Си++

6.1. Работа с манипулятором «мышь»

При работе на персональном компьютере часто используется манипулятор «мышь». Для организации взаимодействия с этим манипулятором необходим специальный драйвер. Загрузка драйвера осуществляется двумя путями. С помощью команды вида `DEVICE=` в файле конфигурации `CONFIG.SYS` или путем загрузки резидентной программы (например, в файле автозапуска `AUTOEXEC.BAT`). Поддержка работы манипулятора «мышь» осуществляется через функции прерывания `0x33`.

Рассмотрим сначала, как определить наличие драйвера и мыши в персональном компьютере. Для того чтобы проверить установку драйвера, необходимо просмотреть содержимое вектора прерывания `0x33`. Это можно сделать, вызвав функцию `0x35` прерывания `0x21`. Она позволяет получить вектор прерывания. На входе в регистре `AL` необходимо задать номер прерывания. В данном случае он равен `0x33`. На выходе в регистрах `ES:BX` будет содержаться вектор прерывания. Если этот вектор равен `0000:0000`, то драйвер мыши не установлен. Таким образом, проверка в языке Си может быть выполнена, например, так:

```
_AH = 0x35; // это номер функции
_AL = 0x33; // это номер прерывания для мыши.
geninterrupt (0x21); // это вызов прерывания 0x21
// если _BX == 0, то драйвер мыши не установлен
```

В некоторых компьютерах в векторе `0x33` записана точка входа в программу, которая сразу же содержит команду возврата. На языке Ассемблера эта программа будет выглядеть примерно так:

```
mouse_driver PROC far
    iret
mouse_driver ENDP
```

Машинным кодом ассемблерной команды `iret` является `0xCF` или в ассемблерной записи `OCFh`. Тогда для проверки

190

установки драйвера необходимо выполнить следующие действия:

```
_AH = 0x35; // это номер функции
_AL = 0x35; // это номер прерывания для мыши
geninterrupt (0x21); // это вызов прерывания 0x21
b_x = _BX; // сохранение регистра BX во внутренней переменной,
// поскольку значение этого регистра может быть изменено
asm {cmp byte ptr ES:[BX],0CFh
     jne short_hump}
b_x = 0;
short_jump:
if(b_x == 0) { puts("Не установлен драйвер \"мыши\"");
              exit(1); }
```

Здесь функция exit(1) обеспечивает завершение программы с кодом возврата 1.

Теперь необходимо убедиться в том, что манипулятор подключен к компьютеру. Чтобы это сделать, необходимо вызвать функцию 0 прерывания 0x33. Если манипулятор не подключен, то в регистре AX вернется значение 0. Рассмотрим фрагмент программы, который выполняет эти проверки.

```
_AX = 0;
geninterrupt(0x33);
a_x = AX;
if(a_x == 0) { puts("К ПЭВМ не подключена \"мышь\"");
              exit(1); }
```

Ниже приводится программа на языке Си, которая показывает перемещение курсора мыши и отображает в центре экрана ее текущие горизонтальную и вертикальную координаты (для выхода из программы нужно нажать правую кнопку манипулятора).

```
/* пример EX6_1 */
#include <conio.h> /* для функций clrscr, gotoxy */
#include <dos.h> /* для функции geninterrupt */
#include <stdio.h> /* для функций puts, printf */
#include <process.h> /* для функции exit */
#pragma inline /* для команды asm */
void main(void)
{ int old_x = 0, old_y = 0, c_x, d_x, b_x, a_x;
  /* проверка установки драйвера для манипулятора мышь */
  _AH = 0x35; _AL = 0x33;
  geninterrupt(0x21);
  b_x = _BX;
  asm { cmp byte ptr es:[bx],0CFh
       jne short_jump }
  b_x = 0;
short_jump:
  if(b_x == 0) { puts("Не установлен драйвер \"мыши\"");
                exit(1); }
  /* проверка установки манипулятора мышь */
  _AX = 0;
```

```

geninterrupt(0x33);
a_x = _AX;
if(a_x == 0) { puts("К ПЭВМ не подключена \"мышь\"");
exit(1); }
/* очистка экрана */
clrscr();
/* вывод курсора манипулятора мышь на экран */
_AX = 1;
geninterrupt(0x33);
/* ВЫВОД НА ЭКРАН ТЕКУЩИХ КООРДИНАТ МАНИПУЛЯТОРА МЫШЬ */
/*****
while( _AX = 3, geninterrupt(0x33), c_x = _CX,
d_x = _DX, _BX != 2)
{ if(c_x != old_x || d_x != old_y)
{ old_x = c_x; old_y = d_x; }
else continue;
/* скрытие курсора манипулятора мышь */
_AX = 2;
geninterrupt(0x33);
gotoxy(30,10); /* установка курсора в позицию (30,10) */
printf("Горизонтальная координата: %3d",c_x);
gotoxy(30,11); /* установка курсора в позицию (30,11) */
printf("Вертикальная координата: %3d",d_x);
/* вывод курсора манипулятора мышь на экран */
_AX = 1;
geninterrupt(0x33);
}
*****/
/* переустановка драйвера (повторная инициализация мыши) */
_AX = 0;
geninterrupt(0x33);
clrscr();
}

```

Библиотечная функция `clrscr()` осуществляет очистку экрана (см. приложение П.3). Рассмотрим используемые в программе функции прерывания 0x33.

Функция 0 проверяет, установлена ли мышь, переустанавливает драйвер и возвращает число кнопок манипулятора. На входе в регистре AX задается значение 0. На выходе будут следующие значения:

- AX = 0, если мышь не установлена;
- AX = -1, если мышь установлена;
- BX — число кнопок манипулятора (два для манипуляторов Microsoft и три для некоторых других манипуляторов).

Функция 1 показывает курсор мыши на экране дисплея. На входе в регистре AX задается значение 1. На выходе не возвращаются никакие значения.

Функция 2 скрывает курсор мыши. На входе в регистре AX задается значение 2. На выходе не возвращаются никакие значения.

Функция 3 возвращает текущую позицию мыши и состояние ее кнопок. На входе в регистре AX задается значение 3. На выходе будут следующие значения:

BX — состояние кнопок;
CX — горизонтальная координата;
DX — вертикальная координата.

С другими функциями прерывания 0x33 можно познакомиться в руководстве по операционной системе MS-DOS, например в книгах [7, 13, 18, 29].

Ниже приводится аналогичная программа, которая работает в графическом режиме. В начале программы с помощью функции 0xF прерывания 0x10 определяется текущий режим работы дисплея, который сохраняется в переменной `old_video_mode`. Затем с помощью функции 0 прерывания 0x10 устанавливается новый режим с номером 0x10 (графический режим с разрешением экрана 640x350). Далее все действия с манипулятором мышь выполняются в этом режиме. Текст программы приведен ниже.

```
/* пример EX6_2 */
#include <conio.h>      /* для функций clrscr, gotoxy */
#include <dos.h>        /* для функции geninterrupt */
#include <stdio.h>      /* для функций puts, printf */
#include <process.h>    /* для функции exit */
#pragma inline         /* для команды asm */
void main(void)
{
    int old_x = 0, old_y = 0, c_x, d_x, b_x, a_x;
    unsigned char old_video_mode;
    /* проверка установки драйвера для манипулятора мышь */
    _AH = 0x35; _AL = 0x33;
    geninterrupt(0x21);
    b_x = _BX;
    asm { cmp byte ptr es:[bx],0CFh
          jne short_jump }
    b_x = 0;
short_jump:
    if(b_x == 0) { puts("Не установлен драйвер \\"мышь\\"");
                  exit(1); }
    /* проверка установки манипулятора мышь */
    _AX = 0;
    geninterrupt(0x33);
    a_x = _AX;
    if(a_x == 0) { puts("К ЛЭВИ не подключена \\"мышь\\"");
                  exit(1); }
    /* очистка экрана */
    clrscr();
    /* сохранение в переменной old_video_mode
       старого видеорежима */
    _AH = 0xF;
    geninterrupt(0x10);
    old_video_mode = _AL;
    /* установка графического режима */
    _AH = 0;
    _AL = 0x10;
    geninterrupt(0x10);
    /* вывод курсора манипулятора мышь на экран */
    _AX = 1;
    geninterrupt(0x33);
    /* ВЫВОД НА ЭКРАН ТЕКУЩИХ КООРДИНАТ МАНИПУЛЯТОРА МЫШЬ */
}
```

```

/*****
while( _AX = 3, geninterrupt(0x33), c_x = _CX,
      d_x = _DX, _BX != 2)
{ if(c_x != old_x || d_x != old_y)
  { old_x = c_x; old_y = d_x; }
  else continue;
/* сккрытие курсора манипулятора мышь */
_AX = 2;
geninterrupt(0x33);
gotoxy(30,10); /* установка курсора в позицию (30,10) */
printf("Горизонтальная координата: %3d",c_x);
gotoxy(30,11); /* установка курсора в позицию (30,11) */
printf("Вертикальная координата: %3d",d_x);
/* вывод курсора манипулятора мышь на экран */
_AX = 1;
geninterrupt(0x33);
}
/*****
/* переустановка драйвера (повторная инициализация мыши) */
_AX = 0;
geninterrupt(0x33);
/* восстановление старого видеорежима */
_AH = 0;
_AL = old_video_mode;
geninterrupt(0x10);
clrscr();
}

```

6.2. Программирование последовательного интерфейса

Персональный компьютер может взаимодействовать с внешним оборудованием с помощью последовательного и параллельного интерфейсов. При параллельной связи в любой момент времени можно переслать или получить байт данных по восьми различным линиям. В случае последовательной связи в каждый момент времени пересылается один бит по единственной линии.

Устройство, посылающее данные, называется *передатчиком*, а устройство, принимающее данные, — *приемником*. Имеется два типа последовательной связи: *синхронная связь* и *асинхронная связь*. В первом случае передаваемые и принимаемые данные синхронизируются импульсами от специального источника. Во втором случае импульсы синхронизации отсутствуют. Асинхронная связь нашла более широкое распространение и будет рассматриваться в этом разделе. В этом случае передаваемые и принимаемые данные делятся на следующие части:

- стартовый бит;
- биты данных (от 5 до 8);
- бит паритета;
- стоп-биты (1, 1.5 или 2).

Для того чтобы данные передавались и принимались правильно, они должны иметь одинаково организованные части на передающей и приемной сторонах. Кроме того, передатчик и приемник должны передавать и принимать данные с одинаковой скоростью. Поддержка работы асинхронного последовательного канала осуществляется специальной микросхемой UARN (Universal Asynchronous Receiver/Transmitter). В персональных компьютерах для этих целей обычно используется микросхема 8250. В составе этой микросхемы десять программно-доступных регистров, доступ к которым осуществляется через семь портов ввода/вывода. Большинство портов используется для инициализации и программирования UART и только некоторые из них для передачи данных. Дадим краткую характеристику каждому из программно-доступных регистров UART..

Регистр передачи данных THR (Transmitter Holding Register) будет доступен, когда бит 7 регистра управления LGR (Line Control Register) установлен в нуль. Этот регистр является базовым для последовательного интерфейса. Как известно, персональный компьютер без специальных средств расширения может иметь до четырех последовательных каналов, которые называются COM1, COM2, COM3 и COM4. Базовые номера портов для этих каналов хранятся по следующим адресам оперативной памяти:

- 0040:0000 — COM1 (обычно здесь записано значение 03F816);
- 0040:0002 — COM2 (обычно здесь записано значение 02F8M16);
- 0040:0004 — COM3 (обычно здесь записано значение 03E816);
- 0040:0006 — COM1 (обычно здесь записано значение 02E816).

Базовый адрес (например, 03F816 для COM1) и является адресом порта THR. Любой следующий программно-доступный регистр задается некоторым смещением относительно базового адреса. В регистр THR записываются те данные, которые должны быть переданы от передатчика к приемнику. В этот регистр разрешается записывать данные только тогда, когда бит 5 регистра статуса LSR (Line Status Register) установлен в единицу.

Регистр приема данных RDR (Receiver Data Register) доступен, когда бит 7 регистра LCR установлен в нуль. Этот регистр тоже задается базовым адресом (для него задано смещение 0). Получение данных из этого регистра может осуществляться, когда бит 0 LSR установлен в единицу.

Регистры установки скорости BRDL (Baud Rate Divisor Low — для младшего байта) и BRDH (Baud Rate Divisor High — для старшего байта). Доступ к обоим регистрам осуществляется, когда бит 7 LCR установлен в единицу. В этом

случае BRDL имеет смещение 0 и BRDH — 1. Требуемая скорость вычисляется по следующей формуле:

$$\text{значение_в_BRD} = \frac{\text{скорость работы UART}}{16 * \text{скорость_приема передачи}} :$$

Скорость работы UART всегда постоянна и равна 1.8432 МГц. Тогда для того, чтобы обеспечить скорость приема/передачи 9600 бит в секунду, получим следующее значение в BRD:

$$\text{значение_в_BRD} = \frac{1843200}{16 * 9600} = 12 = C16.$$

Регистр разрешения прерываний IER (Interrupt Enable Register) имеет смещение 1 и доступен, когда бит 7 LCR установлен в нуль. Этот регистр задает типы прерываний, которые будут возникать в персональном компьютере. Для задания типов прерываний используются четыре младших бита этого регистра:

- если бит 0 установлен в единицу, то прерывание возникнет при наличии данных в RDR;

- если бит 1 установлен в единицу, то прерывание возникнет, если THR пуст;

- если бит 2 установлен в единицу, то прерывание возникнет при наличии ошибки;

- если бит 3 установлен в единицу, то прерывание возникнет при наличии изменений в регистре статуса модема MSR (Modem Status Register).

Регистр идентификации прерываний IIR (Interrupt Identification Register) имеет смещение 2. Содержимое этого регистра используется для установления типа прерывания генерируемого UART. Когда возникает прерывание, можно проверить содержимое младших трех битов этого регистра и определить тип прерывания:

- если бит 0 в 1, то произошло более одного события, вызывающего прерывание;

- если биты 2, 1 имеют значение 00, то регистр MSR был изменен;

- если биты 2, 1 имеют значение 01, то регистр THR пуст;

- если биты 2, 1 имеют значение 10, то в RDR есть данные;

- если биты 2, 1 имеют значение 11, то произошла ошибка.

Регистр управления LCR (Line Control Register) имеет смещение 3 и используется для установки различных параметров обмена:

- если биты 1, 0 имеют значение 00, то данные имеют длину 5 бит;

- если биты 1, 0 имеют значение 01, то данные имеют длину 6 бит;

- если биты 1, 0 имеют значение 10, то данные имеют длину 7 бит;

- если биты 1, 0 имеют значение 11, то данные имеют длину 8 бит;

- если бит 2 имеет значение 0, то 1 стоп-бит;

если бит 2 имеет значение 1, то 1.5 стоп-бита для данных длиной 5 и 2 стоп-бита в противном случае;
если бит 3 имеет значение 1, то вырабатывается бит паритета, если значение 0, то бит паритета не вырабатывается;
если бит 4 имеет значение 0, то используется контроль на нечетность (нечетный паритет), если 1, то контроль на четность (четный паритет);
бит 5 используется для управления паритетом;
бит 6 задает тип приостановки;
бит 7 позволяет устанавливать доступ к другим портам.

Регистр управления модемом MCR (Modem Control Register) имеет смещение 4 и используется для управления модемом. Этот регистр используется также при установке порта для работы в режиме прерываний. Рассмотрим возможные значения битов этого регистра:

если в бите 0 единица, то вырабатывается сигнал DTR (Data Terminal Ready). Этим сигналом компьютер информирует модем, что включено питание и он готов для организации связи;

если в бите 1 единица, то вырабатывается сигнал RTS (Request To Send). Этим сигналом компьютер информирует модем, что он хочет переслать данные;

бит 2 определяет дополнительный выход 1. Он должен устанавливаться в ноль. Этот бит может быть использован в будущем, если возможности модемов будут расширены;

бит 3 определяет дополнительный выход 2. Он устанавливается в единицу только тогда, когда для поддержки работы последовательного канала используются аппаратные прерывания (векторы 0xВ и 0xС);

бит 4 устанавливается в единицу для специальной проверки микросхемы UART. В этом случае передаваемые UART сигналы возвращаются назад без каких-либо внешних проводных связей;

биты 7—5 должны быть всегда установлены в ноль.

Регистр статуса LSR (Line Status Register) имеет смещение 5 и показывает состояние последовательного канала. Его биты могут иметь следующие значения:

если в бите 0 значение 1, то данные доступны в RDR;

если в бите 1 значение 1, то произошла ошибка повторного чтения;

если в бите 2 значение 1, то произошла ошибка паритета;

если в бите 3 значение 1, то произошла ошибка из-за неправильной установки скорости на передающей и приемной сторонах;

если в бите 4 значение 1, то произошла остановка;

если в бите 5 значение 1, то регистр THR пуст;

если в бите 6 значение 1, то передающий регистр сдвига пуст.

Таким образом, если в этом регистре записаны единицы в 1-, 2-, 3- или 4-м битах, то это говорит о наличии ошибки.

Регистр статуса модема MSR (Modem Status Register) имеет смещение 6 и показывает состояние модема:

если бит 0 в единице, то вырабатывается сигнал CTS (Clear To Send). В этом случае модем информирует компьютер, что он готов для передачи данных;

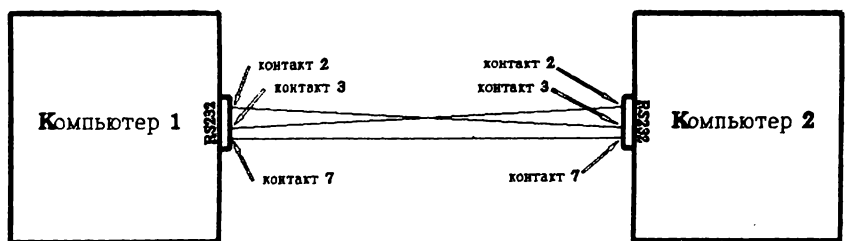


Рис. 6.1. Соединение двух компьютеров для организации последовательной связи

если бит 1 в единице, то вырабатывается сигнал DSR (Data Set Ready). В этом случае модем информирует компьютер, что включено питание и он готов к работе;

если бит 2 в единице, то вырабатывается сигнал RI (Ring Indicator). В этом случае модем информирует компьютер о наличии сигнала (звонка) от связанной с модемом телефонной линии;

если бит 3 в единице, то вырабатывается сигнал DCD (Data Carrier Detect). В этом случае модем информирует компьютер, что он связан с другим модемом;

биты 4—7 повторяют биты 0—3 (бит 4 CTS, бит 5 DSR, бит 6 RI, бит 7 DCD). Биты 0—3 устанавливаются только тогда, когда произошло изменение статуса с того времени, когда этот регистр последний раз был прочитан. Эти биты автоматически устанавливаются в нуль после последней операции чтения.

Рассмотрим примеры простых программ на языке Си для работы с последовательным каналом. Первая программа позволяет передавать данные между двумя компьютерами. Эти компьютеры должны быть соединены кабелем в соответствии с рис. 6.1.

Программа должна быть запущена одновременно на одном и другом компьютере. После запуска программы на экран дисплея будет выведено сообщение:

```
/* пример EX6_3 */
#pragma inline
#include <bios.h>
#include <stdio.h>
#include <conio.h>
/* скорость 9600 - 0xE, четный паритет - 0x18,
   один стоп-бит - 0x0, 8 бит данных - 0x3 */
#define INIT (0xE0 ; 0x18 ; 0x00 ; 0x03)
/* 0 - код команды для инициализации канала */
#define CMD_INIT 0
/* 3 - код команды для получения статуса канала */
#define CMD_STATUS 3
/* данные получены, если в нулевом бите (старшего) байта
   статуса записана 1 */
#define data 0x100
/* 27 - ASCII код клавиши ESC */
#define ESC 27
void main(void)
{ int no_ESC = 1, status, transmitted_data, COM;
```



```

puts("Укажите номер порта: 0 - COM1, 1 - COM2, ... ");
scanf("%d",&COM);
bioscom(CMD_INIT,INIT,COM); /* инициализация порта */
puts("Для завершения обмена нажмите клавишу ESC");
while(no_ESC)                { /* НАЧАЛО ЦИКЛА */
    status = bioscom(CMD_STATUS,0,COM); /* определение
                                        статуса канала */

    if(status & data)
        /* получение символа из канала */
        asm { push ds
                push ax
                push dx
                push bx
                mov ax,40h
                mov ds,ax
                mov bx,COM
                add bx,bx
                mov dx,[bx]
                in al,dx
                mov dl,al
                mov ah,2
                int 21h
                pop bx
                pop dx
                pop ax
                pop ds }

    if( kbhit() )                { /* начало if */
        if(( transmitted_data = getch() ) == ESC)
            no_ESC = 0;
        /* посылка символа в канал */
        asm { push ds
                push ax
                push dx
                push bx
                mov ax,40h
                mov ds,ax
                mov bx,COM
                add bx,bx
                mov dx,[bx]
                mov al,transmitted_data
                out dx,al
                pop bx
                pop dx
                pop ax
                pop ds }

        } /* конец if */
    } /* КОНЕЦ ЦИКЛА */
}

```

Укажите номер порта: 0—COM1, 1—COM2, ...

Введите цифру 0, 1, 2 или 3 в зависимости от того, через какой последовательный канал будут передаваться данные (вероятнее всего связь будет осуществляться через канал COM1). Далее можно набирать символы на клавиатуре на одном (любом) компьютере и они будут немедленно передаваться и появляться на экране дисплея в другом компьютере. В программе использованы вставки на языке Ассемблера и

новая библиотечная функция языка Си bioscom, которая пояснена в приложении П.3.

Следующая программа не использует библиотечные функции языка Си и полностью программирует регистры UART. Для этих целей использованы две ассемблерные процедуры: init — для начальной настройки (инициализации) UART и r_s — для передачи и приема данных.

```

/* пример EX6_4 */
#include <stdio.h>
#include <conio.h>
extern "C" void init(int,int);
extern "C" void r_s(int);
void main(void)
{ int COM;
  puts("Укажите номер порта: 0 - COM1, 1 - COM2, ... ");
  scanf("%d",&COM);
  puts("\nДля завершения обмена нажмите клавишу ESC\n");
  puts("Хотите использовать проверку с внутренним соединением\вывода UART на его вход (Y - да/N -нет)");
  init(COM*2,getch());
  r_s(COM*2);
  puts("\nПрограмма завершила работу. Установка последовательного\канала, сделанная в программе, не изменена");
}

.model SMALL
THR EQU 0
RDR EQU 0
BRDL EQU 0
BRDH EQU 1
IER EQU 1
IIR EQU 2
LCR EQU 3
MCR EQU 4
LSR EQU 5
MSR EQU 6
ESC_ EQU 27
data_size EQU 00000011b ; восемь бит данных
stop_bits EQU 00000000b ; один стоп-бит
parity EQU 00011000b ; проверка на четность
COMn EQU bx
my_set MACRO COM,serial_port_register,value
    mov dx,[COM]
    add dx,serial_port_register
    mov al,value
    out dx,al
ENDM
receive MACRO COM,serial_port_register
    mov dx,[COM]
    add dx,serial_port_register
    in al,dx
ENDM
PUBLIC C init, C r_s
.CODE
no_install DB 'Порт с этим номером не установлен ,10,13, '$'
; bx всегда содержит смещение для базового адреса порта
init PROC C USES BP AX DS DX BX

```

```

mov bp,sp
mov COMn,[bp+2+10]
mov ax,40h
mov ds,ax
cmp [COMn],0
je exit
my_set COMn,LCR,10000000b ; установка бита 7 LCR
my_set COMn,BRDH,0 ; установка скорости обмена
my_set COMn,BRDL,0Ch ; 9600 бит в секунду
; установка размера данных, числа стоп-бит и паритета
xor al,al
or al,data_size
or al,stop_bits
or al,parity
my_set COMn,LCR,al
; установка регистра разрешения прерываний
my_set COMn,IER,0
;*****
cmp [bp+4+10], 'y' ; здесь проверяется введенный*
jne _end_ ; символ и при положительном *
my_set COMn,MCR,00010011b ; результате задается*
; проверка UART внутренним соединением выхода на вход*
;*****
jmp short _end
exit: push cs
pop ds
lea dx,no_install
mov ah,9
int 21h
_end_: my_set COMn,MCR,00000011b
_end: ret
init ENDP
; процедура для приема и передачи данных
r_s PROC C USES BP AX DS DX BX
mov bp,sp
mov COMn,[bp+2+10]
mov ax,40h
mov ds,ax
again: receive COMn,LSR
test al,00011110b ; проверка байта статуса
jnz error_detect ; на наличие ошибки
test al,00000001b ; проверка на получение данных
jnz receive_data ; получены данные
test al,00100000b ; проверка на передачу данных
jz again ; нельзя передавать данные
; можно передавать данные: передача данных
mov ah,1 ; проверка наличия символа в буфере
int 16h
jz again ; если символа нет, то переход к again
xor ah,ah ; получение символа из буфера клавиатуры
int 16h
cmp al,ESC_
je stop ; выход при нажатии клавиши ESC
my_set COMn,THR,al ; передача символа в порт
jmp short again
receive_data:
receive COMn,RDR
cmp al,19 ; проверка наличия приостановки
je xoff
mov dl,al
mov ah,2
int 21h ; вывод полученного символа на экран

```

```

        jmp short again
error_detect: ; здесь выполняется обработка ошибок
xoff:       ; здесь выполняется обработка приостановки и
            ; ожидание сигнала хол
stop:       ret
r_s        ENDP
            END

```

Эта программа выполняет те же действия, что и предыдущая программа, поэтому работа с ней осуществляется точно так же. Дополнительно имеется возможность использования проверки работы программы путем внутреннего соединения выхода микросхемы UART на ее вход. Для этого следует установить в единицу бит 5 MCR (и это при необходимости делается в ассемблерной программе). С использованием конструкций языка Ассемблера можно дополнительно ознакомиться в книгах [4, 5].

Рассмотрим процедуры обмена по последовательному каналу, использующие аппаратные прерывания. Чтобы организовать работу по прерываниям, необходимо выполнить следующие действия:

- определить адреса портов, которые будут использованы;
- установить формат данных;
- установить скорость приема/передачи;
- установить значения DTR и RTS (о них говорилось выше при рассмотрении регистра управления модемом);
- установить обработчик прерывания и разрешить его вызов через аппаратный сигнал.

Три первых действия подробно рассматривались выше. Установка значений DTR/RTS является не обязательной. Оба этих значения устанавливаются в регистре MCR. Ниже приводится фрагмент ассемблерной программы, в котором выполняются эти действия.

mov dx, COMn; COMn — базовый адрес используемого канала

```

add dx, 4 ; выбор регистра MCR
in al, dx ; чтение байта из MCR
or al, 00000011b; включение DTR и RTS
out dx, al ; запись измененного байта в MCR

```

Для того чтобы включить аппаратные прерывания, необходимо выполнить несколько шагов;

- написать программу обработчика прерываний, который будет вызываться при определенных событиях в последовательном канале;

- установить в единицу бит 3 регистра MCR. После этого микросхема UART начнет вырабатывать сигналы прерываний. Для установки этого бита в единицу можно использовать следующий фрагмент ассемблерной программы:

```

mov dx,COMn
add dx,4
in al,dx
or al,00001000b
out dx,al

```

В большинстве практических задач этот шаг можно объединить с предыдущим, на котором устанавливались значения DTR и RTS;

разрешить аппаратные прерывания от последовательного канала. С этой целью необходимо сделать определенные установки в контроллере прерываний. В ПЭВМ IBM PC/AT два таких контроллера. На первый из них поступают следующие аппаратные сигналы, вызывающие прерывания: IRQ0 — сигнал от системного таймера; IRQ1 — сигнал от клавиатуры; IRQ2 — сигнал каскадирования от второго контроллера прерываний; IRQ3 — сигналы от каналов COM2/COM4; IRQ4 — сигналы от каналов COM1/COM3; IRQ5 — сигнал от второго принтера; IRQ6 — сигнал от контроллера гибкого диска; IRQ7 — сигнал от первого принтера. Этим аппаратным сигналам соответствуют векторы прерываний от 8 до 15 (от 8₁₆ до F₁₆). Для того чтобы сигналы от микросхемы UART обрабатывались в контроллере прерываний, необходимо установить в нуль соответствующие биты регистра маски этого контроллера. В нашем случае это бит 3 для каналов COM2/COM4 и бит 4 для каналов COM1/COM2. Это можно сделать с помощью следующего фрагмента ассемблерной программы:

```

in al,21h ;21h — номер порта регистра маски
and al,11100111b ; установка в нуль битов 3 и 4
out 21h,al ; пересылка измененного значения в порт

```

установить биты в регистре IER в соответствии с теми событиями, которые будем анализировать. В большинстве случаев необходимо вырабатывать аппаратное прерывание, когда в UART поступили новые данные. В этом случае необходимо установить в единицу только бит 0. Это можно сделать с помощью следующего фрагмента ассемблерной программы:

```

mov dx, COMn
inc dx ; теперь в dx адрес IER
mov al,00000001b ; установка в единицу бита 0
out dx,al ; пересылка значения 00000001b в порт

```

Рассмотрим пример программы, которая устанавливает аппаратное прерывание. Это прерывание будет вызываться (через вектор 0xB или 0xC в зависимости от номера канала), когда в канал поступают новые данные. Тексты программы

на языке Си и вызываемой из нее программы на языке Ассемблера приведены ниже.

Программа должна быть запущена на выполнение на двух ПЭВМ, связанных кабелем. В любую ПЭВМ можно вводить

```

/* пример EX6_5 */
#include <stdio.h>
#include <process.h>
#include <conio.h>
#define ESC 27
extern "C" void init(int); /* инициализация порта */
extern "C" void n_init(int); /* дополнительная инициа-
                             лизация порта */
extern "C" void send(int); /* посылка символов в порт */
extern "C" void set_int(int); /* установка программы, реа-
                             гирующей на аппаратное прерывание */
extern "C" void del_int(int,int); /* удаление программы,
                             реагирующей на аппаратное прерывание */
void main(void)
{ int COM,com_vect;
  puts("Укажите номер порта: 0 - COM1, 1 - COM2, ... ");
  scanf("%d",&COM);
  puts("Для завершения обмена нажмите клавишу ESC");
  switch(COM) {
    case 0: com_vect = 0xC; break;
    case 1: com_vect = 0xB; break;
    case 2: com_vect = 0xC; break;
    case 3: com_vect = 0xB; break;
    default: puts("Неправильно задан номер порта");exit(1); }
  init(COM*2);
  set_int(com_vect);
  n_init(COM*2);
  send(COM*2);
  del_int(com_vect,COM*2);
}

.model SMALL
THR EQU 0
RDR EQU 0
BRDL EQU 0
BRDH EQU 1
IER EQU 1
IIR EQU 2
LCR EQU 3
MCR EQU 4
LSR EQU 5
MSR EQU 6
ESC_ EQU 27
data_size EQU 00000011b ; восемь бит данных
stop_bits EQU 00000000b ; один стоп-бит
parity EQU 00011000b ; проверка на четность
COMn EQU bx
my_set MACRO COM,serial_port_register,value
    mov dx,[COM]
    add dx,serial_port_register
    mov al,value
    out dx,al
ENDM
receive MACRO COM,serial_port_register
    mov dx,[COM]
    add dx,serial_port_register
    in al,dx

```

```

        ENDM
PUBLIC C init, C send, C set_int, C del_int, C n_init
.CODE
com_vect DW ?
old_offset DW ?
old_segment DW ?
com_number DW ?
no_install DB 'Порт с этим номером не установлен',10,13,'$'
; bx всегда содержит смещение для базового адреса порта

; ОСНОВНАЯ ПРОЦЕДУРА ИНИЦИАЛИЗАЦИИ
init PROC C USES BP AX DS DX BX
    mov bp,sp
    mov COMn,[bp+2+10]
    mov cs:com_number,COMn ; сохранение удвоенного номера
                                ; порта в переменной com_number

    mov ax,40h
    mov ds,ax
    cmp [COMn],0
    je exit ; если по адресу COMn 0, то порт не установлен
    my_set COMn,LCR,10000000b ; установка бита 7 LCR
    my_set COMn,BRDH,0        ; установка скорости обмена
    my_set COMn,BRDH,0Ch      ; 9600 бит в секунду
; установка размера данных, числа стоп-бит и паритета
    xor al,al
    or al,data_size
    or al,stop_bits
    or al,parity
    my_set COMn,LCR,al
    jmp short _end
exit:  push cs
    pop ds
    lea dx,no_install
    mov ah,9
    int 21h
_end:  ret
init  ENDP

; ПРОЦЕДУРА ДЛЯ ПЕРЕДАЧИ ДАННЫХ
send PROC C USES BP AX DS DX BX
    mov bp,sp
    mov COMn,[bp+2+10]
    mov ax,40h
    mov ds,ax
again: receive COMn,LSR
    test al,00011110b ; проверка байта статуса
    jnz error_detect ; на наличие ошибки
    test al,00100000b ; проверка на передачу данных
; здесь может быть процедура выхода из программы после
; истечения некоторого заданного промежутка времени
    jz again ; нельзя передавать данные
; можно передавать данные: передача данных
    mov ah,1 ; проверка наличия символа в буфере
    int 16h
    jz again ; если символа нет, то переход к again
    xor ah,ah ; получение символа из буфера клавиатуры
    int 16h
    cmp al,ESC_
    je stop ; выход при нажатии клавиши ESC
    my_set COMn,THR,al ; передача символа в порт
    jmp short again
error_detect: ; здесь выполняется обработка ошибок
stop:  ret

```

send ENDP

; ПРОЦЕДУРА УСТАНОВКИ ОБРАБОТЧИКА ПЕРЕРЫВАНИЙ

set_int PROC C USES BP AX DS DX BX ES

 mov bp,sp

 mov ax,[bp+2+12]

 mov cs:com_vect,ax

; сохранение старого вектора прерываний в переменных

; old_segment:old_offset

 mov ah,35h

 int 21h

 mov cs:old_offset,bx

 mov cs:old_segment,es

; установка нового вектора прерываний, который будет

; вызывать новый обработчик прерывания my_int

 mov ax,cs:com_vect

 push cs

 pop ds

 mov dx,offset my_int

 mov ah,25h

 int 21h

 ret

set_int ENDP

; ДОПОЛНИТЕЛЬНАЯ ПРОЦЕДУРА ИНИЦИАЛИЗАЦИИ ДЛЯ

; ОРГАНИЗАЦИИ РАБОТЫ ПО ПЕРЕРЫВАНИЯМ

n_init PROC C USES BP AX DS DX BX

 mov bp,sp

 mov COMn,[bp+2+10]

 mov ax,40h

 mov ds,ax

 cli ; запрет внешних прерываний

; установка регистра MCR

 my_set COMn,MCR,00001011b

; снятие маски для входов IRQ3 и IRQ4 в контроллере

; прерываний (21h - порт этого контроллера)

 in al,21h

 and al,11100111b

 out 21h,al

; установка регистра IER

 my_set COMn,IER,00000001b

 sti ; разрешение внешних прерываний

 ret

n_init ENDP

; УДАЛЕНИЕ УСТАНОВЛЕННОГО ОБРАБОТЧИКА ПЕРЕРЫВАНИЙ

del_int PROC C USES BP AX DS DX BX

 mov bp,sp

 mov ax,[bp+2+10]

 mov cs:com_vect,ax

 mov COMn,[bp+4+10]

 mov ax,40h

 mov ds,ax

; установка регистра IER

 my_set COMn,IER,00000000b

; установка маски для входов IRQ3 и IRQ4 в контроллере

; прерываний (21h - порт этого контроллера)

 in al,21h

 or al,00011000b

 out 21h,al

; восстановление старого вектора прерываний

 mov ds,cs:old_segment


```

        mov dx,cs:old_offset
        mov ax,cs:com_vect
        mov ah,25h
        int 21h
        ret
del_int ENDP

; ОБРАБОТЧИК ПРЕРЫВАНИЯ my_int
my_int PROC FAR
        push bx
        push ax
        push dx
        push es
        push ds
        mov ax,40h
        mov ds,ax
        mov COMn,cs:com_number
; получение байта из регистра MCR
        receive COMn,MCR
; выключение сигнала RTS
        and al,11111101b
        out dx,al
; получение данных из регистра RDR
rec:    receive COMn,RDR
; установка в es сегментного адреса экранной памяти
        mov dx,0B800h
        mov es,dx
; вывод полученного символа в левый верхний угол экрана
        mov byte ptr es:[0],al
; чтение байта из регистра IIR
        receive COMn,IIR
; проверка, не было ли другого запроса на прерывание
        test al,00000001b
        jz rec ; был запрос (получение нового символа)
; передача кода завершения аппаратного прерывания
; в контроллер прерываний (20h - порт этого контроллера)
        mov al,20h
        out 20h,al
; чтение байта из регистра MCR
        receive COMn,MCR
; включение сигнала RTS
        or al,00000010b
        out dx,al
        pop ds
        pop es
        pop dx
        pop ax
        pop bx
        iret
my_int ENDP
END

```

символы с клавиатуры и они сразу же появляются в левом верхнем углу экрана дисплея. Завершение работы программы произойдет после нажатия клавиши «ESC». Программа работает следующим образом. Приходящий в канал новый символ сразу же вызывает аппаратное прерывание через вектор 0xB (для каналов COM2/COM4) или 0xC (для каналов

COM1/COM2). Этот символ выводится в левый верхний угол экрана дисплея (все символы выводятся последовательно в одну и ту же позицию экрана). Во время, когда прием символа из канала не производится, можно передавать символы в канал с помощью процедуры send. В это же время можно производить любые другие действия, например обрабатывать приходящие символы. При этом не надо следить за состоянием канала, так как индикация приходящего символа осуществляется аппаратным сигналом. Поступающие символы можно накапливать в специально организованном буфере (по аналогии с буфером клавиатуры). При этом можно выполнять параллельно накопление символов и их обработку (например, с помощью процедуры, подобной send).

6.3. Программирование параллельного интерфейса

Каждое параллельное устройство имеет имя LPTn (n = 1, 2, ...) и свой адаптер, управляемый тремя регистрами: выходных данных, статуса и управления. Найти адреса портов этих регистров очень просто. Предварительно нужно обратиться к началу области переменных BIOS, где хранятся соответствующие базовые адреса (двухбайтовые значения адресов: 0:0408₁₆ — для LPT1, 0:040A₁₆ — для LPT2, 0:40C16 — для LPT3, 0:040E₁₆ — для LPT4). Затем необходимо выбрать требуемый параллельный канал, например LPT1). Тогда значение, прочитанное по соответствующему адресу (в примере 0:0408₁₆), дает номер порта выходных данных. Добавляя к этому значению единицу, получим номер порта статуса; добавляя еще одну единицу, получим номер порта управления.

Каждый посылаемый байт (обычно это байт, посылаемый в принтер) передается через регистр выходных данных. Регистр статуса сообщает различную информацию (о принтере). Его биты принимают следующие значения:

— бит 0:1 — ошибка превышения времени, в течение которого принтер находится в состоянии занятости;

— биты 2—1 не используются;

— бит 3:0 — ошибка принтера;

— бит 4:1 — принтер связан с машиной;

— бит 5:1 — нет бумаги в принтере;

— бит 6:0 — принтер подтверждает прием символа;

— бит 7:0 — принтер занят.

В процессе обмена для правильной передачи данных анализируется состояние регистра статуса. Регистр управ-

ления устанавливает канал в исходное состояние и координирует вывод данных. Его биты принимают следующие значения:

- бит 0: кратковременное единичное значение воспринимается как стробирующий сигнал для вывода байта;
- бит 1:1 — автоматический перевод строки после возврата каретки;
- бит 2:0 — инициализация порта принтера;
- бит 3:1 — включает принтер в режим online (подключения к линии);
- бит 4:1 — разрешает аппаратное прерывание от принтера;
- биты 7—5 не используются.

Рассмотрим пример программы для работы с принтером.

Программа выводит на принтер строку символов, введенную с клавиатуры. Ввод осуществляется в программе на

```

/* пример EX6_6 */
#include <stdio.h>
#include <alloc.h>
extern "C" int myasm(unsigned char*);
void main(void)
{   unsigned char *str;
    str = (unsigned char *)malloc(100); /* выделение
                                        памяти для строки */
    puts("\nВведите строку, заканчивающуюся символом $");
    gets(str); /* ввод строки */
    myasm(str); /* вывод строки на принтер в про-
                рамме на языке АССЕМБЛЕРА */
    fprintf(stderr, "\n"); /* передача на принтер символов
                            возврата каретки и перевода строки */
}

.MODEL SMALL
.CODE
PUBLIC _myasm
_myasm PROC
    push bp
    mov     bp, sp
    mov     bx, [bp+4] ; в bx передается адрес строки
    pop     bp
    sub     ax, ax ; в ax заносится значение ноль
    mov     es, ax ; в es заносится значение ноль
    mov     dx, es: [408H] ; в dx номер порта выходного регистра
b:    mov     al, [bx] ; в al передается очередной символ
    cmp     al, '$' ; если символ $, то переход на метку
    je     rt ; rt для завершения программы
    out     dx, al ; символ в al пересылается в порт
    inc     dx ; к dx дважды прибавляется 1 и теперь
    inc     dx ; в dx номер порта регистра управления
    mov     al, 0DH ; в al код для стробирующего импульса
    out     dx, al ; выдача в порт стробирующего импульса
    mov     al, 0CH ; в al код для отмены строга
    out     dx, al ; отмена стробирующего импульса
    dec     dx ; в dx номер порта регистра статуса
no:    in     al, dx ; чтение состояния регистра статуса

```

```

;*****;
; Здесь следует проверить бит 3, и если он сигнализирует ;
; о наличии ошибки, то перейти к ее обработке ;
;*****;
    test al,80H      ; ожидание готовности принтера (анализ
    jz no            ; бита 7 в регистре статуса)
    inc bx           ; выбор следующего символа в строке
    dec dx           ; в dx номер порта выходного регистра
    jmp b            ; переход для вывода следующего символа
rt:    ret           ; возврат в программу на языке C
_myasm endp
END

```

языке Си, а вывод — в программе на языке Ассемблера, в которой не используются вызовы процедур обработки прерываний. В некоторых принтерах символы не печатаются до тех пор, пока не получен код возврата каретки, завершающей строку, или пока не передана полная строка. В связи с этим в конце программы на языке Си добавлено обращение к функции `fprintf`.

6.4. Резидентные программы

Резидентные программы загружаются в память и остаются в ней во время выполнения других программ. Наилучшим средством для написания резидентных программ безусловно является язык Ассемблера. Однако эти программы можно писать и на языках Си, Си++. Здесь будет показано, как это делается. Во многих случаях будут одновременно приводиться программы на языках Си, Си++ и аналогичные программы на языке Ассемблера.

Существует два основных типа резидентных программ: пассивные и активные. *Пассивные резидентные программы* загружаются в память и не выполняют никаких действий. Они могут быть активизированы только из другой программы. В ассемблере вызов пассивной резидентной программы осуществляется командой вида

```
Int < номер_прерывания >
```

Параметр `номер_прерывания` определяет номер вектора в начальной области оперативной памяти, который хранит адрес точки входа в пассивную резидентную программу. В языке Си аналогичные вызовы осуществляются через функции, например:

```
genInterrupt (номер_прерывания);
```

Действия этой функции практически эквивалентны действиям приведенной выше ассемблерной команды. *Активные резидентные программы* загружаются в память и

периодически активизируются, проверяя некоторые условия (например, нажата или нет некоторая клавиша).

Любая резидентная программа содержит две части: резидентную часть и часть, которая устанавливает резидентную часть и сохраняет ее в памяти. После завершения программы первая часть остается в памяти, а вторая (она обычно называется транзитной) — нет. Транзитная (вторая) часть решает следующие задачи:

- определяет, можно ли установить резидентную часть;
- устанавливает в выбранном векторе прерывания адрес входа в резидентную часть;
- освобождает всю неиспользуемую память;
- оставляет программу резидентной в памяти.

Ниже приводится пример пассивной резидентной программы на языке Си.

```
/* пример KX6_7 */
#include <stdio.h>
#include <dos.h>
#include <conio.h>
#include <process.h>
#pragma inline
#ifdef __cplusplus
    #define space_or_dots ...
#else
    #define space_or_dots
#endif
#define int200 200
void interrupt (*old_vector)(space_or_dots);
void interrupt my_int(space_or_dots)
{
    unsigned ah,bx,ds;
    ds = _DS;
    ah = _AH;
    bx = _BX;
    if(ah == 0) clrscr();
    if(ah == 1)
    {
        _BX = bx;
        _DS = ds;
again:
        asm {
            mov dl,byte ptr [bx]
            inc bx
            cmp dl,0
            je _exit
            mov ah,2
            int 21h
            jmp again
        }
    }
    _exit:
}

void main(void)
{
    old_vector = getvect(int200);
    if((*old_vector) != 0)
```

```

    { puts("Этот вектор уже используется");
      exit(1); }
    setvect(int200, my_int);
    keep(0, (_SS + (_SP/16) - _psp));
}

```

В этой программе две функции: `main` и `my_int`. Функция `main` содержит транзитную часть, а функция `my_int` — резидентную часть. Точка входа в резидентную часть устанавливается в векторе 200₁₀, который обычно свободен. В программе используется несколько новых библиотечных функций, которые рассматриваются ниже.

Функция `getvect` описана в файле `dos.h` и имеет следующий прототип:

```
void interrupt (*getvect(int int_N))();
```

Она читает значение вектора прерываний, определенного номером `int_N`, и возвращает его в виде четырехбайтового значения. Типовое использование этой функции в программах на языках Си и Си++ будет примерно таким:

```

#ifdef __cplusplus
#define space_or_dots ...
#else
#define space_or_dots
#endif
void interrupt (*old_vector)(space_or_dots);
...
old_vector = getvect (номер_вектора);
...

```

Здесь `old_vector` — это указатель на функцию, которая является обработчиком прерываний. Параметр `space_or_dots` — это либо пробел (для языка Си), либо многоточие (для языка Си++). В переменной `old_vect` будет сохранен адрес обработчика прерываний, вызываемый через вектор номер_вектора. Те же самые действия вызывает функция 0x35 (35h) прерывания 0x21 (21h). Для нее на входе в регистре АН задается номер функции (35h), в регистре AL — номер_вектора. На выходе в регистрах ES:BX будет возвращаться адрес соответствующего обработчика.

Функция `setvect` описана в файле `dos.h` и имеет следующий прототип:

```
void setvect(int int_N, void interrupt(*isr));
```

Она устанавливает четырехбайтовый адрес обработчика прерываний, заданный в переменной `isr`, в вектор с номером `int_N`. Типовое использование этой функции в программах на языках Си и Си++ будет примерно таким:

```

...

```

```

void Interrupt my_Int(space_or_dots)
{. . . . .}
. . . . .
setvect(номер_вектора, my_Int);
. . . . .

```

Те же самые действия вызывает функция 0x25 (25h) прерывания 0x21 (21h). Для нее на входе в регистрах DS:DX задаются сегмент и смещение точки входа в новый обработчик прерывания, в регистре AH — номер функции (25h), в регистре AL — номер_вектора. На выходе эта функция не возвращает никаких значений.

Функция keep описана в файле dos.h и имеет следующий прототип:

```
void keep (unsigned char status, unsigned size);
```

Эта функция возвращает управление DOS и при этом оставляет текущую программу резидентной в памяти. Переменная status содержит код возврата при передаче управления DOS. Переменная size содержит размер резидентной части в параграфах (один параграф равен 16 байт). Те же самые действия вызывает функция 0x31 (31h) прерывания 0x21 (21h). Для нее на входе в регистре AH задается номер функции (31h), в регистре AL — код возврата (status), в регистре DX — размер (size) резидентной части в параграфах.

После запуска программы на выполнение проверяется содержимое вектора 200. Если в нем записаны не нули, то этот вектор уже используется и программа завершает работу без установки резидентной части. В противном случае с помощью функции setvect устанавливается новый вектор, который будет адресовать резидентную часть. Затем выполняется функция keep, которая завершает программу и оставляет ее резидентом. Значением выражения

```
_SS + (_SP/16) — _psp
```

является размер резидентной части программы. Действительно, _SS — это начало сегмента стека, который всегда будет последним сегментом программы, _SS — _psp — это размер всех сегментов программы, кроме сегмента стека. Здесь _psp — сегментный адрес префикса программного сегмента, который всегда расположен в начале программы. Значение _SP определяет последний байт сегмента стека. Если поделить значение _SP на 16, то будет получен размер сегмента стека в параграфах.

Резидентная часть программы позволяет выполнить две функции с номерами 0 и 1 (эти значения передаются через регистр AH). Если задать AH = 0, то производится очистка

экрана. Если задать AH = 1, то на экран выводится строка, завершаемая нулевым байтом. Адрес начала строки должен быть передан через регистры DX (сегмент) и BX (смещение). Ниже приводится программа, которая вызывает установленный обработчик прерывания через вектор 200:

```

/* пример EX6_8 */
#include <dos.h>
#include <stdio.h>
#include <process.h>
#ifdef __cplusplus
#define space_or_dots ...
#else
#define space_or_dots
#endif
#define int200 200
void interrupt (*vector_200)(space_or_dots);
void main(void)
{ int i;
  char str[21];
  vector_200 = getvect(int200);
  if((*vector_200) == 0)
  { puts("Вектор 200 не установлен");
    exit(1); }
  puts("Укажите значение AH (0 или 1)");
  scanf("%d",&i);
  if(i == 1) {
    puts("Введите строку до 20 символов");
    scanf("%s",str);
    _BX = (unsigned)&str[0];
  }
  _DX = _DS;
  _AH = i;
  geninterrupt(200);
}

```

Сначала необходимо выполнить программу EX6_7, а затем EX6_8.

Ниже приводится еще одна программа на языке Ассемблера, которая полностью эквивалентна программе EX6_7:

```

; пример EX6_9
.model TINY
.stack ; по умолчанию размер стека 1 Кбайт
vector EQU 200
Env EQU 2ch
.code
use_v DB 'Этот вектор уже используется',10,13,0

; УСТАНОВЛИВАЕМЫЙ ОБРАБОТЧИК ПРЕРЫВАНИЯ
my_int: cmp ah,0 ; если AH=0, то очистка экрана
       jne next
       call cls_scr ; вызов процедуры очистки экрана
       jmp _end_
next: cmp ah,1 ; если AH=1, то вывод сообщения
     jne _end_
     call display_mes ; вызов процедуры вывода сообщения
_end_: iret

```


; ПРОЦЕДУРА ОЧИСТКИ ЭКРАНА

```
cls_scr PROC C USES ax bx cx dx
    mov ah,8 ; чтение ASCII кода символа и его атрибута
    xor bh,bh ; все выполняется для нулевой страницы
    int 10h
    mov bh,ah ; атрибут в регистр bh
    mov ah,6 ; выбор функции скроллинга
    xor al,al ; для очистки экрана
    xor cx,cx
    mov dh,24
    mov dl,79
    int 10h ; очистка экрана
    xor dx,dx
    xor bh,bh
    mov ah,2 ; выбор функции установки курсора
    int 10h ; установка курсора в левый верхний угол
    ret
cls_scr ENDP
```

; ПРОЦЕДУРА ВЫВОДА СООБЩЕНИЯ

```
display_mes PROC C USES ax bx dx ds
    mov ds,dx ; эту команду можно убрать, если пе-
                ; редавать данные сразу же через ds
again:    mov dl,byte ptr [bx]
            inc bx
            cmp dl,0
            je _exit
            mov ah,2 ; функция вывода символа на экран
            int 21h ; вывод символа на экран
            jmp again
_exit:    ret
display_mes ENDP
```

; УСТАНОВКА НОВОГО ВЕКТОРА ПРЕРЫВАНИЯ

```
set_vect:
    mov al,vector
    mov ah,35h
    int 21h
    mov ax,es ; в ES:BX возвращается вектор vector
    add ax,bx ; если ES+BX=0, то вектор не установлен
    cmp ax,0
    je set_v ; можно устанавливать новый вектор
    push cs ; передача сегмента выводимого сооб-
    pop dx ; щения в регистр dx
    lea bx,cs:use_v
    call display_mes
    mov ax,4c01h
    int 21h ; завершение программы с кодом ошибки 1
set_v:    mov ax,seg my_int
    mov ds,ax
    lea dx,my_int
    mov al,vector
    mov ah,25h ; функция установки нового вектора
    int 21h ; установка нового вектора
    mov ah,62h ; эта функция возвращает сегмент PSP
    int 21h ; в BX возвращается сегмент PSP
    mov ds,bx
    mov es,ds:[Env]
    mov ah,49h ; функция освобождения памяти
    int 21h ; освобождение памяти под окружение DOS
    mov dx,offset set_vect ; в dx - конец обработчика
    mov ax,seg set_vect ; в ax - сегмент обработчика
```

```

sub ax,bx ; в ax учитывается размер PSP
mov cl,4  ; в cl число бит для сдвига вправо
shr dx,cl ; сдвиг dx на 4 бита вправо (деление на 16). Теперь в dx размер
           ; вого обработчика в параграфах
inc dx    ; добавление одного параграфа
add dx,ax ; добавление размера PSP
xor al,al ; обнуление al
mov ah,31h ; функция завершения и сохранения
int 21h    ; резидентом
END set_vect

```

Чтобы проверить ее работу, необходимо после включения ПЭВМ сначала выполнить программу EX6_9 и затем программу EX6_8.

Рассмотрим основные правила написания активных резидентных программ. Они являются более сложными, чем пассивные программы. Операционная система MS-DOS является нереентерабельной (не повторно входимой) системой. Это означает следующее. Предположим, что вызвана некоторая сервисная функция DOS (например, некоторая функция прерывания 21h). В процессе ее выполнения происходит прерывание и запускается на выполнение другой обработчик, который снова вызывает эту же сервисную функцию n. В нереентерабельных системах такое повторное вхождение запрещено, и если это сделано, то может произойти нарушение нормальной работы системы. Поясним основные причины нарушения нормальной работы MS-DOS. Если функция ОС вызывается из резидентной программы в то время, когда выполняется другая функция, то возникают проблемы с использованием стека. Когда вызывается прерывание 21h, ОС использует один из трех системных стеков. Конкретный стек определяется номером группы функций прерывания 21h и он не может быть определен прерывающей программой. Когда выполняется функция ОС, то все временные данные и адрес возврата помещаются в стек для этой функции. Предположим, что некоторая из таких функций прерывается резидентной программой, которая затем снова вызывает эту же функцию. В результате регистры SS:SP снова устанавливаются в исходное состояние и адресуют начало того же самого стека. При этом каждое новое обращение к стеку будет разрушать старые данные. Функция ОС, которая вызвана из резидентной программы, будет работать правильно. Все проблемы будут возникать тогда, когда управление будет возвращено прерванной функции. В это время содержимое стека изменено, что приводит к нарушению нормальной работы системы. Использование активных резидентных программ может привести к возникно-

влению подобной ситуации. В результате необходимо предусматривать средства, обеспечивающие в этом случае нормальную работу операционной системы. Активная программа должна быть построена по специальным правилам. Рассмотрим сначала сервисные средства MS-DOS, которые могут быть использованы для обеспечения условий безопасности активной резидентной программы. К их числу относятся:

- флаг активности DOS, который размещается по адресу, возвращаемому функцией 0x34 34(h) прерывания 0x21 (21h);

- прерывание 28h, которое периодически вызывается, когда MS-DOS занята только выполнением функций 1,...,C16 прерывания 0x21 (21h). Отметим, что это недокументированное средство MS-DOS и использовать его не рекомендуется.

Перечислим основные правила, которые должны быть выполнены при разработке активной резидентной программы:

- запрещение передачи управления собственному обработчику прерываний (собственной активной резидентной программе) во время выполнения дисковых операций (прерывание 13h). В некоторых случаях передача управления также должна быть запрещена во время работы с видеомонитором (прерывание 10h) и с коммуникационным каналом (прерывание 14h);

- определение того факта, что ваш обработчик уже был загружен ранее для исключения повторной загрузки (это можно сделать через мультиплексное прерывание 2Fh);

- передача управления первоначальному обработчику прерываний, когда ваш обработчик выполнил возложенные на него функции (в некоторых случаях может быть исключение из этого правила);

- использование собственного стека, если размер данных, сохраняемых в стеке, превышает несколько слов;

- переключение префикса программного сегмента (program segment prefix — PSP) и области передачи данных (disk transfer area — DTA) в резидентной части программы;

- сохранение и восстановление фрагмента стека ОС в резидентной части программы.

Рассмотрим каждое из перечисленных средств и правил более подробно.

Флаг активности используется DOS для того, чтобы указать, когда она занята. Флаг активности имеет ненулевое значение, когда выполняется какая-либо функция DOS, и нулевое значение, когда никакая функция DOS не выполняется. Ниже приводится фрагмент ассемблерной программы,

который позволяет получать и проверять флаг активности DOS.

```
push es ; сохранение ES, поскольку он будет изменен
mov ah, 34h ; по адресу в ES:BX будет возвращен
int 21h ; байт с флагом активности DOS
mov al, es:[bx] ; в AL флаг активности DOS
pop es ; восстановление ES
cmp al, 0 ; если 0, то система свободна и безопасна
jne no_safe ; переход к no_safe — система небезопасна
```

Обычно адрес флага активности ОС получается и сохраняется в транзитной части программы (например, ниже в программе он сохраняется в четырехбайтовой переменной `DOS_flag_address`). Далее в резидентной части программы осуществляется проверка этого флага.

Прерывание 28h периодически активизируется, когда ОС выполняет функции 1—C16 прерывания 21h. Они обеспечивают ввод/вывод данных с консоли (клавиатура и дисплей) и с коммуникационного канала, а также вывод данных на принтер. Здесь тратится очень много времени на ожидание (например, ввода данных с клавиатуры). Так, при непосредственной работе в среде MS-DOS всегда активен командный процессор (`COMMAND.COM`). В это время он находится в состоянии ожидания ввода данных с клавиатуры с помощью функции A16 прерывания 21h. Эта функция является одной из наиболее «неприятных», поскольку выход из соответствующего обработчика произойдет только после нажатия клавиши «ВВОД». Таким образом, если вы что-то набрали на клавиатуре, но не нажимали клавишу «ВВОД» и затем активизировали резидентную программу, то вы прервали выполнение функции A16. Теперь если в резидентной части вы обратитесь к функциям ОС, имеющим общий стек с функцией A16, то почти наверняка вы нарушите нормальную работу системы. В связи со сказанным во многих случаях целесообразно переписать эту функцию (это сделано в приведенном ниже примере). Если выполняется программа типа `DOSEDIT.COM`, то ожидание ввода осуществляется с помощью функции 8 прерывания 21h. Особенностью этой функции, а также функций 1, 6, 7 является то, что при вводе расширенного кода с клавиатуры функция должна вызываться два раза. Это, в частности, учтено в приведенной ниже программе. Напомним, что расширенные коды формируются при нажатии клавиш управления перемещением курсора, функциональных клавиш и т. п.

Рассмотрим типовую процедуру использования прерывания 28h. В векторе 28h устанавливается адрес входа в

собственный обработчик, который на языке Ассемблера имеет примерно следующий вид:

```
new_28: cmp cs:TSR_active,TRUE ; если программа уже выполняется,  
        je exit_28             ; то выход из этого обработчика  
        cmp cs:my_req,TRUE     ; если нет запроса на выполнение,  
        jne exit_28           ; то выход из этого обработчика  
        cmp cs:flag_13,0       ; если производится работа с диском,  
        jne exit_28           ; то выход из обработчика  
        mov cs:TSR_active,TRUE ; установка флага активности  
                                ; это программы  
        call TSR_main          ; вызов всплывающей программы, которая  
                                ; выполняет все основные действия  
        mov cs:TSR_active,FALSE ; сброс флага активности  
        exit_28: jmp dword ptr cs:old_28 ; вызов старого обработчика
```

Здесь:

TSR_active — признак, который равен TRUE при выполнении всплывающей программы TSR_main. Его анализ позволяет предотвратить повторный запуск всплывающей программы:

my_req — запрос на выполнение всплывающей программы. Он может быть установлен, например, в новом обработчике прерывания 9 (нажатие клавиши), когда активизирована нужная клавиша или комбинация клавиш. Как только запускается всплывающая программа, этот признак должен быть сброшен в теле этой программы;

flag_13 — признак, имеющий ненулевое значение, если производится работа с диском (выполняется прерывание 13h). Он устанавливается в перехваченном обработчике прерывания 13h.

Обычно обработчик ОС, вызываемый через вектор 28h, содержит только ассемблерную команду iret. Но он может быть изменен и другими программами. Поэтому ваш обработчик, активизируемый через вектор 28h, должен всегда вызывать ранее установленный обработчик.

Заметим, что прерывание 28h используется почти всегда, когда рассматриваются активные резидентные программы (см., например, [7, 13, 29]). Однако следует учесть, что его можно применять только при полной уверенности в хорошей информированности о внутренних недокументированных структурах и переменных операционной системы. В противном случае результат будет почти всегда непредсказуем. В целом использование подобных недокументированных средств можно рассматривать как плохой прием в программировании. Ниже приводится пример активной резидентной программы, построенной другими способами.

Запрещение передачи управления собственному обработчику прерываний во время выполнения дисковых операций осуществляется довольно просто. Вводится некоторая переменная-флаг (в примере flag_13), имеющая сначала нулевое значение. При каждом вызове прерывания 13h она увеличивается на единицу (инкрементируется), после завершения прерывания она уменьшается на единицу (декрементируется). При этом оригинальный обработчик 13h должен быть заменен новым обработчиком следующего вида:

```
inc cs_flag_13      ; инкремент введенного флага
pushf               ; сохранение регистра флагов в стеке
call dword ptr cs:old_13 ; вызов старого обработчика
dec cs:flag_13      ; декремент введенного флага
sti                 ; разрешение аппаратных прерываний
retf 2              ; сохранение регистра флагов после выхода
```

Покажем теперь, как определить, что ваша резидентная программа уже установлена. Это можно сделать используя мультиплексное прерывание 2Fh. В транзитной части заменяется вектор 2Fh и вводится новая (неиспользуемая ранее) функция мультиплексного прерывания. Если вызывается эта функция, то после завершения мультиплексного прерывания возвращается определенный признак. Транзитная часть вызывает мультиплексное прерывание (до установки новой функции) и проверяет этот признак. В зависимости от его значения определяется, была ли установлена ранее резидентная часть. В этом случае можно использовать примерно такие фрагменты ассемблерной программы:

```
.code
; резидентная часть программы
install DB 'Эта программа уже установлена $@'
old_2F DW 0000,0000 ; здесь будет сохранен старый вектор
               ; мультиплексного прерывания

; НОВЫЙ ОБРАБОТЧИК ДЛЯ МУЛЬТИПЛЕКСНОГО ПРЕРЫВАНИЯ
new_2F: cmp ax,OFF00h ; проверка вызова нового обработчика
je CF_1              ; если равно, то программа уже установлена
clc                  ; сброс признака переноса CF
jmp dword ptr cs:old_2F ; если программа не установлена,
               ; то вызов старого обработчика
CF_1: stc             ; если программа установлена, то CF = 1
      retf 2          ; эта команда в отличие от iret не восстанавливает
               ; лишает регистр флагов со стека

; транзитная часть программы
; ПРОВЕРКА ТОГО ФАКТА, ЧТО ПРОГРАММА УЖЕ УСТАНОВЛЕНА
mov ax,OGG00h ; вызов мультиплексного прерывания со значением
int 2Fh        ; AX = OFF00h
jnc no_inst     ; если CF = 0, то эта программа ранее не была
               ; установлена
               ; программа не установлена
mov ax,4c01h
```

```

    int 21h ; завершение программы с кодом ошибки 1
no_inst: . . . . .
    ; СОХРАНЕНИЕ СТАРОГО ВЕКТОРА 2Fh В ПАМЯТИ ПО АДРЕСУ old_2F
    mov al,2Fh
    mov al,35h
    int 21h
    mov cs:[old_2F+2],es
    mov cs:[old_2F],dx
    ; УСТАНОВКА НОВОГО ВЕКТОРА 2F
    lea dx,new_2F ; предполагается, что в регистр DS записан
    mov al,2Fh    ; сегмент нового обработчика new_2F
    mov ah,25h
    int 21h
    . . . . .

```

Различные векторы прерывания могут быть перехвачены многократно. В результате различные обработчики будут вызываться по цепочке. Чтобы не разрушать такие цепочки, новый обработчик должен всегда (за исключением некоторых специальных случаев) после выполнения необходимых действий передавать управление старому обработчику.

Если новый обработчик не имеет стека, то он использует стек прерванной программы. Размер этого стека неизвестен. Поэтому, если вы собираетесь сохранять в стеке больше, чем одно-два слова, необходимо строить собственный стек. Если обработчик использует собственный стек, то он должен сохранять на входе и восстанавливать на выходе значения регистров SS и SP.

Переключение PSP и DTA необходимо в связи с тем, что в момент активизации всплывающей программы в ОС установлены PSP и DTA прерванной программы. При этом необходимо выполнить следующие действия:

в транзитной части программы получить и сохранить во внутренних переменных PSP и DTA резидентной части программы. Для этого используются функции 2Fh и 62h прерывания 21h;

при входе во всплывающую программу получить и сохранить во внутренних переменных PSP и DTA прерванной программы. Из внутренних переменных получить и восстановить PSP и DTA резидентной программы. Для этого используются функции 2Fh, 62h, 1Ah, 50h прерывания 21h;

при выходе из всплывающей программы получить из внутренних переменных и восстановить PSP и DTA прерванной программы. Для этого используются функции 1Ah и 50h прерывания 21h.

Сохранение при входе во всплывающую программу фрагмента стека прерванной программы и его последующее восстановление при выходе из всплывающей программы (см.

пример ниже) позволяет предотвратить разрушение общего стека (имеется в виду один из трех системных стеков ОС). В книге [29] такой подход предлагается для сохранения стека при вызове прерывания 28h. К сожалению, имеется много примеров, когда это не приводит к положительным результатам. В правильности сказанного можно убедиться при попытке модификации приведенной ниже программы. Одним из предположений на этот счет может быть следующее: видимо, при вызове функции 28h происходит переключение на другой стек. И фактически происходит сохранение этого другого стека, а не стека прерванной функции. Интересный подход по принудительному переключению стека указан в книге [13, с. 657]. Если в программе установить флаг критической ошибки CritErrFlag, то ОС будет использовать альтернативный стек.

Рассмотрим пример безопасной активной резидентной программы на языке Ассемблера. Надо сказать, что на языках Си и Си⁺ написать такую программу практически невозможно. Однако эта программа дает возможность устанавливать резидентными рассмотренные выше программы на языках Си и Си++. Программа позволяет:

вывести директорию текущего диска при нажатии клавиши «F7». С этой целью с помощью функции 4Bh прерывания 21h загружается и выполняется копия командного процессора COMMAND.COM, которому в качестве аргумента передается команда dir. При этом предполагается, что файл COMMAND.COM находится в корневой директории диска C:. Если это не так, то необходимо изменить строку prog_name;

удалить себя из памяти при нажатии клавиши «F8». Предварительно осуществляется проверка возможности удаления. Критерием такой возможности является отсутствие изменения уже измененных векторов (8h, 9h, 13h, 21h, 2Fh) после установки резидентной программы. Соответствующая проверка предполагает сравнение значения сегментного адреса в векторе прерывания с сегментным адресом резидентной программы. Если они совпадают, то нового изменения векторов не было;

установить при нажатии клавиши «F9» любую выполняемую программу (типа EXE, COM или BAT) в виде всплывающей резидентной программы. Размер этой программы не должен превышать установленного значения Size-of_program (он может быть легко изменен, в результате можно определять в виде резидента даже очень сложные программы).


```

/* пример EX6_10 */
;*****
;* РЕЗИДЕНТНЫЙ МОДУЛЬ ДЛЯ ЗАГРУЗКИ И ВЫПОЛНЕНИЯ ПРОГРАММ *;
;* (позволяет установить резидентом любую программу раз- *;
;* мером до Size_of_program*16 байт) *;
;*****
;* Автор: Складов В.А. (Минский радиотехнический институт) *;
;* Дата: 30 марта 1993 года. Ассемблер: TASM *;
;* Copyright 1993 by V.A.Sklyarov *;
;*****
;-----
;Так помечены блоки, которые можно удалить, если не будет вы-
;полнено переназначение входных потоков
;-----
.model TINY
.stack
JUMPS ; Директива для увеличения расстояния условных переходов
TRUE EQU 0FFFFh
FALSE EQU 0
Hot_F7 EQU 65 ; Активная клавиша F7
Hot_F8 EQU 66 ; Активная клавиша F8
Hot_F9 EQU 67 ; Активная клавиша F9
Env EQU 2Ch ; Смещение окружения DOS в PSP
my_stack EQU 512 ; Размер стека для резидентной части
Size_of_program EQU 4000 ; Максимальный размер новой
; загружаемой программы

;* MACRO для сохранения копии экрана и позиции курсора *;
;*****
save_screen MACRO
;---- сохранение экранной памяти в буфере screen_buffer ----
mov bx,0B800h ; ПРЕДПОЛАГАЕТСЯ, что сегментный ад-
; рес экранной памяти равен 0B800h
mov ds,bx ; Регистры DS:SI адресуют байты в
mov si,0 ; экранной памяти
mov bx,seg cs:screen_buffer ; В регистр ES заносит-
mov es,bx ; ся сегментный адрес буфера памяти
mov di,offset cs:screen_buffer ; Регистры ES:DI ад-
; ресуют байты в буфере памяти
mov cx,4000 ; 4000 - число байтов для сохранения
cld ; Признак направления для инкремента SI и DI
rep movsb ; Сохранение экранной области в буфере памяти
;---- определение и сохранение текущих координат курсора ----
mov ah,3 ; Функция для чтения координат курсора
mov bh,0 ; Выбирается страница с номером 0
int 10h ; Вызов прерывания 10h
push es ; В ES сегментный адрес буфера памяти
push dx ; В DX координаты курсора
ENDM
;*****

;* MACRO для восстановления экрана и позиции курсора *;
;*****
restore_screen MACRO
;----- восстановление позиции курсора -----
pop dx ; Восстанавливаются координаты курсора
mov ah,2 ; Функция 2 устанавливает курсор
mov bx,0 ; Выбирается страница с номером 0
int 10h ; Вызов прерывания 10h
;-- восстановление экранной памяти из буфера screen_buffer --
pop ds ; В DS сегмент буфера screen_buffer
mov bx,0B800h

```

```

mov es,bx ; В ES сегмент экранной памяти ;
mov si,offset cs:screen_buffer ;
mov di,0 ;
mov cx,4000 ;
cld ;
rep movsb ; Копирование 4000 байтов из DS:SI в ES:DI ;
ENDM ;
;***** ;
;**** MACRO для выдачи звукового сигнала при ошибке **** ;
;***** ;
error MACRO ;
mov ah,2 ; Вывод звукового сигнала с помощью ;
mov dl,7 ; функции 2 прерывания 21h ;
cli ; Запрещение прерываний ;
pushf ;
call dword ptr cs:old_21 ;
sti ; Разрешение прерываний ;
ENDM ;
;***** ;

.CODE
;*** СООБЩЕНИЯ, КОТОРЫЕ ВЫВОДЯТСЯ В ПРОГРАММЕ *** ;
;***** ;
install DB 'Эта программа уже установлена$',10,13 ;
my_mes DB 'Программа установлена',10,13 ;
DB 'F7 - активизация программы',10,13 ;
DB 'F8 - удаление программы из памяти',10,13 ;
DB 'F9 - изменение выполняемых действий$' ;
mes_no_delete DB 'Эту программу удалить нельзя',10,13 ;
DB 'Нажмите любую клавишу',10,13,'$' ;
delete_OK DB 'Программа удалена из памяти',10,13 ;
press_any_key DB 'Нажмите любую клавишу',10,13,'$' ;
error_4B DB 'Ошибка при вызове функции 4B прерывания 21h' ;
DB 10,13,'Нажмите любую клавишу',10,13,'$' ;
load_error DB 'Ошибка при установке резидента',10,13 ;
DB 'Перегрузите операционную систему',10,13,'$' ;
set_prog DB 'Имя программы введено',10,13 ;
DB 'Нажмите любую клавишу',10,13,'$' ;
set_prompt DB 'Введите программу с правильным именем:',10,13 ;
DB 'маршрут\имя.тип, например, C:\WORK\EX6_1.EXE' ;
DB 10,13,'$' ;
set_com DB 'Введите командную строку с правильным именем' ;
DB 'или нажмите клавишу ВВОД.',10,13,'Пример ко' ;
DB 'мандной строки: /C dir',10,13,'$' ;
;***** ;

;*** НОВЫЙ ОБРАБОТЧИК ДЛЯ ПРЕРЫВАНИЯ 9 *** ;
;***** ;
new_9: push ax ;
cli ; Установка запрещения прерываний ;
again: in al,64h ; Получение статуса клавиатуры ;
test al,2 ; Проверка возможности приема данных ;
jnz again ; Переход, если данные принимать нельзя ;
in al,60h ; Прием данных в регистр AL ;
sti ; Установка разрешения прерываний ;
cmp al,Hot_F7 ; Если нажата клавиша Hot_F7, то ;
je request ; переход к метке request ;
cmp al,Hot_F8 ; Если нажата клавиша Hot_F8, то ;
je del_req ; переход к метке del_req ;
cmp al,Hot_F9 ; Если нажата клавиша Hot_F9, то ;
je enter_string ; переход к метке enter_string ;
pop ax ;

```

```

        jmp dword ptr cs:old_9 ; Переход к оригинальному      ;
                                ; обработчику                  ;
enter_string:                    ; string_flag активизирует   ;
        mov cs:string_flag,TRUE ; процесс ввода имени новой ;
                                ; всплывающей программы     ;
        jmp short request
del_req: mov cs:del_flag,TRUE ; Установка признака для пос- ;
                                ; ледующего удаления программы ;
request: cmp cs:TSR_active,TRUE ; Программа уже выполняется ;
        je no_key ; Если программа выполняется, то выход ;
        mov cs:my_req,TRUE ; Установка флага запроса ;
                                ; для активизации программы ;
no_key: mov al,20h ; Завершение аппаратного прерывания ;
        out 20h,al ;
        pop ax ;
        iret ;
;*****
redirected DW FALSE ; Флаг переназначения входных потоков ;
; (типа prog < file -- в этом случае ввод осуществляется не ;
; с клавиатуры, а из файла). Если redirected = TRUE, то ;
; было выполнено переназначение входных потоков
redirected_is_running DW FALSE ; Выполняется процедура про- ;
                                ; верки переназначения потоков
temp_flag DW FALSE ; временная переменная-флаг

;* ВЫЗОВ ПРОЦЕДУРЫ TSR_main ПУТЕМ АНАЛИЗА ФЛАГА АКТИВНОСТИ *;
;* DOS ПРИ КАЖДОМ АППАРАТНОМ ОБРАЩЕНИИ К ТАЙМЕРУ *;
;*****
;*** НОВЫЙ ОБРАБОТЧИК ДЛЯ ПЕРЕРЫВАНИЯ 8 ***
;*****
new_8: pushf ;
        call dword ptr cs:old_8 ; Вызов старого обработчика ;
        push es ;
        push bx ;
        cmp cs:TSR_active,TRUE ; Если программа уже выполня- ;
        je exit_8 ; ется, то выход из этого обработчика ;
        cmp cs:flag_13,0 ; Если производится работа с диском ;
        jne exit_8 ; (13h), то выход из обработчика ;
        les bx,dword ptr cs:DOS_flag_address ; Загрузка в ;
                                ; ES:BX адреса флага активности ОС ;
        cmp byte ptr es:[bx],0 ; Если 0, то DOS свободна ;
        jne exit_8 ; Если ОС занята, то выход из обработчика ;
;
        cmp cs:redirected_is_running,TRUE ; Выход, если ;
        je exit_8 ; программа уже выполняется ;
        call check_redirected ; Вызов процедуры проверки ;
;
        cmp word ptr cs:my_req,TRUE ; Если нет запроса на ;
        jne exit_8 ; выполнение, то выход из обработчика ;
        mov cs:TSR_active,TRUE ; Установка флага активности ;
                                ; этой программы ;
        call TSR_main ; Вызов программы, которая выполняет ;
                                ; все основные действия ;
        mov cs:TSR_active,FALSE ; Сброс флага активности TSR ;
exit_8: pop bx ;
        pop es ;
        iret ;
;*****

```

```

;*****
;* Процедура проверки переназначения входных потоков *
;*****
check_redirected PROC C USES ax bx dx
    mov cs:redirected_is_running,TRUE ; Установка при-
    mov cs:redirected,FALSE ; знака выполнения программы
;-----
    cmp cs:temp_flag,FALSE ; Переменная temp_flag меняет;
    jne set_false ; свое значение от FALSE к ;
    mov cs:temp_flag,TRUE ; TRUE и наоборот при каждом ;
    jmp short the_next ; вызове этой процедуры. Это ;
set_false: ; необходимо для временной ;
    mov cs:temp_flag,FALSE ; задержки в новом обработчи-;
the_next: ; ке прерывания 21h
;-----
    mov ah,44h ; Получение информации об устройстве с ;
    mov al,0 ; помощью подфункции 0 функции 44h ;
    mov bx,0 ; Тип устройства: консоль ввода ;
    pushf ; Вызов старого обра- ;
    call dword ptr cs:old_21 ; ботчика прерывания 21h ;
    jnc no_errors ; Проверка отсутствия ошибки ;
    error ; Выдача звукового сигнала при ошибке ;
    jmp short no_redirection ; Переход к концу при ошибке;
no_errors: ; Переход к этой метке при отсутствии ошибки ;
    test dx,10000000b ; Переход к redirected_OK, если ;
    jz redirected_OK ; будет выполнено обращение не к ;
    ; символному драйверу ;
    test dx,00000011b ; Проверка того факта, что будет ;
    ; выполнено обращение к драйверу консоли ;
    jnz no_redirection ; Переход если нет переназ- ;
    ; начения входного потока ;
redirected_OK: ; Установка флага переназначения ;
    mov cs:redirected,TRUE ; входного потока ;
no_redirection: ; Нет переназначения входного потока ;
    mov cs:redirected_is_running,FALSE ; Сброс признака ;
    ret ; выполнения программы ;
check_redirected ENDP
;*****

```

```

;*** НОВЫЙ ОБРАБОТЧИК ДЛЯ ПРЕРЫВАНИЯ 13h ***;
;*****
new_13: inc cs:flag_13 ; Увеличение значения flag_13 при ;
    ; каждом обращении к этому обработчику ;
    pushf ;
    call dword ptr cs:old_13 ; Вызов старого обработчика ;
    dec cs:flag_13 ; Уменьшение значения flag_13 при ;
    ; каждом обращении к этому обработчику ;
    sti ; Разрешение аппаратных прерываний ;
    retf 2 ; Сохранение регистра флагов после выхода ;
;*****

```

```

;*** НОВЫЙ ОБРАБОТЧИК ДЛЯ МУЛЬТИПЛЕКСНОГО ПРЕРЫВАНИЯ 2F ***;
;*****
; AX=FF00h - эта программа установлена, если CF=1
new_2F: cmp ax,0FF00h ; Проверка вызова нового обработчика ;
    je CF_1 ; Если равно, то CF=1 ;
    cnc ; Сброс признака переноса CF ;
    jmp dword ptr cs:old_2F ; Вызов старого обработчика ;
CF_1: stc ; Установка флага CF ;
    retf 2 ; Выход без изменения регистра флагов ;
;*****

```

```

;***** данные для процедуры обработчика new_21 *****;
number_of_characters DB 0 ; Максимальное число символов,
; которые можно ввести
;
p DW FALSE ; Переменная для выделения расширенных ASCII-кодов;
;*****
;*** НОВЫЙ ОБРАБОТЧИК ДЛЯ ПРЕРЫВАНИЯ 21h ***;
;*****
; в стеке прерванной программы должно быть 24 свободных байта ;
;*****
new_21: sti ; Установка разрешения прерываний ;
        cmp ah,8 ; Функции 8, 1, 6, 7 используются ;
        je input ; различными программами для ввода ;
        cmp ah,1 ; символов с клавиатуры. Здесь вызов ;
        je input ; этих функций задерживается до тех ;
        cmp ah,6 ; пор, пока в буфере клавиатуры не ;
        je input ; появится символ. При отсутствии ;
        cmp ah,7 ; вызова флаг активности DOS будет в ;
        je input ; состоянии "DOS свободна" ;
        cmp ah,10 ; функцию 10 вызывает командный процессор ;
        je f_10 ; при ожидании ввода строки с клавиатуры ;
        jmp cont21 ; Вызов другой функции прерывания 21h ;
; переход к этому фрагменту осуществляется при ожидании ввода ;
;-----;
;-- если с клавиатуры принимается расширенный ASCII-код, то --;
;-- функций 8, 1, 6, 7 должны вызываться дважды на каждое --;
;-- нажатие клавиши. Это учитывается в следующем фрагменте --;
;-----;
input: push ax
myloop: cmp word ptr cs:p,TRUE ; Если введен не расширенный ;
        jne call16 ; код, то можно вызывать прерывание 16h ;
        mov word ptr cs:p,FALSE ; Если введен расширенный ;
        jmp short next ; код, то надо пропустить ;
call16: ; один вызов прерывания 16h ;
;-----;
; cmp cs:redirected,TRUE ; Если было переназначение, ;
; je next ; то переход к оригинальному обработчику ;
;-----;
        mov ah,1 ; Проверка наличия символа в буфере ;
        int 16h ; Символ из буфера не удаляется ;
        jz myloop ; Если в буфере нет символа, то переход ;
        ; к метке myloop ;
        cmp al,0 ; Если AL=0, то введен расширенный код ;
        jne next ; Переход, если не расширенный код ;
        mov word ptr cs:p,TRUE ; Признак расширенного кода ;
next: pop ax
;-----;
i21: cmp cs:redirected,TRUE ; Если не было переназначе- ;
     jne cont21 ; ния потоков, то пропуск фрагмента ниже ;
;-----;
        cmp cs:temp_flag,TRUE ; Здесь выполняется задержка на ;
        jne wait_true ; одно обращение к процедуре ;
wait_false: ; check_redirected. Для уста- ;
        cmp cs:temp_flag,FALSE ; новления этого факта исполь- ;
        jne wait_false ; зуется переменная temp_flag. ;
        jmp short my_check ; Это необходимо делать в свя- ;
wait_true: ; зи с тем, что после выхода в ;
        cmp cs:temp_flag,TRUE ; среду COMMAND.COM (или другой ;
        jne wait_true ; подобной программы) произой- ;
my_check: ; дет преждевременный вход в оригинальную функцию ;
        cmp cs:redirected,FALSE ; прерывания 21h. В результа- ;
        je f_10 ; те некоторое время программа не будет реа- ;
        ; гировать на активные клавиши ;
;-----;

```

```

cont21: cli ; запрещение прерываний ;
        jmp dword ptr cs:old_21 ; оригинальный обработчик
;*****
;*****
;----- переход к этому фрагменту осуществляется при -----
;----- выполнении функции 10 (0Ah) -----
;- Здесь полностью изменена оригинальная функция 0Ah прерывания 21h, которую использует командный процессор для ввода данных с клавиатуры
f_10:  push ax ;
        push bx ;
        push es ;
        push di ;
        push cx ;
        push ds ; Через регистры DS:DX передается адрес буфера для записи символов, поступающих с клавиатуры. Значение DS переписывается в DX ;
        pop di ; ES, а значение DX в DI ;
        inc di ; Пропуск двух байтов в буфере, в которых задается максимальное и действительное число символов, введенных с клавиатуры ;
        mov bx,dx ;
        push bx ; сохранение BX для использования в будущем ;
        mov al,[bx] ; лов в переменную number_of_characters ;
        mov cs:number_of_characters,al ; characters ;
        xor cx,cx ; Обнуление счетчика введенных символов cx ;
i16:
;-----
;----- cmp cs:redirected,TRUE ; Если было переназначение,
;----- je next1 ; то переход к оригинальному обработчику
;-----
        mov ah,1 ; Проверка наличия символа в буфере клавиатуры с помощью функции 1 прерывания 16h ;
        int 16h ; туры с помощью функции 1 прерывания 16h ;
        jz i16 ; Если в буфере нет символа, то переход к метке i16 ;
        mov ah,0 ; Чтение символа клавиатуры с помощью функции 0 прерывания 16h (ASCII-код в AL) ;
        int 16h ; ;
        cld ; Установка признака направления ;
        stosb ; Запись байта из AL в память по адресу ES:DI ;
        inc cx ; Инкремент счетчика числа записанных байтов ;
        mov ah,2 ; функция вывода символа на экран ;
        mov dl,al ; выводимый символ заносится в DL ;
        cli ; Запрещение прерываний ;
        pushf ; Вызов оригинального обработчика ;
        call dword ptr cs:old_21 ;
        sti ; Разрешение прерываний ;
        cmp al,13 ; Проверка наличия символа ввода ;
        je cont ; Если был "ВВОД", то завершение функции ;
        cmp cl,byte ptr cs:number_of_characters ; Если число введенных символов меньше максимального, то переход к метке i16 ;
        jne i16 ;
        dec di ; Резервирование последнего байта для возможного символа "ВВОД" ;
        dec cx ; ;
        error ; МАСКО для выдачи звукового сигнала ;
cont:  pop bx ; Восстановление ранее сохраненного BX ;
        inc bx ; В BX теперь адрес первого байта в буфере ;
        mov byte ptr [bx],cl ; Сохранение числа действительных введенных байтов ;
        pop cx ;
        pop di ;

```

```

        pop es
        pop bx
        pop ax
        iret ; Завершение обработчика для функции 0Ah
;
next1:  pop bx ; Восстановление регистров из стека
        pop cx
        pop di
        pop es
        pop bx
        pop ax
        jmp i21
;
;*****
;***   ГЛАВНАЯ ПРОЦЕДУРА, ДЕЙСТВИЯ КОТОРОЙ ОПРЕДЕЛЯЮТСЯ   ***;
;***   НАЖАТОЙ АКТИВНОЙ КЛАВИШЕЙ                           ***;
;*****
TSR_main proc C USES ax bx cx dx si di ds es
        mov cs:my_req,FALSE ; Сброс признака запроса на вы-
                                ; полнение этой программы
;----- переключение стека -----
        cli ; Запрет прерываний
        mov cs:old_SS,SS ; Сохранение старых значений SS и
        mov cs:old_SP,SP ; SP в переменных old_SS и old_SP
        mov SS,CS:new_SS ; Установка нового стека путем
        mov SP,CS:new_SP ; изменения значений в SS и SP
        sti ; Разрешение прерываний
;-- сохранение во внутреннем стеке 64 слов системного стека --
        mov cx,64
        mov ds,cs:old_ss
        mov si,cs:old_sp
save_DOS_stack: ; Сохранение 64 слов (128
                push word ptr [si] ; байтов) стека DOS во
                inc si ; внутреннем стеке этой
                inc si ; программы
                loop save_DOS_stack
;----- действия по выгрузке этой программы из памяти -----
        cmp cs:del_flag,TRUE ; Если del_flag=TRUE, то
        jne check_enter_string ; необходимо выгрузить
        call check_delete ; эту программу из памяти
        cmp ax,TRUE ; Если ax не TRUE, то эту
        jne no_delete ; программу удалить нельзя
        call delete ; Удаление этой программы
        jmp continue
no_delete: save_screen ; MACRO сохранения экрана
        call cls_scr ; Очистка экрана
        push cs ; Выдача сообщения о том,
        pop ds ; что эту программу нельзя
        lea dx,mes_no_delete ; выгрузить из памяти
        mov ah,9
        int 21h
        mov cs:del_flag,FALSE ; Сброс флага запроса
        mov ah,0 ; Ожидание нажатия
        int 16h ; клавиши клавиатуры
        restore_screen ; Восстановление экрана
        jmp continue
;--- действия по установке новой всплывающей программы ---;
check_enter_string: ;# Если string_flag=TRUE,
        cmp cs:string_flag,TRUE ;# то надо установить новую;
        ;# всплывающую программу ;
        jne main_function ;# Если нет, то переход ;

```

```

save_screen                ;# MACRO сохранения экрана ;
call cls_scr               ;# Очистка экрана ;
call my_string             ;# Вызов основной процедуры ;
lea dx,cs:set_prog        ;# Выдача сообщения о за- ;
mov ah,9                  ;# вершении этого процесса ;
int 21h                   ;# ;
mov cs:string_flag,FALSE  ;# Сброс флага запроса ;
mov ah,0                  ;# Ожидание нажатия ;
int 16h                   ;# клавиши клавиатуры ;
restore_screen            ;# Восстановление экрана ;
;----- основные действия этой процедуры -----;
main_function:
;----- сохранение сегмента PSP -----;
mov ah,62h ; Получение сегмента PSP прерванной ;
int 21h ; программы с помощью функции 62h пре- ;
mov cs:interrupted_psp_seg,bx ; рывания 21h ;
;----- сохранение сегмента окружения -----;
push es ;
mov es,bx ; Сегмент PSP в регистр ES ;
mov bx,Env ; Смещение сегмента окружения в BX ;
mov dx,word ptr es:[bx] ; Сегмент окружения в DX ;
mov cs:env_seg,dx ; Сегмент окружения в env_seg ;
pop es ;
;----- сохранение сегмента и смещения DTA -----;
mov ah,2Fh ; Получение сегмента и смещения DTA с ;
int 21h ; помощью функции 2Fh прерывания 21h ;
mov cs:interrupted_dta_seg,es ;
mov cs:interrupted_dta_off,bx ;
;----- установка PSP этой программы -----;
mov ah,50h ; Установка PSP с помощью ;
mov bx,cs:my_psp_seg ; функции 50h прерывания 21h ;
int 21h ;
;----- установка сегмента и смещения DTA этой программы -----;
mov ah,1ah ; Установка сегмента и смеще- ;
mov dx,cs:my_dta_off ; ния DTA с помощью функции ;
mov ds,cs:my_dta_seg ; 1ah прерывания 21h ;
int 21h ;
;----- освобождение ранее выделенной памяти для программы -----;
mov es,cs:buf_seg ; Освобождение сегмента buf_seg ;
mov ah,49h ; памяти с помощью функции 49h ;
int 21h ; прерывания 21h ;
jnc no_error ; Переход к no_error, если нет ошибки ;
error ; Звуковой сигнал при наличии ошибки ;
jmp my_exit ; Выход из программы при ошибке ;
;- вызов MACRO для сохранения копии экрана и позиции курсора ;
no_error: save_screen ;
;----- вызов основных процедур резидентной части -----;
call cls_scr ; Очистка экрана ;
call dir_d ; Процедура вызова установленной всплываю- ;
; щей резидентной программы ;
jnc dir_OK ; Переход к dir_OK, если нет ошибки ;
error ; Звуковой сигнал при наличии ошибки ;
lea dx,error_4B ; Сообщение об ошибке ;
jmp short dir_e ; Вывод сообщения об ошибке ;
;----- вывод сообщения "Нажмите любую клавишу" -----;
dir_OK: lea dx,press_any_key ;
dir_e: push cs ;
pop ds ;
mov ah,9 ;
int 21h ;
;----- ожидание нажатия клавиши клавиатуры -----;
mov ah,0 ; функция 0 прерывания 16h ожидает ;

```



```

        int 16h      ; нажатия клавиши клавиатуры
;- вызов MACRO для восстановления экрана и позиции курсора -;
        restore_screen
;----- выделение ранее освобожденной памяти -----;
        mov ah,48h
        mov bx,Size_of_program
        int 21h
        mov cs:buf_seg,ax
        jnc my_exit ; Переход к my_exit, если нет ошибки
        error       ; Звуковой сигнал при наличии ошибки
;----- восстановление DTA прерванной программы -----;
my_exit: mov ah,lah ; С помощью функции lah прерывания 21h
        mov dx,cs:interrupted_dta_off
        mov ds,cs:interrupted_dta_seg
        int 21h
;----- восстановление PSP прерванной программы -----;
        mov ah,50h ; С помощью функции 50h прерывания 21h
        mov bx,cs:interrupted_psp_seg
        int 21h
;восстановление из внутреннего стека 64 слов системного стека;
continue:
        mov cx,64
        mov ds,cs:old_ss
        mov si,cs:old_sp
        add si,128
yyy:    dec si
        dec si
        pop word ptr [si]
        loop yyy
;----- обратное переключение стека -----;
        cli
        mov SS,cs:old_SS ; Восстановление старого стека
        mov SP,cs:old_SP
        sti
        ret
TSR_main endp
;*****
;***** ПРОЦЕДУРА ОЧИСТКИ ЭКРАНА И УСТАНОВКИ КУРСОРА В *****;
;***** ЛЕВЫЙ ВЕРХНИЙ УГОЛ *****;
;*****
cls_scr PROC C USES es dx
        mov ah,8 ; Чтение ASCII кода символа и его атрибута
        xor bh,bh ; Все выполняется для нулевой страницы
        int 10h
        mov bh,ah ; Атрибут в регистр bh
        mov ah,6 ; Выбор функции скроллинга
        xor al,al ; для очистки экрана
        xor cx,cx
        mov dh,24
        mov dl,79
        int 10h ; Очистка экрана
        xor dx,dx
        xor bh,bh
        mov ah,2 ; Выбор функции установки курсора
        int 10h ; Установка курсора в левый верхний угол
        ret
cls_scr ENDP
;*****

```

```

;***** ПРОЦЕДУРА УСТАНОВКИ НОВОЙ ВСПЛЫВАЮЩЕЙ ПРОГРАММЫ *****;
;***** И ЗАДАНИЯ КОМАНДНОЙ СТРОКИ ДЛЯ ЭТОЙ ПРОГРАММЫ *****;
;*****
my_string PROC C USES dx ax si
    push cs ; Установка DS используется и после завер- ;
    pop ds ; шения этой процедуры
    lea dx,cs:set_prompt ; Выдача сообщения о необходи- ;
    mov ah,9 ; мости и правилах ввода имени ;
    int 21h ; новой всплывающей программы ;
    lea dx,cs:two_bytes_before_prog_name ; В первых двух;
    ; байтах буфера содержатся числовые значения (см.;
    ; функцию 0Ah прерывания 21h или фрагмент f_10) ;
    mov si,dx ; Смещение буфера в SI
    mov ah,0Ah ; Ввод маршрута и полного имени прог- ;
    int 21h ; раммы с помощью функции 0Ah
    cld ; Признак направления для инкремента SI;
new_b: lodsb ; Запись байта в AL из памяти по адресу DS:SI ;
    cmp al,13 ; Проверка наличия ввода (клавиша ВВОД);
    jns new_b ; Если нет, то запись очередного байта;
    dec si ; Теперь SI адресует байт с кодом ВВОД ;
    mov byte ptr [si],0 ; Запись в этот байт нуля ;
    lea dx,cs:set_com ; Вывод сообщения о необходимос- ;
    mov ah,9 ; ти и правилах ввода новой ;
    int 21h ; командной строки ;
    lea dx,cs:byte_before_com ; Здесь надо пропустить ;
    mov ah,0Ah ; только один байт. Далее идет ввод ко- ;
    int 21h ; мандной строки с помощью функции 0Ah ;
    ret
my_string ENDP
;*****

;***** данные для процедуры dir_d *****;
par DB 0,0,0,0,0,0,0ffh,0ffh,0,0,0ffh,0ffh,0,0 ;
two_bytes_before_prog_name DB 40,0 ; 40 - максимальное число ;
; символов в строке с именем программы ;
prog_name DB 'c:\command.com',0,30 dup (0) ;
byte_before_com DB 55 ; 55 - максимальное число символов в ;
; в строке с опциями и аргументами ;
com DB 6,'/C dir',13,50 dup(0) ;
;*****
;* ПРОЦЕДУРА ВЫЗОВА УСТАНОВЛЕННОЙ РЕЗИДЕНТНОЙ ПРОГРАММЫ *;
;*****
dir_d PROC C USES es dx
    push cs
    pop es ; Сегмент кода в регистр ES
    push cs
    pop ds ; Сегмент кода в регистр DS
    lea bx,cs:par ; В BX смещение блока параметров
    lea ax,cs:com ; В AX смещение командной строки
    mov dx,cs:env_seg ; Сегмент окружения в блок
    mov es:[bx],dx ; параметров по смещению 0
    mov es:[bx+2],ax ; Данные из AX по смещению 2
    mov ax,seg cs:com ; Сегмент командной строки в блок
    mov es:[bx+4],ax ; параметров по смещению 4
    lea dx,cs:prog_name ; В DX смещение имени программы
    mov ah,4Bh ; В AH номер функции - 4Bh
    xor al,al ; В AL номер подфункции - 0
;----- сохранение SS:SP во внутренних переменных -----;
    cli
    mov cs:ss_4B,ss
    mov cs:sp_4B,sp
    sti

```

```

;----- вызов функции 4Bh прерывания 21h -----;
    int 21h
;----- восстановление SS:SP из внутренних переменных -----;
    cli
    mov ss,cs:ss_4B
    mov sp,cs:sp_4B
    sti
    ret
dir_d ENDP
;*****
;***** данные для процедуры check_delete *****;
change_vect DB 8h,9h,13h,21h,2Fh,0h
;*****
;***** ПРОВЕРКА ВОЗМОЖНОСТИ ВЫГРУЗКИ ИЗ ПАМЯТИ ЭТОЙ ПРОГРАММЫ *****;
;*****
check_delete PROC C USES di
    mov dx,cs:my_prog_seg ; В DX сегмент этой программы
    mov di,offset change_vect-1
check:
    mov al,cs:[di] ; В DI смещение вектора из данных выше
    or al,al ; В AL очередной проверяемый вектор
    je OK ; Выход если достигнут заключительный
    mov ah,35h ; нуль (в приведенном выше списке)
    int 21h ; Чтение очередного вектора с помощью функции 35h прерывания 21h
    mov cx,es ; Проверка совпадения сегмента возвращен-
    cmp dx,cx ; ного вектора в ES с сегментом программы
    je check ; Если равно, то этот вектор не изменен
    mov ax,FALSE ; В противном случае AX=FALSE
    ret ; и выход из программы
OK:
    mov ax,TRUE ; Программу можно выгружать
    ret
check_delete ENDP
;*****
;* ВЫГРУЗКА ИЗ ПАМЯТИ ЭТОЙ ПРОГРАММЫ *;
;*****
delete PROC C USES ds
;----- восстановление всех измененных векторов -----;
    cli ; Запрещение прерывание
    mov al,8h ; Восстановление вектора 8
    mov ah,25h ; с помощью функции 25h прерывания 21h
    lds dx,dword ptr cs:old_8
    int 21h
    mov al,9h ; Восстановление вектора 9
    mov ah,25h
    lds dx,dword ptr cs:old_9
    int 21h
    mov al,13h ; Восстановление вектора 13h
    mov ah,25h
    lds dx,dword ptr cs:old_13
    int 21h
    mov al,21h ; Восстановление вектора 21h
    mov ah,25h
    lds dx,dword ptr cs:old_21
    int 21h
    mov al,2Fh ; Восстановление вектора 2Fh
    mov ah,25h
    lds dx,dword ptr cs:old_2F
    int 21h
    sti ; Разрешение прерываний
;----- освобождение памяти -----;

```

```

mov es,cs:my_psp_seg
mov cx,es
mov es,es:[2ch]
mov ah,49h
int 21h ; Освобождение памяти под окружение
jnc no_err ; Переход к no_err, если нет ошибки
error ; Звуковой сигнал при наличии ошибки
jmp short error_exit ; Выход по ошибке
no_err: mov es,cx
mov ah,49h
int 21h ; Освобождение памяти под программу
jnc no_er ; Переход к no_er, если нет ошибки
error ; Звуковой сигнал при наличии ошибки
jmp short error_exit ; Выход по ошибке
no_er: mov es,cs:buf_seg
mov ah,49h ; Освобождение памяти под выделенный блок
int 21h ; размером Size_of_program параграфов
jnc no_e ; Переход к no_e, если нет ошибки
error ; Звуковой сигнал при наличии ошибки
jmp short error_exit ; Выход по ошибке
;----- вывод сообщения "Программа удалена из памяти" -----;
no_e: save_screen ; MACRO для сохранения экрана
call cls_scr ; Очистка экрана
push cs
pop ds
lea dx,cs:delete_OK
mov ah,9
int 21h
mov ah,0 ; Ожидание нажатия
int 16h ; клавиши клавиатуры
restore_screen ; MACRO для восстановления экрана
error_exit: ; Переход к этой метке произойдет при ошибке
ret
delete ENDP
;*****

;* ОСНОВНЫЕ ДАННЫЕ ДЛЯ ЭТОЙ ПРОГРАММЫ *;
;*****
my_req DW 0000 ; Флаг активизации этой программы
TSR_active DW 0000 ; Здесь ненулевое значение если обра-
; ботчик активизирован
old_8 DW 0000,0000 ; Адрес старого обработчика int 8h
old_9 DW 0000,0000 ; Адрес старого обработчика int 9h
old_13 DW 0000,0000 ; Адрес старого обработчика int 13h
old_21 DW 0000,0000 ; Адрес старого обработчика int 21h
old_2F DW 0000,0000 ; Адрес старого обработчика int 2Fh
DOS_flag_address DW 0000,0000 ; Указатель на флаг актив-
; ности DOS
flag_13 DB 0 ; Флаг активности прерывания 13h
ss_4B DW 0000 ; Временное сохранение SS в функции 4Bh
sp_4B DW 0000 ; Временное сохранение SP в функции 4Bh
old_SS DW 0000 ; Переменная для старого значения SS
old_SP DW 0000 ; Переменная для старого значения SP
new_SS DW 0000 ; Переменная для нового значения SS
new_SP DW 0000 ; Переменная для нового значения SP
TSR_stack DB my_stack DUP(0) ; Стек для резидентной части
my_dta_seg DW 0000 ; Сегмент DTA этой программы
my_dta_off DW 0000 ; Смещение DTA этой программы
my_psp_seg DW 0000 ; Сегмент PSP этой программы
interrupted_dta_seg DW 0000 ; Сегмент и смещение DTA прер-
interrupted_dta_off DW 0000 ; ванной программы
interrupted_psp_seg DW 0000 ; PSP прерванной программы

```

```

env_seg DW 0000 ; Сегмент окружения для вызываемой программы ;
my_prog_seg DW 0000 ; Сегмент этой программы ;
del_flag DW FALSE ; Признак (флаг) выгрузки этой программы ;
string_flag DW FALSE; Признак установки новой всплывающей ;
; программы (pop-up программы) ;
buf_seg DW 0000 ; Сегмент выделяемого буфера ;
screen_buffer DB 4000 dup(?) ; Буфер для экранной памяти ;
; размером 4000 байт ;
;*****;

;*          ТРАНЗИТНАЯ ЧАСТЬ ПРОГРАММЫ          *;
;*****;
set_vect PROC
;----- получение и сохранение PSP и DTA этой программы -----;
    push cs ; Передача значения из регистра ;
    pop ds ; CS в регистр DS ;
    mov cs:my_prog_seg,ds ; Сохранение сегмента этой ;
    ; программы в переменной my_prog_seg ;
    mov ah,2Fh ; Получение адреса DTA этой программы ;
    int 21h ; с помощью функции 2Fh прерывания 21h ;
    mov cs:[my_dta_seg],es ; Сохранение сегмента DTA ;
    mov cs:[my_dta_off],bx ; Сохранение смещения DTA ;
    mov ah,62h ; Получение адреса PSP этой программы ;
    int 21h ; с помощью функции 62h прерывания 21h ;
    mov cs:[my_psp_seg],bx ; Сохранение сегмента PSP ;
; проверка того факта, что эта программа уже была установлена;
    mov ax,OFF00h ; функция проверки установки программы;
    int 2Fh ; Вызов мультиплексного прерывания ;
    jnc no_inst ; Переход, если программа не установлена;
    mov dx,OFFSET cs:install ; Вывод сообщения о том, ;
    mov ah,09h ; что программа уже ус- ;
    int 21h ; тановлена ранее ;
    mov ah,4Ch ; Завершение программы с кодом ;
    mov al,1 ; ошибки 1 ;
    int 21h ;
; изменение вектора 8 и сохранение адреса старого обработчика;
no_inst: mov al,8h ;
    mov di,OFFSET cs:old_8 ;
    mov dx,OFFSET new_8 ;
    call change_v ; Процедура изменения вектора ;
; изменение вектора 9 и сохранение адреса старого обработчика;
    mov al,09h ;
    mov di,OFFSET cs:old_9 ;
    mov dx,OFFSET new_9 ;
    call change_v ;
;изменение вектора 13 и сохранение адреса старого обработчика;
    mov al,13h ;
    mov di,OFFSET cs:old_13 ;
    mov dx,OFFSET new_13 ;
    call change_v ;
;изменение вектора 21 и сохранение адреса старого обработчика;
    mov al,21h ;
    mov di,OFFSET cs:old_21 ;
    mov dx,OFFSET new_21 ;
    call change_v ;
;изменение вектора 2F и сохранение адреса старого обработчика;
    mov al,2Fh ;
    mov di,OFFSET cs:old_2F ;
    mov dx,OFFSET new_2F ;
    call change_v ;
; установка признака отсутствия активности резидентной части ;
    mov cs:TSR_active,FALSE ;

```

```

; получение и сохранение в памяти адреса флага активности DOS;
mov ah,34h ; Функция 34h прерывания 21h возвращает ;
int 21h ; в ES:BX адрес флага активности DOS ;
mov cs:DOS_flag_address[0],BX ;
mov cs:DOS_flag_address[2],ES ;
; сохранение сегмента и смещения стека для резидентной части ;
mov ax,cs ;
mov cs:new_SS,ax ;
mov ax,OFFSET cs:TSR_stack+my_stack ;
mov cs:new_SP,ax ;
;-- вычисление размера резидентной части программы (в DX) --;
mov dx,offset set_vect ; В dx - конец обработчика ;
mov ax,seg set_vect ; В ax - сегмент обработчика ;
sub ax,cs:[my_psp_seg] ; В ax учитывается размер PSP ;
mov cl,4 ; В cl число бит для сдвига вправо ;
shr dx,cl ; Сдвиг dx на 4 бита вправо ;
inc dx ; Добавление одного параграфа ;
add dx,ax ; Добавление размера PSP ;
;- сокращение занимаемой памяти до размера резидентной части ;
mov ah,4ah ; Функция 4ah для изменения блока памяти ;
mov bx,cs:my_psp_seg ; Сегмент программы в BX ;
mov es,bx ; и далее в регистр ES ;
mov bx,dx ; Новый размер блока памяти в параграфах ;
int 21h ; Вызов прерывания 21h ;
jnc no_e0 ; Переход к no_e0, если нет ошибки ;
error ; Звуковой сигнал при наличии ошибки ;
jmp short load_exit ; Выход по ошибке ;
;- выделение блока памяти для будущей загружаемой программы -;
no_e0: mov ah,48h ; Функция 48h выделения блока памяти ;
mov bx,Size_of_program ; Size_of_program - размер ;
; блока в параграфах ;
int 21h ; Вызов прерывания 21h ;
mov cs:buf_seg,ax ; Сохранение сегмента блока из AX ;
jnc no_e1 ; Переход к no_e1, если нет ошибки ;
error ; Звуковой сигнал при наличии ошибки ;
jmp short load_exit ; Выход по ошибке ;
;----- вывод сообщения "Программа установлена" -----;
no_e1: push dx ;
push cs ;
pop ds ;
mov dx,OFFSET cs:my_mes ; Вывод сообщения, что ;
mov ah,09h ; эта программа установлена ;
int 21h ;
pop dx ;
;---- завершение программы и сохранение ее резидентом ----;
xor al,al ; Обнуление al ;
mov ah,31h ; Функция завершения и сохранения ;
int 21h ; резидентом ;
load_exit: push cs ;
pop ds ;
lea dx,load_error ;
mov ah,9 ; Вывод сообщения об ошибке и о не- ;
pushf ; обходимости перезагрузки системы ;
call dword ptr cs:old_21 ;
mov ah,4Ch ; Завершение программы с кодом ;
mov al,1 ; ошибки 1 ;
pushf ;
call dword ptr cs:old_21 ;
set_vect ENDP ;

```

```

;*****;
;----- СОХРАНЕНИЕ СТАРОГО ВЕКТОРА ПО АДРЕСУ В DS:DI -----;
;----- УСТАНОВКА ВЕКТОРА AL ИЗ РЕГИСТРОВ DS:DX -----;
;*****;
change_v PROC C USES AX DI
;----- сохранение старого вектора -----;
    mov ah,35h ; функция для сохранения старого вектора ;
    int 21h ;
    mov ds:[di],bx ; В BX смещение старого вектора ;
    mov ds:[di+2],es ; В ES сегмент старого вектора ;
;----- установка нового вектора -----;
    mov ah,25h ; функция для установки нового вектора ;
    int 21h ;
    ret ;
change_v ENDP
END set_vect
;*****;

```

Приведенная программа довольно сложна, но все ее конструкции подробно прокомментированы. Рассмотрим простой сценарий работы с этой программой (пусть она имеет имя EX6_10.EXE). Запустите эту программу на выполнение. После ее установки резидентом (будет выдано сообщение «Программа установлена») нажмите клавишу «F7». В результате на экран будет выдан директорий текущего диска. Нажмите теперь клавишу «F9». Предположим, что рассмотренные ранее программы EX6_1.EXE или EX6_2.EXE находятся в корневом каталоге диска C: Введите имя C:\EX6_1.EXE (или C:\EX6_2.EXE). В ответ на второе приглашение просто нажмите клавишу «ВВОД». В результате запускается на выполнение программа для работы с мышью. Нажмите левую кнопку «мыши» и завершите работу программы. Войдите в среду Borland C++. В процессе загрузки этой системы либо после загрузки нажимайте клавишу «F7». Вы увидите, что будет активизирована ваша программа работы с «мышью», после завершения которой продолжается нормальная работа Borland C++. Попробуйте выполнить аналогичные действия на фоне работы сложных программ, например форматирования диска (здесь для активизации всплывающей программы придется подождать некоторое время ввиду интенсивной работы с диском). Вы увидите, что приведенная программа работает надежно. По аналогии можно установить в качестве резидента любую другую из рассмотренных в книге программ.

При необходимости можно легко изменить активные клавиши («F7», «F8», «F9») в начальной части программы. Здесь же можно изменить значение Size_of_program. Если необходимо активизировать программу по комбинации клавиш, например «Alt—F7», то следует проверять байты статуса (адреса оперативной памяти 0:417, 0:418), где при нажатии клавиши «Alt» будет установлен соответствующий

бит (см. [1,4]). Для простоты программа настроена на работу только в режиме 3 (текстовый режим работы экрана). Если экран установлен в другой режим, необходимо выполнить команду вида `mode co80` (здесь `mode` — имя утилиты ОС). В случае необходимости программа легко модифицируется для настройки на любой режим работы дисплея.

Дадим общую характеристику основным модулям программы.

Новый обработчик для прерывания 9 (он вызывается при нажатии клавиши клавиатуры) устанавливает признаки активности клавиш «F7», «F8» и «F9». При этом откладывается вызов соответствующих этим клавишам фрагментов программы до тех пор, пока не будут выполнены условия безопасности ОС.

Новый обработчик для прерывания 8 (он вызывается сигналами от таймера примерно через 55 ms) проверяет условия активизации всплывающей программы, и если они выполняются, то запускается на выполнение программа `TSR_main`. К числу таких условий относятся: удовлетворение требований безопасности ОС, отсутствие работы с дисками, наличие запроса на выполнение всплывающей программы. Кроме того, если всплывающая программа (`TSR_main`) уже выполняется, то запрос игнорируется. Процедура `check_redirected`, которая вызывается перед `TSR_main`, проверяет, было или нет переназначение входных потоков, т. е. не была ли введена команда вида

```
C > my_prog < my_file.txt < BВOD >
```

где `my_prog` — имя какой-то выполняемой программы, `my_file.txt` — файл, из которого данные поступают на вход этой выполняемой программы. Если подобная команда была введена, то устанавливается признак `redirected = TRUE`. В этом случае в новом обработчике прерывания 21h будут игнорироваться введенные изменения и управление будет передано старому обработчику. Проверка в процедуре `check_redirected` осуществляется вызовом подфункции 0 функции 44h прерывания 21h (получение информации об устройстве). Через регистр `BX` передается идентификатор файла (file handle). В нашем случае таким идентификатором будет 0, который соответствует вводу данных с клавиатуры. На выходе, если `CF = 0` (признак отсутствия ошибки), в регистре `DX` возвращается информация об устройстве. Нас будут интересовать три бита: 7, 1, 0. Если в бите 7 единица, то вызывается символьный драйвер. Таким образом, здесь будет 0 при замене символьного клавиатурного драйвера

блочным драйвером (например, в случае получения данных из дискового файла). Если в бите 1 единица, то вызывается драйвер работы с экраном, если в бите 0 единица, то вызывается драйвер работы с клавиатурой. В этих битах будут нули в случае вызова другого символьного драйвера, например COM1, при использовании команды вида

```
C > my_prog < COM1 < ВВОД >
```

Основным отличием данной программы от программ, приведенных в работах [7,29], а также от других опубликованных программ является отсутствие работы с прерыванием 28h и написание собственного обработчика для прерывания 21h. В этом случае если система ожидает ввода символа от клавиатуры (делается попытка вызова функций 1, 6, 7, 8), то доступ к этим функциям откладывается до появления символа в буфере клавиатуры. Проверка этого факта осуществляется с помощью функции 1 прерывания 16h. Во время выполнения этого прерывания флаг активности ОС будет установлен в состояние «свободно» и это позволяет активизировать всплывающую программу в новом обработчике прерывания 8. При работе с функцией 1 прерывания 16h необходимо учитывать одну особенность выполнения функций 1, 6, 7, 8. При вводе обычных ASCII-кодов эти функции вызываются один раз, а при вводе расширенных ASCII-кодов — два раза. Это учтено в приведенной выше программе (если игнорировать это требование, то работа программ типа DOSEDIT.COM, DOSKEY.COM будет нарушена). Функция OAh прерывания 21h полностью переписана (при этом вызов оригинального обработчика не производится). С особенностями выполнения этой функции можно ознакомиться в книгах [7,29]. При входе в нее в регистре AH задается номер функции (OAh), в регистрах DS:DX — адрес буфера, в который будут помещены введенные данные. При этом в нулевой байт буфера записывается значение максимального числа байтов, которые можно поместить в этот буфер. Вводимые символы будут помещены в буфер, начиная с байта 2. В байт 1 функция OAh возвращает действительное число прочитанных байтов. Когда число символов в буфере на один меньше максимального числа символов, заданного в байте 0, новые вводимые символы игнорируются и выдается звуковой сигнал (один оставшийся байт может быть использован только для символа ВВОД). Завершение выполнения функции OAh производится после нажатия клавиши «ВВОД», код которой также помещается в буфер.

Макросредства `save_screen` и `restore_screen` необходимы для сохранения текстового экрана при вызове всплывающей программы (которая использует экран) и для восстановления текстового экрана при возврате из всплывающей программы. Если необходимо, чтобы программа выполнялась на фоне графического экрана, то необходимо предусмотреть аналогичные средства для графического режима. В этом случае, к сожалению, потребуются очень большие объемы оперативной памяти.

Процедура `delete` выгрузки из памяти этой программы восстанавливает все измененные векторы и освобождает всю занятую память.

В программе отсутствует отдельный сегмент данных; все используемые данные помещены в сегмент кода. Стек, определяемый директивой `.STACK`, используется только транзитной частью. Резидентная часть устанавливает свой внутренний стек (`TSR_stack`) из сегмента кода. Его размер задан в начале программы (`my_stack`) и равен 512 байт (при необходимости он может быть легко изменен).

Для того чтобы загрузить новую программу, под нее необходимо выделить память. MS-DOS организована так, что после передачи управления командному процессору (после выхода в среду ОС) ему (командному процессору) передается вся память. В результате если после установки резидента выделить память только под код резидента, то другую программу запустить не удастся. В примере перед установкой резидента (в транзитной части программы) выделяется дополнительная память размером `Size_of_program`. Это делается с помощью функции `48h` прерывания `21h`. Далее в процедуре `TSR_main` перед вызовом процедуры `dir_d` (установка всплывающей программы) эта память освобождается с помощью функции `49h` прерывания `21h`. Теперь эта освобожденная память может быть использована для установки нового резидента. После выполнения резидента (после завершения процедуры `dir_d`) снова выделяется память размером `Size_of_program` с помощью функции `48h` прерывания `21h`. Эту же задачу можно решить и по-другому. Перед установкой всплывающей программы необходимо забрать часть памяти у командного процессора. Предварительно содержимое этой памяти переписывается на диск. После вызова резидента память возвращается командному процессору (ее содержимое восстанавливается с диска). Как известно, такая процедура называется своппингом (`swapping`).

6.5. Драйверы устройств

Драйверы — это системные программы, которые осуществляют управление различными устройствами, подключаемыми к ПЭВМ. Они могут быть добавлены или изменены при загрузке ОС и это дает возможность оперативно изменять старые либо добавлять новые устройства. При обращении к устройству MS-DOS обычно используют такую последовательность:

программа → функция_ОС → драйвер → функция_BIOS → устройство.

Например, если выполняется чтение данных с клавиатуры, то эта последовательность конкретизируется следующим образом:

программа → одна_из_функций_прерывания_21h → драйвер_консоли_с_именем_CON → функция_прерывания_16h → буфер_клавиатуры

Подробные сведения о драйверах можно получить в работах [2, 4, 7, 13, 29]. Здесь будет рассмотрено только то, что необходимо для написания программы конкретного драйвера для графопостроителя MP3100, взаимодействие с которым осуществляется по последовательному каналу. При этом будут использованы фрагменты программ для работы с последовательными каналами из § 6.2.

Драйверы устройств делятся на два класса: символьные и блочные. Первые принимают или передают за одно обращение один символ, а вторые блок символов. Примером символьного драйвера является драйвер консоли (клавиатура и дисплей). За одно обращение к этому драйверу либо считывается один символ из буфера клавиатуры, либо выдается один символ на экран дисплея. Примером блочного драйвера является драйвер диска. Здесь будет рассмотрен символьный драйвер.

Любой драйвер включает три части:

заголовок;

программу стратегии;

программу обработчика прерываний.

Структура каждой из этих частей подробно описана в работе [2, с. 61—64].

При обращении к драйверу ОС передает ему структуру, называемую заголовком запроса. Это специальная область памяти, через которую осуществляется обмен информацией между драйвером и вызывающей его программой. Размер заголовка может меняться в зависимости от выполняемых функций (инициализации, чтение данных, запись данных и т. п.).

Однако первые 13 байт всегда одинаковы и включают следующие данные:

- длина заголовка запроса — 1 байт;
- код, определяющий номер для блочных устройств,— 1 байт;
- номер последней команды, посланной драйверу,— 1 байт;
- статус, возвращаемый драйвером и указывающий его состояние после выполнения команды,—2 байт;
- резервная область — 8 байт.

Нас будут интересовать три команды: инициализации с номером 0, чтения данных с номером 4 и записи данных с номером 8. Заголовок запроса для команды инициализации продолжается следующим образом:

- номер узла для блочного устройства — 1 байт;
- адрес конца драйвера — 4 байт;
- указатель на блок параметров BIOS (BPB) для блочного устройства — 4 байт;
- номер следующего доступного узла (0 — А, 1 — В и т. п.) для блочного устройства — 1 байт;
- флаг управления выдачей ошибок для файла CONFIG.SYS — 2 байт (только для MS-DOS версии 4.0).

В процедуре инициализации для символьного драйвера необходимо вернуть байт статуса, характеризующий состояние устройства (в частности, наличие или отсутствие ошибки) и размер драйвера. Процедура инициализации выполняется только на этапе установки драйвера и далее она не нужна. Ее можно поместить в конец драйвера и определять размер драйвера от его начала до начала этой процедуры. Отметим, что процедура инициализации — это единственная часть драйвера, в которой можно обращаться к функциям ОС. В других частях драйвера это не разрешено. Напомним, что MS-DOS является нерентерабельной (не повторно входимой) системой. Среди функций ОС допустимыми являются 01h—0Ch и 30h (прерывание 21h). Другие функции системы могут быть еще не установлены, когда осуществляется установка драйвера.

Заголовок запроса символьного устройства для команд чтения и записи продолжается следующим образом:

- формат диска (только для блочных устройств) — 1 байт;
- адрес буфера, через который передаются данные,— 4 байт;
- число байтов, которые нужно передать (для символьного драйвера здесь будет записано значение 1),— 2 байт.

Выше уже говорилось, что после завершения работы драйвер должен записывать в заголовок запроса по смещению 3 байт статуса, биты которого имеют следующие значения:

если бит 15 установлен в единицу, то в битах 7—0 содержится код ошибки:

- 0—нарушение защиты записи;
- 1—неизвестное устройство;
- 2—устройство не готово;
- 3—неизвестная команда;
- 4—ошибка проверки по контрольной сумме;
- 5—ошибка в длине запроса к устройству;
- 6—ошибка поиска;
- 7—неизвестный носитель;
- 8—сектор не найден;
- 9—нет бумаги в принтере;
- 10—ошибка записи;
- 11—ошибка чтения;
- 12—общая ошибка;
- 13, 14—зарезервированы;
- 15—неверная замена диска;

если бит 15 установлен в нуль, то отсутствует ошибка и биты 7—0 не имеют значения;

если бит 8 установлен в единицу, то драйвер работает нормально;

если бит 9 установлен в единицу, то драйвер занят;

биты 10—14 не используются.

Ниже приводится пример программы драйвера для графопостроителя MP3100. Общие сведения об этом графопостроителе и формат его команд приведены в работах [1,4]. Предполагается, что графопостроитель связан с ПЭВМ по последовательному каналу COM1. Номер канала задается константой COMn в начале программы драйвера. Драйвер имеет имя PLT, заданное в его заголовке:

```
/* пример EX6_11 */
;*****
;* Пример программы драйвера для графопостроителя MP3100 *;
;*****
.MODEL TINY
THR EQU 0
RDR EQU 0
BRDL EQU 0
BRDH EQU 1
IER EQU 1
IIR EQU 2
LCR EQU 3
MCR EQU 4
LSR EQU 5
MSR EQU 6
COMn EQU 0 ; номер канала: 0 - COM1, 2 - COM2, 4 - COM3
; 6 - COM4
wait_delay EQU 300 ; максимальное время ожидания данных от
; графопостроителя (примерно 16,5 сек)
;*****
;* MACRO, описанные в программах для работы с *;
;* последовательными каналами *;
;*****
my_set MACRO COM, serial_port_register, value
    mov dx, [COM]
    add dx, serial_port_register
    mov al, value
```

```

        out dx,al
        ENDM
;
;
;
receive MACRO COM,serial_port_register
        mov dx,[COM]
        add dx,serial_port_register
        in al,dx
        ENDM
;
;*****;
.CODE
;*****;
;* Заголовок драйвера *;
;*****;-----;
        DD -1          ; Указатель на следующий заголовок
        DW 8000h        ; Атрибут (формат IBM - символьное устройство)
        DW STRATEGY     ; Указатель на процедуру "стратегия"
        DW IRPT         ; Указатель на обработчик прерываний
        DB 'PLT'        ; Имя драйвера (8 символов)
;*****;

;*****;
;*          Адрес заголовка запроса,          *;
;* сохраняемый в подпрограмме "стратегия" *;
;*****;-----;
        MEM_ES_BX LABEL DWORD ; MEM_ES_BX - Адрес четырех-
;          ; байтного указателя на заголовок запроса
        MEM_BX DW ?          ; Слово памяти для сохранения BX
        MEM_ES DW ?          ; Слово памяти для сохранения ES
;*****;

;*****;
;* Подпрограмма "стратегия" *;
;*****;-----;
        STRATEGY PROC FAR
                mov cs:MEM_BX,BX ; ES и BX сохраняются в
                mov cs:MEM_ES,ES ; MEM_ES и MEM_BX
                ret
        STRATEGY ENDP
;*****;

;*****;
;* Таблица точек входа в функции *;
;*****;-----;
        CMD_TABLE LABEL WORD
                DW offset init          ; 0 - команда инициализации
                DW offset no_install ; no_install- команда не
                DW offset no_install ; используется
                DW offset no_install ;
                DW offset input        ; 4 - команда ввода (Read)
                DW offset no_install ;
                DW offset no_install ;
                DW offset no_install ;
                DW offset output       ; 8 - команда вывода (Write);
                DW offset no_install ;
                DW offset no_install ; Здесь используются
                DW offset no_install ; только три команды
                DW offset no_install ; 0, 4 и 8
                DW offset no_install ;
                DW offset no_install ;
        MAX_CMD EQU ($-CMD_TABLE)/2 ; Максимальное значение кода;
;          ; команды (длина таблицы делится на 2, поскольку каждой
;          ; команде соответствуют 2 байта)
;*****;

```

```

;*****;
;* Обработчик прерываний *;
;*****;-----;
IRPT  PROC C FAR USES ds es ax bx cx dx di si
      lds bx,cs:MEM_ES_BX ; Передача адреса заголовка
                                ; запроса в DS:BX
      mov al,[bx+2]          ; Номер функции из заголовка
                                ; запроса в AL
      cbw                    ; Преобразование байта в слово (из al в ax)
      cmp al,MAX_CMD         ; Если номер функции больше допустимого,
      ja error_command       ; то перейти по ошибке
      add ax,ax              ; Удваивание номера функции для
                                ; индексации таблицы
      mov di,ax              ; Занесение индекса в регистр DI
      call cs:CMD_TABLE[di] ; Переход на заданную функцию
      jc error_detect        ; Ошибка, если CF=1
      jmp quit
error_command:                  ; Ошибочная команда
      or [bx+3],word ptr 8003h ; Модификация статуса (не-
                                ; известная команда)
error_detect:                  ;
      or [bx+3],word ptr 8002h ; Модификация статуса (ошиб-
                                ; ка, устройство не готово)
quit:
      or [bx+3],word ptr 100h  ; Нормальное завершение
      ret
IRPT  ENDP
;*****;

;*****;
;* Процедура no_install *;
;*****;-----;
no_install PROC
      ret ; В этой процедуре ничего не делается
no_install ENDP
;*****;

;*****;
;* Процедура input (вызывается по команде вида *;
;*      copy plt con) *;
;*****;-----;
input  PROC C USES ds bx
      cld ; Установка инкремента для строковых команд
      les di,[bx+14] ; В ES:DI - указатель на буфер данных
      call receive_  ; Вызов процедуры receive_ для приема
                                ; символа из последовательного канала
      jnc continue   ; Переход к метке, если нет ошибки
      ret            ; Выход при наличии ошибки
continue:
      cmp al,3        ; 3 - символ от графопостроителя, за-
                                ; вершающий прием данных
      je set_Ctrl_Z   ; Если 3, то переход к указанной метке
      stosb           ; Запись в буфер символа от графопостроителя
      jmp quit_       ; Переход к метке quit_
set_Ctrl_Z:
      mov al,26       ; Запись в буфер завершающего кода для сим-
      stosb           ; воляного драйвера (им является 26 - Ctrl-Z)
quit_:
      ret
input  ENDP
;*****;

```

```

;*****;
;* Процедура output вызывается по команде вида *;
;*          copu con plt)          *;
;*****;-----;
output PROC C USES ds bx
    lds si,[bx+14] ; В DS:SI - указатель на буфер данных ;
    cld ;
    lodsb ; Загрузка в регистр AL байта из буфера ;
    cmp al,'@' ; Если получен заключительный символ @, ;
    jne no_3 ; то записать в AL значение 3, что является ;
    mov al,3 ; заключительным кодом для графопостроителя ;
no_3: ; Не получен заключительный символ @ ;
    call send ; Вызов процедуры для передачи символа ;
    ; графопостроителю ;
    ret ;
output ENDP
;*****;

;*****;
;* Процедура send для передачи символа графопостроителю *;
;*****;--;
send PROC C USES DS
    push ax ; Сохранение передаваемого символа (AL) в стеке ;
    mov bx,COMn ; ВСЕ ПОСЛЕДУЮЩИЕ КОМАНДЫ ПОЯСНЕНЫ ;
    mov ax,40h ; В ПРИМЕРЕ EX6_4 ;
    mov ds,ax ;
again: ;
    xor cx,cx ; обнуление CX для цикла ;
delay: ;
    nop ; пустая операция ;
    loop delay ; цикл временной задержки ;
    receive bx,MSR ; Получение статуса готовности гра- ;
    test al,10000000b ; фопостроителя ;
    jz send_error ; Переход к метке, если не готов ;
    receive bx,LSR ; В BX - смещение базового адреса канала ;
    test al,00011110b ; Проверка байта статуса ;
    jnz send_error ; на наличие ошибки ;
    test al,00100000b ; Проверка на передачу данных ;
    jz again ; Нельзя передавать данные ;
    pop ax ; Восстановление передаваемого символа (AL) ;
    my_set bx,THR,al ; Передача символа в порт ;
    cld ; Сброс признака CF в регистре флагов ;
    ret ;
send_error: ;
    pop ax ; установка балланса стека ;
    stc ; Установка признака CF в регистре флагов при ошибке ;
    ret ;
send ENDP
;*****;

save_time DW 0 ; переменная для хранения текущего времени

;*****;
;* Процедура receive_ приема символа от графопостроителя *;
;*****;
receive_ PROC C USES DS
    xor ax,ax ; функция 0 прерывания 1Ah позволяет опреде- ;
    int 1Ah ; лить текущее время ;
    mov cs:save_time,dx ; Сохранение текущего значения ти- ;
    ; ков, возникающих примерно через 18,2 секунды ;
    add cs:save_time,wait_delay ; Увеличение числа тиков на ;

```



```

    ; wait_delay (в этом примере примерно на 16,5 секунды) ;
    mov bx,COMn      ; ВСЕ ПОСЛЕДУЮЩИЕ КОМАНДЫ ПОЯСНЕНЫ ;
    mov ax,40h       ; В ПРИМЕРЕ EX6_4 ;
    mov ds,ax ;
    my_set bx,MCR,00000011b ; Разрешение приема данных ;
again_:
    receive bx,MSR    ; Получение статуса готовности гра- ;
    test al,00010000b ; фопостроителя ;
    jz receive_error  ; Переход к метке, если не готов ;
    receive bx,LSR ;
    test al,00011110b ; Проверка байта статуса ;
    jnz receive_error ; на наличие ошибки ;
    test al,00000001b ; Проверка на получение данных ;
    jnz receive_data  ; Переход к фрагменту приема ;
    xor ax,ax          ; Вызов функции 0 прерывания 1Ah ;
    int 1Ah            ; Пребывание в цикле в течение ;
    cmp cs:save_time,dx ; 16,5 секунд вызовет выход по ;
    je receive_error   ; отсутствию готовности ;
    jmp short again_ ;
receive_data:
    receive bx,RDR      ; Фрагмент приема символа ;
    push ax             ; В AL полученный символ ;
    my_set bx,MCR,00000000b ; Сохранение (AL) ;
    pop ax              ; Запрет приема данных ;
    cld                 ; Восстановление (AL) ;
    cld                 ; Сброс признака CF в регистре флагов ;
    ret ;
receive_error:
    stc ; Установка признака CF в регистре флагов при ошибке ;
    ret ;
receive_ ENDP ;
;***** ;
;***** ;
;* Процедура инициализации драйвера * ;
;***** ;
init PROC ;
    push ds ;
    push cs ;
    pop ds ;
    mov ah,9 ;
    lea dx,The_first_message ;
    int 21h ; вывод первого сообщения ;
    pop ds ;
    mov word ptr ds:[bx+14],offset init ; запись смещения и ;
    mov word ptr ds:[bx+16],cs ; сегментного адреса ;
    ; завершающих резидентную часть драйвера ;
    ret ;
init ENDP ;
;***** ;
;***** ;
;* Сообщение, выводимое на экран при установке драйвера * ;
;***** ;
The_first_message db 10,13,13 ;
db '*****',10,13 ;
db '* Минский радиотехнический институт *',10,13 ;
db '* Сляров В.А. *',10,13 ;
db '* Драйвер PLT для графопостроителя *',10,13 ;
db '* MP3100 *',10,13 ;
db '*****',10,13,'$' ;
END ;
;***** ;

```

Полученный файл EX6_11.EXE необходимо преобразовать к типу SYS[2,4]. Для этого можно воспользоваться системной утилитой EXE2BIN.EXE:

```
C >exe2bin ex6_11 ex6_11.sys <ВВОД>
```

Драйвер должен быть задан в командном файле CONFIG.SYS:

```
DEVICE = EX6_11.SYS
```

Как обычно, при необходимости перед EX6_11.SYS указываются имя диска и соответствующий путь.

Драйвер устанавливается при загрузке операционной системы с использованием информации из файла CONFIG.SYS. Сразу же после загрузки на экране появится сообщение, приведенное в конце программы.

Прежде чем начать работу с драйвером, необходимо установить одинаковые параметры последовательных каналов в ПЭВМ и графопостроителе. В графопостроителе это можно сделать установкой микропереключателей, которые имеют маркировку SW1 и SW2. Параметры канала ПЭВМ устанавливаются системной утилитой mode. Например, чтобы задать для канала COM1 скорость обмена 9600 бод, 8 бит данных, 1 стоп-бит и проверку на четность, необходимо выполнить команду:

```
C >mode com1 baud = 96 data = 8 stop = 1 parity = E <ВВОД>
```

Чтобы установить аналогичные параметры в графопостроителе, необходимо задать: SW1 = 10011100, SW2 = 10000010.

Если ввести команду вида

```
C >copy con plt <ВВОД>
```

то далее можно передавать любые команды, которые воспринимаются графопостроителем. Цепочка команд завершается нажатием клавиш «Ctrl—Z» и «ВВОД». Например, если ввести

```
: < Ctrl — Z> <ВВОД>
```

то пишущий узел графопостроителя переместится в исходное состояние (в левый нижний угол). Напомним, что система команд графопостроителя MP3100 приведена в работе [4]. Если ввести команду G или C, то можно получить координаты текущего положения пишущего угла графопостроителя (если была введена команда C, то для получения координат на панели графопостроителя надо дополнительно нажать кнопку

«ENTER»). Получение координат и вывод их на экран дисплея осуществляется с помощью команды вида

```
C > copy plt con <ВВОД>
```

Дадим краткое пояснение программы драйвера. Ввиду того что это упрощенная программа, в ней в байт статуса возвращаются только некоторые коды ошибок, к которым относятся:

неизвестная команда (код 3);

устройство не готово (код 2).

Статус отсутствия готовности будет выдаваться в двух случаях:

графопостроитель не подключен к каналу либо на него не подано напряжение питания;

текущие координаты графопостроителя не поступали в течение примерно 16,5 с.

После получения такого кода ошибки ОС будет выдавать сообщение:

```
Not ready reading device PLT
Abort, Retry, Ignore, Fail?
```

Для ответа можно нажать клавишу «А». Формирование байта статуса осуществляется в обработчике прерываний.

Многие команды графопостроителя должны завершаться символом терминатора (им является 3). Код значения 3 неудобно вводить с клавиатуры. С этой целью программа воспринимает вместо цифры 3 символ @. В процедуре output введенный с клавиатуры символ @ изменяется на цифру 3 для графопостроителя.

Прием текущих координат графопостроителя также должен завершаться после поступления от него цифры 3. В этом случае в буфер драйвера необходимо записать завершающий код символа Ctrl—Z (код 26). Это делается в процедуре input.

Размер драйвера определяется от его начала до начала процедуры init. Он вычисляется в процедуре init и возвращается в ней в байты 14, 15, 16, 17 заголовка запроса для процедуры init (напомним, что эта процедура вызывается только при установке драйвера).

Рассмотрим пример программы на языке Си, которая использует написанный драйвер для выдачи данных на графопостроитель и получения данных от графопостроителя:

```
/* пример EX6_12 */
#include <stdio.h>
#include <dos.h>
void main(void)
{ FILE *fp,*fr;
  unsigned char str1[50];
  fp = fopen("plt","w"); /* открытие устройства PLT
```

```

                                для записи */
fr = fopen("plt","r"); /* открытие PLT для чтения */
fprintf(fp,""); /* инициализация графопостроителя */
fprintf(fp,"M1500,2000,"); /* перемещение поднятого
пишущего элемента в точку с координатами (1500,2000) */
fprintf(fp,"G"); /* выдача текущих координат */
fgets(str1,23,fr); /* прием текущих координат */
str1[21] = 0; /* запись заключительного нуля */
fprintf(fp,"PMinsk Radioengineering Institute@");
/* вывод текста Minsk Radioengineering Institute */
fprintf(fp,"O200,200@"); /* перемещение поднятого пишущего
элемента в точку с относительными координатами
(200,200) */
fprintf(fp,"S50,50,PComputer Department@");
/* вывод текста Computer Department */
fclose(fp); /* закрытие PLT для записи */
printf("\nТекущие координаты: %s\n",str1);
/* вывод на экран текущих координат */
fclose(fr); /* закрытие PLT для чтения */
}

```

В результате ее работы пишущий узел графопостроителя перемещается в точку с координатами 1500, 2000; эти координаты возвращаются и выдаются на экран дисплея (в функции printf); графопостроитель выводит текст «Minsk Radioengineering Institute»; происходит перемещение пишущего узла и выводится новый текст: Computer Department.

Приведенный выше драйвер может быть легко переделан таким образом, чтобы он использовал параллельный канал для передачи данных на графопостроитель. В этом случае необходимо использовать фрагмент программы из § 6.3.

ПРИЛОЖЕНИЯ

П. 1. Общие сведения о примененных конструкциях языка Ассемблера

Здесь рассматриваются только те конструкции ассемблера, которые использованы в приведенных выше программах. Для более детального изучения этого языка следует воспользоваться книгами, приведенными в списке литературы.

В языке Ассемблера имеются операторы двух видов: команды, которые транслируются в машинные инструкции, и директивы, выполняющие различные указания (выделение памяти и т. п.).

Общий формат ассемблерной команды имеет вид

Метка Мнемоника_команды Операнды ; комментарии

Минимум один пробел должен быть там, где его отсутствие приводит к неоднозначности соответствующей записи. Наличие метки в команде не обязательно. Мнемоника — это обязательный параметр. Число операндов может быть от 0 до 4 и зависит от вида команды. Различные операнды должны быть согласованы по длине. Рассмотрим пример типичной команды пересылки данных:

```
mov AX,BX
```

Здесь содержимое регистра BX пересылается в AX.

Другая команда:

```
mov AH,AL
```

пересылает младший байт регистра AX в его старшую половину (AH).

Команда вида

```
mov BL,ES:[17H]
```

заносит в регистр BL содержимое байта со смещением 17H в сегменте ES. Здесь ES — префикс замены, указывающий, что второй операнд находится в сегменте ES. Квадратные скобки говорят о том, что значение необходимо взять по адресу (в примере — из байта со смещением 17H в сегменте ES). В подобных командах часто используются директивы вида BYTE PTR, WORD PTR и т. п., например:

```
mov BL, byte ptr ES:[17H]
```

Здесь явно указано, что из памяти необходимо передать один байт. В случае директивы WORD PTR будет передаваться слово.

При записи инструкций большие и малые буквы не различаются. Для шестнадцатеричных значений в конце добавляется буква H и они должны начинаться с цифры, например 0F6EH, 55H. По умолчанию все значения являются десятичными.

В большинстве случаев ассемблерный модуль будет состоять из трех секций или менее: кода, инициализированных данных и неинициализированных данных. Каждая секция организована в сегмент, который использует определенные соглашения об именах. Если для указанных секций используются сегментные директивы .CODE, .DATA, .DATA?, то выбираются стандартные имена, которые определяются заданной моделью памяти (директива .MODEL). Например, для модели SMALL общая структура программы может быть представлена в виде

```
.MODEL SMALL ; объявление малой модели памяти
.STACK 20H   ; объявление стека размером 20H = 32
.DATA        ; начало инициализированного сегмента данных
              ; объявления данных
.DATA?       ; начало неинициализированного сегмента данных
              ; объявления данных

.CODE        ; начало сегмента кода (программы)
start:       ; метка начала программы
имя PROC     ; начало программного модуля «имя»
              ; тело программы
имя ENDP     ; конец программного модуля «имя»
              ; другие программные модули
END start    ; конец программы
```

При необходимости можно для требуемой модели памяти изменить имена сегментов, заданные по умолчанию. Ниже приведен общий формат ассемблерного файла:

```
code         SEGMENT BYTE PUBLIC 'CODE'
ASSUME       CS:code, DS:dseg
              сегмент кода
code         ENDS
dseg         GROUP _DATA, _BSS
data         SEGMENT WORD PUBLIC 'DATA'
              инициализированный сегмент данных
data         ENDS
_BSS         SEGMENT WORD PUBLIC 'BSS'
              неинициализированный сегмент данных
_BSS         ENDS
END
```

Идентификаторы code, data и dseg будут заменены специальными именами, которые определяются табл. П1.1 в соответствии с выбранной моделью памяти. Параметр

filename в табл. П1.1— это имя модуля (директивы DD и DW описаны ниже). Поясним другие используемые директивы:

SEGMENT — определяет начало сегмента с заданным именем;

ENDS — определяет конец сегмента с заданным именем;

ASSUME — указывает ассемблеру соответствие между сегментными регистрами и сегментами программы (но не устанавливает сегментные регистры);

BYTE, WORD — задают способ выравнивания сегмента по границе байта (BYTE) или слова (WORD);

PUBLIC — указывает на необходимость объединения всех сегментов с одним и тем же именем в единый большой сегмент;

'CODE', 'DATA', 'BSS' — типы сегментов, соответственно кода или программы ('CODE'), инициализированных данных ('DATA') и неинициализированных данных ('BSS');

EQU — определяет имя name, которое является результатом вычисления заданного выражения expression. Формально эта директива записывается в следующем виде: name EQU expression. Она подобна директиве языка C вида #define name expression.

GROUP — связывает имя группы с именами одного или более сегментов (в примере с сегментом инициализированных данных — _DATA и сегментом неинициализированных данных — _BSS).

Таблица П1.1

Модель памяти	Способ замены идентификатора	Указатели на код (Code) и данные (Data)
Tiny, Small	code = _TEXT data = _DATA dseg = DGROUP	Code: DW _TEXT:xxx Data: DW DGROUP:xxx
Compact	code = _TEXT data = _DATA dseg = DGROUP	Code: DW _TEXT:xxx Data: DD DGROUP:xxx
Medium	code = filename_TEXT data = _DATA dseg = DGROUP	Code: DDxxx Data: DW DGROUP:xxx
Large	code = filename_TEXT data = _DATA dseg = DGROUP	Code: DDxxx Data: DD DGROUP:xxx
Huge	code = filename_TEXT data = filename_DATA	Code: DDxxx Data: DDxxx

MACRO — определяет макросредство с заданным именем name и записывается в виде

```
name MACRO [parameter[,parameter]...]
    тело MACRO
ENDM
```

Если далее в теле программы появляется имя name, то на его место будет подставлено тело MACRO. В тело MACRO могут передаваться параметры, которые будут подставлены на место соответствующих символических имен.

USES — сохраняет в стеке регистры, указанные в качестве операндов при вызове процедуры, и после завершения процедуры восстанавливает эти регистры из стека.

Инструкция вида

```
mov AX, @DATA  
mov DS, AX
```

устанавливает сегментный регистр DS на раздел .DATA.

Аналогичные инструкции:

```
mov AX, DGROUP  
mov DS, AX
```

устанавливают DS на секцию GROUP, включающую инициализированные и неинициализированные данные. Заметим, что во многих программах раздел неинициализированных данных отсутствует.

К часто применяемым директивам языка относятся:

KODL DB 'строка' — для выделения памяти под текст "строка" и наименования начала этой области посредством идентификатора KODL;

DB 13 DUP(0) — для выделения памяти под 13 байт и занесения в нее нулевых значений;

DB 27 — для выделения памяти размером один байт со значением 27;

DB ? — для выделения памяти размером один байт (здесь начальные значения не задаются);

если вместо DB использовать DW, то память измеряется не в байтах, а в словах (2 байт), а если DD, то в двойных словах (4 байт).

Охарактеризуем команды ассемблера, используемые в приведенных выше программах:

add — сложить, например, add AL, 85 (к AL прибавляется 85 и результат сохраняется в AL);

and — выполняет операцию поразрядное "И" двух операндов. Записывает результат на место первого операнда;

call — вызов процедуры, например call scr_sim;

cld — установка запрета внешних прерываний;

crr — вычитает второй операнд из первого, но не формирует результат, а только устанавливает признаки, которые далее можно использовать для выполнения условных переходов;

in — передает байт или слово из порта, заданного вторым операндом, в регистр al или ah, указанный в качестве первого операнда;

int — вызов прерывания, например, int 21H;

iret — возврат из процедуры — обработчика прерываний;

jmp — переход на метку, например jmp quit;

jxx — выполняет команды условных переходов. Здесь на место символов xx могут записываться различные буквы; lea — загрузить эффективный адрес, например lea ax, [bp-4] (в них загружается адрес, который определяется смещением bp-4);

mov — переслать операнд справа в операнд слева, например mov AH, 31H;

mul — умножить без знака, например, mul BL (здесь AL умножается на BL и результат сохраняется в AX; операнды могут быть и словами, например mul BX, тогда результат будет в DX:AX);

por — холостая команда (нет операции);

or — выполняет операцию поразрядное "ИЛИ" двух операндов. Записывает результат на место первого операнда;

out — передает байт или слово, заданное вторым операндом (в регистре al или ax) в порт, номер которого определяется первым операндом;
 pop — извлечь слово из стека, например pop DS;
 push — поместить слово в стек, например push DS;
 ret — возврат из процедуры;
 shr — сдвинуть логически вправо, например, shr AX,2 (сдвиг всех битов в AX на две позиции вправо, крайний правый бит выдвигается во флажок CF);
 sti — установка разрешения внешних прерываний;
 sub — вычесть, например, sub AX,BX (из AX вычитается BX и результат сохраняется в AX);
 test — выполняет поразрядное логическое "И" двух операндов, но не формирует результат, а только устанавливает признаки, которые далее можно использовать для выполнения условных переходов;
 xchg — обменивает значения двух операндов. Один из операндов должен быть в регистре микропроцессора, а другой может быть в регистре или в памяти;
 xor — выполняет операцию поразрядное "исключающее ИЛИ" двух операндов. Записывает результат на место первого операнда.

П.2. Краткие сведения об опциях для различных программ в системе программирования Borland C++

Программа BC.EXE. Общий синтаксис вызова программы:

BC [исходный_файл] [имя_проекта] [опция[опция...]]

Опции:

/b — вызывает повторную компиляцию и компоновку файлов проекта.

После этого осуществляется автоматический выход в среду ОС;

/m — делает то же, что и /b, но учитывает даты создания файлов;
 /d — позволяет работать с двумя мониторами (дисплеями);
 /e — позволяет использовать память типа expanded;
 /x — позволяет использовать память типа extended;
 /gx — позволяет использовать память типа extended или expanded (здесь x — логическое имя диска с этой памятью);
 /l — позволяет использовать жидкокристаллический экран;
 /p — позволяет восстанавливать регистры палитры адаптера EGA.

Программа BCC.EXE. Представляет собой автономный компилятор. Запускается из командной строки операционной системы; позволяет выполнять компиляцию и компоновку программ на языках Си и Си++.

Общий синтаксис вызова программы:

BCC [опция[опция...]] имя_файла [имя_файла...]

Опции:

-A — разрешает использование только таких ключевых слов Borland C++, которые соответствуют ANSI-стандарту;
 -A — или -AT — разрешает использование любых ключевых слов (устанавливается по умолчанию);
 -AK — разрешает использование только таких ключевых слов, которые введены Керниганом и Ритчи;

- AU — разрешает использование только таких ключевых слов, которые есть в стандарте UNIX;
- a — задает выравнивание по границе слова;
- a — задает выравнивание по границе байта (задается по умолчанию);
- B — включает режим inline (однострочного ассемблера);
- b — объекты типа enum будут иметь размер в одно слово (задается по умолчанию);
- C — разрешает использование вложенных комментариев;
- c — выполняется только компиляция программы;
- Димя — определяет "имя";
- Димя-строка — определяет "имя" через параметр "строка";
- d — включает режим слияния одинаковых строк;
- d — выключает режим слияния одинаковых строк (задается по умолчанию);
- Еимя_файла_с_компилятором_ассемблер — используется для задания нового компилятора ассемблер (не TASM);
- еимя_файла — выполняет компоновку программы;
- f — эмуляция режима плавающей точки (задается по умолчанию);
- f — отмена режима плавающей точки;
- f87 — использование инструкций сопроцессора 8087;
- f287 — использование инструкций сопроцессора 80287;
- G — оптимизация программы по скорости выполнения;
- G — оптимизация программы по размеру;
- gn — прекращение компиляции после n предупреждений;
- имя_каталога — задание каталога для включаемых файлов;
- in — длина идентификатора будет ограничена значением n;
- jn — прекращение компиляции после n ошибок;
- K — символьные переменные будут иметь тип unsigned по умолчанию;
- K — символьные переменные будут иметь тип signed по умолчанию (задается по умолчанию);
- k — используется стандартный фрейм стека (задается по умолчанию);
- Лимя_каталога — задание каталога для библиотечных файлов;
- lx — передает опцию X компоновщику;
- l-x — запрещает опцию x для компоновщика;
- M — указывает компоновщику на необходимость создания map файла;
- mx — устанавливает модель памяти x. Здесь x — одна из шести букв: c, h, l, m, s, t (в соответствии с требуемой моделью памяти);
- mm! — устанавливает модель medium (здесь DS не равно SS);
- ms! — устанавливает модель small (здесь DS не равно SS); _mt! — устанавливает модель tiny (здесь DS не равно SS);
- N — включается проверка на переполнение стека;
- пвыходной_каталог — задается выходной каталог;
- O — включается режим оптимизации переходов;
- O — выключается режим оптимизации переходов (задается по умолчанию);
- оимя_файла — задает имя файла после компиляции — имя_файла.OBJ;
- P — для любых файлов включает компилятор Borland C++;
- р — используется паскалевское соглашение о передаче параметров;
- р — используется соглашение языка C о передаче параметров (задается по умолчанию);
- Qe — разрешает компилятору использовать всю доступную expanded память (задается по умолчанию);
- Qe — запрещает компилятору использовать expanded память;
- Qx — разрешает компилятору использовать всю доступную extended память (задается по умолчанию);
- Qx-n — требует, чтобы компилятор оставил n Кбайт extended памяти для других программ;

- Qx — запрещает компилятору использовать extended память;
- r — разрешает использование регистровых переменных (задается по умолчанию);
- r — запрещает использование регистровых переменных;
- rd — разрешает сохранять во внутренних регистрах микропроцессора только объявленные регистровые переменные;
- S — строит ассемблерный выходной файл (*.ASM);
- Топция — передает опцию ассемблеру;
- T — удаляет все предыдущие ассемблерные опции;
- Имя — делает неопределенным ранее определенное "имя";
- u — автоматическая генерация подчеркиваний перед именами в языке Си++ (задается по умолчанию);
- v — задание символьной информации для отладчика (при этом запрещается использование встроенных функций);
- v — удаление символьной информации для отладчика (при этом разрешается использование встроенных функций);
- vl — разрешает использование встроенных (inline) функций;
- v — запрещает использование встроенных функций;
- w — отображает сообщения с предупреждениями;
- w — не отображает сообщения с предупреждениями;
- wx — разрешает вывод предупреждения x;
- w-x — запрещает вывод предупреждения x;
- X — удаляет дополнительную информацию компилятора из выходных файлов (обычно используется только для файлов, включаемых в библиотеку);
- Y — разрешает генерацию оверлейного кода;
- Yo — компилируемый файл является оверлейным модулем;
- y — включает нумерацию линий в объектном файле (это может быть необходимо для символьного отладчика);
- Z — разрешает регистровую оптимизацию;
- ZAnnn — изменяет имя класса (по умолчанию CODE) для сегмента кода на nnn;
- ZBnnn — изменяет имя класса (по умолчанию BSS) для неинициализированного сегмента данных на nnn;
- ZCnnn — изменяет имя (по умолчанию TEXT) сегмента кода на nnn (для моделей medium, large и huge по умолчанию задается имя filename_TEXT, где filename — имя исходного файла);
- ZDnnn — изменяет имя (по умолчанию BSS) неинициализированного сегмента данных на nnn;
- ZEnnn — изменяет имя сегмента с дальними объектами (по умолчанию FAR) на nnn;
- ZFnnn — изменяет имя класса на дальних объектах (по умолчанию FAR_DATA) на nnn;
- ZGnnn — изменяет имя группы (по умолчанию DGROUP) для неинициализированного сегмента данных на nnn;
- ZHnnn — требует, чтобы дальние объекты попадали в группу nnn;
- ZPnnn — любой выходной файл создается в сегменте кода с именем nnn;
- ZRnnn — изменяет имя (по умолчанию DATA или filename_DATA) инициализированного сегмента данных на nnn;
- ZSnnn — изменяет имя группы (по умолчанию DGROUP) для инициализированного сегмента данных на nnn;
- ZTnnn — изменяет имя класса (по умолчанию DATA) для инициализированного сегмента данных на nnn;
- ZX* — использует для объекта X имя, заданное по умолчанию;
- 1 — используются команды микропроцессора 80186;
- 1- — используются команды микропроцессора 8086 и команды микропроцессора 80286 для реального режима (задается по умолчанию);

-2 — используются команды микропроцессора 80286 для защищенного режима.

Все перечисленные режимы можно также установить в интегрированной среде Borland C++ (файл BC.EXE).

Программа TLIB.EXE. Представляет собой программу-библиотекарь. Позволяет создавать, изменять и редактировать библиотечные файлы типа LIB.

Общий синтаксис вызова программы:

TLIB имя_библиотеки [/C[/E[/Размер[Опции[.список_файлов]

Опции:

/C — различает символы верхнего и нижнего регистров в именах файлов, включаемых в библиотеку;

/E — создание расширенного словаря для больших библиотечных файлов (это повышает скорость работы компоновщика);

/P — размер — установка размера библиотечной страницы (параметр "размер" может принимать значения от 16 до 32768);

+ — добавить файл с заданным именем в библиотеку (обычно это файл типа OBJ). Если задается библиотечный файл (типа LIB), то все модули этого библиотечного файла будут включены в библиотеку;

- — удалить файл с заданным именем из библиотеки;

* — создает файл с заданным именем путем копирования в него соответствующей программы из библиотеки;

-* или *- — извлекает файл с заданным именем из библиотеки;

+ или + — замещает файл в библиотеке (файл с указанным именем заменяет соответствующий файл в библиотеке).

Программа TLINK.EXE. Представляет собой автономный компоновщик. Запускается из командной строки операционной системы.

Общий синтаксис вызова программы:

TLINK OBJ-файлы EXE-файлы MAP-файлы LIB-файлы

Здесь MAP-файл — это специальный файл, который всегда создается компоновщиком для выполняемой программы. Он включает список сегментов программы, стартовые адреса, сообщения о предупреждениях и ошибках, которые возникли в процессе компоновки.

Опции:

/m — в MAP-файл будет добавлен список внешних символов;

/x — не создается MAP-файл;

/I — присоединение к выполняемому файлу сегментов с неинициализированными данными;

/I — создание в MAP-файле секции с номерами линий исходного кода;

/s — в дополнение к опции /m включает в MAP-файл подробную карту сегментов;

/n — отсутствуют библиотеки по умолчанию;

/d — выдача предупреждений при наличии повторяющихся символов в библиотеке;

/с — различает символы нижнего и верхнего регистров для внешних переменных (типа public и external);

/3 — позволяет использовать 32-битовый код микропроцессора 80386;

/v — включение символьной информации для отладчика;
/e — игнорирование расширенных каталогов для библиотечных файлов;
/t — создание COM-файла;
/o — поддержка оверлеев;
/ye — использование памяти типа expanded;
/yx — использование памяти типа extended.

П. 3. Справочные сведения по библиотечным функциям языков Си и Си++, примененным в программах книги

Ниже дается краткое пояснение всех библиотечных функций, которые применены в рассмотренных выше программах. Функции для работы с файлами описаны в § 2.3.

Функция `bioscom` описана в файле `bios.h` и имеет следующий прототип:

```
int bioscom(int cmd, char abyte, int port);
```

Она выполняет различные операции с коммуникационным каналом RS232. Значение переменной `port` определяет номер порта (0—COM1, 1—COM2 и т. п.). Переменная `cmd` может принимать следующие значения:

- 0— для надстройки канала с помощью значения `abyte`;
- 1— для отправки символа `abyte` в канал;
- 2— для получения символа из канала;
- 3— для получения статуса канала.

В переменную `abyte` могут быть записаны следующие значения: 0x02— в отсылке 7 бит данных, 0x03— в отсылке 8 бит данных, 0x00— 1 стоп-бит, 0x04— 2 стоп-бита, 0x00— отсутствует проверка, 0x08— проверка на нечетность, 0x18— проверка на четность, 0x00— скорость 110 бод, 0x20— скорость 150 бод, 0x40— скорость 300 бод, 0x60— скорость 600 бод, 0x80— скорость 1200 бод, 0xA0— скорость 2400 бод, 0xC0— скорость 4800 бод, 0xE0— скорость 9600 бод. Например, значение 0xEB (0xE0:0x08:0x00:0x03) задает скорость 9600 бод, нечетный паритет, 1 стоп-бит и 8 бит данных. Возвращаемое функцией значение содержит в старшем байте биты статуса канала. Значение младшего байта определяется типом выполняемой команды.

Функция `circle` описана в файле `graphics.h` и имеет следующий прототип:

```
void far circle(int x, int y, int radius);
```

Она позволяет отобразить окружность на экране дисплея и имеет три параметра целого типа. Первые два задают горизонтальную и вертикальную координаты центра окружности, а третий — ее радиус.

Функция `closegraph` описана в файле `graphics.h` и имеет следующий прототип:

```
void far closegraph(void);
```

Эта функция закрывает графическую систему.

Функция `clrscr` описана в файле `conio.h` и имеет следующий прототип:

```
void clrscr(void);
```

Она очищает текущий текстовый экран (текущее текстовое окно) и помещает курсор в левый верхний угол экрана.

Функция `exit` описана в файлах `stdlib.h`, `process.h` и имеет следующий прототип:

```
void exit(int status);
```

Она завершает программу и возвращает значение `status`. Обычно при нормальном завершении задается `status = 0`.

Функция `fread` описана в файле `stdio.h` и имеет следующий прототип:

```
unsigned fread(void *ptr, unsigned size, unsigned n, FILE *st);
```

Она читает `n` элементов данных, длиной `size` байт каждый, из заданного входного потока `st` в блок, на который указывает `ptr`. Общее число прочитанных байт равно `n*size`. При успешном завершении `fread` возвращает число прочитанных элементов данных, при ошибке — 0.

Функция `fwrite` описана в файле `stdio.h` и имеет следующий прототип:

```
unsigned fwrite(const void *ptr, unsigned size, unsigned n, FILE *st);
```

Она добавляет `n` элементов данных, длиной `size` байт каждый, в заданный выходной файл `st`. Данные записываются начиная с позиции `ptr`. При успешном завершении возвращается число записанных элементов данных, при ошибке — неверное число элементов данных.

Функция `getch` описана в файле `conio.h` и имеет следующий прототип:

```
int getch(void);
```

Она читает символ с клавиатуры без вывода его на экран. Если прочитан ноль, то следующий вызов функции `getch` позволяет прочитать расширенный код символа.

Функция `getche` аналогична `getch` и отличается тем, что читает символ с клавиатуры с выводом его на экран.

Функция `getcolor` описана в файле `graphics.h` и имеет следующий прототип:

```
int far getcolor(void);
```

Она возвращает целое значение, равное номеру установленного цвета.

Функция `gets` описана в файле `stdio.h` и имеет следующий прототип:

```
char *gets(char *s);
```

Она получает строку *s* из входного потока *stdin*. При успешном завершении возвращается указатель на введенную строку, при ошибке — значения 0 или EOF.

Функция *gotoxy* описана в файле *conio.h* и имеет следующий прототип:

```
void gotoxy(int x,int y);
```

Она перемещает курсор в заданную координатами (*x*, *y*) позицию текстового экрана или текстового окна. Если координаты ошибочны, то вызов этой функции игнорируется.

Функция *initgraph* описана в файле *graphics.h* и имеет следующий прототип:

```
void far initgraph(int far *gd,int far *gm,char far *path);
```

Функция *initgraph* инициализирует графическую систему путем загрузки графического драйвера с диска и установки системы в графический режим. Графические драйверы находятся в файлах *.BGI. Первый параметр функции *initgraph* — это указатель на переменную (*gd*) целого типа. Значение этой переменной позволяет выбрать нужный графический драйвер. Если задать *gd* = DETECT (DETECT — это константа нуль), то требуемый драйвер будет выбран автоматически. Второй параметр *initgraph* — это тоже указатель на переменную (*gm*) целого типа. Эта переменная будет определять установленный графический режим. Если задать *gd* = DETECT, то *gm* определит режим с наибольшей разрешающей способностью для автоматически выбранного драйвера. Третий параметр *initgraph* — это указатель на символ, с которого начинается строка с маршрутом и именем графического драйвера. Если эта строка задана в виде "", то поиск драйвера будет осуществлен в текущем директории.

Функция *putch* описана в файле *conio.h* и имеет следующий прототип:

```
int putch(int ch);
```

Она выводит символ *ch* в текущее текстовое окно экрана. При успешном завершении возвращается выводимый символ *ch*, при ошибке константа EOF.

Функция *putpixel* описана в файле *graphics.h* и имеет следующий прототип:

```
void far putpixel(int x,int y, int color);
```

Эта функция позволяет вывести точку на экран дисплея и имеет три параметра целого типа. Первые два из них задают горизонтальную и вертикальную координаты точки, а третий — цвет точки.

Функция `setcolor` описана в файле `graphics.h` и имеет следующий прототип:

```
void far setcolor(int color);
```

Она делает текущим цвет, номер которого задан в виде параметра целого типа.

Функция `strcmp` описана в файле `string.h` и имеет следующий прототип:

```
int strcmp(const char *s1, const char *s2);
```

Она выполняет беззнаковое сравнение строк `s1` и `s2`, начиная с первого символа и кончая символом, для которого произошло несравнение, или концом строки. Эта функция возвращает следующие значения:

< 0, если `s1` меньше, чем `s2`;

0, если `s1` равна `s2`;

> 0, если `s1` больше, чем `s2`.

Функция `strcpy` описана в файле `string.h` и имеет следующий прототип:

```
char *strcpy(char *dest, const char *src);
```

Она копирует строку `src` в строку `dest`. Копирование выполняется до достижения заключительного нуля. Функция возвращает указатель на `dest`.

П. 4. Пример анализа и проектирования объектно-ориентированной программы

Языки программирования можно рассматривать как инструмент для создания различных программных систем. Каждый язык поддерживает конкретную технологию программирования. Сложность современных программных систем постоянно увеличивается. Каждая конкретная технология программирования имеет определенный предел, характеризующий максимально-допустимую сложность создаваемого программного продукта. Детальный анализ существующих технологий программирования и их возможностей выполнен в книге [33]. Широко используемый подход к созданию программных систем базируется на технологиях процедурного и модульного программирования. Основная идея этих подходов может быть выражена в форме:

ПРОГРАММА = АЛГОРИТМ + ДАННЫЕ

Разработчик языка C++ Bjarne Stroustrup определил идеи процедурного и модульного программирования следующим образом: «Decide which procedures you want; use the best algorithms you can find» — выберите необходимые Вам процедуры, используйте лучший алгоритм, который Вы можете найти: «Decide which modules you want; partition the program

so that data is hidden in modules» — выберите необходимые Вам модули, стройте программу таким образом, чтобы модули использовали свои собственные внутренние (скрытые) данные [34]. Языки, поддерживающие процедурное и модульное программирование (Си, Паскаль, Модула и др.), в первую очередь ориентированы на задачи, имеющие явно выраженные алгоритмические особенности (математические вычисления и т. п.). В настоящее время большинство современных задач не являются вычислительными в традиционном смысле этого слова. К ним относятся многие задачи автоматизации проектирования, системы управления базами данных, системы, поддерживающие интерфейс с пользователем, операционные системы и т. п. Использование языков процедурного и модульного программирования для разработки подобных систем не всегда эффективно и имеет явно выраженные ограничения по сложности решаемых задач. Указанные особенности явились основной причиной разработки новой технологии объектно-ориентированного программирования.

Основная идея объектно-ориентированного программирования (ООП) впервые была реализована в языке Simula-67 и далее развита в языках семейства Smalltalk. В действительности именно Smalltalk можно рассматривать как первый язык объектно-ориентированного программирования (см. § 3.1). Технология ООП с самого начала была ориентирована на решение задач очень большой размерности. Книга [33] содержит детальное обоснование этого утверждения.

Основная идея ООП может быть выражена в форме:

ПРОГРАММА = ОБЪЕКТЫ + СООБЩЕНИЯ

Bjarne Stroustrup определил идею ООП следующим образом: «Decide which classes you want; provide a full set of operations for each class; make commonality explicit by using inheritance» — выберите необходимые Вам классы, определите все необходимые функции для объектов каждого класса (определите полный набор операций или компонентов-функций для каждого класса), используйте свойство наследования при построении сложных иерархических систем (при построении новых классов пытайтесь использовать то, что было создано ранее через свойство наследования) [34].

Выше отмечалось, что основная цель ООП — это создание очень сложных программных систем. В книге [33] приведены результаты исследований об ограничении человеческих способностей по созданию таких систем. Один из подходов по преодолению этих ограничений основан на декомпозиции, которая предполагает иерархическое разделение сложной системы на подсистемы до тех пор, пока сложность очередной подсистемы не будет ниже критического предела, определяющего возможность ее проектирования. Технология ООП

непосредственно поддерживает объектно-ориентированную декомпозицию, которая имеет много отличий от алгоритмической декомпозиции. Результатом декомпозиции являются множества объектов, которые имеют некоторое уникальное (собственное и характеризующее эти объекты) поведение. Объект можно рассматривать как ощущаемую реальность. Он включает данные, над которыми выполняются операции, и функции, обычно выполняющие эти операции и обеспечивающие внешний интерфейс. В различных литературных источниках программы-функции объекта называют по-разному: компоненты-функции, методы, просто функции и т. п.

Рассмотрим содержательный пример проектирования объектно-ориентированной программы, приведенный в работе [35]. Предположим, что необходимо построить модели устройств, используемых в изделиях вычислительной техники. Рассмотрим объект, который назовем РЕГИСТР. Будем рассматривать этот объект как программную модель реальной цифровой электронной схемы, используемой для хранения информации (см., например, регистры микропроцессора 8086, описанные в § 2.5). Предположим, что наш регистр является 10-разрядным (10-битным) и что мы можем выполнять с ним операции чтения 10-разрядного слова и записи 10-разрядного слова. Построим объект РЕГИСТР, который будет включать следующие компоненты:

- state (компонент данные);
- READ (компонент функция);
- WRITE (компонент функция).

Теперь можно предложить простейший сценарий для работы с нашим объектом:

записать значение N в РЕГИСТР (поскольку РЕГИСТР — это имя нашего объекта, то соответствующую команду можно выразить в следующем сообщении «запиши в себя значение N»);

прочитать значение из РЕГИСТРА.

Для выполнения этих простейших действий необходимо послать сообщения объекту, например «запиши в себя значение N».

При проектировании сложного цифрового устройства требуется много типов простейших регистров, например регистры, имеющие различную разрядность (а не точно 10 бит). С одной стороны, разные регистры слегка различаются. Однако, с другой стороны, они имеют полностью однотипное и хорошо определенное поведение. В результате можно создать класс регистров. Классы и объекты являются главными нововведениями ООП. В действительности их нельзя рассматривать независимо друг от друга. Главное различие между ними заключается в следующем: объект — это конкретная ощущаемая реальность, существующая во времени и в

пространстве; класс — это только абстракция [33]. Grady Booch определил класс следующим образом: «A class is a set of objects that share a common structure and a common behaviour» — класс — это множество объектов, имеющих общую структуру и общее поведение [33]. Дадим нашему классу имя REGISTER. Предположим, что мы хотим определить объект класса REGISTER. Когда мы определяем объект, было бы полезно сделать некоторые начальные установки, например задать конкретную разрядность регистра и его начальное состояние. Чтобы задать разрядность, необходимо выделить память (напомним, что наш объект — это модель регистра в памяти компьютера). Другими словами, необходимо:

выделить память для задания разрядности конкретного регистра;

выполнить начальную установку регистра.

Язык Си++ непосредственно поддерживает эти операции с помощью конструктора, который строит конкретный объект и инициализирует его данные. В нашем примере конструктор строит конкретный регистр с заданной разрядностью (например, SIZE) и устанавливает его в заданное начальное состояние (например, INITIAL_STATE). В результате наш класс может быть определен следующим образом:

```
class REGISTER
{   BOOLEAN *state;
    unsigned size;
public:
    REGISTER(unsigned SIZE, BOOLEAN *INITIAL_STATE);
    void WRITE(BOOLEAN *NEW_STATE);
    void READ(BOOLEAN *CURRENT_STATE);
    . . . . .
};
```

Выше отмечалось, что конструктор (REGISTER) имеет то же самое имя, что и класс, и он не возвращает значения (даже типа void). Наш конструктор может быть описан в следующем виде:

```
REGISTER::REGISTER(unsigned SIZE, BOOLEAN *INITIAL_STATE)
{   size = SIZE;
    state = new BOOLEAN [SIZE];
    for (int i = 0; i < SIZE; state[i++] = (INITIAL_STATE ?
        INITIAL_STATE[i] : 0);
```

Здесь BOOLEAN — это определенный пользователем тип, который представляет одномерный булев (двоичный) массив (каждый бит этого массива соответствует биту регистра), new — это оператор языка Си++, обеспечивающий динамическое выделение памяти.

Давайте усложним модель нашего регистра. В большинстве практических применений его следует устанавливать первоначально в нулевое состояние. В результате

можно ввести в конструктор параметр, заданный по умолчанию. Тогда объявление конструктора в классе REGISTER представится в следующем виде:

```
REGISTER (unsigned SIZE, BOOLEAN *INITIAL_STATE = NULL);
```

Теперь можно определить объект двумя возможными путями (предположим, что SIZE = 10):

```
REGISTER register1 (10);  
REGISTER register2 (10, INITIAL_STATE);
```

В первой строке второй параметр конструктора равен NULL по умолчанию. Во второй строке второй параметр — это указатель на заданное начальное состояние регистра.

При создании объекта для него выделяется фрагмент оперативной памяти компьютера. Чтобы освободить эту память, необходимо уничтожить объект. Выполнение подобных функций возложено на деструктор, который имеет то же самое имя, что и класс со знаком тильда (~) перед ним. Рассмотрим возможное описание деструктора для нашего примера.

```
class REGISTER  
{ . . . . .  
public:  
    REGISTER (...);  
    REGISTER (void);  
    { delete [size] state;}  
    . . . . .  
};
```

Здесь delete — это оператор Си++, обеспечивающий динамическое освобождение ранее выделенной памяти. Наш (и любой другой) деструктор не имеет аргументов.

Выше уже говорилось, что объект включает данные и функции. Внутренние данные можно рассматривать как собственность конкретного объекта. Поэтому внешний доступ к этим данным может быть в значительной степени ограничен. Для этих целей в языке Си++ введены атрибуты private, public и protected. Примеры использования этих атрибутов приведены в основных разделах книги.

Продолжим развитие нашей модели. Предположим, что мы хотим рассмотреть сдвигающий регистр, осуществляющий сдвиг хранимого двоичного слова влево или вправо. По сути сдвигающий регистр — это разновидность регистров. Рассмотрим предыдущее определение класса REGISTER. Функции WRITE и READ нужны также и для сдвигающего регистра. Дополнительно сдвигающий регистр требует новую функцию, выполняющую операцию сдвига. Рассмотрим следующую иерархию классов: базовый класс с именем REGISTER — производный класс с именем SHIFT_REGISTER. Класс SHIFT_REGISTER будет наследовать функции READ и WRITE

базового класса REGISTER. Дополнительно новый класс может включать некоторые новые компоненты, например функцию с именем SHIFT, выполняющую операцию сдвига. Рассмотрим возможное объявление класса SHIFT_REGISTER:

```
class SHIFT_REGISTER: public REGISTER
{
public:
    SHIFT_REGISTER(unsigned SIZE, BOOLEAN *INITIAL_STATE):
        REGISTER(SIZE, INITIAL_STATE) {}
    void SHIFT(unsigned number);
};
```

Когда объявляется производный класс, можно использовать спецификаторы доступа (private, public или protected) перед именами базовых классов. В результате можно менять атрибуты наследуемых компонентов. Если базовый класс имеет конструктор хотя бы с одним аргументом, который не может быть задан по умолчанию, то производный класс тоже должен иметь конструктор.

Предположим, что функция READ является одинаковой для базового и производного классов, а функция WRITE производного класса имеет небольшие отличия. Язык Си++ позволяет выполнить переопределение (перегрузку) этой функции. Мы хотим использовать базовую версию функции WRITE для класса REGISTER и производную версию функции WRITE для класса SHIFT_REGISTER. Рассмотрим следующие объявления:

```
REGISTER reg(...), *pointer1_to_register, *pointer2_to_register;
SHIFT_REGISTER shift_register(...);
```

Далее можно записать выражения:

```
pointer1_to_register = &reg;
pointer2_to_register = &shift_register;
```

Теперь использование указателя pointer2_to_register позволяет обращаться ко всем компонентам объекта shift_register, наследуемым из класса REGISTER. Рассмотрим следующее выражение:

```
pointer2_to_register -> WRITE(...);
```

Какая версия функции WRITE будет вызвана? Если pointer2_to_register определен как указатель на объект базового класса, то вызывается функция WRITE базового класса. Чтобы вызвать функцию WRITE производного класса, необходимо объявить ее как виртуальную функцию.

Предположим, что мы хотим ввести два новых класса для определения регистров сдвига влево и вправо. Наследуем эти классы из класса SHIFT_REGISTER и назовем их

соответственно: LEFT_SHIFT_REGISTER и RIGHT_SHIFT_REGISTER. Рассмотрим функцию SHIFT, которая будет общей для обоих классов. Класс SHIFT_REGISTER имеет смысл лишь как промежуточный базовый класс для двух новых классов, поскольку не существует регистра, выполняющего просто сдвиг (сдвиг хранимого двоичного слова может осуществляться либо влево, либо вправо). В результате можно объявить этот класс как абстрактный класс в следующем виде:

```
class SHIFT_REGISTER:public REGISTER
{
    . . . . .
public:
    virtual void SHIFT(unsigned number) = 0;
    . . . . .
};
class LEFT_SHIFT_REGISTER:public SHIFT_REGISTER
{
    . . . . .
public:
    void SHIFT(unsigned number)
    { // выполнение сдвига влево на number бит
    }
    . . . . .
};
class RIGHT_SHIFT_REGISTER:public SHIFT_REGISTER
{
    . . . . .
public:
    void SHIFT(unsigned number)
    { // выполнение сдвига вправо на number бит
    }
    . . . . .
};
```

Рассмотрим следующие выражения:

```
(состояние регистра) <<= number;
(состояние регистра) >>= number;
```

Поскольку состояние регистра определено как булев массив, мы не можем использовать стандартные операторы >>= и <<= языка Си++. Однако Си++ позволяет переопределить действия большинства стандартных операторов. Ниже приводится пример переопределения оператора <<=.

```
class LEFT_SHIFT_REGISTER:public SHIFT_REGISTER
{
    . . . . .
public:
    . . . . .
    BOOLEAN* operator <<= (unsigned number)
    {
        for(unsigned i = 0; i < size-number; i++)
            state[i] = state[i + number];
        for(i = size-number; i < size; i++)
            state[i] = 0;
        return state;
    }
    . . . . .
};
```

Заметим, что атрибут компонента `state` базового класса должен быть изменен на `protected`. Теперь можно определить функцию `SHIFT` в следующем виде:

```
void SHIFT(unsigned number)
{ (this <<= number; }
```

Далее можно использовать выражения:

```
имя_объекта.SHIFT(number);
указатель_на_объекта->SHIFT(number);
```

Предположим, что мы хотим использовать другие выражения, например:

```
имя_объекта <<= number;
```

В этом случае необходимо изменить оператор `<<=` функцией следующим образом:

```
LEFT_SHIFT_REGISTER& operator<<=(unsigned number)
{ // см. рассмотренные выше инструкции
  return *this;
}
```

Наконец, для использования выражения:

```
имя_объекта << number;
```

необходимо переопределить оператор `<<` в следующем виде:

```
void operator << (unsigned number)
{ // см. рассмотренные выше инструкции, кроме «return...»
}
```

Рассмотрим другую возможную задачу. Предположим, что необходимо сравнивать состояния двух объектов, один из которых является регистром сдвига влево, а другой — регистром сдвига вправо. Для этих целей введем функцию с именем `comp_two_reg`. Компонент `state` имеет атрибут `protected` и является поэтому скрытым компонентом, доступ к которому за пределами класса невозможен. Для того чтобы наша функция `comp_two_reg` могла иметь доступ к скрытым компонентам двух объектов разных классов, она должна быть объявлена со спецификатором `friend`. Необходимо помнить, что такие функции не являются компонентами классов и поэтому они не имеют указателя `this`.

Продолжим усложнение нашей модели. Предположим, что необходимо получать информацию о некотором фиксированном состоянии с именем `state_fixed`, которое определено независимо и имеет силу для любого объекта класса. Для определения таких состояний можно использовать статическую область памяти. Рассмотрим следующее объявление статической переменной `state_fixed` (например, в пределах класса `LEFT_SHIFT_REGISTER`):

```
static BOOLEAN *state_fixed;
```

Статические компоненты имеют иные свойства по сравнению с нестатическими компонентами. В случае нестатических компонентов их копия создается в памяти компьютера для каждого объекта класса. В случае статических компонентов в памяти компьютера строится только одна копия для всех объектов класса. Таким образом эти компоненты можно использовать для хранения данных, являющихся общими для всех объектов одного класса. Если некоторый компонент является статической функцией, то существует только одна копия этой функции. Необходимо помнить, что статические функции не имеют указателя `this`. Для нашего примера статические функции тоже можно использовать для хранения фиксированных состояний.

Заключительный этап нашего примера будет посвящен построению регистровой памяти. Предположим, что необходимо построить следующие массивы регистров:

массив регистров сдвига влево;
массив регистров сдвига вправо.

Такая регистровая память может рассматриваться как логическая модель физической схемы, размещаемой на кристалле большой интегральной схемы. Наша задача посвящена проектированию семейства связанных классов. Первый класс позволяет представить массив регистров сдвига влево, а второй — массив регистров сдвига вправо. Конструирование параметризованного класса, который позволяет описать семейство классов, обеспечивается использованием программных шаблонов с помощью оператора `template` языка Си++. Этот оператор можно так же применять для задания параметризованной функции, позволяющей описать семейство связанных функций. Базовым аргументом параметризованного класса является тип конкретного класса (из семейства описанных классов), который передается как параметр. Хорошим примером использования оператора `template` являются контейнеры классов (см. Borland C++, версия 4.0), такие, как стеки и массивы.

Предположим, что `I_or_r` это тип класса, который может быть либо `LEFT_SHIFT_REGISTER`, либо `RIGHT_SHIFT_REGISTER`. Объявление:

```
template <class I_or_r>
```

говорит, что `I_or_r` это имя типа. С его помощью можно задать один из двух рассматриваемых классов. Ниже приводится пример полного описания нашего параметризованного класса.

```
template<class I_or_r> class ARRAY:
    public LEFT_SHIFT_REGISTER,
    public RIGHT_SHIFT_REGISTER
{ I_or_r **data;
```



```

    unsigned array_size;
public:
    ARRAY(unsigned ARRAY_SIZE, unsigned SIZE, BOOLEAN
          *INITIAL_STATE = NULL);
    ~ARRAY(void);
    l_or_r& operator [ ] (unsigned x)
    { return *data[x]; }
};
template < class l_or_r ARRAY < l_or_r >::
    ARRAY (unsigned ARRAY_SIZE, unsigned SIZE,
           BOOLEAN *INITIAL_STATE):
    LEFT_SHIFT_REGISTER(SIZE, INITIAL_STATE),
    RIGHT_SHIFT_REGISTER(SIZE, INITIAL_STATE)
{
    data = new l_or_r* [ARRAY_SIZE];
    for (int i = 0; < ARRAY_SIZE; i++)
        data[i] = new l_or_r(SIZE);
    array_size = ARRAY_SIZE;
}
template < class l_or_r > ARRAY < l_or_r >:: ~ARRAY()
{
    for(int i = 0; < array_size; i++)
        delete data[i];
    delete [ ] data;
}

```

Нашу модель можно развивать и дальше. Например, можно рассмотреть другие виды устройств вычислительной техники, такие, как двоичные счетчики, декодеры и т. п. В результате будут построены новые иерархические структуры. Далее эти устройства можно определить как элементы для построения более сложных цифровых схем, таких, как микропроцессоры, микроконтроллеры и микрокомпьютеры. На любом возможном уровне объектная модель является применимой в естественной и привычной форме. Мы начали использовать эту модель с очень простого устройства, которое постепенно усложнялось на каждом этапе. Построенные программные модели цифровых устройств являются естественными и удобными для решения различных практических задач, таких, например, как логическое моделирование и синтез дискретных схем [36, 37]. Объектная модель тесно связана с конечно-автоматной моделью [36, 37]. Об этом также говорится в книге [33]. Давайте вернемся к нашему примеру. С одной стороны, регистр — это объект. С другой стороны, его можно рассматривать как дискретное устройство с позиций конечно-автоматных моделей. Цифровые схемы, содержащие такие устройства, как регистры, двоичные счетчики и т. п., можно описать множеством классов, используя базовые принципы ООП. Однако они могут быть также рассмотрены, как сеть конечных автоматов. Более того, один из подходов, который был применен для проектирования параметризованных матричных цифровых схем [36], фактически использует идею, реализованную в рамках ООП через оператор `template`.

П.5. Модели памяти и оверлейные программы

П.5.1. Модели памяти для программ на языке Си ++ .

Система программирования Borland C++ позволяет использовать шесть моделей памяти: tiny (минимальная); small (малая); medium (средняя); compact (компактная); large (большая); huge (максимальная). Дадим их краткую характеристику.

Модель tiny (рис. П.5.1) предполагает, что все сегментные регистры (CS, DS, SS, ES) содержат один и тот же адрес. В результате код программы, данные и стек располагаются в одном сегменте (их суммарный размер ограничен значением 64 Кбайт). Здесь всегда используются указатели типа near. Программы, использующие модель tiny, можно преобразовать к формату COM-файла, путем задания опции /t при компоновке.

Модель small (рис. П.5.2) использует два сегмента кода и данных. Таким образом, размер кода может быть до 64 Кбайт;

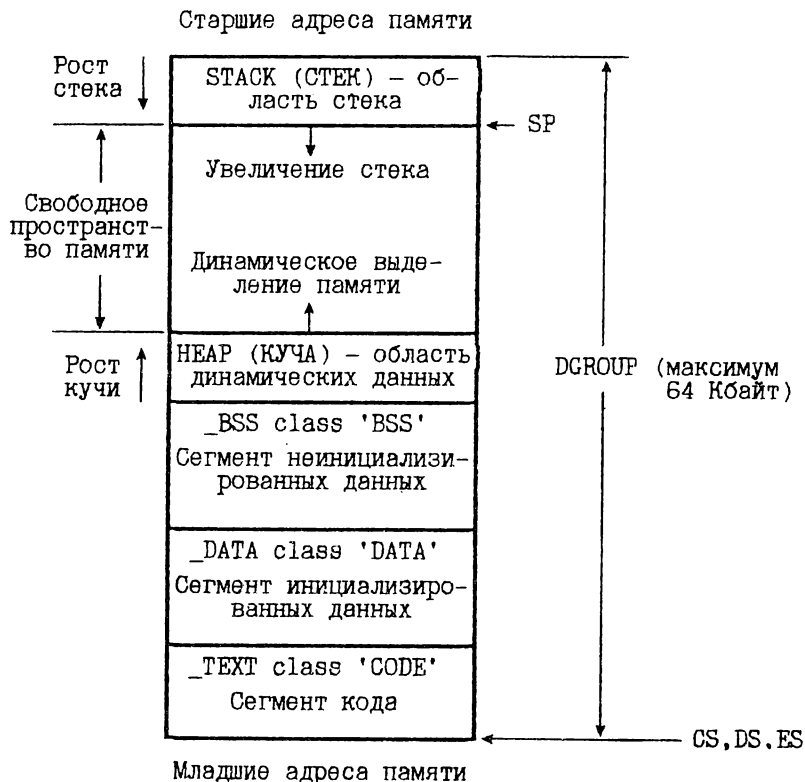


Рис. П.5.1. Организация сегментов программы для модели tiny

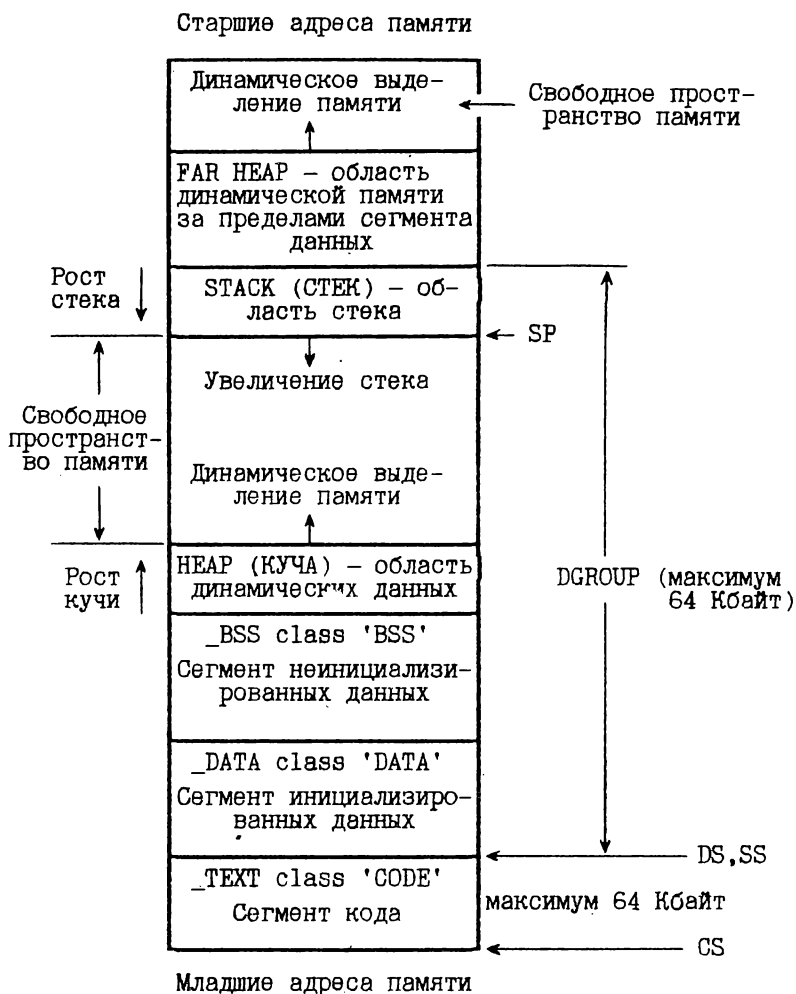


Рис. П.5.2. Организация сегментов программы для модели small

размер данных и стека тоже до 64 Кбайт. Здесь всегда используются указатели типа near.

Модель medium (рис. П.5.3) использует дальние указатели (типа far) для кода и ближние указатели (типа near) для данных. В результате данные и стек могут занимать до 64 Кбайт памяти, а код — до 1 Мбайт.

Модель compact (рис. П.5.4) является противоположной по отношению к модели medium. Она использует дальние указатели (типа far) для данных и ближние указатели (типа near) для кода.

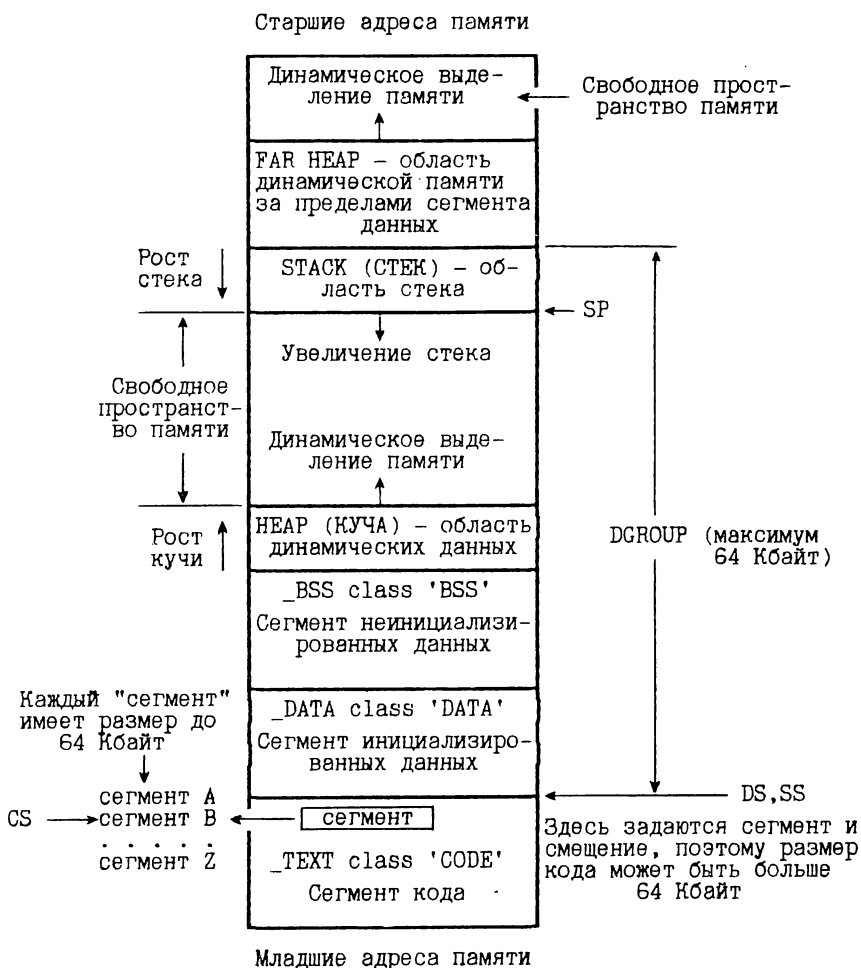


Рис. П.5.3. Организация сегментов программы для модели medium

В результате код может иметь размер до 64 Кбайт, а данные — до 1 Мбайт.

Модель large (рис. П.5.5) использует дальние указатели и для кода и для данных. Таким образом, и код и данные могут иметь размер до 1 Мбайт. Однако все статические (не динамические) данные могут иметь размер до 64 Кбайт.

Модель huge (рис. П.5.6) подобна модели large. Ее отличием является то, что объем статических (не динамических) данных может превышать 64 Кбайт.

Для того чтобы выбрать ту или иную модель памяти, можно сделать соответствующие установки в среде инструментальной

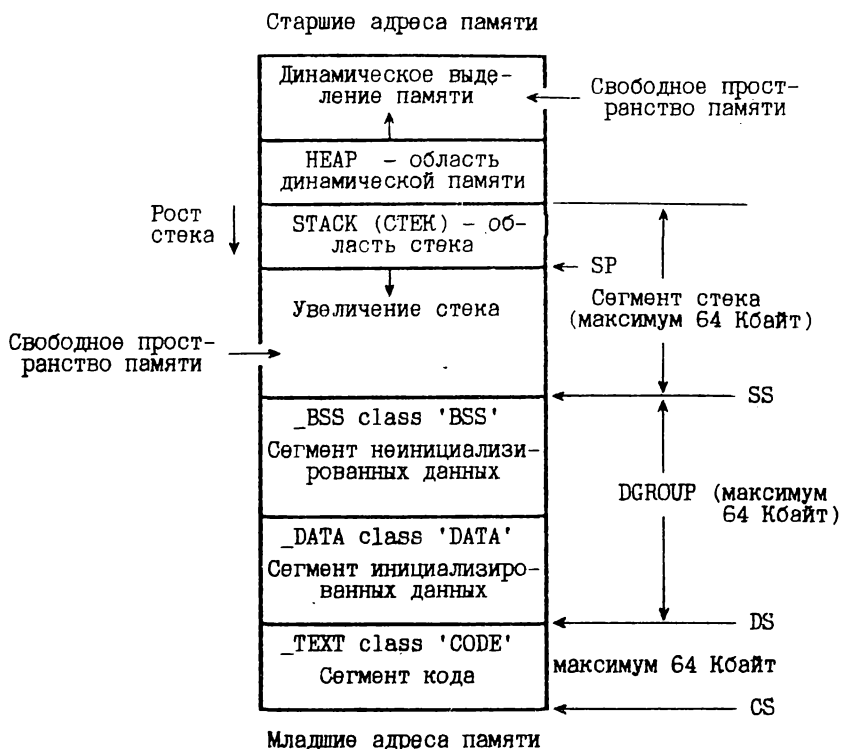


Рис. П.5.4. Организация сегментов программы для модели *compact*

системы программирования Borland C++ (выбрать последовательно меню Options — Compile — Code generation и затем установить нужную модель). При использовании компилятора BCC.EXE необходимо задать соответствующие опции в командной строке.

Предположим, что создан исходный модуль программы, который помещен в некоторый файл. Он может включать и вызовы некоторых процедур из этого же файла. Если скомпилировать и скомпоновать этот модуль, то его размер для любой модели не может превышать 64 Кбайт (даже для моделей *medium*, *large* или *huge*). Если предполагают создать программу размером больше 64 Кбайт, то выполняются следующие действия:

исходный текст большой программы размещается в нескольких исходных файлах;

каждый полученный исходный файл компилируется по отдельности (размер полученного объектного кода не должен превышать 64 Кбайт);

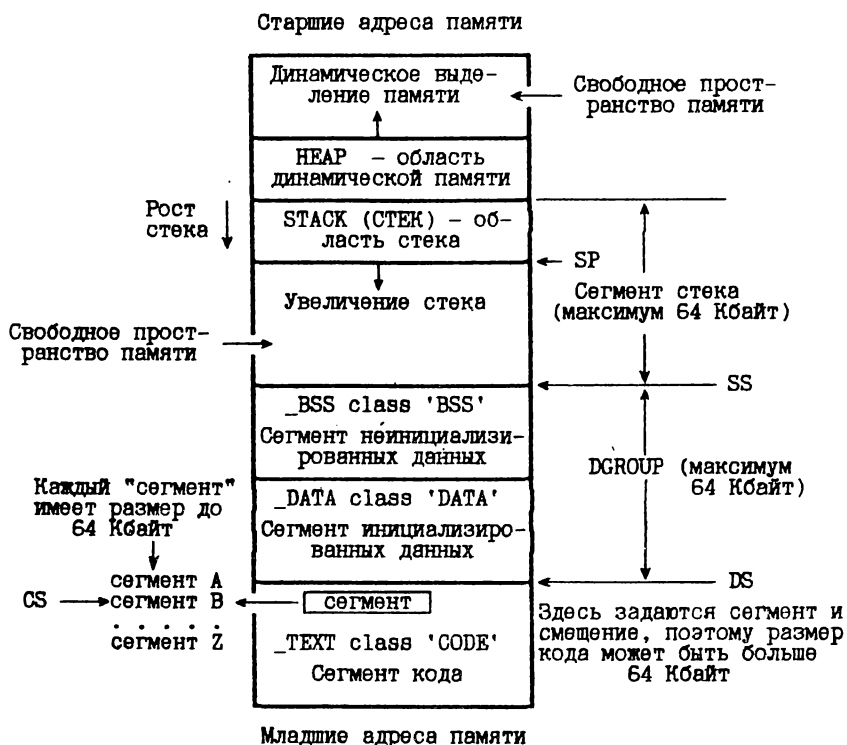


Рис. П.5.5. Организация сегментов программы для модели large

полученные объектные файлы компонуются вместе, в результате чего строится один загрузочный файл (типа EXE), размер которого будет превышать 64 Кбайт.

Несмотря на то что для модели huge объем статических данных может превышать 64 Кбайт, для одного модуля он должен быть не больше 64 Кбайт.

П.5.2. Программы, использующие возможности разных моделей памяти

Язык Borland C++ содержит восемь ключевых слов, которые позволяют модифицировать значения указателей: near, far, huge, _cs, _ds, _ss, _es, _seg.

Модификатор near определяет 16-битовый ближний указатель (смещение относительно некоторого сегментного регистра). Если он задает указатель на функцию, то соответствующее 16-битовое смещение суммируется со значением регистра CS. Ближний указатель на данные задает смещение относительно регистра DS.

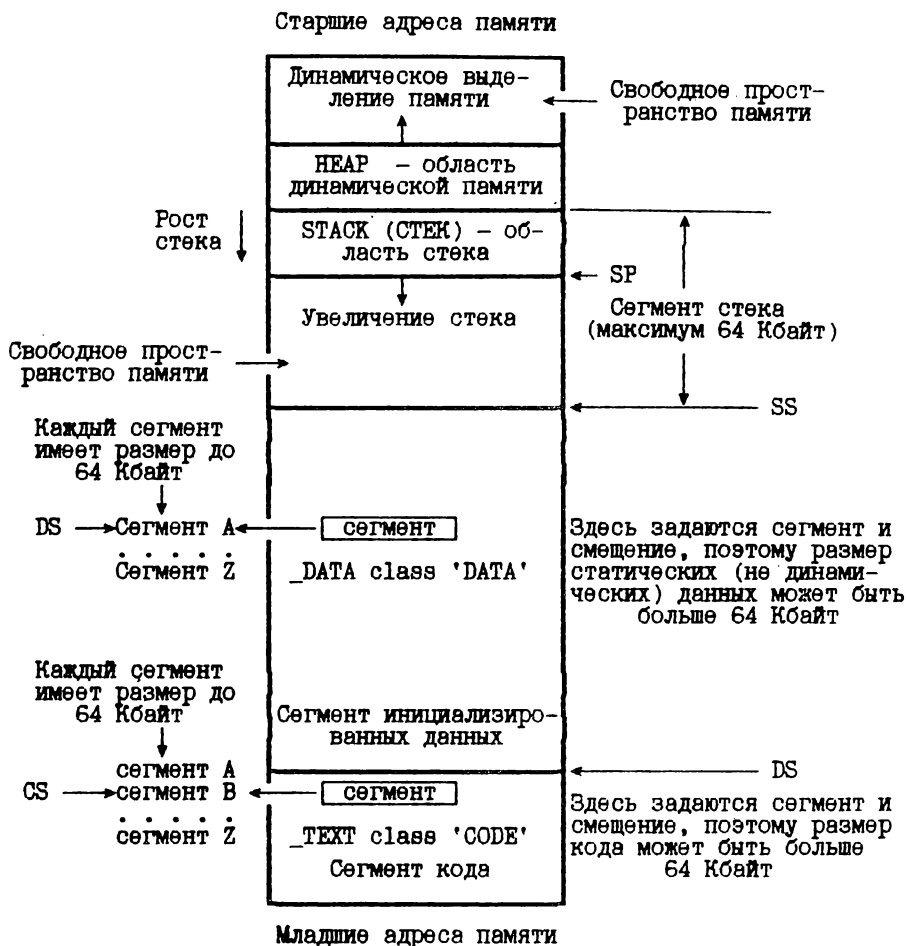


Рис. П.5.6. Организация сегментов программы для модели huge

Модификатор `far` определяет 32-битовый дальний указатель, включающий 16-битовое значение сегмента и 16-битовое смещение. Одному и тому же физическому адресу могут соответствовать много пар «сегмент: смещение», например: 0000:1230; 0100:0230; 0120:0030; 0123:0000. Предположим, что приведенным парам соответствуют дальние указатели `a`, `b`, `c`, `d`. Тогда любое выражение вида

`if (== c)...`

будет задавать ложное значение (хотя переменные `a` и `c` имеют значения одного и того же физического адреса). Другие операторы

ры отношения: $>$, $>=$, $<$, $<=$, $!=$ — тоже будут давать неверные результаты. Здесь следует учесть, что операторы $==$ и $!=$ используют соответствующее 32-битовое значение как объект типа `unsigned long` (не как физический адрес памяти). Операторы $>$, $>=$, $<$, $<=$ используют только 16-битовое смещение, поэтому для приведенных выше пар будут истинны все выражения: $(a > b)$, $(b > c)$, $(c > d)$.

Когда к указателю типа `far` добавляется некоторое значение, то изменяется только смещение. Например, если добавить 1 к значению `013D:FFFF`, то результат будет `013D:0000` (не `113D:0000`). Если вычесть 1 из значения `013D:0000`, то результат будет `013D:FFFF` (не `013C:000F`).

Модификатор `huge` определяет 32-битовый указатель. В отличие от модификатора типа `far` указатель поддерживается в нормализованном виде. После нормализации 16-битовый сегментный адрес будет иметь максимально возможное значение. В результате допустимые значения смещения лежат в диапазоне от 0 до 15. После нормализации указатель фактически преобразуется к 20-битовому физическому адресу (16 старших бит берется из значения сегмента, а младшие 4 бита задают соответствующую шестнадцатеричную цифру от 0 до 15 из смещения). Например, значение `0000:1234` после нормализации примет вид `0123:0004`. Рассмотрим примеры:

```
1234:ABCD — 1CF0:000D;  
3000:CD2F — 3CD2:000F;  
2222:1111 — 2333:0001.
```

Нормализация позволяет получить следующие преимущества: для любого физического адреса будет существовать только одна пара «сегмент:смещение»;

операторы $>$, $>=$, $<$, $<=$ используют полное 32-битовое значение указателя. В результате все операции отношения работают правильно;

при увеличении или уменьшении значения указателя могут изменяться как смещение, так и сегмент (в отличие от указателей типа `far`). Например, если добавить 1 к значению `1234:000F`, то результат будет `1235:0000`. Если вычесть 1 из значения `1235:0000`, то результат будет `1234:000F`.

Следует учитывать, что работа с указателями типа `huge` выполняется медленнее, чем с указателями типов `far` и `near`, за счет дополнительных затрат времени на нормализацию.

Ключевые слова `_cs`, `_ds`, `_ss` и `_es` позволяют задать четыре специальных указателя типа `near`. Эти указатели определенным образом связаны с соответствующим сегментным регистром (отдаленно это напоминает префикс замены сегмента при прог-

раммировании на языке Ассемблера). Любое из приведенных слов задает сегментный регистр, относительно которого отсчитывается смещение. Например, следующее объявление:

```
unsigned int _ss *stack_of;
```

говорит о том, что указатель `stack_of` — это смещение относительно регистра SS. Этот указатель адресует беззнаковые целые числа (unsigned int или просто unsigned). Предположим, что в переменную `stack_of` занесено некоторое значение X, тогда команда `*stack_of = 125;` запишет число 125 в сегмент стека по адресу SS:X. Ниже приводятся четыре примера программ, демонстрирующих использование ключевых слов `_cs`, `_ds` и `_ss` (ключевое слово `_es` используется по аналогии и задает экстра-кодированный сегмент ES).

```
/* пример EX5_1 */
#include <stdio.h>
#pragma inline
void main(void)
{
    unsigned int _ss *stack_of;
    unsigned int i;
    asm push 1234h;
    stack_of = (unsigned int _ss *)_SP;
    i = *stack_of;
    printf("i = %x\n", i);
    asm push 5678h;
    stack_of = (unsigned int _ss *)_SP;
    i = *stack_of;
    printf("i = %x\n", i);
    asm pop ax;
    asm pop ax;
}
```

В этой программе сначала в стек записывается значение 123h. Затем переменная `stack_of` получает значение SP. После команды `i = *stack_of;` из сегмента стека будет читаться значение по адресу SP (и это будет значение 1234h). Далее по аналогии из стека читается значение 5678h. Результаты работы программы будут следующими:

```
i = 1234
i = 5678
```

Последовательность команд языка Си++ `stack_of = (unsigned int _ss *)_SP; i = *stack_of;` преобразуется в следующие ассемблерные команды:

```
mov SI, SP
mov AX, SS:[SI]
mov [BP-2], AX
```

Здесь предполагается, что значение указателя `stack_of` помещается в регистр `SI`, локальная переменная `i` хранится в стеке по адресу `BP-2`, регистр `AX` используется для передачи значения.

Выше говорилось о том, что компилятор языка Си++ помещает локальные переменные в стек. Вторая программа показывает изменение локальной переменной `i` со значения 10 на значение 20. Такое изменение осуществляется косвенно через переменную `p`.

```
/* пример EX5_2 */
#include <stdio.h>
#pragma inline
void main(void)
{   unsigned int _ss *p;
    unsigned int i = 10;
    printf("i = %u\n",i);
    asm {   lea ax,i      // смещение i в регистр ax
            mov p,ax     } // смещение i в переменную p
// ниже значение 20 записывается по адресу SS:p
    *p = 20;             // теперь i = 20
    printf("i = %u\n",i);
}
```

Результаты работы этой программы будут следующими:

```
i = 10
i = 20
```

Третья программа показывает косвенное изменение значений в сегменте данных.

```
/* пример EX5_3 */
#include <stdio.h>
#pragma inline
unsigned char a = 100;
void main(void)
{   unsigned char _ds *p;
    printf("a = %u\n",a);
    asm {   lea ax,a      // смещение a в регистр ax
            mov p,ax     } // смещение a в переменную p
// ниже значение 200 записывается по адресу DS:p
    *p = 200;            // теперь a = 20
    printf("a = %u\n",a);
}
```

Следует обратить внимание на то, что для модели памяти `small` изменение ключевого слова `_ds` на `_ss` (и наоборот) не приведет к изменению результатов, поскольку данные и стек находятся в одном сегменте. Если же использовать большие модели памяти для данных (например, `large`), то указанные изменения будут приводить к ошибке (если для модели `large` в последней программе вместо `_ds` записать `_ss`, то значение `a` не изменится).

Последняя программа демонстрирует использование ключевого слова `_cs`. В результате она изменяет свой код.

```
/* пример EX5_4 */
#include <stdio.h>
#pragma inline
void main(void)
{   unsigned _cs *p;
    asm { lea bx,sstr
          mov p,bx   } // теперь p содержит адрес sstr
// замена команд в сегменте CS, начиная со смещения p
*p = 0x07B2;         // B207 - код команды mov dl,7
*(p+1) = 0x02B4;     // B402 - код команды mov ah,2
*(p+2) = 0x21CD;     // CD21 - код команды int 21h
puts("ПРИВЕТ ИЗ МИНСКА");
asm sstr: // смещение в сегменте CS, начиная с кото-
          // рого производится замена команд
// команда por - однобайтная с кодом 90h. Шесть следующих
// однобайтных команд заменяются тремя двухбайтными коман-
// дами:   mov dl,7       mov ah,2       int 21h
asm {   por    // B207 или
        por    // mov dl,7
        por    // B402 или
        por    // mov ah,2
        por    // CD21 или
        por    }          // int 21h
}
```

В конце программы записана последовательность из шести ассемблерных команд `por` (отсутствие операции). Каждая из таких команд в сегменте `CS` кодируется однобайтным кодом. В программе шесть байт такого кода, которые начинаются с ассемблерной метки `sstr`, заменяются тремя двухбайтными ассемблерными командами:

```
mov dl,7
mov ah,2
int 21h
```

В результате вызывается функция 2 прерывания 21h, которая обеспечивает вывод символа из регистра `DL` на экран. В `DL` записывается значение 7. Это управляющий ASCII-код для генерации звукового сигнала. Таким образом, в результате работы программы на экран выводится сообщение ПРИВЕТ ИЗ МИНСКА и генерируется звуковой сигнал. Все изменения в сегменте `CS` осуществляются косвенно через переменную `p`. Эта переменная объявлена как беззнаковая целая, поэтому выражение `p + 1` изменяет указатель на 2 (осуществляется автоматическое масштабирование в соответствии с размером объекта типа `unsigned`).

Ключевое слово `_seg` можно использовать для объявления указателей на сегмент. Общую форму его использования можно записать в следующем виде:

```
тип_данных _seg *идентификатор;
```

Сначала идентификатор будет определять сегмент со смещением 0. С ним нельзя выполнять операции вида ++, --, +=, -=, нельзя также вычитать один указатель из другого. При добавлении ближнего указателя к указателю на сегмент выполняется операция типа far (т. е. сегмент не изменяется, но появляется смещение). Оба таких указателя должны быть одинакового типа или один из них должен иметь тип void. Аналогичные операции выполняются при добавлении к указателю на сегмент целого числа. Указатели на сегмент можно инициализировать, их можно сравнивать, передавать в виде параметров в функции и т. п. Рассмотрим пример программы, использующей ключевое слово `_seg`.

```
/* пример EX5_5 */
void main(void)
{
    unsigned _seg *start = (unsigned _seg *)0xB800;
    // в цикле for к указателю на сегмент start прибавляется
    // беззнаковое целое число
    for(unsigned i = 0; i < 160; i+=2)
        *(start + i) = 0x7030;
    // в цикле for к указателю на сегмент start прибавляется
    // ближний указатель на беззнаковое целое число
    for(unsigned *j=(unsigned*)160; j<(unsigned*)320; j++)
        *(start + j) = 0x7031;
}
```

В результате ее выполнения в верхней части экрана появляются две строки: первая из нулей и вторая из единиц. Значение 0x7030, которое записывается в экранную память с сегментным адресом B800, определяет символ с ASCII-кодом 0x30 (это ноль) и его атрибут 0x70 (реверсивное изображение для монохромного экрана). Значение 0x7031 определяет символ с ASCII-кодом 0x31 (это единица) и с атрибутом 0x70.

При разработке программ, которые могут компилироваться в разных моделях памяти, необходимо правильно объявлять функции и указатели. Например, пусть указатель передается в качестве параметра через стек. Тогда если его тип `near` — передается только смещение, а если `far` — сегмент и смещение. Многие возникающие проблемы снимаются при использовании прототипов функций. По умолчанию тип данных и функций в программе определяется установленной моделью памяти для данных и функций.

Если некоторая функция объявлена как `near`, например:

```
long near my_fun(int a),
```

то в случае выбора больших моделей для кода (`medium`, `large`, `huge`) эту функцию можно будет вызывать только в том модуле, где она определена.

В целом можно создавать объектные модули с использованием разных моделей памяти и компоновать их совместно с целью построения единой выполняемой программы. Важно правильно определять тип каждой функции и передаваемые параметры. Многие проблемы здесь также снимаются при использовании прототипов функций.

П.5.3. Общие сведения об оверлейных программах

Оверлеями называются программные модули, которые могут занимать одно и то же место в оперативной памяти, перекрывая друг друга. При некотором заданном объеме оперативной памяти использование оверлеев позволяет увеличить размер выполняемой программы.

Borland C++ имеет специальное средство, управляющее оверлеями, которое называется VROOMM (Virtual Run—time Object—Oriented Memory Manager). Предположим, что необходимо разработать некоторую сложную программу, включающую оверлеи. Выделим в ней две части. Первая часть включает головную или базовую программу (функцию). Вторая часть включает некоторое подмножество оверлейных подпрограмм. В общем случае в любой момент времени из этого подмножества может выполняться только одна подпрограмма, поскольку различные подпрограммы перекрывают друг друга. В результате в некотором оверлейном модуле можно вызывать подпрограммы только из этого же модуля или из базовой программы. Подпрограммы из других оверлейных модулей вызывать нельзя. Общий размер памяти, необходимый для выполнения программы с оверлеями, равен размеру базы плюс размер наибольшего оверлея. Borland C++ позволяет выполнять с оверлейными функциями практически те же действия, что и с обычными (не оверлейными) функциями. Им можно передавать аргументы, указатели на другие функции и т. п. Отладчики фирмы Borland (Turbo Debugger и встроенный в Borland C++ отладчик) позволяют отлаживать оверлейные программы.

Borland C++ обеспечивает динамическую подкачку сегментов (основным узлом, который может быть перемещен из внешней в оперативную память и наоборот, является сегмент). Каждый сегмент может включать одну или более программ (функций). Память для программы делится на две части для базы и для оверлеев. Когда вызывается функция, которой нет в базе и текущем оверлейном модуле, сегмент, содержащий эту функцию, перемещается в память для оверлеев, возможно замещая другой сегмент. Память для оверлеев (буфер) размещается между сегментом стека и началом дальней динамически выделяемой области (Far

Heap). Размер буфера задается по умолчанию в системе Borland C++ (обычно он равен двойному размеру наибольшего оверлея). Его можно изменить через глобальную переменную `_ovrbuffer`, например:

```
unsigned _ovrbuffer = 0x1500;
```

Здесь задается требуемый размер буфера в параграфах (в примере 1500₁₆ параграфов). Динамическая загрузка сегментов (управление оверлеями) осуществляется программой-обработчиком прерывания Borland C++, которая обычно вызывается через вектор 3F₁₆.

Рассмотрим теперь правила, которые необходимо выполнять при компиляции и компоновке оверлейных программ. В Borland C++ оверлейные программы могут использовать только большие модели памяти для кода (`medium`, `large` или `huge`). Если используется автономный компилятор (файл `BCC.EXE`), то необходимо задать опцию `-Y` для головной (базовой) программы и опцию `-Yo` для оверлеев. Одна опция `-Yo` распространяется на все последующие модули. Для того чтобы отменить ее, используется опция `-Yo-`. При работе в интегрированной среде (файл `BC.EXE`) необходимо выполнить следующие действия:

выбрать такую последовательность команд главного меню: Option — Compiler — Code generation — Overlay support. Затем включить выбранный режим (Overlay support). Эти действия соответствуют опции `-Y` для компилятора `BCC.EXE`;

выбрать такую последовательность команд главного меню: Option — Linker — Overlay EXE. Затем включить выбранный режим (Overlay EXE);

включить файлы с модулями, составляющими оверлейную программу, в проект. Для этого сначала создать проект (выбрать команды: Project — Open project — задать имя проекта и нажать клавишу «ВВОД»). Затем включить файлы с нужными модулями в проект. Режим включения файлов можно задать путем нажатия клавиши «Ins». В появившееся окно последовательно записываются имена файлов и после каждого имени нажимается клавиша «ВВОД». Выйти из режима включения файлов можно после нажатия клавиши «ESC»;

если некоторый файл `F` включает оверлей, то его необходимо пометить (выполнить действия, аналогичные опции `-Yo` для компилятора `BCC.EXE`). Для этого в окне проекта полоска-курсор перемещается на файл `F` и далее выбирается последовательность команд: Project — Local options — Overlay this module и включается соответствующий режим (Overlay this module). Точно так же для файла можно выбрать компилятор (Си++ или Ассемблер).

П.5.4. Пример оверлейной программы

Предположим, что необходимо разработать оверлейную программу, которая размещается в четырех файлах EX5_6.CPP, EX5_7.CPP, EX5_8.CPP, EX5_9.ASM, причем первые три файла содержат программы на языке Си++, а последний на языке Ассемблера. Головная программа находится в файле EX5_6.CPP; все остальные файлы содержат оверлеи. Заметим, что оверлей на языке Ассемблера не должен создавать переменных в сегменте кода, поскольку любое их изменение может быть потеряно при перекрытии сегментов. Необходимо также осторожно использовать указатели на объекты, расположенные в оверлейных модулях. Управляющая программа VROOMM произвольно назначает адреса загрузки перекрывающимся сегментам. Поэтому от одной к другой загрузке оверлейного модуля адреса объектов могут изменить свои значения.

Ниже приводятся тексты всех четырех модулей оверлейной программы.

```
/* пример EX5_6 */
#include <stdio.h>
#include <iostream.h>
int ovl1(char *s,int i, int j);
char *ovl2(int x,int y,char *ss,int *z);
extern "C" my_asm(int i);
void main(void)
{
    char str1[] = "Этот параметр передается в программу ovl1";
    char str2[] = "Этот параметр передается в программу ovl2";
    int a=100,b=200,c=5,d,p;
    cout << "a/4 =" << my_asm(a) << "\n";
    cout << "b/4 =" << my_asm(b) << "\n";
    d = ovl1(str1,a,b);
    puts(ovl2(d,c,str2,&p));
    cout << "результат =" << p << "\n";
    cout << "результат/4 =" << my_asm(p) << "\n";
}
```

```
/* пример EX5_7 */
#include <stdio.h>
int ovl1(char *s,int i, int j)
{
    puts(s);
    return(i * j);
}
```

```
/* пример EX5_8 */
#include <stdio.h>
char *ovl2(int x,int y,char *ss,int *z)
{
    puts(ss);
    *z = x/y;
    return("Передача управления головной программе");
}
```

```

; пример EX5_9
.MODEL LARGE
.CODE
PUBLIC C my_asm
my_asm PROC FAR
    push bp
    mov bp, sp
    mov ax, [bp+6]
    mov cl, 2
    shr ax, cl
    pop bp
    ret
my_asm ENDP
end

```

С помощью компилятора TASM.EXE получим объектный ассемблерный модуль EX5_9.OBJ. Если далее используется автономный компилятор BCC.EXE, то можно задать следующую командную строку:

C > BCC -ml -v -Y EX5_6.CPP -Y EX5_7.CPP EX5_8.CPP EX5_9.OBJ < ВВОД >

Здесь опция -ml задает модель памяти large, опция -v требует сохранения в файле символьной информации для отладчика. Все остальные опции пояснялись в § 5.3. После компиляции и компоновки будет получен единый выполняемый модуль EX5_6.EXE. Если запустить его на выполнение, то на экране дисплея появятся следующие результаты:

a/4 = 25

b/4 = 50

Этот параметр передается в программу ovl1

Этот параметр передается в программу ovl2

Передача управления головной программе

результат = 4000

результат/4 = 1000

Выполним аналогичные действия в интегрированной среде Borland C++ (файл BC.EXE). Сначала создадим следующий файл проекта:

EX5_6.CPP

EX5_7.CPP

EX5_8.CPP

EX5_9.ASM

Для первых трех файлов зададим использование компилятора Си++, а для последнего — Ассемблера (см. §5.3). Назовем файл проекта CH5.PRJ и выполним действия, описанные в §5.3. В результате построим файл CH5.EXE. Если запустить его на выполнение, то будут получены те же самые результаты. Не забудьте перед компиляцией программы на языке Си++ установить модель памяти LARGE.

Проанализировать выполнение оверлейной программы (оценить размер буфера, время выполнения и загрузки различных сегментов и т. п.) можно с помощью программы Turbo Profiler (файл TPROF.EXE) фирмы Borland.

ЛИТЕРАТУРА

1. Складов В.А. Применение ПЭВМ. Кн. 1. Организация и управление ресурсами ПЭВМ.— М.: Высшая школа, 1992.— 158 с.
2. Складов В.А. Применение ПЭВМ. Кн. 2. Операционные системы ПЭВМ.— М.: Высшая школа, 1992.— 144 с.
3. Складов В.А. Применение ПЭВМ. Кн. 3. Программное и информационное обеспечение ПЭВМ.— М.: Высшая школа, 1992.— 128 с.
4. Складов В.А. Программное и лингвистическое обеспечение ПЭВМ. Кн. 1.— Минск: Высшая школа, 1991.— 462 с.
5. Складов В.А. Программное и лингвистическое обеспечение ПЭВМ. Кн. 2.— Минск: Высшая школа, 1991.— 334 с.
6. Юлин В.А., Булатова И.Р. Приглашение к СИ.— Минск: Высшая школа, 1990.— 224 с.
7. Allen L. Wyatt, Sr. Advanced Assembly Language. USA, Que Corporation, 1992, 705 p.
8. Atkinson Lee, Atkinson Mark. Using C. Que Corporation, 1990, 945 p.
9. Berry John. The Waite Group's C++ Programming. Howard W. Sams Company. Indianapolis, Indiana, 1988, 381 p.
10. Burnap Steve. Advanced Turbo C Programming. Computer Books, Greenboro, North Carolina Radnor, Pennsylvania, 1988, 327 p.
11. David Hu. C/C++ For Expert Systems. Management Information Source, Portland, Oregon, 1989, 565 p.
12. David Hu. Object-Oriented Environment in C++. Management Information Source, Portland, Oregon, 1990, 557 p.
13. Dettmann T., Johnson W. DOS Programmer's Reference. 3rd Edition. USA, Que Corporation, 1992, 1062 p.
14. Dewhurst Stephen C., Stark Kathy T. Programming in C++. Prentice Hall software series, New Jersey, 1989, 233 p.
15. Ferraro R.F. Programmer's Guide to the EGA and VGA Cards. USA, Addison-Wesley Publishing Company, Inc, 1990, 1040 p.
16. Goodwin Mark. User Interface in C++ and Object-Oriented Programming. Management Information Source, 1989, 394 p.
17. Hekmatpour Sharam. C++ A Guide For C Programmers. Prentice Hall. New York, London, Toronto, Sydney, Tokio, Singapore, 1990, 264 p.
18. Hogan T. The Programmer's PC Sourcebook. USA, Microsoft press, 1991, 2035 p.
19. Jourdain R. Programmer's Problem Solver. USA, Brady Publishing, 1992, 596 p.
20. Kernighan Brian W., Ritchie Dennis M. The C Programming Language. Prentice Hall, Englewood Cliffs, New Jersey, 1988, 272 p.
21. Lippman Stanley B. C++ Primer. Addison-Wesley Publishing Company, 1989, 464 p.
22. Mullin Mark. Object-Oriented Program Design. Addison-Wesley Publishing Company, 1989, 303 p.
23. Schildt Herbert. C: The Complete Reference. Osborn McGraw-Hill, Berkeley, California, 1990, 823 p.
24. Schindt Herbert. Teach Yourself C. Osborn McGraw-Hill, Berkeley, California, 1990, 681 p.
25. Schindt Herbert. Using Turbo C++. Osborn McGraw-Hill, Berkeley, California, 1990, 755 p.
26. Shanley T. The IBM PS/2 From the Inside Out. USA, Addison-Wesley, 1991, 612 p.
27. Sklyarov V.A. The Revolutionary Guide to Turbo C++. Birmingham, WROX, 1992, 352 p.
28. Tischer Michael. PC System Programming. USA. Abacus, 1990, 919 p.
29. Tischer Michael. PC Intern. USA. Abacus, 1992, 1320 p.
30. Voss Greg, Chui Paul. Turbo C++ DiskTutor. Borland-Osborn McGraw-Hill. Berkeley, California, 1990, 503 p.
31. Wiener Richard S., Pinson Lewis J. The C+ Workbook. Addison-Wesley Company, 1990, 349 p.
32. McGord James W. Borland C++ 3.1. Programmer's Reference, 2nd Edition. USA, Que Corporation. 1992, 1063 p.
33. Grady Booch. Object-Oriented Analysis and Design. Second Edition, The Benjamin/Cummings Publishing Company, 1994, 589 p.
34. Bjarne Stroustrup. The C++ programming language. Second Edition, Addison-Wesley Publishing Company, 1994, 691 p.
35. Valery Sklyarov. From Procedural to Object-Oriented Programming (foundations, distinctions, applications, training, attractive tutorial). Electronica e Telecomunicacoes, Vol 1, N° 3, Janeiro, 1995 (Aveiro, Portugal), pp. 217-223.
36. Складов В.А. Синтез автоматов на матричных БИС.— Минск: Наука и техника, 1984.— 287 с.
37. Баранов С.И., Складов В.А. Цифровые устройства на программируемых БИС с матричной структурой.— М.: Радио и связь, 1986.— 272 с.
38. Namir Clement Shammas. What Every Borland C++ 4 Programmer Should Know. SAMS Publishing, 1994, 898 p.

ОГЛАВЛЕНИЕ

Введение.....	3
Глава 1. Подготовка, компиляция, компоновка, отладка и выполнение программ.....	5
1.1. Введение в интегрированную среду Developer Studio	5
1.2. Использование базовых средств интегрированной среды.....	6
1.3. Использование дополнительных возможностей интегрированной среды	10
1.4. Отладка программ	13
1.5. Введение в интегрированное окружение Borland C++.....	19
1.6. Выполнение программ на языке Си++	24
Глава 2. Общие конструкции языков Си и Си ++	35
2.1. Программы и данные.....	35
2.2. Операторы и выражения	45
2.3. Структурированные типы данных.....	55
2.4. Функции.....	65
2.5. Другие возможности языка Си	70
2.6. Примеры	76
Глава 3. Программирование на языке Си ++	86
3.1. Введение в язык Си++	86
3.2. Объекты и работа с ними	96
3.3. Наследование и защита	99
3.4. Перегрузка функций и операторов	107
3.5. Другие возможности языка Си++.....	121
Глава 4. Интерфейс Си ++ с языком Ассемблера.....	132
4.1. Вызов подпрограмм и передача параметров в языке Си++	132
4.2. Вызов ассемблерных программ из программ на языке Си++.....	137
4.3. Вызов программ на языке Си++ из программ на языке Ассемблера	145
4.4. Вызов библиотечных функций языка Си++ из программ на языке Ассемблера	146
4.5. Упрощенные конструкции для компилятора TASM	148
4.6. Встроенный ассемблер (режим inline в программах на языке Си++)	150
Глава 5. Программирование в среде Windows с использованием библиотек классов.....	153
5.1. Построение простейшей программы	153
5.2. Базовые компоненты программы.....	159
5.3. Построение прикладной программы с помощью инструментов AppWizard и ClassWizard	160
5.4. Анализ и обработка сообщений.....	164
5.5. Графика.....	173
5.6. Взаимодействие с устройствами.....	178
5.7. Ресурсы.....	183
Глава 6. Примеры программ на языках Си и Си ++	190
6.1. Работа с манипулятором «мышь»	190
6.2. Программирование последовательного интерфейса.....	194
6.3. Программирование параллельного интерфейса	208
6.4. Резидентные программы.....	210
6.5. Драйверы устройств	241
Приложения	251
П.1. Общие сведения о примененных конструкциях языка Ассемблера	251
П.2. Краткие сведения об опциях для различных программ в системе программирования Borland C++	255
П.3. Справочные сведения по библиотечным функциям языков Си и Си++, примененным в программах книги	259
П.4. Пример анализа и проектирования объектно-ориентированной программы.....	262
П.5. Модели памяти и оверлейные программы	272
Литература	287

33-

ISBN 5-06-003486-0



9 785060 034868