

**Б. И. Березин**  
**С. Б. Березин**

# **НАЧАЛЬНЫЙ КУРС**



ДИАЛОГ-МЧОН

УДК 519.682  
Б48

Березин Б. И., Березин С. Б.  
Б48 Начальный курс С и С++. - М.: ДИАЛОГ-МИФИ, 2001. - 288 с.

ISBN 5-86404-075-4

Книга является учебным пособием по языкам программирования С и С++. Она может быть использована для изучения языка С, как самостоятельного языка. Язык С++ рассматривается как надстройка к языку С. Изложение проиллюстрировано большим количеством примеров.

Книга написана на основе учебного курса "С++ для начинающих", который в течение нескольких лет читался в учебном центре "Диалог-МИФИ" и ориентирована на начинающих программистов, а также тех, кто хочет самостоятельно изучить языки программирования С и С++. Она также может быть полезна для читателей, знающих язык С и начинающих изучать С++.

*Учебно-справочное издание*  
Березин Борис Иванович  
Березин Сергей Борисович  
Начальный курс С и С++

Редактор О. А. Голубев  
Корректор В. С. Кустов  
Макет О. А. Голубева  
Обложка Н. В. Дмитриевой

Лицензия ЛР N 071568 от 25.12.97. Подписано в печать 30.11.2000.  
Формат 60х84/16. Бум. офс. Печать офс. Гарнитура Таймс.  
Усл. печ. л. 16.74. Уч.-изд. л. 9.9. Тираж 5 000 экз. Заказ 4460.

Акционерное общество "ДИАЛОГ-МИФИ"  
115409, Москва, ул. Москворечье, 31, корп. 2

Подольская типография  
142100, г. Подольск, Московская обл., ул. Кирова, 25

ISBN 5-86404-075-4

© Березин Б. И., Березин С. Б., 1997-2001

© Оригинал-макет, оформление обложки.  
АО "ДИАЛОГ-МИФИ", 2001

## ВВЕДЕНИЕ

---

Язык С, созданный Денисом Ритчи в начале 70-х годов в Bell Laboratory американской корпорации AT&T, является одним из универсальных языков программирования. Язык С считается языком системного программирования. Правильнее сказать, что он наиболее эффективен при решении задач системного программирования, хотя он, безусловно, удобен и при написании прикладных программ. Среди преимуществ языка С можно отметить переносимость программ, написанных на языке С, на компьютеры различной архитектуры и из одной операционной системы в другую, лаконичность записи алгоритмов, логическую стройность и удобочитаемость программ, возможность получить эффективный код программ, сравнимых по скорости с программами, написанными на ассемблере. Удобство языка С основано на том, что он является одновременно и языком высокого уровня, имеющим полный набор конструкций структурного программирования, поддерживающим модульность, блочную структуру программ, возможность раздельной компиляции модулей. В то же самое время язык С имеет набор низкоуровневых средств, позволяющих иметь удобный доступ к аппаратным средствам компьютера, в частности позволяющих добраться до каждого бита памяти. Гибкость и универсальность языка С обеспечивает его широкое распространение.

Первое описание языка было дано в книге Б. Кернигана и Д. Ритчи, которая была переведена на русский язык. Долгое время это описание являлось стандартом, однако ряд моментов допускали неоднозначное толкование, которое породило множество трактовок языка С. Для исправления этой ситуации при Американском национальном институте стандартов (ANSI) был образован комитет по стандартизации языка С и в 1983 году был утвержден стандарт языка С, получивший название ANSI C.

В начале 80-х годов в той же Bell Laboratory Бьерном Строуструпом (Bjarne Stroustrup) в результате дополнения и расширения языка С был создан новый по сути язык, получивший название "С с классами". В 1983 году это название было заменено на C++.

Автор языка создавал его с целью улучшить язык С, поддержать абстракции данных и объектно-ориентированное программирование. Язык C++ является языком объектно-ориентированного программирования. Концепция объектно-ориентированного программирования возникла не вдруг. Идея использования программных объектов развивалась разными исследователями в течение многих лет. Одним из представителей языков такого типа является Simula 67. Более подробно мы расскажем об особенностях объектно-ориентированного программирования ниже, а сейчас коротко остановимся на его достоинствах.

Что же такое объектно-ориентированное программирование?

В словаре по программированию и информатике мы читаем: "Объектно-ориентированный язык - язык программирования, на котором программа задается описанием поведения совокупности взаимосвязанных объектов. Объекты обмениваются запросами; реагируя на полученный запрос, объект посылает запрос другим объектам, получает ответы, изменяет значения своих внутренних переменных и выдает ответ на полученный запрос. Механизм запросов в объектно-ориентированных языках отличается от механизма процедур в процедурных языках тем, что при выполнении запроса объектом непосредственно могут быть изменены только значения переменных этого объекта".

Объектно-ориентированное программирование имеет дело с объектами, включает в себя создание объектов, которые объединяют данные и правила обработки этих данных. Объекты могут включать в себя частные, закрытые, приватные (private) данные и правила их обработки, доступные только объекту и его наследникам, а также общие (public) данные и правила, которые доступны объектам и модулям в других частях программы. Важной чертой объектно-ориентированного программирования является наследование, т. е. возможность создавать иерархическую последовательность объектов от более общих к более специфическим, частным. В этой иерархии каждый объект наследует характерные черты объектов-предшественников, объектов, предшествующих ему.

Таким образом, языки объектно-ориентированного программирования содержат в себе следующие основные черты: наличие объектов и инкапсуляцию данных, наследование, полиморфизм, абстракцию данных. В дальнейшем мы более подробно остановимся на этих понятиях и проиллюстрируем их использование в языке C++.

При создании языка C++ были созданы или использованы понятия, которые затем стали применяться в языке C и вошли в стандарт ANSI C. Таким образом, языки C и C++ оказывали взаимное влияние друг на друга.

Существует два подхода к языку C++. При первом подходе считается, что язык C является составной частью языка C++, и изложение ведется с учетом этого. Примером такого подхода является, например, описание языка C++ в документации к системам программирования, например к Borland C++ v.3.1. При втором подходе предполагается, что язык C++ является надстройкой над языком C, как это и было исторически. Мы при изложении материала в этой книге будем использовать второй подход, так что те, кто хочет изучить только язык C, смогут пользоваться этой книгой.

Практически все современные трансляторы с языков C и C++ используют стандарт ANSI языка C, т. е. все возможности и правила, предусмотренные этим стандартом, присутствуют. В то же время каждая из систем программирования на языках C и C++ содержит дополнительные возможности, связанные с конкретной операционной системой и архитектурой компьютера.



Мы будем ориентироваться на трансляторы, созданные фирмой Borland International Inc., и соответствующие системы программирования. В частности, мы кратко опишем возможности системы Borland C++ 3.1, используя ту ее часть, которая работает под управлением операционной системы MS DOS. Переход из одной системы программирования в другую уже становится более простым делом, особенно если используются только стандартные возможности языков C и C++. Для начального изучения языка не надо стремиться к использованию самой последней версии компилятора. Ее использование становится целесообразным при написании достаточно больших и сложных программ. Для изучения достаточно использования системы Turbo C++ v.1.01, которая замечательна тем, что требует всего около 6 Мбайт жесткого диска, 640 Кбайт оперативной памяти, т. е. может устанавливаться даже на IBM XT. Использование системы Borland C++ 3.1 требует уже 2 Мбайт оперативной памяти.

В своем изложении мы будем больше внимания уделять именно стандартным средствам языков C и C++, специально отмечая, когда те или другие особенности относятся к операционной системе MS DOS или конкретной архитектуре компьютера.

Книга не претендует на полноту и абсолютную строгость. Авторы пытались написать учебное пособие для тех, кто хочет изучить основы языков C и C++, научиться писать программы на этих языках и имеет начальные понятия хотя бы на уровне курса информатики средней школы или вуза.

Те читатели, которые знают язык C, могут пропустить главу, которая относится к языку C, и сразу перейти к изучению языка C++.

Документации и система Help большинства систем программирования используют английский язык. Поэтому мы будем иногда приводить английское написание тех или иных важных понятий или терминов языках C и C++.

Написание программы предусматривает выполнение определенного числа действий, которые с большей или меньшей детализацией можно разделить на следующие важнейшие этапы:

- постановка задачи;
- выбор метода решения задачи;
- написание исходного текста программы на языках C и C++;
- ввод исходного текста программы с помощью текстового редактора; текст может быть разбит на несколько файлов (модулей); на этом этапе мы получаем файлы исходного текста с расширением .c или .cpp;
- компиляция модулей (каждого модуля по отдельности или всех модулей вместе); на этом этапе мы получаем объектный файл, т. е. файл с расширением .obj;
- отладка синтаксиса программы;
- объединение откомпилированных модулей в программу (это часто называют компоновкой или линковкой (линкованием) программы); на этом этапе к программе подсоединяются необходимые стандартные библиотеки и мы получаем выполняемый файл ghjuhfvs с расширением .obj;

- запуск программы на выполнение;
- отладка программы (тестирование программы и отладка алгоритма);
- окончательное оформление программы.

При выполнении каждого из указанных этапов программирования возникает необходимость возврата на предыдущие этапы, иногда вплоть до изменения постановки задачи.

Современные системы программирования позволяют удобно переходить от одного этапа к другому. Это осуществляется наличием так называемой интегрированной среды программирования, которая содержит в себе текстовый редактор, компилятор, компоновщик, встроенный отладчик и, в зависимости от системы или ее версии, предоставляет программисту дополнительные удобства для написания и отладки программ.

Мы кратко опишем интегрированную среду Borland C++ 3.1, работающую под управлением операционной системы MS DOS. Подробное описание можно найти в документации, прилагающейся к этой системе, или воспользоваться системой Help интегрированной среды.

# 1 ИНТЕГРИРОВАННАЯ СРЕДА ПРОГРАММИРОВАНИЯ СИСТЕМЫ BORLAND C++

---

## ОСОБЕННОСТИ СИСТЕМЫ BORLAND C++ 3.1

Нашей задачей является описание основных возможностей и особенностей программирования в системе Borland C++. Работа над этой системой началась в 1988 году фирмой Borland International Inc. Этот программный продукт продолжает серию систем фирмы Borland, таких, как Turbo BASIC, Turbo Pascal, Turbo C++ 1.0, Borland C++ v.2.0, наследует лучшие их черты, имеет похожую интегрированную среду. Наряду с этим система Borland C++ имеет интегрированную среду, обладающую большими возможностями и удобствами для пользователя.

Система Borland C++ упрощает процесс программирования и делает его более эффективным. Запустив программу Borland C++, вы получаете комплекс услуг, который позволяет вам написать, отредактировать, откомпилировать и отладить программу. У вас всегда под рукой развитая система помощи Help, которая позволяет получить справочную информацию по всем вопросам, касающимся программирования в системе Borland C++. Все эти возможности предоставляет пользователю интегрированная развитая среда программирования - ИРС (Integrated Development Enviroment). Заметим, что по сравнению с предыдущими системами фирмы Borland ИРС предоставляет пользователю следующие дополнительные возможности:

- оконный интерфейс: возможность создания на экране нескольких окон, которые можно перемещать по экрану, изменяя их размеры;
- поддержка работы с мышью;
- наличие блоков диалога;
- возможность обмена из окна Help и между окнами редактирования;
- возможность быстрого перехода к другим программам, например к ассемблеру TASM, и быстрого возврата;
- наличие макроязыка редактора;
- подсветка лексем в процессе редактирования.

## РАБОТА В ИНТЕГРИРОВАННОЙ СРЕДЕ BORLAND C++

Система Borland C++ имеет два режима работы.

Первый, наиболее важный, который используется практически всегда, особенно начинающими программистами, - это режим работы с интегрированной средой. При использовании интегрированной среды процесс ре-

дактирования, компиляции, отладки и исполнения программы производится простым нажатием клавиш и использованием меню.

Второй режим работы - использование традиционного метода, когда вначале применяется какой-либо текстовый редактор для создания файла с программой, затем, набирая в командной строке DOS соответствующие команды для компиляции, линкование (компоновка) и, наконец, выполнения программы.

Обсудим, как пользоваться интегрированной развитой средой, используя наиболее важные моменты и не вникая в некоторые детали.

## ЗАПУСК СИСТЕМЫ BORLAND C++

Для того чтобы запустить систему Borland C++, надо, войдя в каталог, в котором расположен файл `bc.exe`, в командной строке DOS набрать `>bc`. При этом мы предполагаем, что система Borland C++ 3.1 уже установлена на компьютере. Если это необходимо, то в командной строке при вызове системы можно задать параметры, устанавливающие режим работы. Формат командной строки следующий:

```
> bc [<имя файла>][<имя проекта>][<парам> [<парам>...]]
```

Здесь `<имя файла>` - имя произвольного файла в формате ASCII (текстового файла), обычно это текст программы на языках C или C++; `<имя проекта>` - имя файла вашего проекта, он обязательно имеет расширение `.PRJ`; `<парам>` - один или несколько параметров.

Возможно использование следующих параметров: `/e`, `/x`, `/rx`, `/b`, `/d`, `/m`, `/l`, `/p`. Опишем коротко смысл использования этих параметров:

`/b` - произвести полную рекомпиляцию всех файлов проекта (build), компоновку и вернуться в DOS;

`/m` - произвести выборочную перекомпиляцию и компоновку (make) файлов проекта;

`/d` - работа в режиме двух мониторов;

`/e` и `/x` - использование дополнительной (expanded) и расширенной (extended) памяти;

`/rx` - вместо `x` следует подставить имя того дисковод, на который можно осуществлять быструю "откачку" данных, если вся дополнительная или расширенная память была использована для создания виртуального диска;

`/l` - при использовании компьютера с экраном на жидких кристаллах;

`/p` - управляет переключением палитры на EGA и VGA мониторах.

## ВЫХОД ИЗ СИСТЕМЫ BORLAND C++

Чтобы окончательно покинуть систему Borland C++, можно воспользоваться `QUIT` в меню `FILE` или нажать комбинацию клавиш `Alt-X`.

Для временного выхода в операционную среду, чтобы выполнить какую-либо команду DOS, оставив при этом программу в памяти машины,

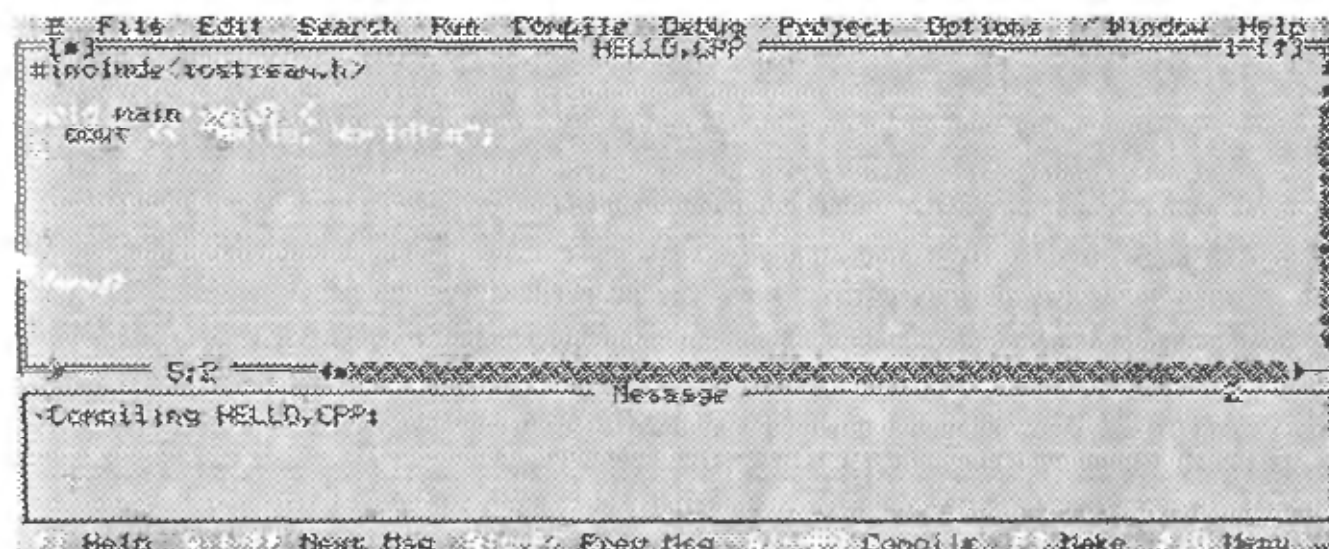


используется опция DOS Shell меню FILE. Для возврата в систему требуется набрать в командной строке DOS > EXIT.

Для временного перехода к другой программе можно использовать системное меню.

## КОМПОНЕНТЫ ИНТЕГРИРОВАННОЙ СРЕДЫ

Войдем в каталог, где находится система, и вызовем систему Borland C++. Вы увидите на экране то, что изображено на рисунке:



Экран содержит 4 основные части:

- главное меню;
- окно редактирования;
- окно сообщений;
- строку состояния.

## ОКНА СИСТЕМЫ BORLAND C++

Большая часть того, что вы видите и делаете в среде системы Borland C++, происходит в окне. Окно представляет собой прямоугольную область экрана, которую можно перемещать, у которой можно изменять размеры, которую можно распахивать на весь экран, перемещать, чтобы не произошло перекрытий с другими окнами, перекрывать другими окнами, закрывать и открывать.

В системе Borland C++ может существовать произвольное число окон (в пределах имеющейся памяти), но в каждый момент времени активным может быть только одно окно. Активным окном является то окно, в котором вы в настоящий момент работаете. Любые выбираемые команды или вводимый текст, как правило, относятся только к активному окну (если один и тот же файл открыт в нескольких окнах, то действие будет относиться ко всем окнам, в которых содержится файл.)

Система Borland C++ позволяет без затруднений определить, какое именно окно является активным. Это достигается при помощи двойных линий бордюра, в которые заключается активное окно. Активное окно редактирования всегда содержит маркер закрытия окна, маркер распахивания окна на весь экран, полосы прокрутки и угол изменения размера окна. Если ваши окна перекрываются, то активное окно всегда будет находиться поверх других окон (всегда располагается на переднем плане).

Существует несколько типов окон, но большая их часть имеет следующие общие элементы:

- строку заголовка;
- маркер закрытия окна;
- полосы прокрутки;
- угол изменения размера окна;
- маркер распахивания окна на весь экран;
- номер окна (от 1 до 9).

Окно редактирования отображает также в нижнем левом углу текущий номер строки и номер столбца. Если вы модифицировали текст в окне, то слева от номеров строки и столбца будет располагаться символ в виде звездочки (\*).

Для того чтобы быстро закрыть окно, достаточно подвести к маркеру закрытия окна курсор мыши и нажать кнопку.

Строка заголовка содержит название данного окна.

Маркер распахивания окна на весь экран содержит пиктограмму, к которой нужно подвести указатель мыши и нажать кнопку мыши, чтобы распахнуть на весь экран или вернуть окно в исходное состояние.

Первые 9 открытых окон имеют номер. Для активизации окна можно воспользоваться комбинацией клавиши Alt и цифры - номера окна.

Полосы прокрутки используются при наличии мыши для прокрутки содержимого окна.

Для увеличения или уменьшения размера окна необходимо "отбуксировать" угол изменения размера окна.

Строка заголовка, которая является самой верхней строкой окна, содержит название данного окна и номер окна. Для того чтобы распахнуть окно на весь экран, можно подвести указатель мыши к строке заголовка и дважды кратковременно нажать кнопку мыши. Для перемещения окна по экрану можно также осуществить "буксировку", захватив заголовок.

Маркер закрытия окна представляет собой прямоугольник, который расположен в верхнем левом углу окна. Для закрытия окна необходимо подвести к нему указатель мыши и кратковременно нажать кнопку мыши (можно также выбрать команду Window|Close (Окно|Закрыть) или нажать комбинацию клавиш Alt-F3). Окна Inspect (Анализ) и Help (Подсказка) считаются временными и могут быть закрыты с помощью нажатия клавиши Esc.

Полосы прокрутки представляют собой горизонтально или вертикально ориентированные полосы. Полосы прокрутки позволяют как работающим с

клавиатурой, так и работающим с мышью пользователям видеть, сколь далеко от начала файла они находятся.

Эти полосы прокрутки используются для прокрутки содержимого окна при помощи мыши. Для прокрутки на одну строку необходимо подвести указатель мыши к символу стрелки на одном из концов полосы прокрутки и нажать кнопку мыши (для непрерывной прокрутки необходимо держать кнопку мыши нажатой). Для однократной прокрутки на страницу в любую из сторон необходимо подвести указатель мыши к заполненной фактурой области по одну из сторон лифта и нажать кнопку мыши. Наконец, можно "отбуксировать" лифт на любое место полосы прокрутки для того, чтобы оперативно переместиться к тому месту в файле, относительная позиция которого от начала файла соответствует положению лифта на полосе прокрутки.

Для изменения размера окна с помощью клавиатуры необходимо выбрать команду Size|Move (Размер|Переместить) из меню Window (Окно) либо нажать комбинацию клавиш Ctrl-F5.

Для того, чтобы распахнуть окно на весь экран с помощью клавиатуры, выберите команду Window|Zoom (Окно|Распахнуть) или нажмите клавишу F5.

Строка состояния располагается у нижнего края экранного кадра системы Borland C++. Она напоминает об основных клавишах и клавишах активации, которые в настоящий момент могут быть применены к активному окну. Она позволяет установить указатель мыши на эти обозначения клавиш и кратковременно нажать кнопку мыши, чтобы выполнить указанное действие, вместо того чтобы выбирать команды из меню или нажимать соответствующие клавиши. Она сообщает, какое действие выполняется программой, и предлагает состоящие из одной строки советы и рекомендации по любой выбранной команде меню и элементам блока диалога. Строка состояния меняется по мере переключения от одного окна к другому или при переходе к различным действиям.

## ГЛАВНОЕ МЕНЮ



Для входа в главное меню можно нажать клавишу F10. В результате на одном из элементов меню появится подсвеченный курсор. Выбор нужного элемента меню можно осуществить двумя способами: перемещением курсора к нужному элементу меню и нажатием клавиши Enter или просто нажатием клавиши с первой буквой нужного элемента меню.

Ниже приводится краткое описание возможностей каждого элемента главного меню:

≡ - системное меню;

FILE - загрузка и создание файлов, сохранение внесенных в программу изменений, управление каталогами, выход в оболочку DOS, выход из системы;

EDIT - реализация различных режимов редактирования текста в активном окне;

SEARCH - поиск фрагментов текста, объявления функций, местоположения ошибок;

RUN - компиляция, компоновка, запись на выполнение программы, находящейся в активном окне редактирования;

COMPILE - компиляция программы из активного окна редактирования;

DEBUG - управление возможностями отладчика;

PROJECT - организация проектов (многофайловых программ);

OPTIONS - установка опций: параметров компиляции, компоновки и др.;

WINDOW - управление окнами;

HELP - обращение к системе оперативной подсказки.

После выбора элемента главного меню на экране появляется спускающееся меню со списком пунктов этого меню, на одном из которых будет подсвеченный курсор. Выбрать нужный пункт подменю можно установив курсор на этот пункт и нажав клавишу Enter или просто нажав клавишу с буквой, выделенной другим цветом.

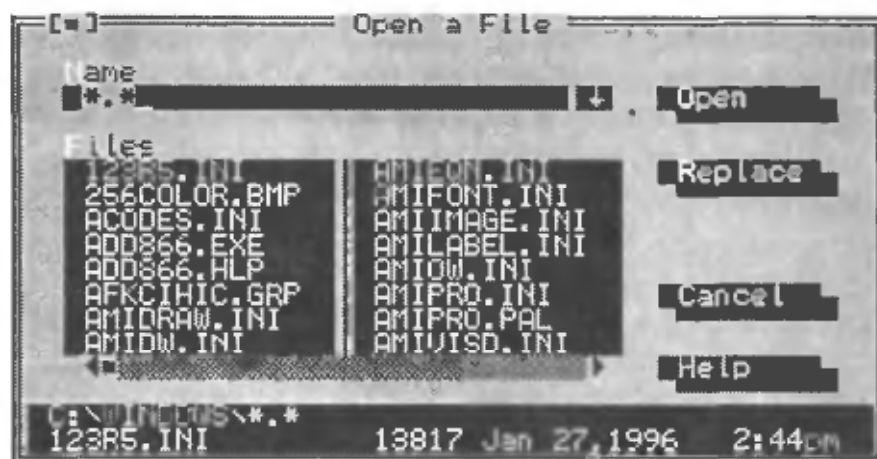
Прервать любое действие при работе с меню можно нажав клавишу Esc. Выполнение некоторых пунктов меню в данный момент становится невозможным. В этом случае нет букв, выделенных другим цветом, и курсор имеет другой цвет.

Некоторые элементы спускающихся меню могут иметь свои подменю, расширяющие детали этого пункта. Такие пункты отмечены темным треугольником справа. Некоторые пункты подменю имеют справа ключи On/Off. Изменить состояние можно подведя курсор к этому пункту и нажав клавишу Enter.

Если за пунктом спускающегося подменю будет три точки, то в результате выбора этого пункта на экране появится блок или окно диалога.

## Блок диалога

Блок диалога представляет собой удобный способ просмотра и задания параметров. Пример блока диалога приведен на рисунке ниже:





При работе в блоке диалога можно пользоваться пятью основными средствами управления: кнопками селективными и триггерными, кнопками действия, блоками ввода и блоками списка. Большинство блоков диалога имеет три стандартные кнопки действия: OK, Cancel, Help. Переход внутри окна диалога осуществляется клавишей Tab и стрелками. Клавиша Esc по действию равносильна кнопке Cancel.

### ТРИГГЕРНЫЕ И СЕЛЕКТИВНЫЕ КНОПКИ

В блоке диалога содержатся также триггерные кнопки. Когда вы выбираете триггерную кнопку, то у нее внутри появляется символ X, который указывает, что параметр, соответствующий данной кнопке, установлен в состояние On (Используется, Задан). Пустая триггерная кнопка указывает, что соответствующий ей параметр установлен в состояние Off (Не используется, Не задан). Если несколько триггерных кнопок относятся к некоторой теме, то они появляются объединенными в группу. В этом случае перемещение с помощью клавиши табуляции приводит к переходу к этой группе. После того как оказывается выбранной вся группа в целом, следует воспользоваться клавишами управления курсором, чтобы выделить световым маркером необходимый элемент, а затем нажать клавишу пробела, чтобы выбрать его.

В блоке диалога содержатся также селективные кнопки. Селективные кнопки отличаются от триггерных кнопок тем, что они представляют взаимно исключающие варианты выбора. По этой причине селективные кнопки всегда объединяются на экране в логически связанные группы, и в каждый момент в любой отдельно взятой группе в состоянии On (Задано, Используется) может быть только одна селективная кнопка. Далее приводится пример того, как выглядят триггерные и селективные кнопки в состоянии On (Используется, Задан) и в состоянии Off (Не используется, Не задан).

### БЛОКИ ВВОДА И БЛОКИ СПИСКА

Блоки ввода позволяют вводить текст и использовать большую часть клавиш редактирования текста. Если места в окне недостаточно, то производится автоматическая прокрутка текста. Переход в блок ввода осуществляется клавишей Tab. После ввода текста нажмите клавишу Enter.

Блок списка позволяет с помощью курсора выбрать нужный элемент из списка (например, нужный файл при загрузке в окно). Выбор производится стрелками после того, как сделали этот блок активным при помощи клавиши Tab.

## ОПИСАНИЕ ЭЛЕМЕНТОВ ГЛАВНОГО МЕНЮ

### СИСТЕМНОЕ МЕНЮ - ≡



Системное меню будет появляться у самого левого края строки меню. Самым быстрым способом обратиться к системному меню является использование комбинации клавиш Alt-клавиша пробела. Когда вы вызываете это меню, вы видите несколько команд, имеющих системное значение Repaint Desktop (Восстановить рабочую область) и имена тех программ, которые установлены с помощью команды

Options|Transfer (Параметры|Перейти в). Вы можете добавить в нижнюю часть системного меню вызов своих программ.

### КОМАНДА REPAINT DESKTOP (ВОССТАНОВИТЬ РАБОЧУЮ ОБЛАСТЬ)

В результате выбора команды Repaint Desktop (Восстановить рабочую область) система Borland C++ осуществляет перерисовку изображения на экране. Это может потребоваться, например, в том случае, если какая-либо программа оставила на экране "мусор" в результате работы программ, особенно при прямой работе с видеопамятью.

### ЭЛЕМЕНТЫ ПОДМЕНЮ TRANSFER (ПЕРЕЙТИ В)

В этом подменю будут появляться имена всех тех программ, которые вы устанавливали с помощью блока диалога Transfer (Перенести в), который вызывается на экран командой Options|Transfer (Параметры|Перенести в). Для того чтобы запустить одну из перечисленных здесь программ, необходимо выбрать ее имя из системного меню. Для того чтобы установить программы, которые в последующем будут появляться в этом подменю, необходимо выбрать команду Options|Transfer (Параметры|Перейти в).

### МЕНЮ FILE (ФАЙЛ)



Меню File (Файл) позволяет вам открывать и создавать файлы программ в окнах редактирования. Данное меню позволяет также сохранять внесенные изменения, выполнять другие действия над файлами, выходить в оболочку DOS и покидать систему Borland C++.

### КОМАНДА NEW (НОВЫЙ)

Команда File|New (Файл|Новый) позволяет вам открывать новое окно редактирования со стандартным именем NONAMExx.C (где вместо букв xx

задается число в диапазоне от 00 до 99. Эти файлы с именем NONAME (Безымянный) используются в качестве временного буфера для редактирования: когда вы сохраняете на диске файл с подобным именем, система Borland C++ запрашивает действительное имя для этого файла.

### **КОМАНДА OPEN (ОТКРЫТЬ)**

Команда File|Open (Файл|Открыть) отображает блок диалога, предназначенный для выбора файлов, чтобы можно было выбрать файл программы, который будет открыт в окне редактирования.

Этот блок диалога содержит в себе блок ввода, список файлов, кнопки с пометками Open (Открыть), Replace (Заменить), Cancel (Отменить) и Help (Подсказка), а также информационную панель, которая описывает выбранный в настоящий момент файл. Теперь можно выполнить одно из указанных далее действий:

- ввести полное имя файла и выбрать кнопку Replace (Заменить) или Open (Открыть);
- ввести имя файла с метасимволами. Это позволяет отфильтровать список файлов;
- нажать клавишу "Стрелка вниз", чтобы выбрать спецификацию файла из списка "предыстории", который содержит введенные вами ранее спецификации файлов;
- просмотреть содержимое других директорий, выбрав имя директории из списка файлов.

Блок ввода позволяет вам вводить имя файла явным образом или ввести имя файла со стандартными метасимволами, которые используются в операционной системе (\* и ?).

### **Блок списка FILE (ФАЙЛ)**

Для поиска имени файла можно ввести букву по нижнему регистру, а для поиска имени директории - ввести букву по верхнему регистру. Блок списка File (Файл) отображает все имена файлов в текущей директории, которые соответствуют спецификациям в блоке ввода, отображает имя текущей директории, а также имена всех поддиректорий. Подведите к блоку списка указатель мыши и нажмите кнопку мыши либо нажимайте клавишу Tab до тех пор, пока имя блока списка не будет выделено световым маркером. Теперь можно нажать клавишу "Стрелка вверх" или "Стрелка вниз", чтобы выбрать имя файла, а затем нажать клавишу Enter, чтобы открыть его.

Панель информации о файле, расположенная у нижнего края блока диалога Load a File (Загрузить файл), отображает маршрут, имя файла, дату и время создания, а также размер выбранного вами в блоке списка файла. (Ни один из элементов в этой панели нельзя выбрать.) По мере прокрутки по блоку списка содержимое этой панели обновляется, отображая информацию по каждому из файлов.

**Команда SAVE (СОХРАНИТЬ)**

Команда File|Save (Файл|Сохранить) осуществляет запись на диск того файла, который находится в активном окне редактирования.

**Команда SAVE AS (СОХРАНИТЬ ПОД ИМЕНЕМ)**

Команда File|Save As (Файл|Сохранить под именем) позволяет вам сохранить файл в активном окне редактирования под другим именем, в другой директории или на другом дисковом диске.

Введите новое имя (возможен также ввод идентификатора дискового диска и имени директории), а затем подведите указатель мыши к кнопке ОК (Выполнить) и нажмите кнопку мыши, либо выберите кнопку ОК (Выполнить). Будут обновлены заголовки всех окон, в которых содержится данный файл.

**Команда SAVE ALL (СОХРАНИТЬ ВСЕ)**

Команда File|Save All (Файл|Сохранить все) действует аналогично команде Save (Сохранить) с тем исключением, что она осуществляет запись на диск содержимого всех модифицированных файлов, а не только того файла, который находится в активном окне редактирования.

**Команда CHANGE DIR (СМЕНИТЬ ДИРЕКТОРИЮ)**

Команда File|Change Dir (Файл|Сменить директорию) позволяет вам задать идентификатор дискового диска и имя директории, которые следует сделать текущими. Текущей директорией является та директория, которая используется системой Borland C++ для сохранения файлов и поиска файлов.

Сменить директорию можно двумя методами:

- вводя маршрутное имя новой директории в блок ввода и нажимая клавишу Enter;
- выбирая необходимую вам директорию в дереве директорий.

**Команда PRINT (ПЕЧАТАТЬ)**

Команда File|Print (Файл|Печатать) позволяет вам вывести на печать содержимое активного окна редактирования. Команда посылает файл на устройство печати, заданное в операционной системе.

Для вызова можно воспользоваться комбинацией клавиш Ctrl-K-P.

**Команда DOS SHELL (ВЫХОД В ОБОЛОЧКУ DOS)**

Команда File|DOS Shell (Файл|Выход в оболочку DOS) позволяет вам временно выйти из системы Borland C++, чтобы выполнить команду DOS или запустить какую-либо программу. Для того чтобы возвратиться в систему Borland C++, необходимо ввести с клавиатуры EXIT и нажать клавишу Enter.



### Команда QUIT (Выйти)

Команда File|Quit (Файл|Выйти) осуществляет выход из системы Borland C++, удаляет ее из памяти и возвращает вас к запросу со стороны DOS. Если вы внесли какие-либо модификации, которые еще не были сохранены, то система Borland C++ запросит вас, хотите ли вы, чтобы они были сохранены перед выходом. Можно выйти используя клавиши Alt-X.

### Меню EDIT (РЕДАКТИРОВАНИЕ)

Undo	Alt+BkSp
Redo	Shift+Alt+BkSp
Cut	Shift+Del
Copy	Ctrl+Ins
Paste	Shift+Ins
Clear	Ctrl+Del
Copy example	
Show clipboard	

Меню Edit (Редактирование) позволяет выполнять вырезание, копирование и вставку текста в окна редактирования. Можно также открыть окно текстового буфера для просмотра или редактирования его содержимого. Более подробно редактор системы Borland C++ будет рассмотрен в дальнейшем.

### Команда UNDO (ВОССТАНОВИТЬ)

Команда Edit|Undo (Редактировать|Восстановить) отменяет действие последней команды редактирования, которая была применена к какой-либо строке.

### Команда REDO (ПОВТОРИТЬ)

Команда Edit|Redo (Редактировать|Повторить) повторяет действие последней команды редактирования.

### Команда CUT (ВЫРЕЗАТЬ)

Команда Edit|Cut (Редактировать|Вырезать) удаляет выделенный фрагмент текста из вашего документа и заносит его в текстовый буфер. Затем можно вклеить текст в любой другой документ (или в какое-либо другое место того же самого документа) путем выбора команды Paste (Вставить). Текст в текстовом буфере остается выделенным, поэтому можно вклеивать тот же самый текст многократно.

### Команда COPY (КОПИРОВАТЬ)

Команда Edit|Copy (Редактировать|Копировать) оставляет выделенный текст нетронутым, но заносит в текстовый буфер точную копию этого текста. Затем можно вклеить текст в любой другой документ (или в какое-либо другое место того же самого документа) путем выбора команды Paste (Вклеить). Можно также скопировать текст из окна Help (Подсказка): при работе с клавиатурой следует воспользоваться комбинацией клавиши Shift и клавиш управления курсором; при работе с мышью следует подвести указатель мыши в нужную позицию, кратковременно нажать кнопку мыши, а

затем "отбуксировать" указатель по тексту до того места, где кончается необходимый фрагмент.

### КОМАНДА PASTE (ВСТАВИТЬ)

Команда Edit|Paste (Редактировать|Вставить) вставляет текст, расположенный в текстовом буфере, в текущее окно в ту позицию, где располагается курсор. Тот текст, который вы вклеиваете, представляет собой помеченный в настоящий момент фрагмент текста в окне Clipboard (Текстовый буфер).

### КОМАНДА CLEAR (СТЕРЕТЬ)

Команда Edit|Clear (Редактировать|Стереть) удаляет выбранный фрагмент текста, но не заносит его в текстовый буфер.

### КОМАНДА COPY EXAMPLE (СКОПИРОВАТЬ ПРИМЕР)

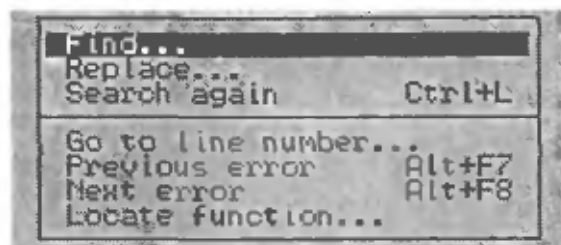
Команда Edit|Copy Example (Редактировать|Скопировать пример) копирует предварительно выбранный в окне Help (Подсказка) текст примера в окно Clipboard (Текстовый буфер). Примеры уже определены как предназначенные для вклейки блоки.

### КОМАНДА SHOW CLIPBOARD (ПОКАЗАТЬ СОДЕРЖИМОЕ ТЕКСТОВОГО БУФЕРА)

Команда Edit|Show Clipboard (Редактировать|Показать содержимое текстового буфера) открывает окно Clipboard (Текстовый буфер), в котором хранятся фрагменты текста, отсеченного и скопированного вами из других окон. Тот текст, который в настоящий момент выделен, представляет собой текст, который система Borland C++ будет использовать при выборе команды Paste (Вклеить).

Окно Clipboard (Текстовый буфер) аналогично любому другому окну редактирования: его можно перемещать, его размеры можно менять, а содержимое - прокручивать и редактировать. Единственное отличие, которое вы обнаружите в окне Clipboard (Текстовый буфер), будет заключаться в том, как оно хранит результаты выполнения команд Cut и Copy. Когда вы выбираете текст в окне Clipboard (Текстовый буфер), а затем выполняете команду Cut или Copy, выбранный текст немедленно появляется у нижнего края окна. (Помните, что любой текст, который вы вырезаете или копируете, добавляется в конец окна Clipboard (Текстовый буфер) - поэтому в дальнейшем можно использовать его для вклейки.)

### МЕНЮ SEARCH (ПОИСК)



Меню Search (Поиск) позволяет вам осуществлять поиск текста, объявлений функций, местоположения ошибок в ваших файлах.

## Команда Find (Найти)

Команда Search|Find (Поиск|Найти) отображает блок диалога Find (Найти), который позволяет вам ввести образец для поиска и задать параметры, оказывающие влияние на процесс поиска. (Эта команда может также быть вызвана с помощью комбинации клавиш Ctrl-Q F).

Блок диалога Find (Найти) содержит несколько кнопок и блоков:

☐ Case sensitive ☐ чувствительность к регистру |

В блок Case sensitive (Чувствительность к регистру) следует занести маркер, если вам необходимо, чтобы система Borland C++ различала символы верхнего и нижнего регистров.

☐ Whole words only ☐ только целые слова |

В блок Whole words only (Только целые слова) следует занести маркер, если вам необходимо, чтобы система Borland C++ осуществляла бы поиск только целых слов (т. е., строк, с обеих концов которых располагаются символы пунктуации или символы пробела).

☐ Regular expression ☐ регулярные выражения |

В блок Regular expression (Регулярные выражения) следует занести маркер, если вам необходимо, чтобы система Borland C++ распознавала используемые утилитой GREP метасимволы в строке поиска. В число метасимволов входят следующие символы: ^, \$, ., \*, +, [] и \. Далее приводятся их значения:

^ - этот символ в начале строки соответствует началу строки текста;

\$ - символ доллара в конце выражения соответствует концу строки текста;

. - точка соответствует любому символу;

\* - символ, предшествующий звездочке, соответствует любому числу экземпляров (в том числе нулевому) этого символа. Например, последовательности bo\* будет соответствовать строка bot, b, boo, а также be;

+ - символ, предшествующий плюсу, соответствует любому числу экземпляров (но не нулевому) этого символа. Например, последовательности bo+ будет соответствовать строка bot или boo, но не be или b;

[] - символы, заключенные в такие скобки, соответствуют любому одному символу, который присутствует в скобках, но не другим символам. Например, [bot] соответствует символам b, o или t;

[^] - символ "крышечки" в начале строки, заключенной в квадратные скобки, соответствует префиксу "не". Следовательно, последовательность [^bot] соответствует любым символам за исключением b, o или t;

[-] - дефис, заключенный в угловые скобки, означает диапазон символов. Например, [b-o] соответствует любому символу в диапазоне от b до o;

\ - обратная косая перед метасимволом указывает системе Borland C++, чтобы она интерпретировала данный символ буквально, а не как метасимвол. Например, \^ соответствует символу ^, а не представляет начало строки.

Для того чтобы начать поиск, необходимо ввести строку в блок ввода и выбрать кнопку ОК (Выполнить). Для того чтобы отменить поиск строки, необходимо выбрать кнопку Cancel (Отмена). Если вы хотите ввести строку, поиск которой вы уже осуществляли, необходимо нажать клавишу "Стрелка вниз", чтобы отобразить список "предыстории", из которого следует осуществить выбор.

Можно также выбрать то слово, на котором в настоящий момент позиционирован курсор в окне редактирования, и использовать его в блоке Find (Найти), вызвав лишь команду Find (Найти) из меню Search (Поиск). Можно также выбрать из текста окна редактирования дополнительные символы, нажав клавишу "Стрелка вправо".

<i>Direction</i>		<i>Направление</i>	
(*)	Forward	(*)	Вперед
()	Backward	()	Назад
<i>Scope</i>		<i>Область поиска</i>	
(*)	Global	(*)	Глобальная
()	Selected text	()	Выделенный текст

Задайте область поиска, чтобы указать системе Borland C++, в какой части файла необходимо осуществлять поиск. Поиск может производиться во всем файле (Global) или только в выбранном тексте.

<i>Origin</i>		<i>Начало поиска</i>	
(*)	From cursor	(*)	От курсора
()	Entire scope	()	Весь диапазон

Осуществите выбор из группы селективных кнопок Origin (Начало поиска), чтобы определить, где начинается поиск. Если выбрана кнопка Entire scope (Весь диапазон), то селективные кнопки Direction (Направление) определяют, с конца или с начала выбранного диапазона должен начинаться поиск. Требуемый диапазон поиска задается с помощью селективных кнопок Scope (Область поиска).

### КОМАНДА REPLACE (ЗАМЕНИТЬ)

Команда Search|Replace (Поиск|Замена) отображает блок диалога, который позволяет вам вводить искомый текст и текст, на который его следует заменить.

Блок диалога Replace содержит несколько селективных кнопок и триггерных кнопок - многие из них идентичны тем, которые имеются в блоке диалога Find, который описан выше.

Дополнительная триггерная кнопка Prompt to Replace определяет, будет ли выдаваться запрос на каждое изменение.

Для того чтобы начать процесс поиска, необходимо ввести искомую строку и строку замены в блоки ввода и выбрать кнопку ОК (Выполнить) или кнопку Change All (Заменить все). Нажатие клавиши Cancel (Отме-



нить) приводит к отмене поиска и удалению этого блока диалога. Если вы хотите ввести строку, которую вы использовали ранее, нажмите клавишу "Стрелка вниз", чтобы отобразить список "предыстории", из которого следует сделать выбор.

Если система Borland C++ обнаруживает искомый текст, она запрашивает, хотите ли вы осуществить замену. Если вы выбрали кнопку ОК, система будет осуществлять поиск и замену только первого экземпляра искомой строки. Если вы выбираете кнопку Change All, то система будет осуществлять поиск и замену всех обнаруженных экземпляров строки в направлении, которое задается параметрами Direction (Направление), Scope (Область поиска) и Origin (Начало поиска).

Аналогично тому как это делается в блоке диалога Find, можно выбрать слово, расположенное у курсора в окне редактирования, и воспользоваться им в блоке ввода Text to Find. Для этого необходимо лишь вызвать команду Find or Replace из меню Search (Поиск). Кроме того, можно также выбрать из текста окна редактирования дополнительные символы, нажав клавишу "Стрелка вправо".

#### **КОМАНДА SEARCH AGAIN (ПОВТОРИТЬ ПОИСК)**

Команда Search|Search Again (Поиск|Повторить поиск) повторяет действие последней команды Find или последней команды Replace. Все параметры, которые были заданы при последнем обращении к использованному блоку диалога (Find или Replace), остаются при выборе команды Search Again действительными.

#### **КОМАНДА GO TO LINE NUMBER (ПЕРЕЙТИ К СТРОКЕ С НОМЕРОМ)**

Команда Search|Go to Line Number (Поиск|Перейти к строке с номером) запрашивает у вас номер той строки, к которой требуется осуществить переход. Система Borland C++ отображает текущий номер строки и номер столбца в левом нижнем углу каждого окна редактирования.

#### **КОМАНДА PREVIOUS ERROR (ПРЕДЫДУЩАЯ ОШИБКА)**

Команда Search|Previous Error (Поиск|Предыдущая ошибка) перемещает курсор на позицию возникновения предыдущего сообщения об ошибке или предупреждающего сообщения.

#### **КОМАНДА NEXT ERROR (СЛЕДУЮЩАЯ ОШИБКА)**

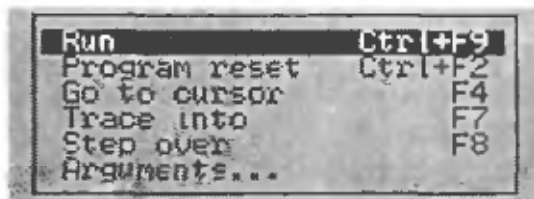
Команда Search|Next Error (Поиск|Следующая ошибка) перемещает курсор на позицию возникновения следующего сообщения об ошибке или предупреждающего сообщения. Эта команда оказывается доступной только в том случае, если в окне Message имеются сообщения, с которыми связаны какие-либо номера строк.

### КОМАНДА LOCATE FUNCTION (МЕСТОПОЛОЖЕНИЕ ФУНКЦИИ)

Команда Search|Locate Function (Поиск|Местоположение функции) отображает блок диалога, в который вы должны ввести имя искомой функции. Данная команда оказывается доступной только в период проведения отладочного сеанса.

Введите имя функции или нажмите клавишу "Стрелка вниз", чтобы выбрать какое-либо имя из списка "предыстории". В противоположность команде Find (Найти) данная команда ищет объявление функции, а не случаи ее вызова.

### МЕНЮ RUN (ВЫПОЛНЕНИЕ)



Меню Run (Выполнение) содержит команды, которые осуществляют выполнение вашей программы, а также инициализируют и завершают сеанс отладки.

### КОМАНДА RUN (ВЫПОЛНЕНИЕ)

Команда Run|Run (Выполнение|Выполнение) осуществляет выполнение вашей программы, используя те аргументы, которые переданы программе с помощью команды Run|Arguments. Если с момента последней компиляции исходный код был модифицирован, эта команда вызовет также Менеджер проектов (Project Manager), чтобы рекомпилировать и перекомпоновать программу. (Менеджер проектов представляет собой инструментальное средство создания программ, которое встроено в интегрированную среду.)

Если вы хотите, чтобы все возможности (отладочные) системы Borland C++ были вам доступны, необходимо, чтобы параметр Source Debugging (Отладка на уровне исходного кода) находился бы в состоянии On.

### КОМАНДА PROGRAM RESET (РЕИНИЦИАЛИЗАЦИИ ПРОГРАММЫ)

Команда Run|Program Reset (Выполнение|Реинициализация программы) прекращает текущий сеанс отладки, освобождает память, которая была выделена вашей программе, и закрывает все открытые файлы, которые использовались в вашей программе. Этой командой следует пользоваться также в том случае, если вы проводите отладку и обнаруживаете нехватку памяти для выполнения программы переноса или вызова оболочки DOS.

### КОМАНДА GO TO CURSOR (ВЫПОЛНИТЬ ДО КУРСОРА)

Команда Run|Go to Cursor (Выполнение|Выполнить до курсора) осуществляет выполнение вашей программы от маркера выполнения (run bar) (строки, которая выделена с помощью светового маркера в исходном коде) до той строки, в которой позиционирован курсор в текущем окне редактирования. Если курсор находится в той строке, которая не содержит выполняемого оператора, выполнение этой команды приводит к выдаче предупреждающего сообщения.

Командой **Go to Cursor** (Выполнить до курсора) следует пользоваться для продвижения маркера выполнения до той части вашей программы, которую вы хотите отладить. Если вам необходимо, чтобы ваша программа останавливалась бы на определенном операторе каждый раз, когда она достигает этого оператора, вам следует установить на этом операторе точку приостанова.

### **КОМАНДА TRACE INTO (ВХОЖДЕНИЕ ПРИ ТРАССИРОВКЕ)**

Команда **Run|Trace Into** (Выполнить|Вхождение при трассировке) осуществляет пооператорное выполнение вашей программы. Когда при пооператорном выполнении достигается вызов какой-либо функции, то будет выполняться каждый оператор этой функции вместо того, чтобы выполнить эту функцию за один шаг.

### **КОМАНДА STEP OVER (ОДНОШАГОВОЕ ВЫПОЛНЕНИЕ ФУНКЦИИ)**

Команда **Run|Step Over** (Выполнить|Одношаговое выполнение функции) выполняет следующий оператор в текущей функции. Она не осуществляет трассирующего вхождения в вызовы функций более низкого уровня, даже в том случае, если они доступны отладчику.

Командой **Step Over** (Одношаговое выполнение функции) следует пользоваться в тех случаях, когда необходимо отладить функцию в пооператорном режиме выполнения без вхождения в другие функции.

### **КОМАНДА ARGUMENTS (АРГУМЕНТЫ)**

Команда **Run|Arguments** (Выполнить|Аргументы) позволяет вам задавать выполняемой вами программе аргументы командной строки в точности так же, как если бы они вводились в командной строке DOS. Команды переадресации DOS будут игнорироваться. Когда вы выбираете данную команду, появляется блок диалога с одним единственным блоком ввода.

### **МЕНЮ COMPILE (КОМПИЛЯЦИЯ)**



Команды из меню **Compile** (Компиляция) используются для компиляции программы, присутствующей в активном окне, а также для полной или избирательной перекompиляции всех файлов вашего проекта. Для того, чтобы использовать команды **Compile**, **Make**, **Build** и **Link**, необходимо,

чтобы в активном окне редактирования был открыт какой-либо файл или чтобы был задан какой-либо проект (для команд **Make**, **Build** и **Link**). Например, если вы открываете окно **Message** (Сообщение) или **Watch** (Промотр выражений), указанные команды меню будут запрещены.

### **КОМАНДА COMPILE TO OBJ (КОМПИЛИРОВАТЬ В ОБЪЕКТНЫЙ ФАЙЛ)**

Команда **Compile|Compile to OBJ** (Компиляция|Компилировать в объектный файл) осуществляет компиляцию активного файла редактора (файла с

расширением .c или .cpp в файл с расширением .OBJ). Это меню всегда отображает имя того файла, который должен быть создан в результате, например:

Compile to OBJ C EXAMPLE OBJ

Когда система Borland C++ выполняет компиляцию, на экран выдается блок статуса, в котором отображается прохождение процесса компиляции и результаты. Когда будет завершена компиляция/компоновка, для удаления с экрана данного блока следует нажать произвольную клавишу. Если возникают какие-либо сообщения об ошибках или предупреждающие сообщения, активным становится окно Message, в котором отображается и выделяется световым маркером первое сообщение об ошибке.

#### **КОМАНДА MAKE EXE FILE**

##### **(ИЗБИРАТЕЛЬНАЯ ПЕРЕКОМПИЛЯЦИЯ В EXE-ФАЙЛ)**

Команда Compile|Make EXE file (Компиляция|Избирательная перекомпиляция в выполняемый файл) вызывает Менеджер проектов для создания EXE-файла. Данная команда меню всегда отображает имя того EXE-файла, который должен быть создан в результате.

#### **КОМАНДА LINK EXE FILE (КОМПОНОВАТЬ EXE-ФАЙЛ)**

Команда Compile|Link EXE file (Компиляция|Компоновать EXE-файл) использует текущий .obj и .lib файлы (задаваемые либо по умолчанию, либо в текущем файле проекта) и компоует их, не производя избирательной компиляции; в результате получается новый EXE-файл.

#### **КОМАНДА BUILD ALL (ПОЛНАЯ ПЕРЕКОМПИЛЯЦИЯ ВСЕХ ФАЙЛОВ)**

Команда Compile|Build All (Компиляция|Полная перекомпиляция всех файлов) осуществляет полную перекомпиляцию всех файлов, составляющих ваш проект, вне зависимости от того, как их дата исходного кода соотносится с датой объектного кода.

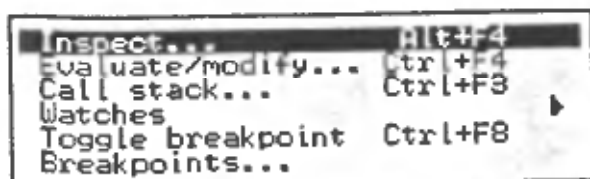
#### **КОМАНДА INFORMATION... (ИНФОРМАЦИЯ)**

Команда Compile | Information... (Компиляция|Информация) выдает окно диалога с информацией о текущем состоянии системы.

#### **КОМАНДА REMOVE MESSAGES (УДАЛИТЬ СООБЩЕНИЯ)**

Команда Compile|Remove Messages (Компиляция|Удалить сообщения) удаляет все сообщения из окна Message.

## МЕНЮ DEBUG (ОТЛАДКА)



Команды из меню Debug (Отладка) управляют всеми возможностями интегрированного отладчика. Можно изменить стандартные значения параметров для этих команд с помощью блока диалогов Options\Debugger.

### КОМАНДА INSPECT (ИНСПЕКТИРОВАТЬ)

Команда Debug\Inspect (Отладка\Инспектировать) открывает окно Inspector (Инспекция), которое позволяет вам проанализировать значения объекта. Окно особенно удобно для просмотра содержимого объектов при отладке программ на языке C++.

### КОМАНДА EVALUATE/MODIFY (ВЫЧИСЛИТЬ/МОДИФИЦИРОВАТЬ)

Команда Debug\Evaluate/Modify (Отладка\Вычислить/Модифицировать) вычисляет значение переменной или выражения, отображает их значение и, если это возможно, позволяет вам модифицировать это значение. Данная команда открывает блок диалогов, в котором содержится три поля: поле Expression, поле Result и поле New Value.

### КОМАНДА CALL STACK (СТЕК ВЫЗОВОВ)

Команда Debug\Call Stack открывает блок диалогов, в котором содержится стек вызовов. Окно Call Stack отображает последовательность функций, которые вызывались вашей программой для достижения той функции, которая выполняется в настоящий момент. У основания стека располагается функция main (основная); у вершины стека находится та функция, которая выполняется в настоящий момент. Каждый элемент стека отображает имя вызванной функции, а также значения параметров, которые были ей переданы.

При первом обращении к этому окну расположенный у верхнего края окна элемент будет выделен с помощью цветового или светового маркера. Для того чтобы отобразить текущую строку любой другой функции, в стеке вызовов следует выбрать имя этой функции и нажать клавишу Enter. Курсор перемещается к строке, содержащей вызов той функции, которая расположена непосредственно над исследуемой в стеке вызовов.

### КОМАНДА WATCHES (ПРОСМОТР ВЫРАЖЕНИЙ)

Команда Debug\Watches (Просмотр выражений) открывает всплывающее меню, содержащее команды, которые управляют использованием точек просмотра. В представленных далее разделах описываются команды, которые входят в это меню.

**Команда Add Watch (Добавить выражение просмотра)**

Команда Add Watch (Добавить выражение просмотра) вставляет в окно Watch еще одно выражение просмотра. При выборе данной команды отладчик открывает блок диалога и просит вас ввести выражение просмотра. В качестве стандартного выражения используется слово, на котором в настоящий момент позиционирован курсор в текущем окне редактирования. Если вы хотите оперативно ввести то выражение, которое уже использовалось ранее, можно воспользоваться списком "предыстории".

**Команда Delete Watch (Удалить выражение просмотра)**

Команда Delete Watch (Удалить выражение просмотра) удаляет из окна Watch текущее выражение просмотра.

Перед использованием данной команды окно Watch должно быть активным окном. Если окно Watch является активным, то текущее выражение просмотра оказывается выделенным. Если окно Watch в настоящий момент не является активным, а активным является другое окно, меню или блок диалога, то это выражение будет помечено точкой в левом поле.

**Команда Edit Watch (Редактировать выражение просмотра)**

Команда Edit Watch (Редактировать выражение просмотра) позволяет вам редактировать текущее выражение просмотра в окне Watch. Для того чтобы сохранить время, затрачиваемое на повторный ввод, можно воспользоваться списком "предыстории".

Когда вы выбираете эту команду, отладчик открывает блок диалога, содержащий копию текущего выражения просмотра. Отредактируйте это выражение и нажмите клавишу Enter. Отладчик заменит оригинальную версию выражения на отредактированную вами.

**Команда Remove All Watches (Удалить все выражения просмотра)**

Команда Remove All Watches (Удалить все выражения просмотра) удаляет все выражения просмотра из окна Watch.

**Команда Toggle Breakpoint (Триггерная установка точки останова)**

Команда Debug|Toggle Breakpoint (Отладка|Триггерная установка точки останова) позволяет вам установить или отменить безусловную точку останова на той строке, где позиционирован курсор. Когда точка останова установлена, она отмечается с помощью маркера точки приостанова.

**Команда Breakpoints... (Точки останова)**

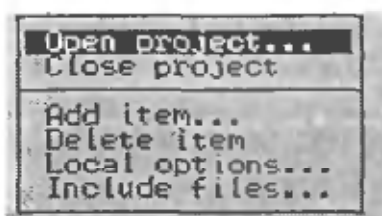
Команда Debug|Breakpoints... (Отладка|Точки останова) открывает блок диалога, который позволяет вам управлять использованием точек останова - как условных (вызывающих приостанов выполнения программы в зависимости от выполнения определенного условия), так и безусловных



(которые вызывают останов выполнения программы вне зависимости от каких-либо условий).

Блок диалога показывает вам все точки останова, которые были заданы, соответствующие им номера строк, а также условия, при которых будет совершаться приостанов выполнения программы.

### МЕНЮ PROJECT (ПРОЕКТ)



Меню Project (Проект) содержит все команды управления проектом, которые необходимы для компиляции многофайловой программы, чтобы:

- создать проект;
- добавить или удалить из проекта файлы;
- указать, с помощью какой программы будет транслироваться ваш исходный файл;
- задать параметры для какого-либо файла в проекте;
- указать, какие параметры переопределения командной строки должны использоваться для транслирующей программы;
- указать, какое имя должен получить результирующий объектный модуль, куда он должен быть помещен, должен ли данный модуль быть оверлейным и будет ли этот модуль содержать отладочную информацию;
- просмотреть включаемые файлы для конкретного файла, входящего в проект.

### КОМАНДА OPEN PROJECT (ОТКРЫТЬ ПРОЕКТ)

Команда Project|Open Project (Проект|Открыть проект) отображает блок диалога Load Project File, который позволяет вам выбрать и загрузить какой-либо проект или создать новый проект, введя его имя. Обычно файл проекта получает расширение имени .PRJ.

### КОМАНДА CLOSE PROJECT (ЗАКРЫТЬ ПРОЕКТ)

Команду Project|Close Project (Проект|Закрыть проект) следует использовать, когда вы хотите удалить свой проект и вернуться к используемому по умолчанию проекту.

### КОМАНДА ADD ITEM (ДОБАВИТЬ ЭЛЕМЕНТ)

Команду Project|Add Item (Проект|Добавить элемент) следует использовать, когда вы хотите добавить к списку проекта какой-либо файл. В результате выбора этой команды на экране появляется блок диалога Add Item to Project List.

### **КОМАНДА DELETE ITEM (УДАЛИТЬ ЭЛЕМЕНТ)**

Команду Project|Delete Item (Проект|Удалить элемент) следует использовать, когда вы хотите удалить имя какого-либо файла в окне Project (Проект). Если окно Project (Проект) находится в активном состоянии, то для удаления файла можно нажать клавишу Del.

### **КОМАНДА LOCAL OPTIONS (ЛОКАЛЬНЫЕ ПАРАМЕТРЫ)**

Команда Project|Local Options (Проект|Локальные параметры) открывает блок диалога Override Options (Переопределение параметров). Все программы, которые вы устанавливали в блоке диалога Transfer (Перенос) с "промаркированными" триггерными кнопками Translator (Транслятор), появляются в списке Project File Translators (Трансляторы объектных файлов в проекте).

### **КОМАНДА INCLUDE FILES (ВКЛЮЧАЕМЫЕ ФАЙЛЫ)**

Команду Project|Include Files (Проект|Включаемые файлы) следует использовать для отображения блока диалога Include Files; она нужна в случае, когда вы хотите увидеть, какие файлы включены в тот файл, который вы выбрали из окна Project. Когда вы находитесь в окне Project, для отображения блока диалога Include Files можно нажать клавишу пробела. Если вы еще не создали проект, эта команда будет запрещена.

### **МЕНЮ OPTIONS (ПАРАМЕТРЫ)**

Меню Options (Параметры) содержит команды, которые позволяют вам просматривать и модифицировать различные стандартные параметры, определяющие функционирование системы Borland C++. Выбор большинства команд в этом меню приводит к открытию какого-либо блока диалога.

Более подробно имеет смысл говорить о настройке системы после того, как будут изучены особенности языков C и C++. Многие понятия, используемые при настройке параметров компилятора, интегрированной среды и других параметров, сейчас вам еще не известны. Рекомендуем после того, как вы изучите основные понятия языков C и C++, просмотреть меню Options (Параметры) и прочитать документацию к системе Borland C++ (User Guide).

## МЕНЮ WINDOW (ОКНО)



Меню Window (Окно) содержит команды управления окнами. Большая часть окон, которую вы открываете из этого меню, содержит все стандартные элементы окна, в число которых входят полосы прокрутки, маркер закрытия окна и маркеры увеличения окна до размера экрана. У нижнего края меню Window появляется команда Window/List (Окно/Список). Для того чтобы получить список всех открытых окон, а также недавно закрытых окон, следует выбрать эту команду. (Информация о недавно закрытом окне появляется со словом closed (закрыто) перед именем этого окна; чтобы повторно открыть его, его необходимо выбрать).

### Команда SIZE/MOVE (ИЗМЕНИТЬ РАЗМЕР/ПЕРЕМЕСТИТЬ)

Команда Window/Size/Move (Окно/Изменить размер/Переместить) используется для того, чтобы изменить размер или местоположение активного окна.

### Команда ZOOM (УВЕЛИЧИТЬ ОКНО ДО РАЗМЕРОВ ЭКРАНА)

Команда Window/Zoom (Окно/Увеличить окно до размеров экрана) используется для изменения размера окна, в результате чего оно получает максимально допустимый размер. Если это окно уже увеличено до максимально возможного состояния, то выбор данной команды приводит к восстановлению его предыдущего размера. Для того чтобы увеличить окно на весь экран или вернуть его к предыдущему состоянию, можно также подвести мышь к произвольному месту верхней строки (за исключением мест размещения пиктограмм) и дважды подряд нажать кнопку мыши.

### Команда TILE (МОЗАИЧНОЕ РАСПОЛОЖЕНИЕ)

Команда Window/Tile (Окно/Мозаичное расположение) располагает все открытые окна таким образом, чтобы они были одновременно видны и не перекрывали одно другое.

### Команда CASCADE (КАСКАДНОЕ РАСПОЛОЖЕНИЕ)

Команда Window/Cascade (Окно/Каскадное расположение) располагает все открытые окна каскадом (уступами), чтобы край последующего окна выступал из-под предыдущего.

### **КОМАНДА NEXT (СЛЕДУЮЩЕЕ)**

Команда Window|Next (Окно|Следующее) делает активным следующее окно, в результате выбранное окно расположится на самой вершине стека окон.

### **КОМАНДА CLOSE (ЗАКРЫТЬ)**

Команда Window|Close (Окно|Заккрыть) используется для того, чтобы закрыть активное окно. Для закрытия окна можно также подвести указатель мыши к маркеру закрытия окна в верхнем левом углу окна и нажать кнопку мыши.

### **КОМАНДА CLOSEALL (ЗАКРЫТЬ ВСЕ)**

Команда Window|CloseAll (Окно|Заккрыть все) используется для того, чтобы закрыть все окна и очистить поверхность desktop (рабочего стола).

### **КОМАНДА MESSAGE (СООБЩЕНИЕ)**

Команда Window|Message (Окно|Сообщение) используется для того, чтобы открыть окно Message и сделать его активным. В окне Message (Сообщение) отображаются сообщения об ошибках и предупреждающие сообщения, которые можно использовать для справки. Кроме того, можно выбрать одно из этих сообщений, в результате чего в окне редактирования соответствующее место будет выделено световым маркером. Если сообщение относится к файту, который в настоящий момент не загружен в память, то для загрузки этого файла можно нажать клавишу пробела. В этом окне можно также отобразить данные, которые выводятся программой переноса.

### **КОМАНДА OUTPUT (ВЫВОДИМЫЕ ДАННЫЕ)**

Команда Window|Output (Окно|Выводимые данные) используется для того, чтобы открыть окно Output и активизировать его. Она отображает текст любой введенной в операционной системе командной строки, а также текстовую выдачу, которая генерируется вашей программой (графические данные не отображаются).

Команда Output удобна при отладке, поскольку можно одновременно просматривать ваш исходный код, переменные и выводимые данные. Эта возможность особенно полезна в том случае, если в блоке диалога Options|Environment (Параметры|Среда работы) устанавливается режим отображения 43/50 строк и вы осуществляете отладку стандартной программы, которая работает с 25-строчным режимом отображения. В результате можно видеть практически все выводимые программой данные, и в то же самое время иметь достаточное число строк, чтобы просматривать ваш исходный код и значения переменных.

### **Команда Watch (Выражение просмотра)**

Команда Window|Watch (Окно|Выражение просмотра) используется для того, чтобы открыть окно Watch (Выражение просмотра) и сделать его активным. Окно Watch отображает выражения и их изменяющиеся значения, поэтому можно следить за тем, как программа осуществляет вычисление значений ключевых переменных.

Эта команда используется во всплывающих меню Debug|Watches для добавления в это окно или удаления из этого окна выражений просмотра. Для получения подробной информации по этому меню следует обратиться к описанию окна Watch.

### **Команда User Screen (Экранный кадр пользователя)**

Команда Window|User Screen (Окно|Экранный кадр пользователя) используется для просмотра выводимых вашей программой данных в полноэкранном режиме. Если вы предпочитаете видеть выводимые вашей программой данные в окне системы Borland C++, то вместо данной команды следует выбрать команду Window|Output. При выборе данной команды нажатие любой клавиши или кнопки мыши приводит к возврату в интегрированную среду.

### **Команда Register (Регистры)**

Команда Window|Register (Окно|Регистры) используется для того, чтобы открыть и сделать активным окно Register. Окно Register отображает содержимое регистров центрального процессора и используется при отладке встроенных ассемблерных модулей и модулей Турбо ассемблера вашего проекта.

### **Команда Project (Проект)**

Команда Window|Project (Окно|Проект) используется для открытия окна Project, которое позволяет вам просмотреть файлы, используемые вами для создания программы.

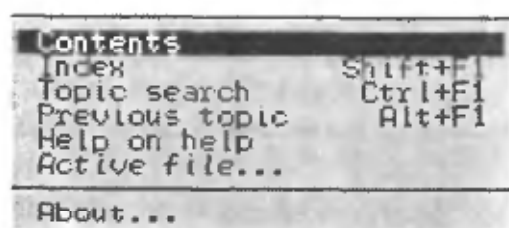
### **Команда ProjectNotes (Примечания)**

Команда Window|ProjectNotes (Окно|Примечания) используется для записи каких-либо подробностей, создания перечней того, что следует сделать, или иных деталей, которые касаются файлов вашего проекта.

### **Команда List All... (Список)**

Команда Window|List... (Окно|Список) используется для получения перечня всех окон, которые были вами открыты. Этот список содержит имена всех файлов, которые открыты в настоящий момент, а также файлов, которые были вами открыты в окне редактирования, но впоследствии были закрыты. Закрытые в последнее время файлы появляются в этом перечне с префиксом "closed" перед именем файла.

## МЕНЮ HELP (ПОДСКАЗКА)



Меню Help (Подсказка) позволяет вам обратиться к оперативной подсказке, которая выдается в специальное окно. Система подсказки содержит информацию практически по всем аспектам интегрированной среды и системы Borland C++.

Для открытия окна Help следует:

- в произвольный момент времени нажать клавишу F1 (в том числе, когда вы находитесь в блоке диалога или когда выбрана какая-либо команда меню);
- нажать комбинацию клавиш Ctrl-F1 для получения подсказки по языку, когда активным является окно редактирования и курсор позиционирован на каком-либо слове;
- подвести указатель мыши к слову Help, когда оно появляется в строке состояния или в блоке диалога, и нажать кнопку мыши.

Если за подсказкой вы обращаетесь из блока диалога или из меню, вы не можете изменять размер окна подсказки или копировать его содержимое в окно текстового буфера. В этом случае нажатие клавиши Tab производит перемещение к средствам управления блоком диалога, а не к очередному ключевому слову.

Экранный кадр подсказки часто содержит ключевые слова (выделенный световым атрибутом текст), который можно выбрать, чтобы получить дополнительную информацию. Нажатие клавиши Tab приводит к перемещению к любому из ключевых слов; после этого для получения дополнительной информации необходимо нажать клавишу Enter (альтернативный способ заключается в том, чтобы переместить курсор к выделенному световым атрибутом слову и нажать клавишу Enter). При работе с мышью для перехода к тексту подсказки, на который ссылается данный элемент, можно подвести указатель мыши к этому слову и дважды подряд нажать кнопку мыши. Кроме того, можно перемещать курсор по окну подсказки к любому слову и нажимать комбинацию клавиш Ctrl-F1 в целях получения информации по данному слову. Если это слово в системе подсказки не обнаружено, то в индексе (тематическом указателе) выполняется инкрементальный поиск и отображается наиболее близкое слово.

Когда окно Help находится в активном состоянии, можно копировать текст из этого окна и вклеивать его в окно редактирования. Эта процедура осуществляется точно так же, как и при работе в окне редактирования.

Для того, чтобы выбрать текст в окне Help, необходимо "отбуксировать" мышью поверх нужного текста или позиционировать курсор в начало требуемого блока, а затем с помощью комбинаций клавиши Shift и клавиш управления курсором маркировать блок.



**Команда CONTENTS (СОДЕРЖАНИЕ)**

Команда Help|Contents (Подсказка|Содержание) открывает окно Help, в котором отображается перечень тем (содержание). Из этого окна можно перейти к любой другой части системы подсказки.

**Команда I INDEX (ТЕМАТИЧЕСКИЙ УКАЗАТЕЛЬ)**

Команда Help|Index (Подсказка|Тематический указатель) открывает блок диалога, в котором отображается полный перечень ключевых слов, по которым может быть получена подсказка (выделенный световым атрибутом в экранных кадрах подсказки текст, который позволяет вам быстро перейти к связанной с данным ключевым словом теме).

**Команда TOPIC SEARCH (ТЕМАТИЧЕСКИЙ ПОИСК)**

Команда Help|Topic Search (Подсказка|Тематический поиск) отображает справку по выбранному элементу синтаксиса языка.

**Команда PREVIOUS TOPIC (ПРЕДШЕСТВУЮЩАЯ ТЕМА)**

Команда Help|Previous Topic (Подсказка|Предшествующая тема) открывает окно Help, в котором отображается тот текст, который вы просматривали в последний раз при обращении к системе подсказки.

**Команда HELP ON HELP (ИНФОРМАЦИЯ ПО СИСТЕМЕ ПОДСКАЗКИ)**

Команда Help|Help on Help (Подсказка|Информация по системе подсказки) открывает экранный кадр, в котором объясняется, как пользоваться системой подсказки в Borland C++. Если вы уже находитесь в системе подсказки, этот кадр может быть вызван нажатием клавиши Enter.

**Команда ACTIVEFILE (АКТИВНЫЙ ФАЙЛ)**

Команда Help|ActiveFile (Подсказка|Активный файл) открывает окно диалога, в котором можно выбрать подсказку по одной из тем.

**Команда ABOUT... (О...)**

Команда Help|About... (Подсказка|О...) открывает окно с информацией о версии системы.

**РЕДАКТИРОВАНИЕ ФАЙЛОВ В СИСТЕМЕ BORLAND C++**

Далее будет представлен набор команд редактирования системы Borland C++. Следует помнить, что данные команды имеют отношение к редактору. Интегрированная среда системы Borland C++ по-прежнему позволяет пользоваться принятыми в продуктах корпорации Borland уже знакомыми вам комбинациями клавиш для перемещения в файле, вставки, копирования и удаления текста, а также для поиска и замены. Однако кроме знакомых средств он предлагает вам совершенно новые меню в строке меню - меню Edit и меню Search. В дополнение к этому система Borland C++

обеспечивается поддержкой мыши при выполнении большинства операций перемещений курсора и команд маркирования блока текста.

Меню Edit содержит команды для выполнения операций отсечения, копирования и вклеивания фрагментов текста в файл, копирования примеров из окна Help в окно редактирования, а также просмотра этих фрагментов в окне Clipboard. Когда вы впервые запускаете систему Borland C++, окно редактирования уже находится в активном состоянии. Для того чтобы открыть другие окна редактирования, необходимо перейти в меню File и выбрать команду Open. Находясь в окне редактирования, можно нажать клавишу F10, чтобы вернуться в главное меню; для возврата в окно редактирования надо нажимать клавишу Esc до тех пор, пока вы не выйдете из системы меню. Если у вас имеется мышь, можно также перевести указатель мыши в произвольное место окна редактирования и нажать кнопку мыши.

В редакторе имеется возможность восстановления прежнего текста. Предлагаемый редактор имеет больше возможностей, чем можно описать в этой главе.

Команды редактора можно подразделить на следующие группы:

- команды перемещения курсора;
- операции вставки и удаления текста;
- операции над блоками;
- различные команды редактирования.

### Команды перемещения курсора

Символ вправо	->
Символ влево	<-
Слово влево	Ctrl <-
Слово вправо	Ctrl ->
Строка вверх	"Стрелка вверх"
Строка вниз	"Стрелка вниз"
Прокрутка вверх на одну строку	Ctrl-W
Прокрутка вниз на одну строку	Ctrl-Z
Страница вверх	PgUp
Страница вниз	PgDn

### Перемещения на большие расстояния

К началу строки	Home
К концу строки	End
К верхнему краю окна	Ctrl Home
К нижнему краю окна	Ctrl End
К началу файла	Ctrl PgUp
К концу файла	Ctrl PgDn
К началу блока	Ctrl-Q B
К концу блока	Ctrl-K K
К последней позиции курсора	Ctrl-Q P

**Команды вставки и перемещения**

Задание/снятие режима вставки	Ins
Удалить символ слева от курсора	Backspace
Удалить символ у курсора	Del
Удалить слово справа	Ctrl-T
Вставить строку	Ctrl-N
Удалить строку	Ctrl-Y
Удалить символы до конца строки	Ctrl-Q Y

**Команды обработки блоков**

Отметить блок	Shift- "Стрелки",
Отметить начало блока	Ctrl-KB
Отметить конец блока	Ctrl-K K
Отметить слово	Ctrl-K T
Скопировать блок	Ctrl-Ins
Вставить блок	Shift-Ins
Вырезать блок	Shift- Del
Удалить блок	Ctrl-Del
Считать блок из файла	Ctrl-K R
Записать блок в файл	Ctrl-K W
Скрыть/отобразить блок	Ctrl-K H
Вывести блок на принтер	Ctrl-K P
Сдвинуть блок вправо	Ctrl-K I
Сдвинуть блок влево	Ctrl-K U

**Другие команды редактирования**

Найти маркер позиции	Ctrl-Q <n>
Перейти к строке меню	F10
Новый файл	File  New
Открыть файл	File  Open
Найти парный символ	Ctrl-Q [Ctrl-Q]
Печатать файл	File  Print
Выход	Alt-X
Повторить последний поиск	Ctrl-L
Восстановить сообщение об ошибке	Ctrl-Q W
Восстановить строку	Edit  Restore Line
Восстановить строку	Ctrl-Q L
Возвратиться к редактору из меню	Esc
Сохранить	F2
Искать	Ctrl-Q F
Искать и заменить	Ctrl-Q A
Задать маркер позиции	Ctrl-K <n>
Табуляционное перемещение	Tab

**Переходы внутри файла**

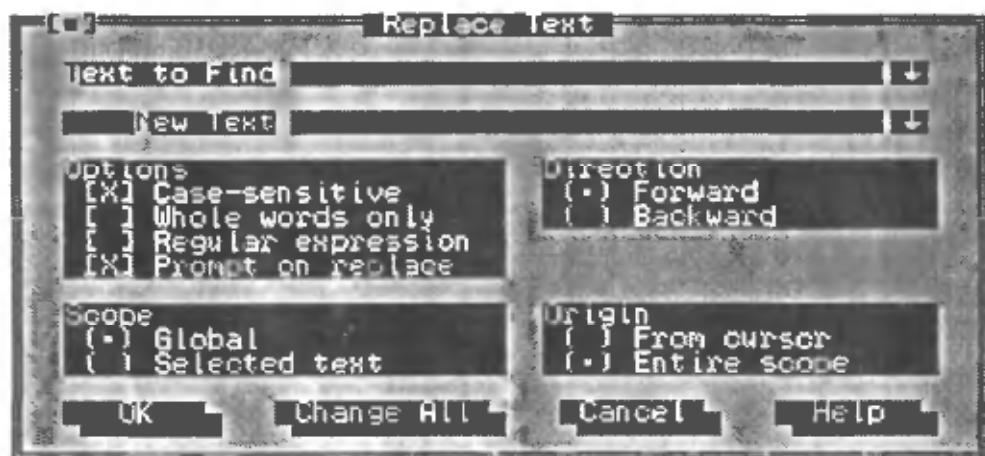
К началу блока	Ctrl-Q B
----------------	----------

К концу блока  
К последней позиции курсора

Ctrl-Q K  
Ctrl-Q P

Команда Ctrl-Q P осуществляет перемещение к той позиции, которую занимал курсор перед выполнением последней команды. Эта команда оказывается особенно полезной после выполнения операции поиска или операции поиска и замены, если вы хотите вернуться туда, где вы находились перед началом поиска.

## ПОИСК И ЗАМЕНА



1. Выберите команду Search/Replace. В результате будет открыт блок диалога Replace Text.
2. Введите строку, которую вы ищете, в блок ввода Text to Find.
3. Нажмите клавишу табуляции или воспользуйтесь мышью, чтобы перейти к блоку ввода New Text. Введите замещающую строку.
4. Установите параметры поиска, как и в блоке диалога Find.
5. Выберите кнопку OK или кнопку Change All, чтобы начать поиск, либо кнопку Cancel, чтобы отменить эту процедуру. Система Borland C++ выполнит указанную операцию. Выбор кнопки Change All приведет к замене каждого найденного экземпляра.
6. Чтобы прекратить выполнение операции, нажмите клавишу Esc в любой момент, когда в процессе поиска наступит пауза.

## ПОИСК ПАРНЫХ СИМВОЛОВ

Данное средство предоставляет вам возможность синтаксического контроля исходного файла, в котором присутствует значительное число функций, заключенных в круглые скобки выражений, вложенных комментариев и масса всяких других конструкций, в которых используются парные символы-ограничители. Фактически ваш файл содержит много парных символов:

{ }	- фигурные скобки;
< >	- угловые скобки;

( )	- круглые скобки;
[ ]	- квадратные скобки;
/* */	- скобки комментариев;
“ ”	- двойные кавычки;
‘ ’	- одинарные кавычки.

Поиск парного символа к конкретному символу в такой конструкции может оказаться непростым делом. Предположим, что у вас имеется сложное выражение с рядом вложенных выражений и вы хотите убедиться в том, что все круглые скобки сбалансированы должным образом. Или предположим, что вы находитесь в начале функции, которая простирается на несколько экранов кадров, и хотите перейти к концу этой функции. При работе в системе Borland C++ в ваших руках находится удобное решение этой проблемы: команды поиска парных символов. Далее приводится последовательность действий, которую нужно выполнить.

1. Поместить курсор на требуемый разделитель (например, на открывающую скобку какой-либо функции, которая простирается на несколько экранов кадров).

2. Для нахождения пары для этого выбранного разделителя необходимо нажать комбинацию клавиш Ctrl-Q[. (В данном случае парный символ должен находиться в конце функции.) Для поиска вперед подходит комбинация клавиш Ctrl-Q ].

3. Редактор немедленно перемещает курсор к тому разделителю, который соответствует выбранному символу. Если он перемещается именно к предполагаемому символу, то можно быть уверенным, что в разделяющем их коде не содержится ни одного непарного разделителя этого типа.

Тот метод, посредством которого редактор осуществляет поиск разделителей комментария (/\* \*/), несколько отличается от других способов поиска. Если для выбранного символа не существует парного символа, редактор не перемещает курсор.

Существование двух команд поиска парных символов связано с тем, что для некоторых разделителей понятие направления явно не существует.

Предположим, например, что вы указываете редактору осуществить поиск открывающей фигурной скобки { или открывающей квадратной скобки [. Редактору известно, что парные символы-разделители не могут быть обнаружены перед тем символом, который был вами выбран, поэтому в поисках парного символа редактор продвигается вперед по тексту. Если вы указали редактору найти парный символ для закрывающей фигурной скобки } или закрывающей круглой скобки ), то редактору известно, что парный символ-разделитель не может быть обнаружен после выбранного символа, поэтому в процессе поиска парного символа он автоматически продвигается назад по тексту.

Разделитель	Являются вложенными	Направление
{ }	Да	Да
( )	“	“
[ ]	“	“
< >	“	“
/* */	“	Да и Нет
" "	Нет	Нет
' '	“	“

Вложенность означает, что при поиске редактором парного символа для "направленного" разделителя редактор учитывает, в какое число уровней разделителей он входит при поиске и из какого числа уровней выходит.

### КОМПИЛЯТОР КОМАНДНОЙ СТРОКИ

Компилятор командной строки системы Borland C++ (программа BCC.EXE) позволяет вам реализовать все возможности компилятора системы Borland C++ из командной строки DOS.

В дополнение к использованию интегрированной среды разработки программ можно компилировать и выполнять свои программы системы Borland C++ с помощью задания командных строк. В то время как интегрированная среда, как правило, представляет идеальное средство для разработки и выполнения ваших программ, иногда вы, возможно, предпочтете использовать средства, которые предоставляются пользователю командной строкой. В некоторых изолированных программах компиляция с помощью командной строки может оказаться единственным способом реализации каких-либо "хитростей" (например, утилита MAKE может использовать пакетные (командные) файлы, а менеджер проектов не обладает такой возможностью).

Программа BCC компилирует исходные файлы, написанные на языках C и C++, и компоует их для образования единого выполняемого файла. Для того чтобы вызвать компилятор командной строки системы Borland C++, введите BCC в ответ на запрос DOS. За этой командой должен следовать набор аргументов командной строки. В число аргументов командной строки входят параметры компилятора и компоновщика и имена файлов. Далее приводится типовой формат командной строки:

```
>bcc [<параметр>[<параметр>...]]<имя-файла>[<имя-файла>...]
```

За исключением двух случаев каждому параметру командной строки предшествует знак дефиса. Каждый параметр должен отделяться от команды BCC, других параметров и последующих имен файлов по меньшей мере одним символом пробела.

В документации можно найти перечень всех параметров компилятора командной строки системы Borland C++.



## О ДРУГИХ КОМПИЛЯТОРАХ ЯЗЫКА C++

В настоящее время появилось большое количество компиляторов и систем программирования на языке C++ для IBM-компьютеров. Фирма Borland после системы Borland C++ 3.1 выпустила компиляторы Borland C++ 4.0 и 4.5. Уже объявлено о компиляторе Borland C++ 5.0. Среди компиляторов других фирм следует отметить компиляторы фирмы Symantec (Symantec C++ 6.0 и 7.0), компиляторы фирмы Microsoft (Visual C++ 1.0, 2.0 и 4.0), компиляторы фирмы Watcom (Watcom C++ 9.5, 10.0). Каждый из компиляторов имеет свои достоинства и свои недостатки. Каждый новый компилятор добавляет новые возможности и в то же время требует больше ресурсов и более мощного компьютера. Поэтому каждый раз следует задуматься о необходимости перехода на новый, более совершенный компилятор. Все вышеуказанные компиляторы поддерживают стандарт ANSI языка C++ и в то же время добавляют свои дополнительные возможности. Компиляторы фирмы Borland обладают дружественной интегрированной средой и являются естественным развитием предыдущих версий, поэтому переход на новые компиляторы той же фирмы проходит менее болезненно.

Следует также отметить, что последние версии компиляторов содержат интегрированную среду, работающую только под управлением Windows, хотя, конечно, позволяют создавать программы и для MS DOS, и для Windows.

## 2 ВВЕДЕНИЕ В ЯЗЫК C

---

### НЕКОТОРЫЕ ОСОБЕННОСТИ ЯЗЫКА C

Версия C, на которой базируется язык C++, отвечает стандарту ANSI. Те возможности языка Borland C++, которые не отвечают стандарту ANSI, в дальнейшем будут отмечаться.

Прежде чем перейти к подробному изучению языка C, рассмотрим несколько простых программ на языке C.

Желательно все приводимые программы набрать и выполнить на компьютере. При компиляции программы выдается два рода сообщений (Messages): сообщения об ошибках (Errors) и предупреждения (Warnings). Предупреждения не препятствуют дальнейшему выполнению программы. Чтобы не загромождать экран лишними (на первом этапе) предупреждениями, следует отключить сообщения о предупреждениях. Для этого нужно выполнить следующую последовательность действий: выберем раздел Options в главном меню, а в спускающемся меню выберем раздел Compiler. В очередном спускающемся меню выберем опцию Messages. Выберем верхний пункт меню Display. В открывшемся диалоговом окне в разделе Display Warnings установим None. Затем нажмем кнопку ОК или клавишу Enter. В дальнейшем мы вернем эту опцию в исходное состояние.

### ОСНОВНЫЕ ПОНЯТИЯ

При написании программ в языке C используются следующие понятия:

- алфавит,
- константы,
- идентификаторы,
- ключевые слова,
- комментарии.

Алфавитом языка называется совокупность символов, используемых в языке.

Очень важно знать и помнить, что язык C различает прописные и строчные буквы. Язык C, как говорят, является чувствительным к регистру (case sensitive). В языке C имена COLOR, Color и color определяют три различных имени переменных. При написании программ будьте внимательны к использованию регистров при написании имен переменных. Удобнее всего принять некоторые соглашения относительно использования прописных и строчных букв в идентификаторах. Например, имена переменных

содержат только строчные буквы (нижний регистр), константы и макросы - прописные буквы (верхний регистр) и т. д.

В именах переменных можно использовать символ подчеркивания. Обычно с символа подчеркивания начинаются имена системных зарезервированных переменных и констант.

В библиотечных функциях также часто используются имена, начинающиеся с этого символа. Это делается в предположении, что пользователи вряд ли будут применять этот символ в качестве первого символа. Старайтесь не использовать имен, начинающихся с символа подчеркивания, и вам удастся избежать возможных конфликтов и пересечений с множеством библиотечных имен.

Идентификаторы в языке программирования используются для обозначения имен переменных, функций и меток, применяемых в программе. Идентификатором может быть произвольная последовательность латинских букв (прописных и строчных), цифр и символа подчеркивания, которая начинается с буквы или символа подчеркивания. В языке C идентификатор может состоять из произвольного числа символов, однако два идентификатора считаются различными, если у них различаются первые 32 символа. В языке C++ это ограничение снято.

В языках C и C++ некоторые идентификаторы употребляются как служебные слова (keywords), которые имеют специальное значение для компилятора. Их употребление строго определено, и эти слова не могут использоваться иначе. Ключевыми словами стандарта ANSI языка C являются:

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Каждый компилятор может увеличивать количество ключевых слов, так как компилятор учитывает дополнительные возможности того типа компьютеров, для которых он создан. Например, компилятор Borland C++ 3.1 добавляет к ключевым словам стандарта языка C дополнительные слова, предназначенные для работы с памятью и регистрами процессоров семейства Intel, а также позволяющих использовать прерывания и фрагменты программ на другом языке. Эти дополнительные ключевые слова приведены ниже:

asm	_asm	__asm	cdecl
_cdecl	__cdecl	_cs	__cs
_ds	__ds	_es	__es
_export	__export	far	_far
__far	__fastcall	__fastcall	huge

<code>_huge</code>	<code>__huge</code>	<code>interrupt</code>	<code>_interrupt</code>
<code>__interrupt</code>	<code>__loadds</code>	<code>__loadds</code>	<code>near</code>
<code>_near</code>	<code>__near</code>	<code>pascal</code>	<code>_pascal</code>
<code>__pascal</code>	<code>__saveregs</code>	<code>__saveregs</code>	<code>_seg</code>
<code>__seg</code>	<code>__ss</code>	<code>__ss</code>	

Язык C++ в дополнение к ключевым словам языка C добавляет еще несколько:

<code>asm</code>	<code>catch</code>	<code>class</code>	<code>friend</code>	<code>inline</code>	<code>new</code>
<code>operator</code>	<code>private</code>	<code>protected</code>	<code>public</code>	<code>template</code>	<code>this</code>
<code>throw</code>	<code>try</code>	<code>virtual</code>			

Часть символов язык C рассматривает как пробельные символы. Это не только символ пробела ' ', но и символы табуляции, символы перевода строки (новой строки), возврата каретки, символ перевода страницы (новой страницы).

Комментарий - это часть программы, которая игнорируется компилятором и служит для удобочитаемости исходного текста программы. В процессе компиляции комментарий заменяется пробелом; следовательно, комментарий может располагаться в любом месте программы, где допустимо использование пробела. Комментарием в языке C является любая последовательность символов, заключенная между парами символов /\* и \*/. В стандарте языка C запрещены вложенные комментарии, хотя во многих реализациях компиляторов, в частности в Borland C++, вложенные комментарии разрешены. В языке C++ появился еще один вид комментариев: так называемый однострочный комментарий. Все символы, располагающиеся за парой символов // и до конца строки, рассматриваются как комментарий. Компилятор языка C, встроенный в систему Borland C++, позволяет использовать комментарий в стиле C++ в программах на языке C.

О константах, которые могут использоваться при написании программы на языке C, мы поговорим позже.

Рассмотрим несколько простейших примеров программ на языке C, а затем вернемся к систематическому изучению языка.

## ДВЕ ПРОСТЫЕ ПРОГРАММЫ

Вызовем систему Borland C++ и введем в режиме редактирования следующую программу:

```
# include <stdio.h>
/* Пример 1. */
main()
{
    int year, month;
    year = 1996;

    printf("Сейчас %d год\n", year);
}
```

Когда вы введете эту программу, нажмите клавишу F10, чтобы перейти к главному меню. Откомпилируйте и выполните программу, выбрав опцию RUN в главном меню и опцию подменю RUN (или нажмите CTRL-F9). Система Borland C++ откомпилирует программу, подключит необходимые библиотечные функции и выполнит ее. Когда начнется процесс компиляции, откроется окно, позволяющее следить за процессом компиляции. Когда компиляция закончится, это окно пропадет и на экране высветится строка "Сейчас 1996 год".

Сразу после этого на экран возвратится окно редактирования системы Borland C++. Если вы хотите снова увидеть экран с результатом работы программы, нажмите клавиши ALT-F5.

Теперь детально рассмотрим нашу первую программу.

Первая строка:

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

Она сообщает компилятору о необходимости подключить файл `stdio.h`. Этот файл содержит информацию, необходимую для правильного выполнения функций библиотеки стандартного ввода/вывода языка C. Язык C предусматривает использование некоторого числа файлов такого типа, которые называются заголовочными файлами (header files). В файле `stdio.h` находится информация о стандартной функции вывода `printf()`, которую мы использовали.

Вторая строка:

```
/* Пример 1. */
```

является комментарием.

При внимательном рассмотрении программы заметим, что между 5-й и 6-й строками находится пустая строка. Пустые строки в языке C не оказывают никакого влияния и могут быть вставлены для удобочитаемости программы.

Строка

```
main()
```

определяет имя функции. Любая программа на языке C состоит из одной или нескольких функций. Выполнение программы начинается с вызова функции `main()`. Поэтому каждая программа на языке C должна содержать функцию `main()`.

Следующая строка,

```
{
```

содержит открывающую фигурную скобку (brace), обозначающую начало тела функции `main()`. Фигурные скобки в языке C всегда используются парами (открывающая и закрывающая). Закрывающую скобку мы еще встретим в нашей программе.

Строка

```
int year, month ;
```

объявляет (`declare`) переменную, называемую `year`, и сообщает компилятору, что эта переменная целая. В языке C все переменные должны быть объявлены прежде, чем они будут использованы. Процесс объявления переменных включает в себя определение имени (идентификатора) переменных (`year`, `month`) и указание типа переменных (`int`).

Строка

```
year = 1996;
```

является оператором присваивания. В этой строке переменной с именем `year` присваивается значение 1996. Заметим, что в языке C используется просто знак равенства в операторе присваивания. Все операторы в языке C заканчиваются символом "точка с запятой".

Строка

```
printf("Сейчас %d год\n", year);
```

является вызовом стандартной функции `printf()`, которая выводит на экран некоторую информацию. Эта строка состоит из двух частей: имени функции `printf()` и двух ее аргументов "Сейчас %d год\n" и `year`, разделенных запятой. В языке C нет встроенных функций ввода/вывода. Но библиотеки языка C и Borland C++ содержат много полезных и удобных функций ввода/вывода. Функция `printf()`, которую мы использовали, является универсальной функцией форматного вывода.

Для вызова функции нужно написать имя функции и в скобках указать необходимые фактические аргументы. Первый аргумент функции `printf()` - это строка в кавычках "Сейчас %d год\n", которую иногда называют управляющей строкой (`control string`). Эта строка может содержать любые символы или спецификации формата, начинающиеся с символа '%'. Обычные символы просто отображаются на экран в том порядке, в котором они следуют.

Спецификация формата, начинающаяся с символа '%', указывает формат, в котором будет выводиться значение переменной `year`, являющейся вторым аргументом функции `printf()`. Спецификация `%d` указывает, что будет выводиться целое число в десятичной записи. Комбинация символов '\n' сообщает функции `printf()` о необходимости перехода на новую строку. Этот символ называется символом новой строки (`newline`).

Последняя строка программы:

```
}
```

содержит закрывающую фигурную скобку. Она обозначает конец функции `main()`.

Если при наборе программы вы не допустили опечаток, то вы получите результат, о котором написано выше. Если, кроме того, вы отключили выдачу предупреждений так, как это описано в начале главы, то никаких сообщений вам не будет выдано.

Попробуем намеренно ввести ошибку в нашу программу. Например, не поставив одну из точек с запятой. Попробуем еще раз выполнить уже неправильную программу. Нажмем комбинацию клавиш `Ctrl-F9`. При ком-



пиляции программы будет обнаружена ошибка, которая будет подсвечена в окне сообщений (message window). Причем в окне сообщений ошибка будет подсвечена более яркой строкой, а в окне редактирования курсор устанавливается в том месте программы, где компилятор системы Borland C++ обнаружил ошибку. На самом деле курсор будет находиться в следующей строке, что бывает всегда, когда компилятор не находит точки с запятой.

Одним из наибольших удобств интегрированной среды является то, что в интерактивном режиме пользователь может обнаружить и исправить ошибку. Для перехода к следующей ошибке нужно набрать комбинацию клавиш ALT-F8.

Нажав комбинацию клавиш ALT-F7, можно перейти к предыдущей ошибке. Нажатием клавиши Enter можно активизировать окно редактирования и исправить выделенную курсором ошибку.

Наличие ошибок (errors) не позволяет выполнить программу. Необходимо исправить найденные ошибки и снова компилировать программу. Однако, даже если в программе нет синтаксических ошибок, некоторые ситуации могут вызвать подозрение у компилятора. Когда Borland C++ встречается с одной из таких ситуаций, то печатается предупреждение (warning). Пользователь должен проанализировать указанную ситуацию и принять соответствующее решение.

Например, была объявлена переменная month, но она не была использована в программе. Предупреждение об этом должно быть сделано системой Borland C++. Однако вспомним, что при установке опций для режима компиляции соответствующий блок был отключен. Наличие предупреждений не является препятствием для выполнения программы.

Рассмотрим второй пример, в котором будет реализовываться ввод данных с клавиатуры. Для этого будет использоваться библиотечная функция scanf(), которая позволяет пользователю вводить информацию с клавиатуры во время выполнения программы.

```
#include <stdio.h>
/* Пример 2.
Вычисление длины окружности */
main()
{
    int radius;
    float length;
    printf("Введите значение радиуса:\n");
    scanf("%d", &radius);
    length = 3.1415 * 2 * radius;

    printf("Радиус - %d \n длина - %f\n",
           radius, length);
}
```

В этой программе по сравнению с предыдущей использовано несколько важных новшеств.

Во-первых, объявлены две переменные двух разных типов: `radius` - типа целое (`int`); `length` - типа с плавающей запятой (`float`), содержащую дробную часть.

Во-вторых, используется функция `scanf()` для ввода с клавиатуры значения радиуса окружности. Первый аргумент функции `scanf()` `"%d"` указывает, что будет вводиться целое десятичное число. Вторым аргументом - имя переменной, которой будет присвоено введенное значение. Символ `&` (амперсанд, *ampersand*) перед именем переменной `radius` необходим для правильной работы функции `scanf()`. Более подробно необходимость использования символа `&` перед именем переменной будет обсуждаться в дальнейшем.

Обратим внимание на то, что в следующей строке целые числа 2 и `radius` умножаются на число с плавающей запятой 3.1415 и результат присваивается переменной типа `float`. В отличие от многих других языков, язык C допускает использование в выражениях переменных разных типов. Для вывода результатов применяется функция `printf()`. При употреблении в первой программе функция `printf()` содержала при вызове 2 аргумента, а во втором примере - 4 аргумента. Спецификатор формата `%f` используется для печати значения переменной `length` типа `float`.

В рассмотренном примере длина окружности вычисляется только для целых радиусов. Можно изменить эту программу так, чтобы она вычисляла длину окружности для любых радиусов и вычисляла, кроме этого, площадь круга.

```
#include <stdio.h>
/* Пример 3.
Вычисление длины окружности и площади круга */
main()
{
    float radius, length, area;
    printf("Введите значение радиуса:\n");
    scanf("%f", &radius);
    length= 3.1415 * 2 * radius;
    area= 3.1415 * radius*radius;
    printf("Радиус=%f, длина окружности=%f, площадь круга=%f\n",
        radius, length, area);
}
```

В этой программе тип переменной `radius` изменен на `float`, соответственно изменены спецификаторы формата ввода и вывода переменной `radius` в функциях `scanf()` и `printf()`.

## Немного о функциях языка C

Принципы программирования на языке C основаны на понятии функции. Функция - это самостоятельная единица программы, созданная для

решения конкретной задачи. Функция в языке C играет ту же роль, что и подпрограммы или процедуры в других языках.

Каждая функция языка C имеет имя и список аргументов. По соглашению, принятому в языке C, при записи имени функции после него ставятся круглые скобки (parentheses). Это соглашение позволяет легко отличать имена переменных от имен функций.

Рассмотрим модельный пример программы, в которой кроме функции `main()` содержится еще три функции.

```
#include <stdio.h>
/* Пример 4. Пример программы с функциями */
main()      /* Главная функция */
{           /* Начало тела функции */
    function1(); /* вызов первой функции */
    function2(); /* вызов второй функции */
    function3(); /* вызов третьей функции */
}           /* Конец тела функции main() */

function1() /* Начало определения первой функции */
{           /* Начало тела первой функции */
    printf("вызвали первую функцию\n");
}           /* Конец тела первой функции */

function2() /* Начало определения второй функции */
{           /* Начало тела второй функции */
    printf("вызвали вторую функцию\n");
}           /* Конец тела второй функции */

function3() /* Начало определения третьей функции */
{           /* Начало тела третьей функции */
    printf("вызвали третью функцию\n");
}           /* Конец тела третьей функции */
```

Здесь 4 функции: `main()`, `function1()`, `function2()`, `function3()`. Эти функции не имеют аргументов. В дальнейшем будет рассказано, как можно создать функции, которые имеют аргументы. Аргументы функции - это величины, которые передаются функции во время ее вызова. Мы уже сталкивались с функциями, имеющими аргументы, при использовании функций `printf()` и `scanf()`. Создадим свою функцию, имеющую аргументы. Пусть это будет функция, которая вычисляет длину окружности и выводит вычисленное значение на экран.

```
#include <stdio.h>
/* Пример 5. Программа, использующая функцию с аргументом:
   вычисление длины окружности */
void length(float radius); /* Объявление функции length(), ее прототип */
```

```
main()
{
    float radius;
    radius=5;
    length(radius); /* вызов функции length() */
    scanf("%f", &radius);
}
void length (float r) /* Описание функции length() */
{
    printf("Длина окружности радиуса %f равна %f\n",
           r, 3.1415*2*r);
}
```

Данная программа состоит из двух функций: функции `main()` и функции `length()`. После комментария с названием программы следует объявление функции `length()`. По правилам, принятым в языке С, каждый идентификатор, в том числе и имя функции, должен быть объявлен до его использования в программе. Заголовок функции, заканчивающийся точкой с запятой называется прототипом функции и является объявлением функции. После фигурной скобки, заканчивающей тело функции `main()`, следует описание функции `length()`.

Описание функции `length()` состоит из имени функции, за которым в круглых скобках следует аргумент функции с указанием его типа (функции языка С могут иметь несколько аргументов). Аргументы, стоящие в заголовке функции, называются формальными параметрами функции. За заголовком функции следует тело функции, заключенное в операторные скобки `{ }`, состоящее из одного оператора. Конкретное значение аргумент получает при вызове функции `length()` в функции `main()`. Аргумент, стоящий в операторе вызова функции, называется фактическим параметром. Следует помнить, что при вызове функции тип фактического параметра должен совпадать с типом формального параметра.

Ключевое слово `void` в заголовке функции говорит о том, что функция не возвращает никакого значения. Не менее важно, чтобы функция могла вернуть какие-либо значения. В языке С функция может возвращать значение в вызывающую программу посредством оператора `return`.

В старой версии языка С прототип функции не должен был объявляться, что создавало неопределенность относительности правильности использования количества параметров функции и их типа. Стандарт ANSI C требует обязательного использования прототипа, если, конечно, описание функции не идет раньше ее вызова. Язык С++ еще более строго требует использования прототипов функции. В новом стандарте функция полностью определяется не только по ее имени, но и по количеству параметров и их типу. При этом при объявлении функции, т. е. при написании прототипа, имена формальных параметров не используются и не рассматриваются компилятором и их можно опустить.

Прототип функции можно использовать так:

```
float func(int n, float f, long double ld);
```

или так:

```
float func(int, float, long double);
```

Объявление функции может встречаться в программе несколько раз. Важно только, чтобы прототипы функций не отличались друг от друга.

Для примера рассмотрим вариант программы, которая печатает длину окружности, используя другой вариант функции `length()`, возвращающей значение.

```
#include <stdio.h>
/* Пример 6. Программа, использующая функцию с аргументом и
   возвращающее значение длины окружности */
float length(float radius): /* Объявление функции length(), ее прототип
                             отличается от предыдущего примера типом
                             возвращаемого значения */

main()
{
    float radius;
    radius=5;
    printf("Длина окружности радиуса %f равна %f\n",
           r, length(radius)); /* вызов функции length() */
    scanf("%f", &radius);
}
float length (float r) /* Описание функции length() */
{
    return 3.1415 * 2 * r;
}
```

В этом примере функция `length()` возвращает значение  $3.1415 * 2 * r$  с использованием оператора `return`.

Основная форма описания функции в языке C имеет следующий вид:

```
<тип возвращаемого значения> <имя функции>(<список параметров>)
{
    //тело функции
}
```

## ДВА ПРОСТЫХ ОПЕРАТОРА: IF И FOR

Продолжая общий обзор языка C, а также для того, чтобы были понятны последующие примеры, рассмотрим два оператора языка C, `if` и `for`,

в случае их простейшего использования. Более подробно об этих операторах будет рассказано в дальнейшем.

Простейшая форма оператора `if` имеет следующий вид:

`if (условие) оператор;`

Условие - это логическое выражение, которое принимает значение либо "истинно", либо "ложно". В языке C "истинно" - это ненулевая величина, "ложно" - нуль.

Оператор

`if (0<1) printf("0 меньше 1");`

выводит на экран сообщение: "0 меньше 1".

Оператор

`if (0>1) printf("0 больше 1");`

не выводит на экран никакого сообщения, так как значение выражения `(0>1)` ложно (0) и оператор, следующий за условием, не выполняется. Таким образом, оператор, следующий за условием, выполняется лишь тогда, когда выражение, стоящее в условии, принимает значение истина.

Заметим, что отношение "равно" в языке C записывается двумя знаками равенства `"=="`.

Оператор

`if (0==1) printf("0 равно 1");`

не выводит на экран никакого сообщения, так как значение выражения `(0==1)` ложно (0).

Следующий важный оператор - это оператор цикла `for`.

Формат оператора `for`:

`for(инициализация; условие; изменение) оператор ;`

Здесь:

- инициализация - используется для установки начального значения параметра цикла;
- условие - выражение, значение которого проверяется при выполнении каждого шага цикла. Цикл выполняется, пока значение этого выражения истинно (ненулевое);
- изменение - выражение, изменяющее параметр цикла.

Например, следующая программа выводит на экран числа от 10 до 1 (обратный счет).

```
#include <stdio.h>
/* Пример 7. */
main ( )
{
    int i;
    for(i=10; i>0; i--) printf("%d\n", i);
}
```



```
printf("Старт!\n");  
}
```

Как видно из примера, параметр цикла *i* сначала устанавливается равным 10. В начале каждого шага цикла проверяется условие *i* > 0. Если условие истинно, то на экран функцией `printf()` будет выведено текущее значение *i*. Два минуса в последней части заголовка цикла указывают на то, что параметр цикла должен быть уменьшен на единицу. Когда значение *i* станет равным нулю, условие принимает значение "нуль" и цикл прекращается. После оператора цикла выполняется следующий за ним оператор, выводящий на экран выражение "Старт!".

Одним из больших достоинств языка C является его структурированность, позволяющая создавать блоки операторов. Каждый блок трактуется как логически связанная последовательность операторов, находящихся между открывающей и закрывающей скобкой. Такую последовательность операторов принято называть составным оператором. Блок (составной оператор) рассматривается как один оператор. Пример блока приведен ниже:

```
if (i>10) {  
    printf("Слишком много");  
    i=i/2;  
}
```

### Точка с запятой, скобки и комментарии

В языке C точка с запятой обозначает конец оператора. Таким образом, каждый отдельный оператор должен заканчиваться точкой с запятой. В языке C блок (составной оператор) - это набор логически связанных операторов, находящихся между открывающей и закрывающей фигурными скобками (операторными скобками). Если вы рассматриваете блок как набор операторов, то за блоком точка с запятой не ставится.

Конец строки в C не является окончанием оператора. Это означает, что нет ограничений на расположение операторов в программе и их можно располагать так, чтобы программу было удобно читать. Нельзя разрывать идентификаторы. Зато в операторах там, где можно поставить пробел, можно и перенести оператор на другую строку.

Два фрагмента программ, представленные ниже, эквивалентны:

а) <code>x = y; y = y + 1;     mul (x, y);</code>	б) <code>x = y; y = y + 1;     mul (x, y );</code>
---	--

### ОПРЕДЕЛЕНИЕ НЕКОТОРЫХ ПОНЯТИЙ

Прежде чем перейти к подробному изучению языка C, определим некоторые понятия.

Исходный текст (source code) - текст программы на языке программирования.

Объектный код (object code) - текст программы на машинном языке, который не может выполняться компьютером. Получается после компиляции исходного текста файла или программы.

Компоновщик (linker) - программа, строящая выполняемый модуль из объектных модулей. Эта программа собирает откомпилированный текст программы и функции из стандартных библиотек языка C в одну выполняемую программу.

Библиотека (library) - набор функций, в том числе из стандартных библиотек, предопределенных переменных и констант, которые могут быть использованы в программе и хранятся в откомпилированном виде.

Время компиляции (compiler time) - период, во время которого происходит компиляция программы. Во время компиляции обнаруживаются синтаксические ошибки.

Время выполнения (run time) - период, во время которого происходит выполнение программы.

Перейдем теперь к более детальному и систематическому описанию языка C.

## ПЕРЕМЕННЫЕ, КОНСТАНТЫ, ОПЕРАЦИИ И ВЫРАЖЕНИЯ

Операторы языка C манипулируют переменными и константами в виде выражений. В языке C имена, которые используются для обозначения переменных, функций, меток и других определенных пользователем объектов, называются идентификаторами (identifiers). Идентификатор может содержать латинские буквы, цифры и символ подчеркивания, и начинаться обязан с буквы или символа подчеркивания. В стандарте ANSI языка C идентификатор определяется своими первыми 32 символами. Как вы помните, в языке C строчные и прописные буквы рассматриваются как разные символы.

Идентификатор не должен совпадать с ключевыми словами.

## БАЗОВЫЕ ТИПЫ ДАННЫХ

В языке C все переменные должны быть объявлены до их использования. В нем определены 5 типов данных, которые можно назвать базовыми:

char	- символьные,
int	- целые,
float	- с плавающей точкой,
double	- с плавающей точкой двойной длины,
void	- пустой, не имеющий значения.

Это основные, или базовые, типы данных.

Типы char и int являются целыми типами и предназначены для хранения целых чисел. Вас не должно смущать название типа char - символьная переменная. Любой символ в компьютере связан с целым числом - кодом

этого символа, например в таблице ASCII. Сам символ нам необходим, когда информация выводится на экран или на принтер или, наоборот, вводится с клавиатуры. Подобные преобразования символа в код и наоборот производятся автоматически. Тип `char` по умолчанию является знаковым типом, однако настройкой опций интегрированной среды можно установить по умолчанию беззнаковый тип `char`. Тип `int` всегда знаковый, так же как и типы `float` и `double`.

Переменные типа `double` и `float` являются числами с плавающей точкой.

Ключевое слово `void` отсутствовало в языке C стандарта Керниган&Ритчи и было привнесено в стандарт ANSI C из языка C++. Нельзя создать переменную типа `void`. Однако введение этого типа оказалось весьма удачным, что вы в дальнейшем увидите.

На основе этих пяти типов строятся дальнейшие типы данных.

Простейшим приемом является использование модификаторов (`modifiers`) типа, которые ставятся перед соответствующим типом.

В стандарте ANSI языка C такими модификаторами являются следующие зарезервированные слова:

<code>signed</code>	- знаковый,
<code>unsigned</code>	- беззнаковый,
<code>long</code>	- длинный,
<code>short</code>	- короткий.

Модификаторы `signed` и `unsigned` могут применяться к типам `char` и `int`. Модификаторы `short` и `long` могут применяться к типу `int`. Модификатор `long` может применяться также к типу `double`. Модификаторы `signed` и `unsigned` могут комбинироваться с модификаторами `short` и `long` в применении к типу `int`.

В следующей таблице приведены все возможные типы с различными комбинациями модификаторов, использующиеся в языке C. Размер в байтах и интервал изменения могут варьироваться в зависимости от компилятора, процессора и операционной системы (среды).

<i>Тип</i>	<i>Размер в байтах (битах)</i>	<i>Интервал изменения</i>	
<code>char</code>	1 (8)	от -128	до 127
<code>unsigned char</code>	1 (8)	от 0	до 255
<code>signed char</code>	1 (8)	от -128	до 127
<code>int</code>	2 (16)	от -32768	до 32767
<code>unsigned int</code>	2 (16)	от 0	до 65535
<code>signed int</code>	2 (16)	от -32768	до 32767
<code>short int</code>	2 (16)	от -32768	до 32767
<code>unsigned short int</code>	2 (16)	от 0	до 65535
<code>signed short int</code>	2 (16)	от -32768	до 32767
<code>long int</code>	4 (32)	от -2147483648	до 2147483647
<code>signed long int</code>	4 (32)	от -2147483648	до 2147483647

unsigned long int	4 (32)	от 0	до 4294967295
float	4 (32)	от 3.4E-38	до 3.4E+38
double	8 (64)	от 1.7E-308	до 1.7E+308
long double	10 (80)	от 3.4E-4932	до 3.4E+4932

Запись 3.4E-38 соответствует числу  $3.4 * 10^{-38}$ , это так называемый научный формат (scientific format) записи числа с плавающей запятой.

Различие между целыми числами со знаком и целыми числами без знака состоит в том, как интерпретируется старший бит целого числа. Старший бит для целого числа со знаком определяет знак числа. Если старший бит равен нулю -, число положительное, если же старший бит равен единице, то число отрицательное. Целое число +3 типа int будет храниться в памяти компьютера в виде

0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Если объявлено отрицательное целое число, то компилятор генерирует так называемый обратный код. Чтобы получить число -3 надо поменять значения всех битов на обратные, т. е. 0 заменить на 1, 1 заменить на 0 и прибавить к младшему биту 1. Число -3 в двоичной записи в обратном коде будет иметь вид:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

При внимательном рассмотрении этой таблицы можно заметить, что типы int, short int, signed int и signed short int имеют одни и те же пределы изменения. Эти типы, хотя и имеют разные названия, являются совершенно одинаковыми в данной реализации компилятора языка. Типы int и signed int всегда обозначают один и тот же тип, так как тип int всегда знаковый и без модификатора signed. Модификатор short в данной реализации не оказывает никакого воздействия на тип, поэтому в программах, написанных для компиляторов фирмы Borland, вы не встретите это ключевое слова.

В старом стандарте Керниган&Ритчи тип int являлся настолько важным, что его можно было иногда опускать. В частности, в типах signed int, unsigned int, long int, unsigned long int ключевое слово int можно опустить и писать просто signed, unsigned, long, unsigned long.

Для того чтобы понять различие в интерпретации компилятором целых чисел со знаком и целых чисел без знака, рассмотрим следующий пример:

```
#include <stdio.h>
/* Пример 8. */
main()
{
    int i;
    unsigned int j;
    j = 60000;
    i = j;
    printf("i = %d j = %u\n", i, j);
}
```

Результатом работы программы будет строка

```
i = -5536 j = 60000
```

Спецификация `%d` сообщает функции `printf()`, что используется целое число с знаком; спецификация `%i` - другая спецификация формата, которая соответствует беззнаковому целому.

## ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ

Основная форма объявления переменных имеет вид

```
тип <список_переменных> ;
```

Здесь `тип` должен быть одним из существующих в C типов переменных, а `<список_переменных>` может состоять из одной или нескольких переменных, разделенных запятыми. При объявлении переменных компилятор выделяет место в памяти компьютера, необходимое для размещения переменной данного типа. Примеры объявлений переменных:

```
int x, y, z;  
float radius;  
unsigned char ch;  
long double integral;
```

Когда переменная объявляется в программе, очень большое значение имеет то, в каком месте программы она объявляется. Правило, которое определяет, где переменная может быть использована, зависят от того, где переменная была объявлена, и называется правилом видимости (`scope rules`).

В языке C могут быть три места, где переменная может быть объявлена. Во-первых, вне каких-либо функций, в том числе и `main()`. Такая переменная называется глобальной (`global`) и может использоваться в любом месте программы (за исключением глобальных статических переменных, о которых мы поговорим чуть позже). Во-вторых, переменная может быть объявлена внутри блока, в том числе внутри тела функции. Объявленная таким образом переменная называется локальной (`local`) и может использоваться только внутри этого блока. Такая переменная неизвестна вне этого блока. Наконец, переменная может быть объявлена как формальный параметр функции. Кроме специального назначения этой переменной для передачи информации в эту функцию и места ее объявления переменная может рассматриваться как локальная переменная для данной функции.

Рассмотрим пример объявления переменных в разных местах программы:

```
#include <conio.h>  
#include <stdio.h>  
/* Пример 9. */  
/* пример объявления переменных */  
  
char ch;          /* глобальная переменная ch */
```

```
main()
{
    int n;      /* локальная переменная n */
    printf("Введите символ:");
    ch=getche(); /* использование глобальной переменной */
    printf("Введите количество символов в строке:");
    scanf("%d", &n);
    print_str(n);
}

print_str(int m) /* формальный параметр m */
{
    int j;      /* локальная переменная j */
    for (j=0; j<m; j++)
        printf("%c\n", ch); /* использование глобальной переменной ch */
}
```

Эта программа иллюстрирует использование глобальных и локальных переменных и формальных параметров.

Очень важно запомнить следующее:

- две глобальные переменные не могут иметь одинаковые имена;
- локальная переменная одной функции может иметь такое же имя, как локальная переменная другой функции (или формальный параметр другой функции);
- две локальные переменные в одном блоке не могут иметь одинаковые имена, в том числе формальный параметр функции не должен совпадать с локальным параметром, объявленным в функции.

## **Константы в языке C**

В языке C константы представляют фиксированную величину, которая не может быть изменена в программе. Константы могут быть любого базового типа данных. Примеры констант:

Тип данных	Константа
char	'a', '\n', '9'
int	1, 123, -346
unsigned int	60000
long int	75000, -27, 5L
short int	10, 12, -128
float	123.23, 4.34E-3, 4E+5
double	123.23, 12312311, -0.987

К какому типу относится константа 13 - к типу char, int, unsigned или к другому? Для языка C это почти не играет никакой роли. В то же время для языка C++ с его жесткой проверкой типов параметров функций это может сыграть очень большую роль.

Правила определения типа констант таковы:



Целая константа (т. е. константа не имеющая десятичной точки или порядка) относится к типу `int`, если эта константа входит в интервал значений типа `int`.

Если эта константа не входит в интервал значений типа `int`, например 37000, то она считается константой типа `unsigned`. Если же константа не входит в интервал изменения `unsigned`, она считается константой типа `long`.

Константа с десятичной точкой считается константой типа `double`, если она помещается в соответствующий интервал изменения.

В языке C имеется механизм явного задания типов констант с помощью суффиксов. В качестве суффиксов целочисленных констант могут использоваться буквы `u`, `l`, `h`, `U`, `L`, `H`. Для чисел с плавающей запятой - `l`, `L`, `f` и `F`.

Например:

12h 34H	short int
23L -237l	long int
89lu 89 Lu 89ul 7UL	unsigned long
45uh	unsigned short
23.4f 67.7E-24F	float
1.39l 12.0L 2e+10	long double

Так как в программировании важную роль играют восьмеричные и шестнадцатеричные системы счисления, важно уметь использовать восьмеричные (`octal`) и шестнадцатеричные (`hexadecimal`) константы. Для того чтобы отличать шестнадцатеричные константы, перед ними ставится пара символов `0x`. Восьмеричные константы всегда начинаются с нуля. Шестнадцатеричные и восьмеричные константы могут быть только беззнаковыми.

Например:

<i>Шестнадцатеричные константы</i>	<i>Восьмеричные константы</i>
0xFFFF	01
0x10	055
0x1F1A	07777

В качестве цифр шестнадцатеричной константы нам надо использовать 16 символов - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

В дальнейшем нам понадобится двоичная запись шестнадцатеричных чисел, поэтому мы приведем двоичную запись шестнадцатеричных цифр, из которых легко сложить шестнадцатеричные числа.

Десятичное число	Шестнадцатеричное число	Двоичная запись числа	Восьмеричная константа	Шестнадцатеричная константа
0	0	0000	0	0x0
1	1	0001	01	0x1
2	2	0010	02	0x2
3	3	0011	03	0x3
4	4	0100	04	0x4

5	5	0101	05	0x5
6	6	0110	06	0x6
7	7	0111	07	0x7
8	8	1000	10	0x8
9	9	1001	11	0x9
10	A	1010	12	0xA
11	B	1011	13	0xB
12	C	1100	14	0xC
13	D	1101	15	0xD
14	E	1110	16	0xE
15	F	1111	17	0xF

Для представления в двоичном виде шестнадцатеричного числа надо просто заменить двоичной записью каждую цифру этого числа. Например, число 0xAB01 представимо в виде

```
A  B  0  1
1010 1011 0000 0001
```

Пробелы, конечно, надо опустить: 1010101100000001.

Строковые константы (strings) также играют в языке C важную роль. Строковая константа или просто строка представляет собой набор символов, заключенный в двойные кавычки. Например, строковая константа "Это строка". Особенностью представления строковых констант в языке C является то, что в памяти компьютера отводится на 1 байт больше, чем требуется для размещения всех символов строки. Этот последний байт заполняется нулевым значением, т. е. байтом в двоичной записи которого одни нули. Этот символ так и называется - нулевой байт и имеет специальное обозначение '\0'. В английском языке слово *строка* имеет два перевода - "line" и "string". Под термином "line" понимается строка в текстовом редакторе; под термином "string" - строковая константа. Переводчики второго издания книги Керниган, Ритчи предложили ввести термин "стринг", что не лишено определенного смысла.

Нельзя путать строковые константы с символьными константами. Так "a" - это строковая константа, содержащая одну букву, в то время как 'a' - символьная константа, или просто символ. Отличие "a" от 'a' в том, что строка "a" содержит еще один символ '\0' в конце строки; "a" занимает в памяти 2 байта, в то время как 'a' - только 1 байт.

В языке C есть символьные константы, которые не соответствуют никакому из печатных символов. Так, в коде ASCII символы с номерами от нуля до 31 являются управляющими символами, которые нельзя ввести с клавиатуры. Для использования таких символов в языке C вводятся так называемые управляющие константы (backslash character constants). Мы с ними уже встречались выше в функции printf(). Фрагменты программы а) и б) эквивалентны по своему действию:

```
а) ch = '\n';      б) printf("\n");
   printf("%c", ch);
```

и вызывают перевод строки. Управляющие символы представлены в следующей таблице:

<i>Управляющий символ</i>	<i>Значение</i>
<code>\b</code>	BS, забой
<code>\f</code>	Новая страница, перевод страницы
<code>\n</code>	Новая строка, перевод строки
<code>\r</code>	Возврат каретки
<code>\t</code>	Горизонтальная табуляция
<code>\v</code>	Вертикальная табуляция
<code>\"</code>	Двойная кавычка
<code>\'</code>	Апостроф
<code>\/</code>	Обратная косая черта
<code>\0</code>	Нулевой символ, нулевой байт
<code>\a</code>	Сигнал
<code>\N</code>	Восьмеричная константа
<code>\xN</code>	Шестнадцатеричная константа
<code>\?</code>	Знак вопроса

Если за символом обратной косой черты следует символ не из этой таблицы, то эта пара воспринимается просто как соответствующий символ.

Другой способ представления таких констант, и печатных символов в том числе, состоит в том, что указывается после обратной косой черты номер в кодировке ASCII в восьмеричной или шестнадцатеричной системе счисления. Операторы

```
printf("\a");
printf("\07");
printf("\7");
printf("\0x7");
```

вызовут один и тот же эффект - машина издаст звуковой сигнал.

## СИМВОЛЬНЫЕ ПЕРЕМЕННЫЕ И СТРОКИ

Рассмотрим более подробно тип `char`. Символьная переменная - это величина размером в 1 байт, которая используется для представления литер и целых чисел в диапазоне от 0 до 255 или от -128 до 127, в зависимости от того, знаковая это переменная или беззнаковая. Символьные константы заключаются в одинарные кавычки. Примеры символьных констант: `'a'`, `'+'`, `'1'`, `'*'`. Ниже приводится программа, иллюстрирующая использование символьных переменных и констант.

```
# include <stdio.h>
/* Пример 7 */
main()
{
    char ch;
    ch='c';
```

```
printf("%c", ch);
ch = '+';
printf("%c%c", ch, ch);
}
```

Обратим внимание на то, что появилась новая спецификация формата в функции `printf()` - `%c`. В этом формате печатается символ. Этот же формат используется и в функции `scanf()` для ввода символа с клавиатуры. Но в языке C в стандартной библиотеке ввода/вывода есть специальная функция `getche()`. Функция `getche()` ожидает, пока не будет нажата какая-либо клавиша на клавиатуре, и затем вводит код этой клавиши.

Рассмотрим пример.

```
#include <stdio.h>
#include <conio.h>
/* Пример 8 */
main()
{
    char ch;
    printf("Нажмите какую-либо клавишу");
    ch = getche(); /* вводит один символ с клавиатуры */
    if (ch == 'a') printf("вы нажали клавишу a\n");
    printf("вы нажали клавишу %c", ch);
}
```

В языке C строка - это массив символов, заканчивающийся нулевым байтом. В языке C нет стандартного типа строка и строка объявляется как массив символов, но для работы с массивом символов как со строкой имеется набор библиотечных функций.

Рассмотрим несколько основных принципов работы с массивами данных на примере одномерных массивов. Одномерный массив - это упорядоченная последовательность данных одного типа. В программе массив объявляется следующим образом:

```
char str[80];
```

В этом описании `char` - тип элементов массива, `str` - имя массива, за которым в квадратных скобках указывается размер массива 80.

Для обращения к отдельному элементу массива нужно указать после имени массива в квадратных скобках номер элемента, например 10. В языке C все элементы массива нумеруются начиная с нуля, т. е. `str[0]` - 1-й элемент массива, `str[1]` - 2-й элемент массива, `str[79]` - 80-й элемент массива.

Следует помнить, что в C строка - это массив символов, заканчивающийся нулевым байтом, и поэтому при объявлении массива, с которым вы собираетесь работать как со строкой, следует зарезервировать место под нулевой байт. В языке C нулевой байт определяется символьной константой `'\0'`.

Например, если слово `hello` - это символьная строка, то под нее нужно зарезервировать массив из шести символов: 5 символов для букв и 6-й символ для нулевого байта:

H	E	L	L	O	'\0'
---	---	---	---	---	------

Для чтения строки с клавиатуры нужно создать символьный массив и затем использовать библиотечную функцию `gets()`. В качестве аргумента функции `gets()` используется имя массива, куда вводится строка. Функция `gets()` читает символы с клавиатуры до тех пор, пока не будет нажата клавиша `Enter`. Нажатие клавиши `Enter` устанавливает в конец строки нулевой байт.

Следующий пример иллюстрирует ввод строки с клавиатуры.

```
# include <stdio.h>
/* Пример 9 */
main()
{
    char str[80];
    printf("Введите Ваше имя.");
    gets(str);
    printf("Я знаю Ваше имя, Ваше имя %s", str);
}
```

В данном примере мы познакомились еще с одной спецификацией формата - `%s`. Она предназначена для ввода/вывода строк.

## ИНИЦИАЛИЗАЦИЯ ПЕРЕМЕННЫХ

После того как переменная объявлена, ей рано или поздно будет присвоено значение. Язык C предоставляет пользователю возможность присвоить значение переменной одновременно с процессом объявления переменной. Основная форма инициализации переменной имеет вид

тип имя\_переменной = константное выражение;

Например:

```
int pr=24;
char c='c', ch='0';
```

Объявление переменной приводит к выделению памяти в размере, необходимом для размещения переменной данного типа. При этом память, которая выделяется, никак не очищается. Инициализация переменной приводит к тому, что одновременно с выделением памяти в эту память заносится значение инициализации. Глобальные и статические переменные всегда инициализируются либо нулем, либо значением инициализатора.

Глобальные переменные инициализируются только один раз в начале работы программы. Локальные переменные инициализируются при каждом вызове функции. Локальные переменные остаются неопределенными до первого присвоения им значения.

В стандарте ANSI C инициализировать можно только константным выражением (статическая инициализация). В реализации языка Borland C и в языке C++ инициализировать можно не только константой, но и выраже-

нием с использованием значений переменных, которые были ранее определены (динамическая инициализация).

В языках C и C++ имеется гораздо больше возможностей инициализации, чем мы сейчас указали. Об этих возможностях мы расскажем в дальнейшем.

## ВЫРАЖЕНИЯ

Выражение в языке C - это некоторая допустимая комбинация переменных, констант и операций.

Каждое выражение в языке C принимает какое-либо значение.

В выражениях языка C допустимо смешение переменных разного типа. Правила языка C, которые используются для автоматического приведения типов при вычислении арифметического выражения, следующие:

1. Все переменные типа char и short int преобразуются в int, все переменные типа float преобразуются в double.

2. Для любой пары операндов: если один из операндов long double, то и другой преобразуется в long double; если один из операндов double, то и другой преобразуется в double; если один из операндов long, то и другой преобразуется в long; если один из операндов unsigned, то и другой преобразуется в unsigned.

3. В операторе присваивания конечный результат приводится к типу переменной в левой части оператора присваивания, при этом тип может как повышаться, так и понижаться.

Пример преобразований типов переменных при вычислении выражений. Пусть определены переменные следующих типов:

char	ch;
int	i;
float	f;
double	d;
long double	r;

При вычислении выражения произойдут следующие автоматические преобразования типов:

```

r = ch * 2 + (i - 0.5) + (f + d) - 7
char int  int  float  float  double int
|   |   |   |   |   |   |
int  int float float double double
 \ / \ / \ / \ /
  int  float  double double
    |   |   \ /
  float float  double
    \ / \ /
    float
    |
   double
    \
long double <-- long double
    
```



Тип результата вычисления выражения можно изменить, используя конструкцию "приведение" (casts), имеющую следующий вид:

(тип) выражение

Здесь "тип" - один из стандартных типов данных языка C.

Например, если вы хотите, чтобы результат деления переменной `x` типа `int` на 2 был типа `float`, напишите

`(float) x/2`

Значение же выражения

`(float) (x/2)`

будет другим. Почему? Постарайтесь сами ответить на этот вопрос.

Пробелы и круглые скобки в выражениях можно расставлять так, как вы считаете нужным. Это делается для удобства чтения программы на языке C. При компиляции лишние пробелы просто игнорируются.

## ФУНКЦИИ `PRINTF()` И `SCANF()`

Функции `printf()` и `scanf()` осуществляют форматированный ввод и вывод на консоль. Форматированный ввод и вывод означает, что функции могут читать и выводить данные в разном формате, которым вы можете управлять.

Функция `printf()` имеет прототип в файле `STDIO.H`

`int printf(char *управляющая_строка, ...);`

Управляющая строка содержит два типа информации: символы, которые непосредственно выводятся на экран, и команды формата (спецификаторы формата), определяющие, как выводить аргументы. Команда формата начинается с символа `%` за которым следует код формата. Команды формата следующие:

- `%c` - символ,
- `%d` - целое десятичное число,
- `%i` - целое десятичное число,
- `%e` - десятичное число в виде `x.xx e+xx`,
- `%E` - десятичное число в виде `x.xx E+xx`,
- `%f` - десятичное число с плавающей запятой `xx.xxxx`,
- `%F` - десятичное число с плавающей запятой `xx.xxxx`,
- `%g` - `%f` или `%e`, что короче,
- `%G` - `%F` или `%E`, что короче,
- `%o` - восьмеричное число,
- `%s` - строка символов,
- `%u` - беззнаковое десятичное число,
- `%x` - шестнадцатеричное число `5a5f`,
- `%X` - шестнадцатеричное число `5A7F`,
- `%%` - символ `%`,
- `%p` - указатель,
- `%n` - указатель.

Кроме того, к командам формата могут быть применены модификаторы `l` и `h`, например:

`%ld` - печать `long int`,  
`%hu` - печать `short unsigned`,  
`%Lf` - печать `long double`.

Между знаком `%` и форматом команды может стоять целое число. Оно указывает на наименьшее поле, отводимое для печати. Если строка или число больше этого поля, то строка или число печатается полностью, игнорируя ширину поля. Нуль, поставленный перед целым числом, указывает на необходимость заполнить неиспользованные места поля нулями. Вывод

```
printf("%05d", 15);
```

даст результат 00015.

Чтобы указать число десятичных знаков после целого числа, ставится точка и целое число, указывающее на количество десятичных знаков. Когда такой формат применяется к целому числу или к строке, то число, стоящее после точки, указывает на максимальную ширину поля выдачи.

Выравнивание выдачи производится по правому краю поля. Если мы хотим выравнивать по левому знаку поля, то сразу за знаком `%` следует поставить знак минуса. В прототипе функции многоточием обозначен список аргументов - переменных или констант, которые следуют через запятую и подлежат выдаче в соответствующем формате, следующем по порядку.

`scanf()` - основная функция ввода с консоли. Она предназначена для ввода данных любого встроенного типа и автоматически преобразует введенное число в заданный формат. Прототип из файла `STDIO.H` имеет вид

```
int scanf(char *управляющая_строка, ...);
```

Управляющая строка содержит три вида символов: спецификаторы формата, пробелы и другие символы. Команды или спецификаторы формата начинаются с символа `%`. Они перечислены ниже:

`%c` - чтение символа,  
`%d` - чтение десятичного целого,  
`%i` - чтение десятичного целого,  
`%e` - чтение числа типа `float`,  
`%h` - чтение `short int`,  
`%o` - чтение восьмеричного числа,  
`%s` - чтение строки,  
`%x` - чтение шестнадцатеричного числа,  
`%p` - чтение указателя,  
`%n` - чтение указателя в увеличенном формате.

Символ пробела в управляющей строке дает команду пропустить один или более пробелов в потоке ввода. Кроме пробела может восприниматься символ табуляции или новой строки. Ненулевой символ указывает на чтение и отбрасывание (`discard`) этого символа. Все переменные, которые мы

вводим, должны указываться с помощью адресов, как и положено в функциях языка C.

Строка будет читаться как массив символов, и поэтому имя массива без индексов указывает адрес первого элемента.

Разделителями между двумя вводимыми числами являются символы пробела, табуляции или новой строки. Знак \* после % и перед кодом формата дает команду прочесть данные указанного типа, но не присваивать это значение. Так,

```
scanf("%d%*c%d", &i, &j);
```

при вводе 50+20 присвоит переменной i значение 50, переменной j - значение 20, а символ + будет прочитан и проигнорирован.

В команде формата может быть указана наибольшая ширина поля, которая подлежит считыванию. К примеру,

```
scanf("%5s", str);
```

указывает необходимость прочесть из потока ввода первые 5 символов. При вводе 123456789 массив str будет содержать только 12345, остальные символы будут проигнорированы. Разделители: пробел, символ табуляции и символ новой строки - при вводе символа воспринимаются, как и все другие символы.

Если в управляющей строке встречаются какие-либо другие символы, то они предназначаются для того, чтобы определить и пропустить соответствующий символ. Поток символов 5plus10 оператором

```
scanf("%dplus%d", &i, &j);
```

присвоит переменной i значение 5, переменной j - значение 10, а символы plus пропустит, так как они встретились в управляющей строке. К недостаткам, правда преодолимым, функции scanf() относится невозможность выдачи приглашения ко вводу т. е. приглашение должно быть выдано до обращения к функции scanf().

Одной из мощных особенностей функции scanf() является возможность задания множества поиска (scanset). Множество поиска определяет набор символов, с которыми будут сравниваться читаемые функцией scanf() символы. Функция scanf() читает символы до тех пор, пока они встречаются в множестве поиска. Как только символ, который введен, не встретился в множестве поиска, функция scanf() переходит к следующему спецификатору формата. Множество поиска определяется списком символов, заключенных в квадратные скобки. Перед открывающей скобкой ставится знак %. Чтобы увидеть, как используется эта возможность, рассмотрим пример.

```
#include <stdio.h>
/* Пример 9 */
main (void)
{
    char s[10], t[10];
    scanf ("%*[0123456789]%s", s, t);
```

```
printf("\n%s %s", s, t);
}
```

Введем следующий набор символов:

```
"123abc456"
```

На экране программа выдаст

```
123 abc456
```

Так как а не входит в множество поиска (оно состоит только из цифр), то ввод по первому спецификатору формата прерывается и начинается ввод по второму спецификатору формата.

При задании множества поиска можно также использовать символ “дефис” для задания промежутков, а также максимальную ширину поля ввода

```
scanf("%10[A-Z1-5]%", s);
```

Можно также определить символы, которые не входят в множество поиска. Перед первым из этих символов ставится знак ^. И множество символов различает, естественно, строчные и прописные буквы.

## ОПЕРАЦИИ ЯЗЫКА C

Язык C богат на операции. Знак операции - это символ или комбинация символов, которые сообщают компилятору о необходимости произвести определенные арифметические, логические или другие действия. Полный список знаков операций языка C приведен ниже. Сразу оговоримся, что мы здесь не объясним действия всех этих операций. Итак, знаки операций языка C:

[]	()	.	->	++	--
&	*	+	-	~	!
sizeof	/	%	<<	>>	
<	>	<=	>=	==	!=
^		&&		?:	=
*=	/=	%=	+=	-=	<<=
>>=	&=	^=	=	,	

К операциям языка C относят также знаки операций препроцессора

```
# ##
```

Язык C++ добавит к этим знакам операций еще пять:

```
*, ->, ::, new, delete.
```

Для того чтобы снять терминологические неоднозначности, отметим, что английский термин “operator” должен переводиться как “операция”, хотя часто встречается перевод “оператор”.

Русскому термину “оператор языка” обычно соответствует английский термин “statement”. Хотя перевод “оператор” в смысле “человек, обслуживающий ЭВМ”, также правилен в соответствующем контексте.

Для каждой операции языка C определено количество операндов:

- один операнд - унарная операция, например унарный минус  $-x$ , изменяющая знак;
- два операнда - бинарная операция, например операция сложения  $x + y$  или вычитания  $x - y$ ;
- три операнда - операция условие  $?:$ , такая операция только одна. О ней мы расскажем немного позже.

Каждая операция может иметь только определенные типы операндов. Например, операция побитового сдвига определена только для целочисленных операндов.

Каждая бинарная операция также имеет определенный порядок выполнения: слева направо или справа налево. Например, операция сложения выполняется слева направо, а операция присваивания выполняется справа налево.

Наконец, каждая операция имеет свой приоритет. Так, операция сложения имеет более низкий приоритет перед операцией умножения. В то же время операция сложения имеет более высокий приоритет перед операцией присваивания. Ниже в таблице мы укажем приоритет каждой операции, точнее, расположим их в порядке приоритета. В этой же таблице мы укажем порядок выполнения (слева направо или справа налево). Как правило, унарные операции имеют более высокий приоритет, чем бинарные.

Каждое выражение, как мы уже говорили, имеет определенное значение, а значит, и тип этого выражения.

Сначала мы обсудим арифметические, логические операции и операции отношения. К другим операциям мы вернемся позже.

### **АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ**

К арифметическим операциям языка C относятся:

- вычитание и унарный минус;
- + сложение;
- \* умножение;
- / деление;
- % деление по модулю;
- ++ увеличение на единицу (increment);
- уменьшение на единицу (decrement).

Операции сложения, вычитания, умножения и деления действуют так же, как и в большинстве других алгоритмических языков. Они могут применяться ко всем встроенным типам данных. Операции выполняются слева направо, т. е. сначала вычисляется выражение левого операнда, затем выражение, стоящее справа от знака операции. Если операнды имеют один тип, то результат арифметической операции имеет тот же тип. Поэтому, когда операция деления  $/$  применяется к целым переменным или символьным переменным, остаток отбрасывается. Так,  $11/3$  будет равно 3, а выражение  $1/2$  будет равно нулю.

Операция деления по модулю % дает остаток от целочисленного деления. Операция % может применяться только к целочисленным переменным. В следующем примере вычисляется целая часть и остаток от деления двух целых чисел.

```
#include <stdio.h>
/* Пример 10. */
main()
{
    int x, y;
    printf("Введите делимое и делитель:");
    scanf("%d%d", &x, &y);
    printf("\nЦелая часть %d\n", x/y);
    printf("Остаток от деления %d\n", x%y);
}
```

Язык C предоставляет пользователю еще две очень полезные операции, специфичные именно для языка C. Это унарные операции ++ и --. Операция ++ прибавляет единицу к операнду, операция -- вычитает единицу из операнда. Обе операции могут следовать перед операндом или после операнда (префиксная и постфиксная формы). Три написанные ниже оператора дают один и тот же результат, но имеют различие при использовании в выражениях:

`x = x + 1; ++x; x++.`

Простая программа позволит понять это различие.

```
#include <stdio.h>
/* Пример 11. */
main()
{
    int x=5;
    int y=60;
    x++;
    ++y;
    printf("x=%d y=%d\n", x, y);
    printf("x=%d y=%d\n", x++, ++y);
}
```

Результатом работы этой программы будет следующее:

`x= 6, y= 61;`  
`x= 6, y= 62.`

Обратите внимание на то, что напечатанное значение x не изменилось при втором обращении к функции printf(), а значение y увеличилось на единицу. На самом деле значение переменной x также увеличилось на единицу, но уже после выхода из функции printf(). Различие в использовании префиксной ++x и постфиксной x++ форм состоит в следующем:

x++ - значение переменной x сначала используется в выражении, и лишь затем переменная увеличивается на единицу;



`++x` - переменная `x` сначала увеличивается на единицу, а затем ее значение используется в выражении.

Старшинство арифметических операций следующее:

`++`, `--`  
 - (унарный минус)  
`*`, `/`, `%`  
`+`, `-`

Операции, одинаковые по старшинству, выполняются в порядке слева направо. Конечно же, для того чтобы изменить порядок операций, могут использоваться круглые скобки.

#### ОПЕРАЦИИ ОТНОШЕНИЯ И ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Операции отношения используются для сравнения. Полный список операций отношения в языке C следующий:

`<` меньше,  
`<=` меньше или равно,  
`>` больше,  
`>=` больше или равно,  
`==` равно,  
`!=` не равно.

В языке C имеется также три логические операции:

`&&` и (AND),  
`||` или (OR),  
`!` не (NOT).

Операции отношения используются в условных выражениях, или, короче, условиях. Примеры условных выражений:

`a<0`, `101>=105`, `'a'=='A'`, `'a'!='A'`

Каждое условное выражение проверяется: истинно оно или ложно. Точнее следует сказать, что каждое условное выражение принимает значение "истинно" ("true") или "ложно" ("false"). В языке C нет логического (булевого, boolean) типа. Поэтому результатом логического выражения является целочисленное арифметическое значение. В языке C "истинно" - это ненулевая величина, "ложно" - это нуль. В большинстве случаев в качестве ненулевого значения "true" используется единица. Так, из приведенных выше примеров условных выражений 2-е и 3-е приняли значение "нуль", а 1-е и 4-е - ненулевые значения. Рассмотрим пример.

```
#include <stdio.h>
/* Пример 12 */
main()
{
    int tr, fal;
    tr = (101<=105); /*выражение "истинно" */
    fal = (101>105); /*выражение "ложно" */
```

```
printf("true - %d false - %d\n", tr, fal);
return 0;
}
```

Логические операции в языке C соответствуют классическим логическим операциям AND(&&), OR (||) и NOT (!), а их результат - соответствующим таблицам, которые принято называть таблицами истинности:

X	Y	X AND Y	X OR Y	NOT X	X XOR Y
1	1	1	1	0	0
1	0	0	1	0	1
0	1	0	1	1	1
0	0	0	0	1	0

Операция XOR называется операцией "исключающее или". В языке C нет знака логической операции XOR, хотя она может быть реализована с помощью операций AND, OR и NOT. Однако в дальнейшем мы будем рассматривать побитовые операции, среди которых операция "исключающее или" уже есть.

Операции отношения и логические операции имеют приоритет ниже, чем арифметические операции. Это значит, что выражение  $12 > 10 + 1$  рассматривается как выражение  $12 > (10 + 1)$ .

Старшинство логических операций и операций отношения следующее:

Старшая !

> < >= <=  
 == !=  
 &&

Младшая ||

В логических выражениях, как и во всех других, можно использовать круглые скобки, которые имеют наивысший приоритет. Кроме того, круглые скобки позволяют сделать логические выражения более понятными и удобными для чтения. Условные и логические выражения используются в управляющих операторах языка C, таких, как if for и других.

Особенность логических операций && и || состоит в том, что если при вычислении результата операции (выр1) && (выр2) - значение левого операнда (выр1) будет нулевым, то значение второго операнда на результат операции не окажет никакого влияния. В этом случае второй операнд не вычисляется. А следовательно, надежды на то, что при вычислении второго операнда может произойти увеличение какой-либо переменной благодаря операции ++, не оправдаются. То же самое касается операции ||. В этом случае второй операнд не вычисляется, если значение левого операнда не нулевое.

## ОПЕРАЦИЯ ПРИСВАИВАНИЯ

Операция присваивания в языке C обозначается просто знаком `=`. В отличие от других языков в языке C оператор присваивания может использоваться в выражениях, которые также включают в себя операторы сравнения или логические операторы.

В фрагменте

```
if ((f=x-y)>0) printf ("Число x, больше чем y)
```

сначала вычисляется величина  $x-y$ , которая присваивается переменной  $f$ , затем сравнивается ее значение с нулем. Еще одной особенностью использования оператора присваивания в языке C является возможность записать оператор:

```
a=b=c=x*y
```

Такое многократное присваивание в языке C - обычное дело, и выполняется справа налево. Сначала вычисляется значение  $x*y$ , затем это значение присваивается  $c$ , потом  $b$ , и лишь затем  $a$ . В левой части оператора присваивания должно стоять выражение, которому можно присвоить значение. Такое выражение в языке C, например просто переменная, называется величиной *lvalue*. Выражение  $2 = 2$  ошибочно, так как константе нельзя присвоить никакое значение: константа не является величиной *lvalue*.

В языке C имеются дополнительные операции присваивания `+=`, `-=`, `/=`, `*=` и `%=`.

Вместо выражения  $n = n + 5$  можно использовать выражение  $n += 5$ . Здесь `+=` аддитивная операция присваивания, в результате выполнения которой величина, стоящая справа, прибавляется к значению переменной, стоящей слева.

Аналогично

```
m-=20 то же самое, что и m=m-20;  
m*=20 то же самое, что и m=m*20;  
m/=10 то же самое, что и m=m/10;  
m%=10 то же самое, что и m=m%10.
```

Эти операции имеют тот же приоритет, что и операция присваивания `=`, т. е. ниже, чем приоритет арифметических операций. Есть еще несколько дополнительных операций присваивания, о которых мы упомянем ниже. Операция  $x += 5$  выполняется быстрее, чем операция  $x = x + 5$ .

## ПОРАЗРЯДНЫЕ ОПЕРАЦИИ (ПОБИТОВЫЕ ОПЕРАЦИИ)

Поразрядные операции можно проводить с любыми целочисленными переменными и константами. Нельзя использовать эти операции с переменными типа `float`, `double` и `long double`. Результатом побитовой операции будет целочисленное значение.

Поразрядными операциями являются:

AND,

	OR,
^	XOR.
~	NOT,
<<	сдвиг влево,
>>	сдвиг вправо.

Операции AND, OR, NOT и XOR нам уже известны, они полностью аналогичны соответствующим логическим операциям. Только в этом случае мы сравниваем не значения выражений, а значения каждой соответствующей пары разрядов (битов).

Поразрядные операции позволяют, в частности, обеспечить доступ к каждому биту информации. Часто поразрядные операции находят применение в драйверах устройств, программах, связанных с принтером, модемом и другими устройствами.

При выполнении поразрядной операции над двумя переменными, например типа `char`, операция производится над каждой парой соответствующих разрядов. Отличие поразрядных операций от логических и операций отношения состоит в том, что логические операции и операции отношения всегда в результате дают 0 или 1. Для поразрядных операций это не так.

Приведем несколько примеров поразрядных операций. Если надо установить значение старшего разряда переменной типа `char` равным нулю, то удобно применить операцию `&` (AND):

```
ch = ch & 127;
Пусть ch = 'A'
'A'      11000001
127      01111111
```

```
-----
'A' & 127 01000001
```

Если же мы хотим установить старший разряд равным единице, то удобна операция OR:

```
ch = ch | 128;
'A'      11000001
128      10000000
```

```
-----
'A' | 128 11000001
```

Поразрядные операции удобны также для организации хранения в сжатом виде информации о состоянии on/off (включен/выключен). В одном байте можно хранить 8 таких флагов.

Если переменная `ch` является хранилищем таких флагов, то проверить, находится ли флаг, содержащийся в третьем бите, в состоянии on, можно следующим образом:

```
if (ch & 4) printf("3 бит содержит 1, состояние on");
```

Эта проверка основывается на двоичном представлении числа `4=00000100`.

Операции сдвига  $\gg$  и  $\ll$  применимы только к целочисленным переменным. В результате применения этой операции сдвигают все биты левого операнда на число позиций, определяемой выражением справа соответственно вправо или влево. Недостающие значения битов дополняются нулями. Форма этой операции следующая:

value  $\gg$  число позиций,  
value  $\ll$  число позиций.

Пример:

двоичное представление числа  $x=9$ : 00001001, тогда

$x = 9 \ll 3$  01001000,  
 $x = 9 \gg 3$  00000001,  
 $x = 9 \gg 5$  00000000.

При применении операции сдвига может происходить потеря старших или младших разрядов. Применение операций  $\ll$  и  $\gg$  по очереди к одной и той же переменной может изменить значение этой переменной из-за потери разрядов.

Пусть `unsigned char x=255`. Двоичное представление переменной  $x$  имеет вид 11111111.

Значением выражения  $x = x \ll 3$  в двоичном виде будет 11111000.

Значением выражения  $x = x \gg 3$  в двоичном виде будет 00011111.

### ОПЕРАЦИИ ( ) И [ ]

В языке C круглые и квадратные скобки также рассматриваются как операции. Причем эти операции имеют наивысший приоритет.

Поразрядные операции порождают еще несколько сложных операций присваивания:

$|=$ ,  $\&=$ ,  $\wedge=$ ,  $\ll=$ ,  $\gg=$ .

### ОПЕРАЦИЯ УСЛОВИЕ ?:

Операция условие - единственная операция языка C, имеющая три операнда. Эта операция имеет вид:

(выр1)?(выр2):(выр3)

Вычисляется выражение (выр1). Если это выражение имеет ненулевое значение, то вычисляется выражение (выр2). Результатом операции будет значение выражения (выр2).

Если значение выражения (выр1) равно нулю, то вычисляется выражение (выр3) и его значение будет результатом операции. В любом случае вычисляется только одно из выражений: (выр2) и (выр3). Например, операцию условие удобно применять для нахождения наибольшего из двух чисел  $x$  и  $y$ :

$\max = (x > y) ? x : y$

или для нахождения абсолютной величины числа  $x$ :

$\text{abs} = (x > 0) ? x : -x$

Если второй и третий операнды являются величинами типа lvalue, т. е. могут стоять в левой части операции присваивания, то и результат операции условие является величиной типа lvalue. С помощью этой операции можно в одну строчку решить задачу: наибольшее из чисел x или y заменить значением 1:

$(x > y) ? x : y = 1;$

### ОПЕРАЦИЯ ЗАПЯТАЯ

Операция запятая имеет самый низкий приоритет из всех операций языков C и C++. Операция запятая выполняется слева направо, и ее значением является значение правого операнда. В выражении (выр1), (выр2) сначала вычислится значение (выр1), затем - значение (выр2). Это значение и будет результатом операции.

### ОПЕРАЦИЯ sizeof

Операция sizeof имеет две формы: sizeof (тип) или sizeof (выражение).

Результатом этой операции является целочисленное значение величины типа или выражения в байтах. При использовании второй формы значение выражения не вычисляется, а лишь определяется его тип. Примеры использования: sizeof (short int) или sizeof (x). В качестве типа в операции sizeof не может использоваться тип void.

Операции . и -> будут определены ниже.

Дальше мы увидим, что некоторые знаки операций имеют несколько смысловых значений. Так, знак операции & имеет два значения: бинарная операция побитовое AND и унарная операция взятия адреса.

## УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ

Мы уже знаем простейшую форму двух таких операторов: if и for. Рассмотрим эти и другие операторы подробнее. Управляющие операторы можно разделить на три категории:

1. Условные операторы if, if-else и switch.
2. Операторы цикла for, while и do-while.
3. Оператор безусловного перехода goto.

### УСЛОВНЫЙ ОПЕРАТОР IF

Мы уже знаем простейшую форму оператора if. Полная форма этого оператора следующая:

if (условие) оператор;  
else оператор;

Если значение условия "истинно", то выполняется оператор (им может быть составной оператор - блок), следующий за условием. Если же условие принимает значение "ложно" то выполняется оператор, следующий за ключевым словом else. В записи оператора if вторая часть (т. е. оператор else) может отсутствовать. Тогда, если условие принимает значение "ложно", вы-

полняется сразу следующий оператор программы. На самом деле в качестве условия может стоять произвольное выражение. В операторе `if` лишь проверяется, является ли значение этого выражения ненулевым (истинным) или нулевым (ложным). С помощью оператора `if` можно, например, вычислить значение функции `sgn(x)` - знак  $x$ . Функция `sgn(x)` принимает значение 1, если  $x > 0$ , значение -1, если  $x < 0$ , значение 0, если  $x = 0$ .

```
#include <stdio.h>
/* Пример 13. */
main()
{
    int sgn;
    float x;
    printf("Введите число:");
    scanf("%f", &x);
    if(x>0) { sgn=1;printf("Число %f положительное sgn = %d \n", x, sgn); }
    if(x==0) { sgn=0;printf("Число %f равно нулю sgn = %d \n", x, sgn); }
    if(x<0) { sgn=-1;printf("Число %f отрицательное sgn = %d \n", x, sgn); }
}
```

Часто встречается необходимость использовать конструкцию `if-else-if`:

```
if (условие) оператор;
else if (условие) оператор;
    else if (условие) оператор;
...
else оператор;
```

В этой форме условия операторов `if` проверяются сверху вниз. Как только какое-либо из условий принимает значение "истинно", выполнится оператор, следующий за этим условием, а вся остальная часть конструкции будет проигнорирована. Операторы `if` предыдущего примера могут быть записаны в другом виде:

```
#include <stdio.h>
/* Пример 14. */
main()
{
    int sgn;
    float x;
    printf("Введите число:");
    scanf("%f", &x);
    if(x>0) { sgn=1;printf("Число %f положительное \n", x); }
    else if(x<0){sgn=-1;printf("Число %f отрицательное \n", x); }
    else { sgn=0;printf("Число %f равно нулю \n", x); }
}
```

В качестве условия оператора `if` может использоваться, как мы уже сказали, некоторое выражение. Так, для того чтобы проверить, равно число  $x$  нулю или не равно, можно написать



```
if (x == 0) printf("Число равно нулю");
else printf("Число не равно нулю");
```

Тот же результат можно получить следующим оператором:

```
if (!x) printf("Число равно нулю");
else printf("Число не равно нулю");
```

Вложенным оператором `if` называется следующая конструкция:

```
if (x)
    if (y) оператор1;
    else оператор2;
```

В такой форме непонятно, к какому из операторов `if` относится `else`. В языке C оператор `else` ассоциируется с ближайшим `if` в соответствующем блоке. В последнем примере `else` относится к `if(y)`. Для того чтобы отнести `else` к оператору `if(x)`, нужно соответствующим образом расставить операторные скобки:

```
if (x){
    if (y) оператор1;
}
else оператор2;
```

Теперь `if(y)` относится к другому блоку.

#### ОПЕРАТОР SWITCH

Язык C имеет встроенный оператор множественного выбора, называемый `switch`. Основная форма оператора имеет такой вид:

```
switch (выражение) {
    case constant1:
        последовательность операторов
        break;
    case constant2:
        последовательность операторов
        break;
    . . .
    case constantN:
        последовательность операторов
        break;
    default
        последовательность операторов
}
```

Сначала вычисляется выражение в скобках за ключевым словом `switch`. Затем просматривается список меток (`case constant1` и т. д.) до тех пор, пока не находится метка, соответствующую значению вычисленного выражения. Далее происходит выполнение соответствующая последовательности операторов, следующих за двоеточием. Если же значение выражения не соответствует ни одной из меток оператора `switch`, то выполняется последовательность операторов, следующая за ключевым словом `default`.

Допускается конструкция оператора switch, когда слово default и соответствующая последовательность операторов может отсутствовать.

Еще один не встречавшийся нам ранее оператор - break. Когда после последовательности операторов встречается ключевое слово break, то выполнение оператора break приводит к выходу из оператора switch и переходу к следующему оператору программы. Наличие оператора break в операторе switch необязательно. Что будет, если операторов break не будет? Ответ на этот вопрос дадут результаты работы следующих двух вариантов программы:

```
#include <stdio.h>
/* Пример 15. */
/* Пример оператора switch с использованием break */
main()
{
    char ch;
    printf("Введите заглавную букву русского алфавита:");
    ch=getchar();
    if(ch>='А' && ch<='Я')
        switch(ch)
        {
            case 'А':
                printf("Алексеев \n");
                break;
            case 'Б':
                printf("Булгаков \n");
                break;
            case 'В':
                printf("Волюшин \n");
                break;
            case 'Г':
                printf("Гоголь \n");
                break;
            default:
                printf("Достоевский, Зощенко и другие \n");
                break;
        }
    else printf("Надо было ввести заглавную русскую букву \n");
}

#include <stdio.h>
/* Пример 16. */
/* Пример оператора switch без использования break */
main()
{
    char ch;
    printf("Введите заглавную букву русского алфавита:");
    ch=getchar();
    if(ch>='А' && ch<='Я')
```

```
switch(ch)
{
case 'A':
printf("Алексеев \n");
case 'Б':
printf("Булгаков \n");
case 'В':
printf("Волошин \n");
case 'Г':
printf("Гоголь \n");
default:
printf("Достоевский, Зощенко и другие \n");
}
else printf("Надо было ввести заглавную русскую букву \n");
}
```

Предположим, вы запустили первую программу и ввели букву Б. Результатом работы программы будет следующая строка:

Булгаков

Выполнился только один оператор, соответствующий метке 'Б'. В случае запуска другой программы ввода буквы Б результат работы программы будет следующий:

Булгаков  
Волошин  
Гоголь  
Достоевский, Зощенко и др.

Мы видим, что выполнились все операторы, начиная с метки 'Б', включая тот, который следует за словом default.

Оператор break заканчивает последовательность операторов, относящихся к каждой метке. Далее выполняется следующий оператор программы. Если же оператор break отсутствует, то выполнение продолжается до первого оператора break ли до конца оператора switch.

## Циклы

Циклы необходимы, когда нам надо повторить некоторые действия несколько раз, как правило, пока выполняется некоторое условие. В языке C известно три вида оператора цикла: for, while и do-while.

### Цикл for

Основная форма цикла for имеет следующий вид:

for (инициализация ; проверка условия ; изменение) оператор;

на самом деле в общем виде

for (выражение1 ; выражение2 ; выражение 3) оператор;

В простейшей форме инициализация используется для присвоения начального значения параметру цикла. Проверка условия - обычно условное

выражение, которое определяет, когда цикл должен быть завершен. Приращение обычно используется для изменения параметра цикла каждый раз при повторении цикла. Эти три раздела заголовка цикла должны быть разделены точкой с запятой. Выполнение цикла происходит до тех пор, пока условное выражение истинно. Как только условие становится ложным, начинается выполнение следующий за циклом `for` оператор.

Простейший пример оператора цикла `for`:

```
for(i=0; i<10; i++) printf("%d\n", i);
```

В результате выполнения этого оператора будут напечатаны в столбик цифры от 0 до 9. Для печати этих цифр в обратном порядке можно использовать следующий оператор:

```
for(i=9; i>=0; i--) printf("%d\n", i);
```

Цикл `for` похож на аналогичные циклы в других языках программирования, и в то же время этот оператор в языке C гораздо более гибкий, мощный и применим во многих ситуациях.

В качестве параметра цикла необязательно использовать целочисленный счетчик. Приведем фрагмент программы, выводящей на экран буквы русского алфавита:

```
unsigned char ch;  
for (ch='A'; ch<='Я'; ch++) printf("%c ", ch);
```

Следующий фрагмент программы

```
for(ch='0'; ch!='N';) scanf("%c", &ch);
```

будет выполняться до тех пор, пока с клавиатуры не будет введен символ 'N'. Заметим, что место, где должно быть приращение, пусто. Случайно или намеренно может получиться цикл, из которого нет выхода, так называемый бесконечный цикл.

Приведем три примера таких циклов.

```
for (;;) printf("Бесконечный цикл\n");  
for (i=1; i++;) printf("Бесконечный цикл\n");  
for (i=10; i>6; i++) printf("Бесконечный цикл\n");
```

Тем не менее для таких циклов также может быть организован выход. Для этого используется оператор `break`, который мы встречали выше. Если оператор `break` встречается в составном операторе цикла, то происходит немедленное прекращение выполнения цикла и начинается выполнение следующего оператора программы.

```
#include <stdio.h>  
/* Пример 17. */  
main()  
{  
    unsigned char ch;  
    for(;;) {  
        ch=getchar(); /*Прочитать символ */
```

```

    if (ch=='Q') break; /*Проверка символа */
    printf("%c", ch); /*Печать символа */
}
}

```

В этой программе будут печататься введенные символы до тех пор, пока не будет введен символ 'Q'. Возможно, и это синтаксически правильно, наличие пустого оператора (отсутствие оператора) в цикле for. Например

```
for (i=0; i<10000; i++);
```

### Циклы while и do-while

Следующий оператор цикла в языке C - это цикл while. Основная его форма имеет следующий вид:

```
while (условие) оператор;
```

где оператор может быть простым, составным или пустым оператором. "Условие", как и во всех других операторах, является просто выражением. Цикл выполняется до тех пор, пока условие принимает значение "истинно". Когда же условие примет значение "ложно", программа передаст управление следующему оператору программы. Так же как и в цикле for, в цикле while сначала проверяется условие, а затем выполняется оператор. Это так называемый цикл с предусловием.

В отличие от предыдущих циклов в цикле do-while условие проверяется в конце оператора цикла. Основная форма оператора do-while следующая:

```
do {
    последовательность операторов
} while (условие);
```

Фигурные скобки необязательны, если внутри них находится один оператор. Тем не менее они чаще всего ставятся для лучшей читаемости программы, а также чтобы не спутать (программисту, а не компилятору) с оператором while. Оператор do-while называется оператором цикла с постусловием. Какое бы условие ни стояло в конце оператора, набор операторов в фигурных скобках один (первый) раз выполнится обязательно. В циклах for и while оператор может не выполниться ни разу.

```

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
/* Пример 18. */
/* Игра "угадай число" */
/* Программа выбирает случайное число от 1 до 100, вы
должны угадать его */
main()
{
    int s, x;
    int n=0;
    randomize();
    s=random(100)+1;

```

```

do {
    printf("Введите число от 1 до 100. ");
    scanf("%d", &x);
    n++;
    if (s<x) printf("Загаданное число меньше\n");
    if (s>x) printf("Загаданное число больше\n");
} while (s-x);
printf("Вы угадали число !\n");
printf("Затратили на угадывание %d попыток\n", n);
}

```

### Вложенные циклы

Когда один цикл находится внутри другого, то говорят, что это вложенные циклы. Часто встречаются вложенные циклы, например, при заполнении таблиц. В качестве примера рассмотрим программу печати таблицы умножения целых чисел.

```

#include <stdio.h>
/* Пример 19. */
main()
{
    int i, j;
    for (i=1; i<10; i++)
    {
        for (j=1; j<5; j++)
            printf("%d * %d = %2d ", i, j, i*j);
        printf("\n");
    }
}

```

### ИСПОЛЬЗОВАНИЕ ОПЕРАТОРА BREAK В ЦИКЛАХ

Оператор break имеет два применения. Первое - окончание case в операторе switch. Второе - немедленное окончание цикла, не связанное с проверкой обычного условия окончания цикла. Когда оператор break встречается внутри оператора цикла, то происходит немедленный выход из цикла и переход к выполнению оператора, следующего за оператором цикла.

```

#include <stdio.h>
/* Пример 20. */
main()
{
    int i;
    for (i=0; i<1000; i++)
    {
        printf("%d - %d \n", i, i*i);
        if (i*i >= 10000) break;
    }
}

```

## ОПЕРАТОР CONTINUE

Еще один полезный оператор - оператор continue. Если оператор continue встретился в операторе цикла, то он передает управление на начало следующей итерации цикла. В циклах while и do-while - на проверку условия, в цикле for - на приращение. Этот оператор необходим, если вы хотите закончить текущую итерацию цикла и не выполнять оставшиеся операторы, а сразу перейти к следующей итерации цикла. Например, его можно использовать в программе, печатающей натуральные числа кратные семи.

```
#include <stdio.h>
/* Пример 21 */
main()
{
    int i;
    for(i=1; i<1000; i++)
    {
        if (i%7) continue;
        printf("%8d", i);
    }
}
```

## ОПЕРАТОР GOTO

Язык C обладает всеми возможностями для написания хорошо структурированных программ. Апологеты структурного программирования считают дурным тоном использование оператора goto, без которого было тяжело обойтись в таких языках, как FORTRAN или BASIC. В языке MODULA Николас Вирт, автор языков Pascal и MODULA, исключил оператор goto совсем. Тем не менее оператор goto в языке C есть и иногда он может быть полезен, хотя без него можно обойтись в любой ситуации. Для использования оператора goto надо ввести понятие метки (label). Метка - это идентификатор, за которым следует двоеточие. Метка должна находиться в той же функции, что и оператор goto. Одно из полезных применений оператора goto - это выход из вложенных циклов:

```
for ( ) {
    while ( ){
        for ( ) {

            goto exit;

        }
    }
}
exit: printf ("Быстрый выход из вложенных циклов");
```



## МАССИВЫ И УКАЗАТЕЛИ

Ранее мы ввели типы данных в языке C, которые называются иногда базовыми или встроенными. На основе этих типов данных язык C позволяет строить другие типы данных и структуры данных. Массив - одна из наиболее простых и известных структур данных. Под массивом в языке C понимают набор данных одного и того же типа, собранных под одним именем. Каждый элемент массива определяется именем массива и порядковым номером элемента, который называется индексом. Индекс в языке C всегда целое число.

### ОБЪЯВЛЕНИЕ МАССИВА В ПРОГРАММЕ

Основная форма объявления массива размерности N такова:

тип <имя массива>[размер1][размер2]...[размерN]

Чаще всего используются одномерные массивы:

тип <имя массива>[размер];

тип - базовый тип элементов массива,

размер - количество элементов одномерного массива.

При описании двумерного массива объявление имеет следующий вид:

тип <имя массива>[размер1][размер2];

В этом описании можно трактовать объявление двумерного массива как объявление массива массивов, т. е. массив размера [размер2], элементами которого являются одномерные массивы <имя массива>[размер1].

Размер массива в языке C может задаваться константой или константным выражением. Нельзя задать массив переменного размера. Для этого существует отдельный механизм, называемый динамическим выделением памяти. К вопросу о динамическом выделении памяти и работе с массивами переменного размера мы вскоре вернемся.

Сначала обсудим более подробно одномерные массивы.

В языке C индекс всегда начинается с нуля. Когда мы говорим о первом элементе массива, то имеем в виду элемент с индексом 0. Если мы объявили массив

```
int a[100];
```

это значит, что массив содержит 100 элементов от a[0] до a[99]. Для одномерного массива легко подсчитать, сколько байт в памяти будет занимать этот массив:

колич. байт = <размер базового типа> \* <колич. элементов>.

В языке C под массив всегда выделяется непрерывное место в оперативной памяти.

В языке C не проверяется выход индекса за пределы массива. Если массив a[100] описан как целочисленный массив, имеющий 100 элементов, а вы в программе укажете a[200], то сообщение об ошибке не будет вы-

иано, а в качестве значения элемента `a[200]` будет выдано некоторое число, занимающее соответствующие 2 байта. Можно определить массив любого определенного ранее типа, например

```
unsigned arr[40], long double al[1000], char ch[80];
```

### Массивы символов. Строки

Однако массивы типа `char` - символьные массивы - занимают в языке особое место. Во многих языках есть специальный тип данных - строка символов (`string`). В языке C отдельного типа строки символов нет, а реализована работа со строками путем использования одномерных массивов типа `char`. В языке C символьная строка - это одномерный массив типа `char`, заканчивающийся нулевым байтом. Нулевой байт - это байт, каждый бит которого равен нулю. Для нулевого байта определена специальная символьная константа `'\0'`. Это следует учитывать при описании соответствующего массива символов. Так, если строка должна содержать `N` символов, то в описании массива следует указать `N+1` элемент.

Например, описание

```
char str[11];
```

предполагает, что строка содержит 10 символов, а последний байт зарезервирован под нулевой байт. Конечно, мы задали обычный одномерный массив, но если мы хотим трактовать его как строку символов, то это будет строка максимум из 10 элементов.

Хотя в языке C нет специального типа строки, язык допускает строковые константы. Строковая константа - это список литер, заключенных в двойные кавычки. Например,

"Borland C++", "Это строковая константа".

В конец строковой константы не надо ставить символ `'\0'`. Это сделает компилятор, и строка "Borland C++" в памяти будет выглядеть так:

B	o	r	l	a	n	d		C	+	+	\0
---	---	---	---	---	---	---	--	---	---	---	----

Есть два простых способа ввести строку с клавиатуры. Первый способ - воспользоваться функцией `scanf()` со спецификатором ввода `%s`. Надо помнить, что функция `scanf()` вводит символы до первого пробельного символа. Второй способ - воспользоваться специальной библиотечной функцией `gets()`, объявленной в файле `stdio.h`. Функция `gets()` позволяет вводить строки, содержащие пробелы. Ввод оканчивается нажатием клавиши Enter. Обе функции автоматически ставят в конец строки нулевой байт. Не забудьте зарезервировать для него место. В качестве параметра в этих функциях используется просто имя массива.

Вывод строк производится функциями `printf()` или `puts()`. Обе функции выводят содержание массива до первого нулевого байта. Функция `puts()` добавляет в конце выводимой строки символ новой строки. В функции `printf()` переход на новую строку надо предусматривать в строке формата самим.

Рассмотрим пример.

```
# include <stdio.h>
/* Пример 22. */
/* Ввод строки с клавиатуры */
main()
{
    char str[80]; /* Зарезервовали место для строки */
    printf("Введите строку длиной менее 80 символов: ");
    gets(str);    /* читает строку с клавиатуры,
                   пока не нажмете клавишу Enter */
    printf("Вы ввели строку %s \n", str);

    printf("Введите еще одну строку длиной менее 80 символов: ");
    scanf("%s", str); /* читает строку с клавиатуры,
                      пока не встретится пробел */
    printf("Вы ввели строку ");
    puts(str);
}
```

### Функции для работы со строками

Для работы со строками существует специальная библиотека, описание которой находится в файле `string.h`.

Наиболее часто используются функции

`strcpy()`, `strcat()`, `strlen()`, `strcmp()`.

Рассмотрим их подробнее. Вызов функции `strcpy()` имеет вид

```
strcpy(s1, s2);
```

Функция `strcpy()` используется для копирования содержимого строки `s2` в строку `s1`. Массив `s1` должен быть достаточно большим, чтобы в него поместилась строка `s2`. Если места мало, компилятор не выдает указания на ошибку или предупреждения; это не прервет выполнения программы, но может привести к порче других данных или самой программы и неправильной работе программы в дальнейшем.

Вызов функции `strcat()` имеет вид

```
strcat(s1, s2);
```

Функция `strcat()` присоединяет строку `s2` к строке `s1` и помещает ее в массив, где находилась строка `s1`, при этом строка `s2` не изменяется. Нулевой байт, который завершал строку `s1`, будет заменен первым символом строки `s2`. И в функции `strcpy()`, и в функции `strcat()` получающаяся строка автоматически завершается нулевым байтом.

Рассмотрим простой пример использования этих функций.

```
# include <stdio.h>
# include <string.h>
/* Пример 23. */
```

```
main()
{
    char s1[20], s2[20];
    strcpy(s1, "Hello, ");
    strcpy(s2, "World !");
    puts(s1);
    puts(s2);
    strcat(s1, s2);
    puts(s1);
    puts(s2);
}
```

Вызов функции `strcmp()` имеет вид

```
strcmp(s1, s2);
```

Функция `strcmp` сравнивает строки `s1` и `s2` и возвращает значение 0, если строки равны, т. е. содержат одно и то же число одинаковых символов. Под сравнением строк мы понимаем сравнение в лексикографическом смысле, так как это происходит, например, в словаре. Конечно, в функции происходит посимвольное сравнение кодов символов. Код первого символа одной строки сравнивается с кодом символа второй строки. Если они равны, рассматриваются вторые символы, и т. д. Если `s1` лексикографически (в смысле словаря) больше `s2`, то функция `strcmp()` возвращает положительное значение, если меньше - отрицательное значение.

Вызов функции `strlen()` имеет вид

```
strlen(s);
```

Функция `strlen()` возвращает длину строки `s`, при этом завершающий нулевой байт не учитывается. Вызов `length("Hello")` вернет значение 5.

Рассмотрим применение этой функции для вычисления длины строки, вводимой с клавиатуры.

```
# include <stdio.h>
# include <string.h>
/* Пример 24 */

main()
{
    char s[80];
    printf("Введите строку: ");
    gets(s);
    printf("Строка \n %s \n имеет длину %d символов \n", s, strlen(s));
}
```

## ДВУМЕРНЫЕ МАССИВЫ

Как мы уже отмечали, язык C допускает многомерные массивы, простейшей формой которых является двумерный массив (two-dimensional array). Можно сказать, что двумерный массив - это массив одномерных массивов.

Двумерный массив `int a[3][4]` можно представить в виде таблички:

Второй индекс	Первый индекс			
	a[0][0]	a[0][1]	a[0][2]	a[0][3]
	a[1][0]	a[1][1]	a[1][2]	a[1][3]
	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Первый индекс - номер строки, второй индекс - номер столбца. Количество байт памяти, которое необходимо для хранения массива, вычисляется по формуле

Колич. байт = <размер типа данных>\*<колич. строк>\*<колич. столбцов>.

В памяти компьютера массив располагается непрерывно по строкам, т. е.

a[0][0], a[0][1], a[0][2], a[0][3], a[1][0], a[1][1], a[1][2], a[2][1], ..., a[2][3].

Следует помнить, что память для всех массивов, которые определены как глобальные, отводится в процессе компиляции и сохраняется все время, пока работает программа.

Часто двумерные массивы используются для работы с таблицами, содержащими текстовую информацию. Также очень часто используются массивы строк. Рассмотрим пример.

```
# include <stdio.h>
# include <string.h>
/* Пример 25. */
main()
{
    char text[5][20];
    strcpy(text[0], "Turbo Basic");
    strcpy(text[1], "Turbo Pascal");
    strcpy(text[2], "Borland C++");
    strcpy(text[3], "Turbo Prolog");
    strcpy(text[4], "Paradox");
}
```

Мы заполнили массив text[5][20]. Обратим внимание на то, что в функции strcpy() в массиве используется только первый индекс!

Заполнение массива иллюстрирует следующая таблица:

T	u	r	b	o		B	a	s	i	c	\0								
T	u	r	b	o		P	a	s	c	a	l	\0							
B	o	r	l	a	n	d		C	+	+	\0								
T	u	r	b	o		P	r	o	l	o	g	\0							
P	a	r	a	d	o	x	\0												

### ИНИЦИАЛИЗАЦИЯ МАССИВОВ

Очень важно уметь инициализировать массивы, т. е. присваивать элементам массива некоторые начальные значения. В языке C для этого имеются специальные возможности. Самый простой способ инициализации следующий: в процессе объявления массива можно указать в фигурных скобках список инициализаторов:

```
float farr[6]={ 1.1, 2.2, 3.3, 4.0, 5, 6};
int a[3][5]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
```

В другом случае такая форма записи эквивалентна набору операторов:

```
a[0][0]=1; a[0][1]=2; a[0][2]=3; a[0][3]=4;
a[0][4]=5; a[1][0]=6; a[1][1]=7; a[1][2]=8;
```

т. д.

Многомерные массивы, в том числе и двумерные массивы, можно инициализировать, рассматривая их как массив массивов.

Инициализации

```
int a[3][5]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
```

```
int a[3][5]={{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};
```

эквивалентны. Количество инициализаторов не обязано совпадать с количеством элементов массива. Если инициализаторов меньше, то оставшиеся значения элементов массива не определены.

В то же время инициализации

```
int a[3][5]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11};
```

```
int a[3][5]={{1, 2, 3}, {4, 5, 6, 7, 8}, {9, 10, 11}};
```

различны. Соответствующие массивы будут заполнены следующим образом:

В первом случае

1	2	3	4	5
6	7	8	9	10
11				

Во втором случае

1	2	3		
4	5	6	7	8
9	10	11		

В пустых клеточках значения не определены.

Символьные массивы могут инициализироваться как обычный массив:

```
char str[15]='B', 'o', 'r', 'l', 'a', 'n', 'd', ' ', 'C', '+', '+';
```

или могут - как строка символов:

```
char str[15]="Borland C++";
```

Отличие этих двух способов состоит в том, что во втором случае будет добавлен еще и нулевой байт. К тому же второй способ короче.

Допускается также объявление и инициализация массива без явного указания размера массива. Например, для выделения места под символьный массив обычным способом

```
char str[80]="Это объявление и инициализация массива символов";
```

мы должны считать количество символов в строке или указать заведомо больший размер массива.

При инициализации массива без указания его размера

```
char str_[]="Это объявление и инициализация массива символов";
```

компилятор сам определит необходимое количество элементов массива, включая нулевой байт. Можно объявлять таким же способом массивы любого типа:

```
int mass[]={1, 2, 3, 1, 2, 3, 4};
```

и многомерные массивы. При объявлении массивов с неизвестным количеством элементов можно не указывать размер только в самых левых квадратных скобках:

```
int arr[][3]={ 1, 2, 3,
               5, 6, 7,
               8, 9, 0};
```

В качестве примера использования массивов можно привести метод сортировки элементов массива:

```
# include <stdio.h>
/* Пример 26. */
main()
{
    int arr[10]={ 1, 23, 4, 7, 8, 0, 1, 9, 4, 7 };
    int i, j, tmp;

    printf("Неотсортированный массив: ");
    for(i=0; i<10; i++)
        printf("%d ", arr[i]);
    printf("\n");
    for(i=0; i<9; i++)
        for(j=i; j<10; j++)
            if (arr[j] < arr[j+1])
            {
                tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
    printf("Отсортированный массив: ");
    for(i=0; i<10; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

## УКАЗАТЕЛИ

Понимание и правильное использование указателей очень важно для создания хороших программ на языке C. Указатели необходимы для успешного использования функций и динамического распределения памяти. Кроме того, многие конструкции языка Borland C++ требуют применения указателей. Однако с указателями следует обращаться осторожно. Использование в программе неинициализированного, или "дикого" (wild), указате-



я может привести к "зависанию" компьютера. При неаккуратном использовании указателей в программе могут возникнуть ошибки, которые очень трудно найти. Но и обойтись без указателей в программах на языке C нельзя.

#### ОБЪЯВЛЕНИЕ УКАЗАТЕЛЕЙ

Указатель - это переменная, которая содержит адрес некоторого объекта. Здесь имеется в виду адрес в памяти компьютера. Вообще-то это просто целое число. Но нельзя трактовать указатель как переменную или константу елого типа. Если переменная будет указателем, то она должна быть соответствующим образом объявлена. Указатель объявляется следующим образом:

тип <имя переменной>;

В этом объявлении тип - некоторый тип языка C, определяющий тип бъекта, на который указывает указатель (адрес которого содержит); \* - означает, что следующая за ней переменная является указателем.

Например:

```
char *ch;
int *temp, i, *j;
float *pf, f;
```

Здесь объявлены указатели ch, temp, j, pf, переменная i типа int и переменная f типа float.

#### ОПЕРАЦИИ НАД УКАЗАТЕЛЯМИ

С указателями связаны две специальные операции: & и \*.

Обе эти операции являются унарными, т. е. имеют один операнд, перед которыми они ставятся. Операция & соответствует операции "взять адрес". Операция \* соответствует словам "значение, расположенное по указанному адресу".

Особенность языка C состоит в том, что знак \* соответствует двум операциям, не имеющим друг к другу никакого отношения: арифметической операции умножения и операции взять значение. В то же время спутать их в контексте программы нельзя, так как одна из операций унарная (содержит один операнд), другая - умножение - бинарная (содержит два операнда). Унарные операции & и \* имеют наивысший приоритет наравне с унарным минусом.

В объявлении переменной, являющейся указателем, очень важен базовый тип. Откуда компилятор знает, сколько байт памяти занимает переменная, на которую указывает данный указатель? Ответ прост: из базового типа указателя. Если указатель имеет базовый тип int, то переменная занимает 2 байта, char - 1 байт и т. д.

Простейшие действия с указателями иллюстрируются следующей программой:

```
# include <stdio.h>
/* Пример 27 */
/* Работа с указателями */
main()
{
    float x = 10.1, y;
    float *pf;
    pf = &x;
    y = *pf;
    printf("x=%f y=%f", x, y);
    *pf++;

    printf("x=%f y=%f", x, y);
    y = 1 + *pf * y;

    printf("x=%f y=%f", x, y);
    return 0;
}
```

К указателям можно применить операцию присваивания. Указатели одного и того же типа могут использоваться в операции присваивания, как и любые другие переменные.

Рассмотрим пример.

```
# include <stdio.h>
/* Пример 28 */
main()
{
    int x = 10;
    int *p, *g;
    p = &x;
    g = p;
    printf("%p", p);      /* печать содержимого p */
    printf("%p", g);      /* печать содержимого g */
    printf("%d %d", x, *g); /* печать величины x
                           и величины по адресу g */
}
```

В этом примере приведена еще одна спецификация формата функции `printf()` `%p`. Этот формат используется для печати адреса памяти в шестнадцатеричной форме.

Нельзя создать переменную типа `void`, но можно создать указатель на тип `void`. Указателю на `void` можно присвоить указатель любого другого типа. Однако при обратном присваивании необходимо использовать явное преобразование указателя на `void`;

```
void *pv;
float f, *pf;
pf = &f;
pv = pf;
pf = (float*)pv;
```

В языке C допустимо присвоить указателю любой адрес памяти. Однако, если объявлен указатель на целое

```
int *p;
```

а по адресу, который присвоен данному указателю, находится переменная `x` типа `float`, то при компиляции программы будет выдано сообщение об ошибке в строке

```
p=&x;
```

Эту ошибку можно исправить, преобразовав указатель на `int` к типу указателя на `float` явным преобразованием типа:

```
p=(int*)&x;
```

Но при этом теряется информация о том, на какой тип указывал исходный указатель:

```
# include <stdio.h>
/* Пример 29. */
/* Неправильные действия с указателями */
main()
{
    float x=10.1, y;
    int *p;
    p=&x; /* Потом заменим на p=(int*)&x; */
    y=*p;
    printf("x=%f y=%f\n", x, y);
}
```

В результате работы этой программы не будет получен тот ответ, который ожидался. Переменной `y` не будет присвоено значение переменной `x`, так как будут обрабатываться не 4 байта, как положено для переменной типа `float`, а только 2 байта, так как базовый тип указателя - `int`.

Как и над другими типами переменных, над указателями можно производить арифметические операции: сложение и вычитание. (Операции `++` и `--` являются частными случаями операций сложения и вычитания.) Арифметические действия над указателями имеют свои особенности. Выполним простейшую программу

```
# include <stdio.h>
/* Пример 30. */
main()
{
    int *p;
    int x;
    p=&x;
    printf("%p %p", p, ++p);
}
```

После выполнения этой программы мы увидим, что при операции `++p` значение указателя `p` увеличилось не на 1, а на 2. И это правильно, так как

новое значение указателя должно указывать не на следующий адрес памяти, а на адрес следующего целого. А целое, как мы помним, занимает 2 байта. Если бы базовый тип указателя был не `int`, а `double`, то были бы напечатаны адреса, отличающиеся на 8, именно столько байт памяти занимает переменная типа `double`, т. е. при каждой операции `++p` значение указателя будет увеличиваться на количество байт, занимаемых переменной базового типа указателя.

Операции над указателями не ограничиваются только операциями `++` и `--`. К указателям можно прибавлять некоторое целое или вычитать целое. Пусть указатель `p` имеет значение 2000 и указывает на целое. Тогда в результате выполнения оператора

```
p = p + 3;
```

значение указателя `p` будет 2006. Если же указатель `p1=2000` был бы указателем на `float`, то после применения оператора

```
p1 = p1 + 10;
```

значение `p1` было бы 2040.

Общая формула для вычисления значения указателя после выполнения операции `p=p+n`; будет иметь вид

$$\langle p \rangle = \langle p \rangle + n \cdot \langle \text{колич. байт памяти базового типа указателя} \rangle$$

Можно также вычитать один указатель из другого. Так, если `p` и `p1` - указатели на элементы одного и того же массива, то операция `p - p1` дает такой же результат, как и вычитание индексов соответствующих элементов массива.

Другие арифметические операции над указателями запрещены, например нельзя сложить два указателя, умножить указатель на число и т. д.

Указатели можно сравнивать. Применимы все 6 операций:

`<`, `>`, `<=`, `>=`, `=`, `==` и `!=`

Сравнение `p < g` означает, что адрес, находящийся в `p`, меньше адреса, находящегося в `g`.

Если `p` и `g` указывают на элементы одного массива, то индекс элемента, на который указывает `p`, меньше индекса массива, на который указывает `g`.

#### Связь указателей и массивов

В языке С существует важная связь между массивами и указателями. В языке С принято, что имя массива - это адрес памяти, начиная с которого расположен массив, т. е. адрес первого элемента массива. Таким образом, если был объявлен массив

```
int plus[10];
```

то `plus` является указателем на массив, точнее, на первый элемент массива. Операторы

```
p1 = plus;
и
```

```
p1=&plus[0];
```

приведут к одному и тому же результату.

Для того чтобы получить значение 6-го элемента массива `plus`, можно написать

```
plus[5] или *(p1+5).
```

Результат будет один и тот же. Преимущество использования второго варианта состоит в том, что арифметические операции над указателями выполняются быстрее, если мы работаем с подряд идущими элементами массива. Если же выбор элементов массива случайный, то быстрее и более наглядна работа с индексами.

Очень часто приходится работать над обработкой текстов, т. е. с массивами строк. Как мы помним, в языке C строка - это массив символов, заканчивающийся нулевым байтом. Рассмотрим две программы, реализующие практически, одни и те же действия.

```
# include <stdio.h>
# include <ctype.h>
/* Пример 31. */
main()
{
    char str[]="String From Letters in Different Registers";

    /* Строка, Состоящая из Букв в Разных Регистрах; */
    int i;
    printf("Строка Будет Напечатана Заглавными Буквами");
    while (str[i])

        printf("%c", toupper(str [ i++]));
}
# include <stdio.h>
# include <ctype.h>
/* Пример 32 б. */
main()
{
    char str[]="String From Letters in Different Register";

    /* "Строка Состоящая из Букв в Разных Регистрах"; */
    printf("Строка будет напечатана строчными буквами");
    p=str;
    while(*p)

        printf("%c", tolower(*p++));
}
```

Если в этих примерах заменить строку на английском языке на строку, набранную русскими буквами, то никакого преобразования букв в строчные или, наоборот, в прописные не произойдет. Это связано с тем, что

стандартные функции `toupper()` и `tolower()` анализируют значение аргумента и возвращают то же самое значение, если он не является соответственно строчной или прописной буквой латинского алфавита. Если же аргумент является строчной буквой латинского алфавита, то значением функции `toupper()` будет соответствующая прописная буква (точнее, код этой буквы). Функция `tolower()` изменяет код лишь прописных букв латинского алфавита. Прототипы этих функций находятся в заголовочном файле `ctype.h`.

### МАССИВЫ УКАЗАТЕЛЕЙ

Указатели, как и переменные любого другого типа, могут объединяться в массивы. Объявление массива указателей на 10 целых чисел имеет вид

```
int *x[10];
```

Каждому из элементов массива можно присвоить адрес; например, третьему элементу присвоим адрес целой переменной `y`:

```
x[2]=&y;
```

чтобы найти значение переменной `y`, можно написать `*x[2]`.

Приведем пример использования массива указателей. Чаще всего это бывает удобно при обработке массива строк.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <conio.h>
/* Пример 33 */
main()
{
    char *ext[]={"exe", "com", "dat", "c", "pas", "cpp"}
    char ch, s1[80];
    for(;;)
    { do
        { printf("Файлы с расширением:\n");
          printf("1. exe\n");
          printf("2. com\n");
          printf("3. dat\n");
          printf("4. c\n");
          printf("5. pas\n");
          printf("6. cpp\n");
          printf("7. quit\n");
          printf("Ваш выбор: \n");
          ch=getche();

          printf("\n");
        }
        while ((ch<'1' || (ch>'7')));
        if (ch=='7') break;
        strcpy(s1, "dir *.");
        strcat(s1, ext[ch-1]);
```

```

        system(s1);
    }
}

```

Здесь функция `system()` - библиотечная функция, которая заставляет операционную систему DOS выполнить команду, являющуюся аргументом этой функции.

Очень часто массив указателей используется, если надо иметь ссылки на стандартный набор строк. Например, если мы хотим хранить сообщения о возможных ошибках, это удобно сделать так:

```

char *errors[] = {"Cannot open file ",
                  "Cannot close file ",
                  "Allocation error ",
                  "System error "
                  };

```

При таком объявлении строчные константы будут занесены в раздел констант в памяти, массив указателей будет состоять из четырех элементов, под которые будет выделена память, и эти элементы будут инициализированы адресами, указывающими на начало этих строчных констант.

Вообще строчная константа в языке C ассоциируется с адресом начала строки в памяти, тип строки получается `char*` (указатель на тип `char`). Поэтому возможно и активно используется следующее присваивание:

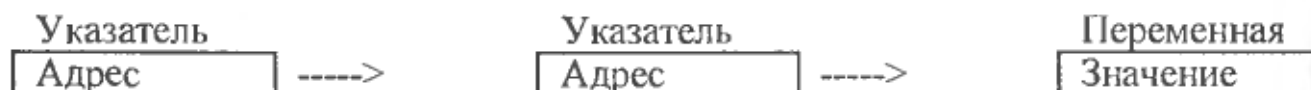
```

char *pc;
pc = "Hello, World !";

```

В языке C возможна также ситуация, когда указатель указывает на указатель. В этом случае описание будет иметь следующий вид:

```
int **point;
```



`point` имеет тип указатель на указатель на `int`. Соответственно, чтобы получить целочисленное значение переменной, на которую указывает `point`, надо в выражении использовать `**point`.

Пример использования:

```

#include <stdio.h>
/* Пример 32. */
main()
{
    int i, pi, ppi;
    i = 7;
    pi = &i;
    ppi = &pi;
    printf("i = %d pi = %p ppi = %p \n", i, pi, ppi);
    *pi += 1;
    printf("i = %d pi = %p ppi = %p \n", i, pi, ppi);
    **ppi = 12;
}

```



```
printf("i = %d pi = %p ppi = %p \n", i, pi, ppi);  
return 0;  
}
```

### ИНИЦИАЛИЗАЦИЯ УКАЗАТЕЛЕЙ

После того как указатель был объявлен, но до того, как ему было присвоено какое-то значение, указатель содержит неизвестное значение. Попытка использовать указатель до присвоения ему какого-то значения является неприятной ошибкой, так как она может нарушить работу не только вашей программы, но и операционной системы. Даже если этого не произошло, результат работы программы будет неправильным и найти эту ошибку будет достаточно сложно.

Принято считать, что указатель, который указывает в "никуда", должен иметь значение `null`, однако и это не делает его "безопасным". После того как он попадет в правую или левую часть оператора присваивания, он вновь может стать "опасным".

С другой стороны нулевой указатель можно использовать, например, для обозначения конца массива указателей.

Если была попытка присвоить какое-либо значение тому, на что указывает указатель с нулевым значением, система выдает предупреждение, появляющееся во время работы программы (или после окончания работы программы) "Null pointer assignment". Появление этого сообщения является поводом для поиска использования неинициализированного указателя в программе.

### ФУНКЦИИ В ЯЗЫКЕ C

Функции - это строительные блоки языка C, самостоятельные единицы программы, спроектированные для решения конкретных задач, обычно повторяющиеся несколько раз.

#### ОБЪЯВЛЕНИЕ ФУНКЦИИ

Основная форма описания (definition) функции имеет вид

```
тип <имя функции>(список параметров)  
{  
    тело функции  
}
```

Тип определяет тип значения, которое возвращает функция с помощью оператора `return`. Если тип не указан, то по умолчанию предполагается, что функция возвращает целое значение (типа `int`). Список параметров состоит из перечня типов и имен параметров, разделенных запятыми. Функция может не иметь параметров, но круглые скобки необходимы в любом случае.

В списке параметров для каждого параметра должен быть указан тип. Пример правильного списка параметров:

```
f(int x, int y, float z)
```

Пример неправильного списка параметров:

```
f(int x, y, float z)
```

### ОПЕРАТОР RETURN

Оператор `return`, с которым мы уже встречались, имеет два варианта использования.

Во-первых, этот оператор вызывает немедленный выход из текущей функции и возврат в вызывающую программу.

Во-вторых, этот оператор может использоваться для возврата значения функции.

Сразу следует отметить, что в теле функции может быть несколько операторов `return`, но может и не быть ни одного. В этот случае возврат в вызывающую программу происходит после выполнения последнего оператора тела функции.

Приведем пример функции, реализующей возведения числа `a` в натуральную степень `b`:

```
float step(float a, int b)
{
    float i;
    if (a < 0) return (-1); /* основание отрицательное */
    a = 1;
    for (i = b; i-- > 0) a *= a;
    return a;
}
```

Эта функция возвращает значение `-1`, если основание отрицательное, и `ab`, если основание неотрицательное. Другой пример - функция для нахождения наибольшего из двух чисел:

```
max(int a, int b)
{
    int m;
    if (a > b) m = a;
    else m = b;
    return m;
}
```

Возможно также написать эту функцию без использования дополнительной переменной:

```
max(int a, int b)
{
    if (a > b) return a;
    else return b;
}
```

Можно еще короче:

```
max(int a, int b)
{
    if(a>b) return a;
    return b;
}
```

А можно и так:

```
max(int a, int b)
{
    return (a>b)? a: b;
}
```

В случае, когда оператора `return` в теле функции нет или за ним нет значения, то возвращаемое функцией значение неизвестно (не определено). Если функция должна возвращать значение, но не делает этого, компилятор выдает предупреждение. Все функции, возвращающие значения, могут использоваться в выражениях языка C, но они не могут использоваться в левой части оператора присваивания, за исключением тех случаев, когда возвращаемым значением является указатель.

Использование функций, возвращающих указатели, имеет некоторые особенности. Указатели не являются ни типом `int`, ни типом `unsigned int`. Их значениями являются адреса памяти данных определенного типа. Соответственно должна быть описана и функция. Рассмотрим пример описания функции, возвращающей указатель на тип `char`. Эта функция находит в строке первый пробел и возвращает его адрес.

```
char* find(char* string)
{
    int i=0;
    while (string[i] != ' ' && (string[i] != '\0')) i++;
    if (string[i]) return &string[i]; /* возвращает адрес первого пробела */
    else return NULL; /* возврат нулевого указателя */
}
```

Когда функция не возвращает никакого значения, она должна быть описана как функция типа `void` (пустая). В стандарте языка C Кернигана & Ритчи ключевого слова `void` не было вообще. Это слово пришло из C++ и оказалось очень удобным. Например, для вывода на экран горизонтальной строки, состоящей из заданного символа, можно использовать следующую функцию, которая, конечно, не должна возвращать никакого значения:

```
void gorizontal_line(char ch)
{
    int i;
    for(i=0; i<80; i++) printf("%c", ch);
}
```

Вы не обязаны объявлять функцию типа `void`, тогда она по умолчанию будет иметь тип `int` и не возвращать никакого значения. Это вызовет предупреждающее сообщение компилятора, но не будет препятствием для ком-

ияции. Однако объявление типа возвращаемого значения функции является хорошим правилом.

### ПРОТОТИПЫ ФУНКЦИЙ

Особенностью стандарта ANSI языка C является то, что для создания правильного машинного кода функции ему необходимо сообщить до ее первого вызова тип возвращаемого результата, а также количество и типы аргументов. Для этой цели в C используется понятие прототипа функции.

Прототип функции задается следующим образом:

тип <имя функции>(список параметров);

Использование прототипа функции является объявлением функции (declaration). Чаще всего прототип функции полностью совпадает с заголовком в описании функции, хотя это и не всегда так. При объявлении функции компилятору важно знать имя функции, количество и тип параметров и тип возвращаемого значения. При этом имена формальных параметров функции не играют никакой роли и игнорируются компилятором. Поэтому прототип функции может выглядеть или так:

```
int func(int a, float b, char* c);
```

или так:

```
int func(int, float, char*);
```

Два этих объявления абсолютно равносильны.

Рассмотрим пример.

```
#include <stdio.h>
/*Пример 33 */
float sqr(float a); /* это прототип функции, объявление функции */
main()
{
    float b;
    b=5.2;
    printf("Квадрат числа %f равен %f", b, sqr(b));
}
float sqr(float a) /* Это описание функции */
{
    return a*a;
}
```

Следующие два примера использования функций вызовут сообщения об ошибке при компиляции. В первом примере препятствием для компиляции будет несоответствие возвращаемого значения объявленному типу функции. Языки C и C++ автоматически преобразуют данные к другому типу, но только тогда, когда это возможно. Целый тип не может быть автоматически преобразовываться в указатель на целое.

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

```

/* Пример 34. Неправильный */
int *sqr(int *i) /* Прототип функции */
main()
{
    int i;
    sqr(&x);
}
int *sqr(int *i) /* Объявление функции */
{
    return *i = (*i)*(*i);
}

```

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

```

/* Пример 35. Неправильный */
int sqr(int *i) /* Прототип функции */
main()
{
    int x = 10;
    sqr(&x, 10); /* Несоответствие количества аргументов */
}
int sqr(int *i)
{
    *i = (*i)*(*i);
}

```

Обратим внимание на то, что если мы исправим эти программы, то функция будет возвращать квадрат числа *i* не через значение функции, а через параметр функции.

Если функция не имеет аргументов, то при объявлении прототипа такой функции следует вместо аргументов писать ключевое слово `void`. В старом стандарте языка C, который должен поддерживаться новыми компиляторами, отсутствие аргументов в скобках не говорило об их отсутствии в данной функции вообще. Чтобы не было недоразумений или путаницы, желательно использовать ключевое слово `void`, если параметры у функции отсутствуют.

Это должно касаться и функции `main()`. Ее объявление должно иметь вид `void main(void)` или `main(void)`.

```

#include <stdio.h>
/* Пример 36 */
void line_(void); /* прототип функции */
main (void)
{
    line_();
}
void line_(void)
{
    int i;
    for(i=0; i<80; i++) printf ("-");
}

```

Мы уже говорили о стандартных заголовочных файлах (header files). Заголовочные файлы языка C содержат два типа информации: первый - это стандартные определения, которые используются функциями. Второй - это прототипы функций, относящиеся к этому заголовочному файлу. Примерами таких заголовочных файлов являются файлы `stdio.h`, `string.h`, `conio.h` и др.

#### Область действия и область видимости

Область действия (scope rules) переменной - это правила, которые устанавливают, какие данные доступны из данного места программы.

В языке C каждая функция - это отдельный блок программы. Попасть в тело функции нельзя иначе, как через вызов данной функции. В частности, нельзя оператором локального перехода `goto` перейти в середину другой функции.

С точки зрения области действия переменных различают три типа переменных: глобальные, локальные и формальные параметры. Правила области действия определяют, где каждая из них может применяться.

Локальные переменные - это переменные, объявленные внутри блока, в частности внутри функции. Язык C поддерживает простое правило: переменная может быть объявлена внутри любого блока программы. Локальная переменная доступна внутри блока, в котором она объявлена. Вспомним, что блок открывается фигурной скобкой и закрывается фигурной скобкой. Область действия локальной переменной - блок.

Локальная переменная существует пока выполняется блок, в котором эта переменная объявлена. При выходе из блока эта переменная (и ее значение) теряется.

```
#include <stdio.h>
/* Пример 37 */
void f(void);
main(void)
{
    int i;
    i = 1;
    f();

    printf("В функции main значение i равно %d\n", i);
}
void f(void)
{
    int i;
    i = 10;
    printf("В функции f() значение i равно %d\n", i);
}
```

Пример показывает, что при вызове функции значение переменной `i`, объявленной в `main()`, не изменилось.

Формальные параметры - это переменные, объявленные при описании функций как ее аргументы. Функции могут иметь некоторое количество параметров, которые используются при вызове функций для передачи значений в тело функции. Формальные параметры могут использоваться в теле функции так же, как локальные переменные, которыми они по сути дела и являются. Область действия формальных параметров - блок, являющийся телом функции.

Глобальные переменные - это переменные, объявленные вне какой-либо функции. В отличие от локальных переменных глобальные переменные могут быть использованы в любом месте программы, но перед их первым использованием они должны быть объявлены. Область действия глобальной переменной - вся программа.

Использование глобальных переменных имеет свои недостатки:

- они занимают память в течение всего времени работы программы;
- использование глобальных переменных делает функции менее общими и затрудняет их использование в других программах;
- использование внешних переменных делает возможным появление ошибок из-за побочных явлений. Эти ошибки, как правило, трудно отыскать.

#### Классы памяти

В языке C есть инструмент, позволяющий управлять ключевыми механизмами использования памяти и создавать мощные и гибкие программы. Этот инструмент - классы памяти. Каждая переменная принадлежит к одному из четырех классов памяти. Эти 4 класса памяти описываются следующими ключевыми словами:

auto - автоматическая,  
extern - внешняя,  
static - статическая,  
register - регистровая.

Тип памяти указывается модификатором - ключевым словом, стоящим перед спецификацией типа переменной. Например,

```
static int sum;  
register int plus;
```

Если ключевого слова перед спецификацией типа локальной переменной при ее объявлении нет, то по умолчанию она принадлежит классу auto. Поэтому практически никогда это ключевое слово не используется. В этом просто нет необходимости.

Автоматические переменные (auto) имеют локальную область действия. Они известны только внутри блока, в котором они определены. Другие функции могут использовать то же имя, но это должны быть переменные, относящиеся к разным блокам. Автоматическая переменная создается (т. е. ей отводится место в памяти программы) при входе в блок функции. При выходе из блока автоматическая переменная пропадает, а область па-



мяти, в которой находилась эта переменная, считается свободной и может использоваться для других целей.

Автоматические переменные хранятся в оперативной памяти машины, точнее, в стеке. Регистровые (register) переменные хранятся в регистрах процессора. Доступ к переменным, хранящимся в регистровой памяти, гораздо быстрее, чем к тем, которые хранятся в оперативной памяти компьютера. В остальном регистровые переменные аналогичны автоматическим переменным. Регистровая память процессора невелика, и если доступных регистров нет, то переменная становится простой автоматической переменной. Описание регистровой переменной имеет вид

```
register int quick;
```

В большинстве случаев создатели компиляторов предусматривают оптимизацию программ, в частности используя регистровую память для размещения в ней переменных, которые активно используются в программе. Авторы компиляторов фирмы Borland утверждают, что оптимизация компиляторов по использованию регистровых переменных сделана так хорошо, что указание использовать переменную как регистровую может только ухудшить эффективность создаваемого машинного кода. В то же время в опциях интегрированной среды есть возможность задавать способы использования регистровых переменных:

- не использовать вообще;
- использовать только тогда, когда есть ключевое слово register;
- по усмотрению компилятора.

Внешняя переменная (extern) относится к глобальным переменным. Она может быть объявлена как вне, так и внутри тела функции:

```
void f(void)
{
    extern int j; /*объявление внешней переменной внутри функции*/
    .....
}
```

Появление ключевого слова extern связано с модульностью языка C, т. е. возможностью составлять многофайловую программу с возможностью раздельной компиляции каждого файла. Когда мы в одном из файлов опишем вне тела функции глобальную переменную

```
float global;
```

то для нее выделится место в памяти в разделе глобальных переменных и констант. Если мы используем эту глобальную переменную в другом файле, то при раздельной компиляции без дополнительного объявления переменной компилятор не будет знать, что это за переменная. Использование объявления

```
extern float global;
```

не приводит к выделению памяти, а сообщает компилятору, что такая переменная будет описана в другом файле. И тогда при компоновке программы, состоящей из нескольких файлов, компоновщик будет искать описание этой переменной и связывать ее с использованием в других файлах. Объявление внешней переменной может быть как вне функции, так и внутри функции. Если это же имя без ключевого слова `extern` будет объявлено внутри функции, то под этим именем будет создана уже другая автоматическая переменная. Можно к объявлению этой переменной добавить ключевое слово `auto`, чтобы показать, что вы не ошиблись, а намеренно продублировали имя. Объявлений переменной как внешней может быть несколько, в том числе и в одной функции или в одном файле. Описание же переменной должно быть только одно.

Следующие примеры демонстрируют разные способы описания внешних переменных:

```
int var;          /* описана внешняя переменная var */
main()
{ extern int var, var1; /* объявлена та же внешняя переменная */
  ...
}
func1()
{ extern int var1; /* объявлена внешняя переменная var1
                  переменная var также внешняя,
                  хотя она и не описана в этом блоке */
  ...
}
func2()          /* переменная var внешняя */
{                /* переменная var1 невидима для
                  этой функции в этом блоке */
}
int var1;        /* описание внешней переменной */
func3()
{                /* для этой функции var1 внешняя */
  int var;       /* переменная var описана как
                  локальная и не связана с
                  глобальной переменной var.
  ...            По умолчанию эта переменная автоматическая */
}
func4(){         /* здесь переменная var является
                  внешней глобальной переменной */
  auto int var1; /* переменная var1 локальная
                  автоматическая переменная */ ...
}
```

При описании статических переменных перед описанием типа ставится ключевое слово `static`. Область действия локальной статической переменной - вся программа. Место в памяти под локальные статические переменные выделяется в начале работы программы в разделе глобальных и стати-

ческих переменных. Однако область видимости локальных статических переменных такая же, как и у автоматических. Значение статических переменных сохраняется от одного вызова функции до другого. Локальные статические переменные инициализируются нулем, если не указан другой инициализатор. При этом описание с инициализацией

```
static int count = 10;
```

локальной статической переменной `count` вызывает однократную инициализацию переменной `count` при выделении под нее места. При последующих вызовах функции, в которой описана эта переменная, инициализации не происходит. Это позволяет использовать такую переменную для счетчика количества вызовов функции.

```
#include <stdio.h>
/* Пример 38 */
/* использование статической переменной */
void trystat(void); /* прототип функции */
main(void)
{
    int i;
    for (i = 1; i <= 3; i++)
    {
        printf("Вызов # %d \n", i);
        trystat();
        printf("Вызов # %d \n", i);
        trystat();
    }
    return 0;
}
void trystat(void)
{
    int auto_l = 1;
    static int stat_l = 1;

    printf("auto_l=%d stat_l=%d \n", auto_l++, stat_l++);
}
```

Можно описать также глобальную(внешнюю) статическую переменную, т. е. описать переменную типа `static` вне любой функции. Отличие внешней переменной от внешней статической переменной состоит в области их действия. Обычная внешняя переменная может использоваться функциями в любом файле, в то время как внешняя статическая переменная только функциями того файла, где она описана, причем только после ее определения. Все глобальные переменные - и статические, и нестатические - инициализируются нулем, если не предусмотрено другой инициализации.

В таблице приведены область действия и продолжительность существования переменных разных классов памяти:

Класс памяти	Ключевое слово	Время существования	Область действия
Автоматический	auto	временно	блок
Регистровый	register	временно	блок
Статический локальный	static	постоянно	блок
Статический глобальный	static	постоянно	файл
Внешний	extern	постоянно	программа

В программе может быть описано несколько переменных с одним и тем же именем. Конечно, в разных блоках. В нижеприведенном модельном примере объявлена одна глобальная переменная и три локальных переменных с одним и тем же именем `var`.

```
#include <stdio.h>
/* Пример 39 */
void f(void);
void fl(void);
int var=7; /* Глобальная переменная, инициализированная
            значением 5 */
main()
{
    int var = 10; /* локальная переменная, закрывающая видимость
                  глобальной переменной */
    printf("var = %d \n", var); /* будет напечатано var = 10 */
    var++;
    {
        int var = 200; /* локальная переменная, закрывающая видимость
                        другой локальной переменной и глобальной переменной */
        printf("var = %d \n", var); /* будет напечатано var = 200 */
        var++;
    }
    printf("var = %d \n", var++); /* будет напечатано var = 11 */
    f();
    printf("var = %d \n", var++); /* будет напечатано var = 12 */
    fl();
    printf("var = %d \n", var++); /* будет напечатано var = 13 */
    fl();
    printf("var = %d \n", var++); /* будет напечатано var = 14 */
    return 0;
}
void f(void)
{
    int var = 77; /*Еще одна локальная переменная в другой функции */
    printf("var = %d \n", var); /* будет напечатано var = 77 */
    var++;
}
void fl(void)
```

```
{
    printf("var = %d \n", var); /* будет напечатано var = 7 или 8.
                               значение глобальной переменной */
    var++;
}
```

#### ПАРАМЕТРЫ И АРГУМЕНТЫ ФУНКЦИИ

В языке C есть особенность, связанная с тем, что все аргументы функции передаются по значению. При вызове функции в стеке выделяется место для формальных параметров функции, и в это выделенное место заносится значение фактического параметра, т. е. значение параметра при вызове функции. Далее функция использует и меняет значения в стеке. При выходе из функции измененные значения параметров теряются.

В языке C вызванная функция не может изменить переменные, указанные в качестве фактических параметров в функции при обращении к ней.

Функция `swap()`, которая должна менять значения параметров местами, не будет фактически это делать.

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Если необходимо, функцию можно приспособить для изменения аргументов. Надо передавать в качестве параметра не значение, а адрес переменной, которую нужно изменять, т. е. передавать указатель на переменную. Такой прием в языке C будет называться передачей параметра по ссылке. Вызванная функция должна описывать аргумент как ссылку и обращаться к фактической переменной косвенно, через эту ссылку. Если в качестве аргумента берется имя массива, то передаваемое функции значение фактически есть адрес первого элемента массива.

Функция `swap1()` будет выглядеть теперь так:

```
void swap1(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Иллюстрацию использования этих двух способов передачи параметров дает следующая программа:

```
#include <stdio.h>
/* Пример 40 */
void swap(int a, int b);
void swap1(int *a, int *b);
void main(void)
```

```

{
    int x=5, y=10;
    printf("Сначала x = %d y = %d\n", x, y);
    swap(x, y);
    printf("Теперь x = %d y=%d\n", x, y);
    printf("Ничего не изменилось ");
    swap1(&x, &y); /* Мы должны передать адреса переменных */
    printf("Теперь x = %d y=%d\n", x, y);
    printf("Значения поменялись ");
    return 0;
}
void swap(int a, int b)
{
    int tmp=a;
    a=b;
    b=tmp;
}
void swap1(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

```

Результатом работы этой программы будет следующее: вначале  $x=5$ ,  $y=10$ , после  $x=5$ ,  $y=10$ .

Значение переменных  $x$  и  $y$  изменились, так как переменные были переданы по ссылке. Мы передали функции `swap1()` адреса переменных  $x$  и  $y$ . Еще одна особенность состоит в том, что в первом случае мы можем вызывать функцию `swap()` с указанием конкретных значений аргументов:

```
swap(5, 10);
```

Вызвать функцию `swap1()` в виде `swap1(&5, &10)`; нельзя.

Если в качестве аргумента функции используется массив, есть лишь один способ - передача параметра по ссылке. В качестве аргумента функции надо указать адрес начала массива. Сделать это можно следующими тремя способами:

```

function(int ar[10]);
function(int ar[]);
function(int *ar);

```

Все три способа приведут к одному и тому же результату.

Следует помнить, что при передаче в качестве аргумента функции массива возможно появление побочного эффекта, а именно изменение значений элементов массива в результате работы функции.

В качестве примера рассмотрим сортировку массива, при этом оформим сортировку в виде функции.

```

#include <stdio.h>
/* Пример 41. Сортировка массива */

```

```

void sort(int arr[], int n)
main()
{
    int mass[10] = {1, 3, -5, 7, 9, 0, 22, 4, 6, 8}
    int size = 10;
    printf("Before sorting ");
    for(i=0; i<10; i++)
        printf("%d ", mass[i]);
    printf("\n");
    sort(mass, size);
    printf("After sorting ");
    for(i=0; i<10; i++)
        printf("%d ", mass[i]);
    return 0;
}

void sort(int arr[], int n) /* можно заголовок заменить
                             на void sort(int arr[10], int n)
                             или void sort(int *arr, int n)
                             */
{
    int i, j, tmp;
    for(i=0; i<n-1; i++)
        for(j=0; j<n-i-1; j++)
            if(arr[j+1]<arr[j]){
                tmp=arr[j];
                arr[j]=arr[j+1];
                arr[j+1]=tmp;
            }
}

```

Как мы видим, элементы массива `mass[]` поменяли свои значения.

Еще одним примером использования массивов в качестве аргументов функции будет умножение двух матриц размером 3 x 3.

```

#include <stdio.h>
/* Пример 42. Умножение матриц */
void multiply(int U[3][3], int V[3][3], int W[3][3]);
main(void)
{
    int A[3][3] = { 0, 1, 2,
                    3, 4, 5,
                    6, 7, 8};
    int B[3][3] = { 1, 2, 3,
                    4, 5, 6,
                    7, 8, 9};
    int i, j, C[3][3];
    multiply (A, B, C);
    printf("Массив C \n");
    for(i=0; i<3; i++)
        printf("%d %d %d \n", C[i][0], C[i][1], C[i][2]);
}

```



```

    return 0;
}
void multiply(int U[3][3], int V[3][3], int W[3][3])
{
    int i, j, k;
    for(i=0; i<3; i++)
        for(j=0; j<3; j++)
        {
            W[i][j]=0;
            for(k=0; k<3; k++)
                W[i][j] += U[i][k]*V[k][j];
        }
}

```

Можно указать в качестве аргумента функции `U[3][3]`, но нельзя указать в качестве такого аргумента `U[3][3]`. Точно так же нельзя указать `**U` или `*U`. Лучшим решением в случае использования в качестве аргумента функции многомерного массива является указание массива со всеми его размерами.

#### АРГУМЕНТЫ ФУНКЦИИ `main()`

Иногда бывает полезно, а то и просто необходимо передать информацию в программу при ее вызове. Основной метод состоит в использовании аргументов командной строки операционной системы для передачи информации функции `main()`. Например, при вызове системы Borland C++ можно сразу задать файл, который будет загружен в активное окно редактирования:

```
c:\bc\bin> bc program.c
```

В языке C заданы два встроенных аргумента функции `main()`: `argc` и `argv`:

```
main(int argc, char *argv[]){...}
```

В системе Borland C++ предусмотрен еще и третий аргумент, который, так же как и второй, является указателем на массив строк:

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

Два первых аргумента используются для передачи аргументов командной строки. Третий аргумент используется для доступа к параметрам среды операционной системы. Эти три аргумента доступны только функции `main()`.

Параметр `argc` содержит количество аргументов командной строки и является параметром типа `int`. Он всегда не меньше единицы, так как имя программы, вызываемой для выполнения, трактуется как первый параметр командной строки.

Параметр `argv` является указателем на массив строк. Каждый элемент массива указывает на аргументы командной строки. Один параметр командной строки отделяется от другого пробелами. Если мы хотим в ка-

честве параметра иметь строку, содержащую пробелы, ее надо заключить в двойные кавычки. Если аргументом командной строки является число, то оно рассматривается как строка и должно быть преобразовано с использованием соответствующей функции, например `atoi()`, `atof()`, `atol()` и др.

Следующая программа иллюстрирует эти возможности:

```
#include <stdio.h>
#include <stdlib.h>
/* Пример 43 */
main(int argc, char argv[])
{
    if (argc != 2) {
        printf("Вы забыли набрать имя \n");
        exit(0);
    }
    printf("HELLO! %s \n", argv[1]);
}
```

Если эту программу создать в файле с именем NAME.EXE и в командной строке указать

> NAME Сергей

то она выдаст на экран сообщение

HELLO! Сергей!

Каждый параметр в командной строке должен отделяться от другого пробелом или символом табуляции. Если аргумент, который вы хотите передать функции, содержит пробелы, то надо заключить соответствующую строку в кавычки. В результате вызова предыдущей программы

> NAME "Вы прекрасно выглядите сегодня"

на экране появится сообщение

HELLO! Вы прекрасно выглядите сегодня!

Borland C++ допускает столько параметров функции `main()`, сколько позволяет операционная система. В MS DOS командная строка может содержать 128 символов.

Параметр `env` объявлен так же, как и `argv`, т. е. это указатель на массив строк, которые содержат установку среды. Последняя строка этого массива пустая. Поэтому можно выдать содержание командной строки и состояние среды:

```
#include <stdio.h>
/* Пример 44 */

void main(int argc, char *argv[], char *env[])
{
    int i;
    printf("Количество аргументов командной строки: %d \n", argc);
```

```

        printf("Аргументы командной строки\n");
    for(i=0; i < argc; i++)
        printf("%s\n", argv[i]);
    printf("Аргументы состояния среды\n");
    for(i=0; env[i]; i++)
        printf("%s\n", env[i]);
}

```

Аргументы `main()` позиционно зависимы. И если `env` может не объявляться (в случае, когда она не используется), то при ее использовании первые два аргумента должны быть объявлены вне зависимости от того, будут они использоваться или нет.

Функция `main()`, как и другие функции, может возвращать целое значение. Оно передается в вызывающий процесс. В наших программах это всегда операционная система MS DOS.

По принятому соглашению для операционной системы MS DOS возврат нулевого значения говорит об успешном окончании работы, другие значения говорят об аварийном окончании работы программы.

Проверить значение, возвращаемое функцией `main()`, можно командой MS DOS

```
> if errorlevel value
```

где `value` - конкретное значение.

Функция `main()` может и не возвращать значения, если она объявлена как

```
void main(void){...}
```

Параметры командной строки можно задать не выходя из интегрированной среды системы. Для этого надо выбрать в меню `Run|Arguments`. В открывшемся диалоговом окне можно задать командную строку. Имя выполняемой программы добавлять не надо.

### РЕКУРСИВНЫЕ ФУНКЦИИ

В языке C функции могут вызывать сами себя. Функция называется рекурсивной, если оператор в теле функции содержит вызов этой же функции. Классический пример рекурсивной функции - это вычисление факториала числа  $N! = 1*2*3*...*N$ .

Назовем эту функцию `factorial()`.

```

factorial(int n)
{
    int a;
    if (n == 1) return 1;
    a = factorial(n-1)*n;
    return a;
}

```

Вызов функции в рекурсивной функции не создает новую копию функции, а создает в памяти новые копии локальных переменных и пара-

метров. Из рекурсивной функции надо предусмотреть выход, иначе это вызовет "зависание" системы.

### ФУНКЦИИ С ПЕРЕМЕННЫМ ЧИСЛОМ ПАРАМЕТРОВ

В предыдущей главе уже встречались функции с переменным числом параметров, в частности `printf()` и `scanf()`. Можно определять свои функции с переменным числом параметров. Для поддержки этого механизма в языке C предусмотрены макросы, определенные в файлах `stdarg.h` и `varargs.h`. Файл `stdarg.h` используется в стандарте ANSI C, файл `varargs.h` - в стандарте UNIX SYSTEM V. Подключать оба файла одновременно нельзя.

Для описания функций с переменным числом параметров используется список с переменным числом параметров, содержащий по крайней мере один параметр и завершающийся многоточием (...), при этом запятая после последнего фиксированного параметра перед многоточием необязательна, например:

```
void f(int a, int b, ...);
```

или

```
void f(int a, int b...);
```

Язык C не предусматривает больше никаких средств для работы с переменным числом аргументов, но в файле `stdarg.h` находятся переносимые средства поддержки функции с переменным числом параметров:

```
тип      va_list
макрос   va_start (list, last_fixed)
макрос   va_arg(list, arg_type)
макрос   va_end (list)
```

Макрос `va_start` производит инициализацию механизма доступа к нефиксированным параметрам, `list` - переменная типа `va_list`, и `last_fixed` - последний фиксированный параметр (поэтому функции с переменным числом параметров должны иметь хотя бы один фиксированный параметр).

Последовательный доступ к нефиксированным параметрам осуществляется макросом `va_arg`, где первый параметр макроса - инициализированная вызовом `va_start` переменная типа `va_list`, а второй параметр - тип очередного параметра; `va_arg` возвращает значение этого параметра. Проверки соответствия типа не производятся, т. е. совпадение между типом в `va_arg` и тем, что было передано при вызове функции, должно контролироваться программистом.

Макрос `va_end` завершает работу с переменной типа `va_list`. Его следует использовать в целях переносимости на другие платформы и компиляторы, хотя никакого действия в Borland C++ он не производит.

Примеры использования функций с переменным числом параметров:

1.

```
int sum_1(int a, ...)
{
    va_list args;
    int result = a, t;
```

```

        va_start (args, a);
        while((t=va_arg(args, int) != 0)
            result += t;
    va_end (args);
    return result;
}
2.
int sum_2(int num_args, ...)
{
    va_list args;
    int result=0, i;
    va_start(args, num_args);
    for(i=0; i< numargs; i++)
        result += va_arg(args, int);
    va_end (args);
    return result;
}
3.
void error_message(char* fmt, ...)
{
    va_list (args);
    printf("error: ")
    va_start (args, fmt);
    vprintf (fmt, args);
    va_end (args);
    exit(1);
}

```

Здесь приведено три основных способа работы с переменным числом параметров. Основная проблема заключается в том, что ни тип передаваемых параметров, ни их количество неизвестны вызываемой функции. Существуют следующие подходы:

- Как в функции `printf()` строка формата полностью определяет тип и число параметров через спецификации формата.
- Можно зарезервировать какое-либо значение как завершающее (пример 1). Функция считает сумму чисел, завершающее число равно нулю.
- Можно передавать число параметров в качестве фиксированного параметра (пример 2).
- В файле `stdio.h` предусмотрено семейство функций `vprintf()`, `vsprintf()`, `vfprintf()`, `vscanf()`, `vsscanf()`, `vfscanf()`, которые можно использовать внутри других функций с переменным числом параметров. В качестве последнего параметра эти функции принимают инициализированную переменную типа `va_list`.

Как работает этот механизм. Передаваемые функции параметры располагаются в памяти непосредственно друг за другом. Тип `va_list` - это указатель на `void`. Макрос `va_start` устанавливает этот указатель за последним фиксированным параметром, а это можно сделать, так как его адрес и тип

известны. Макрос `va_arg`, используя приведение к типу, указанному в качестве второго параметра, возвращает значение, на которое указывает первый параметр, и передвигает указатель на размер переменной данного типа.

Эта реализация накладывает некоторые ограничения. В Borland C++ надо аккуратно использовать нефиксированные параметры `char`, `unsigned char`, так как они занимают в памяти 1 байт, а в стеке под них отводится 2 байта.

### УКАЗАТЕЛЬ НА ФУНКЦИЮ

На функцию, как и на другой объект языка C, можно создать указатель. Указатель `rfunc` на функцию, возвращающую значение типа `type` и имеющую параметры типа `type1`, `type2`, объявляется следующим образом:

```
type (*pfunc)(type1 t1, type2 t2);
```

По определению указатель на функцию содержит адрес первого байта или слова выполняемого кода функции. Над указателями на функцию запрещены арифметические операции. Использование указателей на функцию имеет несколько применений, в частности он используется, если необходимо передать функцию как параметр другой функции. Использование указателя на функцию иллюстрирует следующий модельный пример.

```
#include <stdio.h>
#include <math.h>
/* Пример 45. Использование указателя на функцию */
double f(double x);
double f1(double x);
double f2(double x);
double ff(double (* pf)(double x)); /* Указатель на функцию в качестве аргумента
функции */
main()
{
    double z=1.1, y;
    double (*ptrf)(double x); /* Указатель на функцию */
    ptrf=f; /* Инициализация указателя */

    y=(*ptrf)(z); /* Вызов функции через указатель - первый способ */
    printf("z=%f y=%f\n", z, y);
    y=ptrf(z); /* Вызов функции через указатель - второй способ */
    printf("z=%f y=%f\n", z, y);
    ptrf=sin; /* В качестве функции можно использовать стандартные
библиотечные функции */
    y=ptrf(z);
    printf(" z=%f sin(z)=%f\n", z, y);

    y=(*ptrf)(2.5); /* Можно вызвать функцию и так */
    printf(" z=%f y=%f\n", z, y);

    y=ff(f1); /* Вызов функции ff сп параметром f1 */
}
```

```

printf("z=%f y=%f\n", z, y);

y=f(cos);      /* Другой вызов с другой функцией */
printf("z=%f y=%f\n", z, y);

y=f(ptrf);     /* То же через указатель на функцию */
printf("z=%f y=%f\n", z, y);

return 0;
}
double f(double x)
{
    puts("In f()");
    return x;
}
double f1(double x)
{
    puts("In f1()");
    return x*x;
}
double f2(double x)
{
    puts("In f2()");
    return x*x*x;
}
double ff(double (* pf)(double x)) /* Функция с указателем
                                   на функцию в качестве параметра */
{
    printf(" In ff() %f\n", pf(3));
    return pf(3);
}

```

Так же как и для других типов переменных, можно использовать массив указателей на функцию. Такая структура данных называется jump table.

Если есть объявления

```

int f0(void);
int f1(void);
int f2(void);
int f3(void);
int (*jtable[])(void)={f0, f1, f2, f3};

```

Соответствующую функцию можно вызвать

```
val = (*jtable[i])();
```

или

```
val = jtable[i]();
```



## ТИПЫ ДАННЫХ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ

Мы уже познакомились с пятью базовыми типами данных. Дальнейшее развитие типов данных реализуется двумя способами. Первый способ - введение новых типов данных. Второй способ - применение различных модификаторов типа (type modifiers). Модификаторы типа могут быть применены к основным базовым типам. Таких модификаторов 4 вида:

- модификаторы доступа,
- модификаторы класса памяти,
- Borland C++ модификаторы типа функции,
- Borland C++ модификаторы моделей памяти.

Использование модификаторов типа в общем виде следующее:

модификатор\_типа спецификатор\_типа <список\_переменных>

т. е. модификатор ставится перед именем типа.

Модификаторы класса памяти auto, static, extern и register рассмотрены выше.

К модификаторам доступа относятся const и volatile.

Модификатор const появился в языке C после появления языка C++. Переменные, которые объявлены с модификатором const, не могут быть изменены в процессе работы программы. Хотя, конечно, они могут и должны быть инициализированы при создании. С помощью модификатора const создаются типизированные поименованные константы. Примеры объявления:

```
const float radius=15.5;
const char yes='y';
```

Одно из важных применений переменных типа const - защита аргументов функции от изменений, если аргумент передается по ссылке. Если аргументом функции является указатель на переменную, то возможно изменение значения этой переменной при вызове функции. Если же эту переменную описать как const, то функция не сможет изменить значение этой переменной.

```
#include <stdio.h>
/* Пример 46. Кодирование строки */
void code(const char *str);
main (void)
{
    code (" Это тест ");
    return 0;
}
void code(const char *str)
{
    while (*str) printf("%c", (*str++)+1);
}
```

Модификатор `volatile` используется в том случае, когда эту переменную могут изменять, например, системные функции операционной системы, т. е. переменная может меняться вне действия программы. Например, глобальная переменная, содержащая системное время компьютера.

Интересно, что модификаторы `volatile` и `const` могут использоваться одновременно:

```
const volatile unsigned char *port=0x30;
```

Это объявление может понадобиться, если предположить, что `0x30` значение порта процессора, который может изменяться только извне.

Следующий специфичный именно для Borland C++ вид модификаторов - это модификатор типа функции. Эти модификаторы являются особенностью систем программирования в операционной системе MS DOS. Их 3: `pascal`, `cdecl` и `interrupt`.

Модификаторы `pascal`, `cdecl` сообщают компилятору, что передача параметров функции должна быть организована соответственно как в языке C или Pascal. Необходимость в этом возникает если вы хотите программу, написанную на языке C, использовать вместе с программами на языке Pascal.

```
pascal factorial(register int n);
{
    register int i=1;
    for (;n;n--) i*=n;
    return i;
}
```

Если же вы хотите все функции, кроме одной, компилировать с передачей параметров, как в Pascal, то можно в интегрированной среде установить соответствующую опцию. Для функции, параметры которой вы хотите передавать так, как принято в C, установите перед типом этой функции модификатор `cdecl`.

Модификатор `interrupt` сообщает компилятору, что функция с этим модификатором будет осуществлять обработку прерываний. Обычно этот модификатор используется при создании резидентных программ. Создание резидентных программ не входит в задачу данного издания. С резидентными программами можно ознакомиться в книге Фролов А. В., Фролов Г. В. Библиотека системного программиста т.1.

## **ДИНАМИЧЕСКОЕ РАСПРЕДЕЛЕНИЕ ПАМЯТИ.**

### **ФУНКЦИИ `MALLOC()` И `FREE()`**

Принятое распределение памяти в языке C следующее:

Верхние адреса

СТЕК
СВОБОДНАЯ ПАМЯТЬ
РАЗДЕЛ ГЛОБАЛЬНЫХ ПЕРЕМЕННЫХ И КОНСТАНТ
КОД ПРОГРАММЫ

Нижние адреса

Для глобальных переменных отводится фиксированное время в памяти на все время работы программы. Локальные переменные хранятся в стеке. Между ними находится область свободной памяти для динамического распределения. Наиболее важными функциями для динамического распределения памяти являются `malloc()` и `free()`. Они используются для динамического распределения свободной памяти. Функция `malloc()` выделяет память, а функция `free()` освобождает ее. Прототипы этих функций хранятся в заголовочном файле `STDLIB.H` и имеют вид

```
void *malloc(size_t size);
void *free(void *p);
```

Функция `malloc()` возвращает указатель типа `void`, для правильного использования значение этой функции надо преобразовать к указателю на соответствующий тип. При успешном выполнении `malloc()` возвращает указатель на первый байт свободной памяти требуемого размера. Если достаточного количества памяти нет, то возвращается значение 0 (нулевой указатель). Чтобы определить количество байт, необходимых для переменной, используют операцию `sizeof`.

Действие функции `free()` обратно действию функции `malloc()`. Эта функция освобождает ранее выделенную область памяти. После этого память может снова использоваться функцией `malloc()`.

Пример использования этих функций:

```
#include <stdio.h>
#include <stdlib.h>
/* Пример 47. Динамическое выделение памяти */
main (void)
{
    int *p, t;
    p=malloc(40*sizeof(int)); /*выделение памяти для 40 целых чисел*/
    if(!p) {
        printf ("Недостаточно памяти \n");
        exit(1);
    }

    for(t=0;t<40;++t) *(p+t)=t;          /* Использование памяти */
    for(t=0;t<40;++t) printf("%d", *(p++));
```

```

    free(p);      /* Освобождение памяти */
    return 0;
}

```

Перед использованием указателя, возвращаемого `malloc()`, необходимо убедиться, что памяти достаточно.

Динамическое распределение памяти удобно использовать тогда, когда заранее неизвестно количество используемых переменных. В частности, этот механизм используется для создания массивов с изменяемым количеством элементов.

## НЕЛОКАЛЬНЫЙ ПЕРЕХОД

Оператор `goto` позволяет осуществлять локальный безусловный переход, т. е. переход внутри одной функции. Обычно он используется при выходе из глубоко вложенных операторов.

В языке C предусмотрен и нелокальный переход, который позволяет перейти из одной функции в другую, конечно при достаточно жестких условиях. Нелокальный переход осуществляется при помощи функций, определенных в заголовочном файле `setjmp.h`, входящем в состав стандартных заголовочных файлов языка C.

### Функция

```
int setjmp(jmp_buf jmpb);
```

сохраняет в переменной `jmpb` полное состояние задачи, необходимое для возврата функцией `longjmp()` в точку вызова `setjmp()`. При вызове `setjmp()` возвращает значение 0.

### Функция

```
void longjmp(jmp_buf jmpb, int retval);
```

возвращает управление (восстанавливая состояние задачи) в точку вызова функции `setjmp()` с параметром `jmpb`. В этом случае функция возвращает значение `retval`. Это позволяет различать ситуации, когда `setjmp()` сохранила состояние задачи и когда управление было возвращено функцией `longjmp()`.

Рассмотрим модельный пример использования функций нелокального перехода: обработка ошибок внутри вложенных функций.

```

#include <setjmp.h>
#include <stdio.h>
#include <stdlib.h>
/* Пример 48. Использование нелокального перехода */

#define OUT_OF_MEMORY 1
#define FILE_ERROR 2
/*...*/
/* Не должно быть кодов ошибок с нулевым кодом, так как setjmp()
   возвращает значение 0 при вызове

```

```

*/

jmp_buf err_buf;

void work(void);
int work1(void);
int work2(void);
/*...*/
int workN(void);

main()
{
    puts("Вызываем work()");
    work(); /* Вызов функции, содержащей вызов setjmp() */
    puts("Конец функции main()");
    return 0;
}

void work(void)
{
    int result;
    if((result = setjmp(err_buf))!=0)
    {
        switch(result){
            case OUT_OF_MEMORY:
                puts("Out of memory ");
                return;
            case FILE_ERROR:
                puts("Cannot open file");
                return;
        }
        /*...*/
    }
    printf("Result is %d \n", work1());
}

int work1(void)
{
    /*...*/
    work2(); /*Вложенный вызов */
    /*...*/
}

int work2(void)
{
    void* p;
    /*...*/
    if((p=malloc(4096))==NULL)
        longjmp(err_buf, OUT_OF_MEMORY);
}

```

```

        /*Ошибка - недостаточно памяти. Возвращаем управление в точку
        вызова setjmp() с кодом ошибки OUT_OF_MEMORY */
    work3();
    /*...*/
    return 1;
}

/*, , , */

int workN(void)
{
    FILE* in;
    /*, , , */
    if((in=fopen("DATA001.TXT", "r"))==NULL)
        longjmp(err_buf, FILE_ERROR);
    /*, , , */
    return 2;
}

```

Преимущество функций нелокального перехода состоит в том, что они позволяют передавать управление за пределы функции и позволяют передавать информацию в точку возврата в виде целого числа (retval).

Что имеется в виду, когда говорят, что функция `setjmp()` сохраняет полное состояние задачи? Имеется в виду, что сохраняется информация, позволяющая продолжить выполнение программы с данной точки. Эта информация зависит от операционной системы, типа компьютера и компилятора. В Borland C++ под этим понимается содержимое регистров и флагов процессора, а именно CS, IP, SS, SP, BP, ES, DS, SI, DI, FLAGS. Достаточно посмотреть поля структуры `jmp_buf`, определенной в файле `setjmp.h`. Функция `setjmp()` заполняет поля этой структуры, а функция `longjmp()` восстанавливает значения регистров из этой структуры.

При использовании функций нелокального перехода есть ограничения:

- в состояние задачи не входят значения переменных;
- их нельзя использовать в оверлейных программах;
- до вызова `longjmp()` переменная `jmpb` должна быть инициализирована функцией `setjmp()`;
- при вызове `longjmp()` функция, в которой был вызов `setjmp()`, должна быть активна (т. е. из нее нельзя выходить). При выходе из функции информация в стеке будет разрушена, и результат после возврата управления функцией `longjmp()` будет непредсказуем.

## ТИПЫ, ОПРЕДЕЛЯЕМЫЕ ПОЛЬЗОВАТЕЛЕМ

Кроме известных нам типов данных язык C позволяет создавать еще 5 типов данных:

- структуры (structure),

- объединения (union),
- перечислимый тип (enumeration),
- поля битов (bit fields),
- 5-я возможность - с помощью оператора `typedef` создать новое имя (псевдоним) для уже существующего типа.

### СТРУКТУРА

Первый из новых для нас типов - структура. Структура объединяет несколько переменных, возможно разного типа. Переменные, которые объединены структурой, называются членами, элементами или полями структуры.

Пример определения структуры:

```
struct student {
    char name[30];
    int kurs;
    char group[3];
    int stip;
};
```

Объявление структуры является оператором, и поэтому в конце должна стоять точка с запятой. При этом пока никакая переменная не объявлена. Выделения памяти под переменную не произошло. Под именем `student` задан частный вид структуры; говорят, что задан шаблон структуры и определен новый тип `struct student`. Для того чтобы объявить конкретные переменные типа `struct student`, можно написать

```
struct student stud1, stud2;
```

Теперь объявлены две переменные - `stud1` и `stud2`. Компилятор автоматически выделит под них место в памяти компьютера. Под каждую из переменных типа структуры выделяется непрерывный участок памяти. Задание шаблона структуры и объявление переменных может производиться и в одном операторе:

```
struct student {
    char name[30];
    char kurs;
    char group[3];
    int stip;
} stud1, stud2;
```

Здесь одновременно задается структура с именем `student` и объявляются переменные `stud1` и `stud2`.

Доступ к конкретному элементу структуры осуществляется с помощью операции "точка" (`dot`). Например,

```
strcpy(stud1.name, "Иванов М. С.");
```

Если мы хотим напечатать содержимое третьего поля переменной `stud2` структуры `student`, мы должны написать



```
printf("%s", stud2.group);
```

Структуры, как и переменные другого типа, могут объединяться в массивы структур. Чтобы объявить массив структур, надо сначала задать шаблон структуры (у нас уже есть шаблон `student`), а затем объявить массив:

```
struct student stud1kurs[200];
```

Этот оператор создаст в памяти 200 переменных типа структуры с шаблоном студент и именами `stud1kurs[0]`, `stud1kurs[1]` и т. д.

Для доступа к полю `kurs` 25-го элемента массива используем

```
stud1kurs[24].kurs
```

Если же мы хотим взять 5-й элемент поля `name` переменной `stud1`, необходимо написать

```
stud1.name[4]
```

Если объявлены две переменные типа структуры с одним шаблоном, можно сделать присваивание

```
stud1 = stud2;
```

при этом произойдет побитовое копирование каждого поля одной переменной в соответствующее поле другой структуры. В то же время нельзя использовать операцию присваивания переменных типа структуры, шаблоны которых описаны под разными именами, пусть даже совсем идентично:

/\* Пример 49.

пример неправильный ! \*/

```
main(void)
```

```
{
```

```
struct first {
```

```
int a;
```

```
char b;
```

```
};
```

```
struct second {
```

```
int a;
```

```
char b;
```

```
};
```

```
    struct first a;
```

```
    struct second b;
```

```
    first a = 1;
```

```
    first.b = 'f';
```

```
        second.a = first.a; /* правильное присваивание */
```

```
    second = first; /* неправильное присваивание */
```

```
    return 0;
```

```
}
```

Переменная типа структуры может быть глобальной, локальной переменной и формальным параметром. Можно, естественно, использовать

структуру или ее элемент структуры как любую другую переменную в качестве параметра функции.

Например:

```
func1(first.a); или func2(&second.b);
```

Заметим, что & ставится перед именем структуры, а не перед именем поля.

Можно в качестве формального параметра передать по значению всю структуру:

```
#include <stdio.h>
/* Пример 50. Использование структуры в качестве параметра
функции */
struct stru {
    int x;
    char y;
};
void f(struct stru param); /* прототип */
main (void)
{
    struct stru arg;
    arg.x=1;
    arg.y='2';
    f(arg);
    return 0;
}
void f(struct stru param)
{
    printf(" %d %d \n", param.x, param.y);
}
```

Можно также создать указатель на структуру и передавать аргумент типа структуры по ссылке. Объявить указатель на структуру можно следующим образом:

```
struct stru *adr_pointer;
adr_pointer - переменная типа указатель на структуру struct stru.
```

Если мы передаем структуру по значению, то все элементы структуры заносятся в стек. Если структура простая и содержит мало элементов, то это не так страшно. Если же структура в качестве своего элемента содержит массив, то стек может переполниться. При передаче по ссылке в стек занесется только адрес структуры. При этом копирования структуры не происходит, а также появляется возможность изменять содержимое элементов структуры.

```
struct complex {
    float x;
    float y;
} c1, c2;
struct complex *a; /* объявление указателя */
```

```
a = &c1;
```

указателю `a` присвоится адрес переменной `c1`. Получить значение элемента `x` переменной `c1` можно так:

```
(*a).x;
```

Использование указателей на структуру встречается часто. Поэтому, кроме способа получить значение элемента `a` структуры, используя `(*a).x`, есть еще один. В языке C вводится специальная операция `->` (стрелка, *arrow*).

Операция стрелка употребляется вместо операции точка, когда мы хотим использовать значение элемента структуры с применением переменной указателя.

Вместо `(*a).x` мы можем использовать `a -> x`. Этот способ обычно и применяется.

Чтобы завершить разговор о структурах, надо добавить, что в качестве элементов структуры можно использовать массивы, структуры и массивы структур.

Пусть объявления переменных имеют вид

```
struct addr {  
    char city[30];  
    char street[30];  
    int house;  
};  
struct fulladdr {  
    struct addr address;  
    int room;  
    char name[30];  
} f, g;
```

Здесь `addr` - шаблон структуры, определенный перед объявлением структуры `fulladdr` и объявлением переменной `f` типа структуры `fulladdr`. Для присвоения значения элементу `house` структуры `address` переменной `f` используем

```
a.address.house=101;
```

### Доступ к отдельному биту

В отличие от других языков программирования язык C обеспечивает доступ к одному или нескольким битам в байте или слове. Это имеет свои преимущества. Если многие переменные принимают только два значения, такие переменные иногда называют флагами (например, логические), можно использовать 1 бит.

Один из методов, встроенных в язык C и позволяющих иметь доступ к биту, - это поля битов (*bit-fields*). В действительности поля битов - это специальный тип членов структуры, в котором определено, из скольких бит

состоит каждый элемент. Основная форма объявления такой структуры следующая:

```
struct имя_структуры {
    тип    имя1: длина_в_битах;
    тип    имя2: длина_в_битах;

    ...

    тип    имяN: длина_в_битах;
};
```

В этом объявлении структуры тип может быть одним из следующих: `int`, `unsigned` или `signed`.

ИмяI может быть пропущено, тогда соответствующее количество бит не используется (пропускается). Длина структуры всегда кратна восьми. Так, если указать

```
struct onebit{
    unsigned one_bit: 1;
} obj;
```

То для переменной `obj` будет выделено 8 бит, но использоваться будет только первый.

В структуре могут быть смешаны обычные переменные и поля битов.

### Объединения (UNION)

В языке C определен еще один тип для размещения в памяти нескольких переменных разного типа. Это объединение. Объявляется объединение так же, как и структура, например:

```
union u {
    int i;
    char ch;
    long int l;
};
```

Это объявление не задает какую-либо переменную. Оно задает шаблон объединения. Можно объявить переменную

```
union u alfa, beta;
```

Можно было объявить переменные одновременно с заданием шаблона. В отличие от структуры для переменной типа `union` места в памяти выделяется ровно столько, сколько надо элементу объединения, имеющему наибольший размер в байтах. В приведенном выше примере под переменную `alfa` будет выделено 4 байта памяти. В самом деле, элемент `i` требует 2 байта, элемент `с` - 1 байт, и элемент `l` - 4 байта. Остальные переменные будут располагаться в том же месте памяти. Синтаксис использования элементов объединения такой же, как и для структуры:

```
u.ch = '5';
```

Для объединений также разрешена операция `->`, если мы обращаемся к объединению с помощью указателя.

Программа, приведенная ниже, выдает на экран двоичный код ASCII символа, вводимого с клавиатуры:

```
#include <stdio.h>
#include <conio.h>
/* Пример 51. Использование полей битов и объединений */

struct byte {
    int b1: 1;
    int b2: 1;
    int b3: 1;
    int b4: 1;
    int b5: 1;
    int b6: 1;
    int b7: 1;
    int b8: 1;
}; /* Определена структура - битовое поле */

union bits {
    char ch;
    struct byte b;
} u; /* Определено объединение */

void decode (union bits b); /*прототип функции*/
main (void)
{
    do {
        b.ch=getche();
        printf(": ");
        decode (u);
    } while (u.ch != 'q');
    return 0;
}

void decode (union bits b)
{
    if (b.bit.b8) printf("1");
    else printf("0");
    if (b.bit.b7) printf("1");
    else printf("0");
    if (b.bit.b6) printf("1");
    else printf("0");
    if (b.bit.b5) printf("1");
    else printf("0");
    if (b.bit.b4) printf("1");
    else printf("0");
    if (b.bit.b3) printf("1");
    else printf("0");
    if (b.bit.b2) printf("1");
    else printf("0");
    if (b.bit.b1) printf("1");
```

```

    else printf("0");
    printf("\n");
}

```

### ПЕРЕЧИСЛИМЫЙ ТИП

Перечислимый тип (enumeration) - это множество поименованных целых констант. Перечислимый тип определяет все допустимые значения, которые могут иметь переменные этого типа. Основная форма объявления типа следующая:

```
enum имя_типа {список_названий} список переменных;
```

Список переменных может быть пустым. Пример определения перечислимого типа и переменной данного типа:

```
enum seasons { win, spr, sum, aut };
enum seasons s;
```

Ключом к пониманию сущности перечислимого типа является то, что каждое из имен win, spr, sum и aut представляет собой целую величину. Если эти величины не определены по-другому, то по умолчанию они соответственно равны нулю, единице, двум и трем. Оператор

```
printf ("%d %d", win, aut);
```

выдаст на экран числа 0 и 3. Во время объявления типа можно одному или нескольким символам присвоить другие значения, например:

```
enum value {one=1, two, three, ten=10, thousand=1000, next};
```

Если теперь напечатать значения

```
printf ("%d %d %d %d %d\n", one, two, ten, thousand, next);
```

то на экране появятся числа 1 2 10 1000 1001, т. е. каждый следующий символ увеличивается на единицу по сравнению с предыдущим, если нет другого присваивания.

С переменными перечислимого типа можно производить следующие операции:

- присвоить переменную типа enum другой переменной того же типа;
- провести сравнение с целью выяснения равенства или неравенства;
- арифметические операции с константами типа enum ( $i = \text{win} - \text{aut}$ ).

Нельзя использовать арифметические операции и операции ++ и -- для переменных типа enum.

Основная причина использования перечислимого типа - это улучшение читаемости программ.

### ПЕРЕИМЕНОВАНИЕ ТИПОВ - TYPEDEF

Язык C позволяет, кроме того, дать новое название уже существующим типам данных. Для этого используется ключевое слово typedef. При этом не создается новый тип данных.

Например:

```
typedef char SYMBOL;
typedef unsigned UNSIGN;
typedef float real;
```

Достаточно часто используется оператор typedef с применением структур:

```
typedef struct st_tag{
    char name[30];
    int kurs;
    char group[3];
    int stip;
} STUDENT;
```

Теперь для определения переменной можно использовать

```
struct st_tag avar;
```

а можно использовать

```
STUDENT avar;
```

## МОДЕЛИ ПАМЯТИ

Особенности операционной системы MS DOS требуют дополнительных уточнений относительно распределения памяти компьютера.

Система Borland C++ предусматривает работу с шестью моделями памяти:

tiny, small, medium, compact, large и huge.

Выбор модели определяется требованиями программы. Каждая модель памяти соответствует определенному типу программы и размеру кодов. Микропроцессор INTEL 80 x 86 при работе в реальном режиме имеет сегментированную архитектуру памяти.

Вопросы работы в защищенном режиме процессора в данной книге не затрагиваются. Как правило, компьютер имеет 1 Мбайт памяти, но размер адреса 2-го байта позволяет адресовать 64 Кбайт памяти

$$2^{16} = 2^6 * 2^{10} = 2^6 * 1 \text{ Кбайт} = 64 \text{ Кбайт}$$

Такой участок памяти называется сегментом. Процессор позволяет работать одновременно с четырьмя разными сегментами:

- сегментом кода,
- сегментом данных,
- сегментом стека,
- вспомогательным сегментом.

Полный адрес состоит из двух 16 битовых значений: адреса сегмента и смещения. Адресные регистры процессора имеют 20 разрядов:

$$2^{20} = 2^{10} * 2^{10} = 2^{10} * 1 \text{ Кбайт} = 1 \text{ Мбайт},$$

и позволяют адресовать 1 Мбайт памяти.

Какое отношение имеют указатели к моделям памяти Borland C++? Самое непосредственное. Тип выбранной модели памяти определяет по умолчанию тип указателей, используемых для кода и данных. Указатели бывают трех видов:

near - 16 бит,  
far - 32 бита,  
huge - 32 бита.

**Tiny** - это минимальная из моделей памяти. Общий размер кода, данных и стека 64 Кбайт. Все указатели ближние (near).

**Small** - эта модель хорошо подходит для небольших прикладных программ. Используются только ближние указатели. Общий объем памяти 64 Кбайт для кода и 64 Кбайт для данных и стека.

**Medium** - годится для больших программ, для которых не требуется держать в памяти большой объем данных. Для кода, но не для данных, используются дальние указатели (far)

Объем памяти данных и стека ограничен 64 Кбайт, а программа может занимать до 1 Мбайт.

**Compact** - лучше всего использовать в тех случаях, когда размер кода невелик, но требуется адресация большого объема данных. Дальние указатели используются для данных (1 Мбайт), но не для кода (64 Кбайт).

**Large** и **Huge** - эти модели памяти применяются только для очень больших программ. И для кода, и для данных используются дальние указатели. Модель памяти huge отменяет ограничения размера памяти, выделяемой под статические данные. Обычно объем такой памяти не более 64 Кбайт.

Для выбора модели памяти надо либо воспользоваться соответствующей опцией компилятора командной строки, либо установить соответствующую опцию в интегрированной среде.

Borland C++ вводит 8 новых ключевых слов, которые могут использоваться как модификаторы при объявлении указателей, а также функций. Эти ключевые слова следующие:

near, far, huge, \_cs, \_ds, \_es, \_ss, \_seg.

Функции и указатели бывают ближними или дальними по умолчанию, в зависимости от выбранной модели памяти. Можно, однако, указать модификатор перед типом указателя. Так, все функции графической системы Borland C++ указаны как far (дальние).

## ПРЕПРОЦЕССОР ЯЗЫКА C

Препроцессор языка C - это программа, выполняющая обработку входных данных для другой программы. Препроцессор языка C просматривает программу до компилятора, заменяет аббревиатуры в тексте программы на соответствующие директивы, отыскивает и подключает необходимые файлы, может влиять на условия компиляции. Директивы препро-



цессора языка C в действительности не являются частью языка C. Препроцессор включает в себя следующие директивы:

#define	Определение макроса
#undef	Отмена определения макроса
#include	Включение объекта - заголовка
#if	Компиляция, если выражение истинно
#ifdef	Компиляция, если макрос определен
#ifndef	Компиляция, если макрос не определен
#else	Компиляция, если выражение в if ложно
#elif	Составная директива else/if
#endif	Окончание группы компиляции по условию
#line	Замена новым значением номера строки или имени исходного файла
#error	Формирование ошибок трансляции
#pragma	Действие определяется реализацией
#	null-директива
Операции	Назначение
#	Придание операнду формы строки символов
##	Склеивание лексем
defined	Представление #ifdef в форме выражения

Все директивы препроцессора начинаются с символа #.

#### ДИРЕКТИВА #DEFINE

Директива #define вводит макроопределение или макрос. Общая форма директивы следующая:

```
#define имя_макроса последовательность_символов
```

Последовательность символов называют еще строкой замещения. Когда препроцессор находит в исходном тексте программы имя\_макроса (в дальнейшем будем говорить просто макрос), он заменяет его на последовательность\_символов. Любые вхождения макроса, обнаруженные в строках символов, символьных константах или в комментариях, замене не подлежат.

Можно отменить определение макроса директивой #undef:

```
#undef имя_макроса
```

Данная строка удаляет любую ранее введенную строку замещения. Определение макроса теряется и имя\_макроса становится неопределенным.

К примеру можно определить MAX как величина 100:

```
#define MAX 100
```

Это значение будет подставляться каждый раз вместо макроса MAX в исходном файле. Можно также использовать макрос вместо строковой константы:

```
#define NAME "Turbo C++ v. 1.01"
```

Если последовательность символов в директиве `#define` не помещается на одной строке, то можно поставить в конце строки символ `\` и продолжать последовательность на другой строке:

```
#define STRING "Эта последовательность символов не \
                умещается на одной строке, мы используем две"
```

Среди программистов на языке C принято соглашение, что для имени макроса используются, как правило, идентификаторы из прописных букв. Это соглашение позволяет легко находить в тексте программы макросы. Также лучше все директивы `#define` выносить в начало файла или в отдельный заголовочный файл.

Наиболее часто директива `#define` используется для определения имен "магических чисел", которые встречаются в программе. Например, если вы задаете один размер массивов в программе и в некоторых подпрограммах используете массивы того же размера, то предпочтительно определить размер массива директивой `#define` и использовать в объявлениях массивов этот макрос.

Например:

```
#define MAX 100
float balance[MAX], saldo[MAX];
```

Директива `#define` имеет еще одну важную особенность: макрос может иметь аргументы. Каждый раз, когда происходит замена, аргументы также заменяются на те, которые встречаются в программе. Иллюстрацией может служить следующий пример:

```
#define MIN(a,b) ((a)<(b))?(a):(b)
...
printf("Минимум из x и y %d, MIN(x,y));
printf("Минимум из a и b %d, MIN(m,n));
...
```

Когда программа будет компилироваться, в выражение, определенное `MIN(a, b)` будут подставлены соответственно `x` и `y` или `m` и `n`. Аргументы `a` и `b` заключены в круглые скобки, так как вместо них может подставляться некоторое выражение, а не просто идентификаторы, например

```
printf("Минимум %d, MIN(x*x,x*x*x));
```

Директива `#error` имеет вид

```
#error сообщение_об_ошибке
```

Эта директива прекращает компиляцию программы и выдает сообщение

```
Error: имя_файла номер_строки: error directive:
сообщение_об_ошибке
```

Сообщение об ошибке не заключается в кавычки. Данная директива обычно встраивается в условные конструкции препроцессора.

Директива `#include` подключает к исходному коду заданные в директиве файлы. Эти файлы называются подключаемыми, заголовочными файлами

или заголовками. Часто в качестве подключаемых файлов используются заголовочные файлы библиотек языка C. Форма этой директивы может быть трех видов:

```
#include <имя_заголовка>,  
#include "имя_заголовка",  
#include имя_макроса.
```

Третья форма директивы предполагает, что первым символом после пробела не будут символы < или "; кроме того, предполагается, что существует макроопределение, которое заменит имя макроса либо на <имя\_заголовка>, либо на "имя\_заголовка".

Имя заголовка должно быть допустимым именем файла DOS с расширением (традиционно заголовочные файлы имеют расширение.h, в языке C++ иногда встречается .hpp). Различие между форматами <имя\_заголовка> и "имя\_заголовка" состоит в алгоритме поиска подключаемого файла.

Вариант <имя\_заголовка> задает стандартный включаемый файл. Поиск последовательно производится во всех включаемых директориях, в той последовательности, в которой они определены. Если ни в одной из этих директорий файл не найден, то выдается сообщение об ошибке.

Вариант "имя\_заголовка" задает включаемый файл, созданный пользователем. Сначала он ищется в текущей директории, а если там не найден, то поиск продолжается во всех включаемых директориях так же, как и в первом случае.

### ДИРЕКТИВЫ УСЛОВНОЙ КОМПИЛЯЦИИ

Следующие директивы позволяют проводить выборочную компиляцию вашей программы. Этот процесс называется условной компиляцией.

К директивам условной компиляции относятся

#if, #else, #elif и #endif.

Если выражение, следующее за #if, истинно, то коды, заключенные между #if и #endif, будут компилироваться. В противном случае они будут при компиляции пропущены. Выражение, следующее за #if, проверяется во время компиляции, поэтому оно может содержать только константы и макросы, которые прежде определены. Переменные не могут использоваться.

Директива #else используется так же, как else в языке C.

Приведем простой пример использования #if и #else.

```
/* Пример 52. Использование условной компиляции */  
#include <stdio.h>  
#define MAX 100  
  
main (void)  
{  
    #if MAX>99  
        printf("MAX больше 99 \n");  
    #else
```

```
printf("MAX равно %d \n", MAX);  
#endif  
return 0;  
}
```

Директива `#elif` используется для организации вложенной условной компиляции. Основная форма использования директивы следующая:

```
#if выражение  
последовательность операторов  
#elif выражение1  
последовательность операторов  
#elif выражение2  
последовательность операторов  
...  
#elif выражениеN  
последовательность операторов  
#endif
```

Другой метод условной компиляции состоит в использовании директив `#ifdef` и `#ifndef`.

Основная форма использования директивы

```
#ifdef имя_макроса  
последовательность операторов  
#endif
```

и соответственно

```
#ifndef имя_макроса  
последовательность операторов  
#endif
```

Если макрос определен, то при использовании `#ifdef` компилируется соответствующая последовательность операторов до `#endif`. Если же макрос не определен или был отменен директивой `#undef`, то соответствующая последовательность операторов компилятором игнорируется.

Директива `#ifndef` действует противоположным образом по отношению к директиве `#ifdef`.

С директивами `#ifdef` и `#ifndef` можно использовать директиву `#else`, однако директиву `#elif` использовать нельзя.

Чтобы иметь возможность проверять, определены ли комбинации макросов в препроцессоре, определена операция `defined`. Она допустима только в директивах `#if` и `#elif`. Выражение `defined(имя_макроса)` принимает значение 1("истинно"), если макрос определен, и 0("ложно"), если макрос не определен. Директивы

```
#if defined(MAX)
```

и

```
#ifdef MAX
```

дают один и тот же результат. Преимущество состоит в возможности использования, например, таких директив:

```
#if defined(MAX)&&defined(MIN)
```

Макрос, определенный как имеющий пустое (NULL) значение, считается определенным.

После `#define MAX` макрос MAX считается определенным, но его значение неизвестно.

Директива `#line` используется для изменения содержания предопределенных макросов `__LINE__` и `__FILE__`. Основная форма директивы следующая:

```
#line целая_константа "имя_файла"
```

Директива `#line` служит для задания программе способа нумерации строк и используется при отладке программ. Эта директива игнорируется при работе в интегрированной среде, но используется при компиляции с помощью компилятора командной строки `tcc`.

Директива `#pragma` позволяет использовать специфичные для конкретных реализаций директивы в форме

```
#pragma имя_директивы
```

При помощи `#pragma` Borland C++ позволяет определить любые желаемые директивы, не обращая при этом к другим, поддерживаемых их компилятором. Если компилятор не поддерживает данное имя директивы, то он просто игнорирует директиву `#pragma`.

Borland C++ поддерживает следующие имена директив:

- `argused`,
- `exit`,
- `startup`,
- `inline`,
- `option`,
- `saveregs`,
- `warn`.

Использование директивы `#pragma` требует больше знаний, чем мы используем в этой книге, поэтому приведем лишь одно применение:

```
#pragma inline
```

Эта директива сообщает компилятору, что программа содержит встроенные ассемблерные коды. Эта директива эквивалентна опции компилятора командной строки `-B` или соответствующей опции интегрированной среды.

Более подробную информацию можно найти в документации по Borland C++.

## ПРЕДОПРЕДЕЛЕННЫЕ МАКРОСЫ

Стандарт ANSI языка C определяет 5 макросов:

\_\_LINE\_\_  
\_\_FILE\_\_  
\_\_DATE\_\_  
\_\_TIME\_\_  
\_\_STDC\_\_

В дополнение к ним Borland C++ определяет еще 12 макросов:

\_\_CDECL\_\_  
\_\_COMPACT\_\_  
\_\_HUGE\_\_  
\_\_LARGE\_\_  
\_\_MEDIUM\_\_  
\_\_MSDOS\_\_  
\_\_PASCAL\_\_  
\_\_SMALL\_\_  
\_\_TINY\_\_  
\_\_TURBOC\_\_  
\_\_cplusplus\_\_  
\_\_OVERLAY\_\_  
\_\_DLL\_\_  
\_\_Windows\_\_

Макрос `__DATA__` содержит строку месяц/день/год, которая представляет собой дату трансляции программы в объектный код.

Макрос `__TIME__` - это время начала компиляции программы в виде строки часы:минуты:секунды.

Макрос `__STDC__` имеет значение 1, если компиляция программы производилась с включенной опцией ANSY Keyword Only. В противном случае макрос не определен.

Если программа содержит оверлейные функции, то макрос `__OVERLAY__` принимает значение 1, в противном случае макрос не определен.

Макрос `__MSDOS__` принимает значение 1, если используется версия TurboC в системе MS DOS, в противном случае макрос не определен.

Макрос `__TURBOC__` содержит строку с номером версии Turbo C или Turbo C++. Номер версии представлен в шестнадцатеричном виде.

Макрос `__BCPLUSPLUS__` определен, если выбран C++ компилятор

Макрос `__BORLANDC__` содержит номер версии компилятора orland C++.

Макросы `__CDECL__` и `__PASCAL__` принимают соответственно значение 1 или "не определен" в зависимости от способа передачи параметров функций, используемого при компиляции программы.

Если программа компилировалась на языке C++, то макрос `__cplusplus__` определен. В противном случае он не определен.

Макрос `__DLL__` принимает значение 1, если генерируется код для Windows DLL, иначе - не определен.

Макрос `__Windows__` определен по умолчанию.

Только один из следующих макросов определен в зависимости от используемой модели памяти:

```
__TINY__
__SMALL__
__COMPACT__
__MEDIUM__
__LARGE__
__HUGE__
```

Пример использования макросов:

```
#include <stdio.h>
/* Пример 53. Использование директив препроцессора */
main (void)
{
    printf("%s%s%s%s\n", __FILE__, __LINE__, __DATE__,
        __TIME__);
    printf("Используется версия v.%x TurboC++ \n",
        __TURBOC__);
    return 0;
}
```

## СТАНДАРТНЫЕ ЗАГОЛОВОЧНЫЕ ФАЙЛЫ

Каждая библиотечная функция, определенная стандартом языка C, имеет прототип в соответствующем заголовочном файле. В соответствии со стандартом языка ANSI языка C 15 следующих заголовочных файлов должны быть обязательно:

Заголовочный файл	Назначение
assert.h	Диагностика программ
ctype.h	Преобразование и проверка символов
errno.h	Проверка ошибок
float.h	Работа с числами с плавающей точкой
limits.h	Определение размеров целочисленных типов
locale.h	Поддержка интернациональной среды
math.h	Математическая библиотека
setjmp.h	Возможности нелокальных переходов
signal.h	Обработка сигналов
stdarg.h	Поддержка функций с неопределенным числом параметров
stddef.h	Разное
stdio.h	Библиотека стандартного ввода/вывода
stdlib.h	Функции общего назначения
string.h	Функции работы со строками символов
time.h	Функции работы с датами и временем

На самом деле каждый из компиляторов содержит, как правило, больше заголовочных файлов. Компилятор Borland C++ содержит библиотеки, ориентированные на работу с MSDOS и WINDOWS, дополнительные библиотеки языка C++ и некоторые другие.

Описание всех функций Borland C++, макросов и системных переменных можно найти в документации к системе, а именно в Library Reference.

## БИБЛИОТЕКИ ВВОДА/ВЫВОДА И РАБОТА С ФАЙЛАМИ В ЯЗЫКЕ C

Операции ввода/вывода в языке C организованы посредством библиотечных функций. Мы уже использовали некоторые из этих функций: `printf()`, `scanf()`, `getche()` и др.

Нужно сказать, что система Borland C++ следует стандарту ANSI, называемому также буферизованным (*buffered*) или форматированным (*formatted*) вводом/выводом.

В то же время система Borland C++ поддерживает и другой метод ввода/вывода, так называемый UNIX-подобный, или неформатированный (небуферизованный) ввод/вывод.

Мы уделим главное внимание первому методу - стандарту ANSI.

Язык C++ поддерживает еще и собственный объектно-ориентированный ввод/вывод. Мы обязательно вернемся к нему позже. Однако знание стандартного для C ввода/вывода важно еще и потому, что огромное число программ создано на языке C. И нужно уметь их читать, чтобы при необходимости переделать под ввод/вывод, типичный для C++.

Важно понять, что такое файл (*file*) и поток (*stream*) и каково различие между этими понятиями. Система ввода/вывода языка C поддерживает интерфейс, не зависящий от того, какое в действительности используется физическое устройство ввода/вывода, т. е. есть абстрактный уровень между программистом и физическим устройством. Эта абстракция и называется потоком. Способ же хранения информации на физическом устройстве называется файлом.

Несмотря на то что устройства очень разные (терминал, дисководы, магнитная лента и др.), стандарт ANSI языка C связывает каждое из устройств с логическим устройством, называемым потоком. Так как потоки не зависят от физических устройств, то одна и та же функция может записывать информацию на диск, на магнитную ленту или выводить ее на экран.

В языке C существует два типа потоков: текстовый (*text*) и двоичный (*binary*).

Текстовый поток - это последовательность символов. Однако может не быть взаимоднозначного соответствия между символами, которые передаются в потоке и выводятся на экран. Среди символов пара может соответствовать возврату каретки или символу табуляции.

Двоичный поток - это последовательность байтов, которые взаимоднозначно соответствуют тому, что находится на внешнем устройстве.



Файл в языке C - это понятие, которое может быть приложено ко всему от файла на диске до терминала. Поток может быть связан с файлом с помощью оператора открытия файла. Как только файл открыт, то информация может передаваться между ним и вашей программой.

Не все файлы одинаковы. К примеру из файла на диске вы можете выбрать 5-ю запись или заменить 10-ю запись. В то же время в файл, связанный с печатающим устройством, информация может передаваться только последовательно в том же порядке. Это иллюстрирует самое главное различие между потоками и файлами: все потоки одинаковы, что нельзя сказать о файлах.

Операция открытия файла связывает поток с определенным файлом. Операция закрытия файла разрывает эту связь. Если поток был открыт для вывода, то при выполнении операции закрытия файла соответствующий буфер записывается на внешнее устройство. Если программа закончила работу нормальным образом, все файлы автоматически закрываются.

Каждый поток, связанный с файлом, имеет управляющую структуру, называемую FILE. Она описана в заголовочном файле STDIO.H.

#### Ввод/вывод на консоль

К этим операциям относятся операции ввода с клавиатуры и вывода на экран. Технически функции, осуществляющие эти операции, связывают консоль со стандартными потоками ввода/вывода. Во многих системах стандартный ввод/вывод может быть перенаправлен (в том числе и в MS DOS). Однако мы будем для простоты предполагать, что стандартный ввод - это ввод с клавиатуры, а стандартный вывод - это вывод на экран.

Простейшая функция ввода `getche()`, которая читает символы с клавиатуры. Функция ожидает, пока не будет нажата клавиша, и возвращает код, соответствующий символу. Одновременно происходит отображение введенного символа на экран. Ее прототип

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

находится в файле CONIO.H.

Простейшая функция вывода - `putchar()`. Она выводит символ, который является ее аргументом, на экран в текущую позицию курсора. Прототип этой функции

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

находится в STDIO.H.

Двумя наиболее важными аналогами функции `getche()` являются `getchar()` и `getch()`. Функция `getchar()` производит буферизованный ввод, но требует нажатия клавиши Enter. Прототипы этих функций описаны в файле STDIO.H. Функция `getch()` действует так же, как `getche()`, но не выводит символ на экран, ее прототип находится в CONIO.H. Функцию `getch()` часто используют для остановки действия программы до нажатия какой-либо клавиши именно потому, что она не выдает эхо на экран.

Функции `gets()` и `puts()` осуществляют соответственно ввод и вывод на консоль строки символов, прототип `getc()` имеет вид

```
char &gets(char *s);
```

здесь `s` - указатель на массив символов, который заполняется вводимыми с клавиатуры символами. Окончание ввода осуществляется нажатием клавиши `Enter`. Символ возврата каретки в массив не записывается, зато заносится символ `'\0'`, завершающий строку.

Функция `puts()` выводит на экран строку. Ее прототип -

```
int puts(char *);
```

Эта функция, так же как и `printf()`, распознает специальные символы, например символ табуляции `\t`. Функция `puts()` в отличие от `printf()` может выводить только строку, зато работает быстрее и ее запись короче, чем у `printf()`. В результате действия функции `puts()` всегда происходит переход на новую строку. Если вывод успешно завершен, то функция возвращает ненулевое значение, в противном случае возвращает символ `EOF`. Прототипы функций `puts()` и `gets()` находятся в файле `STDIO.H`.

#### УКАЗАТЕЛЬ НА ФАЙЛОВУЮ ПЕРЕМЕННУЮ

Связующим звеном между файлом и потоком в системе ввода/вывода стандарта ANSI языка C является указатель на файл (file pointer). Указатель на файл - это указатель на информацию, которая определяет различные стороны файла: имя, статус, текущую позицию. Указатель файла определяет имя файла на диске и его использование в потоке, ассоциированном с ним. Указатель файла - это указатель на структуру типа `FILE`, которая определена в файле `STDIO.H`. В файле `STDIO.H` определены также следующие функции

Функция	Действие функции
<code>fopen()</code>	Открыть файл
<code>fclose()</code>	Закрыть файл
<code>putc()</code>	Записать символ в поток
<code>getc()</code>	Прочитать символ из потока
<code>fseek()</code>	Изменить указатель позиции файла на указанное место
<code>fprintf()</code>	Форматная запись в файл
<code>fscanf()</code>	Форматное чтение из файла
<code>feof()</code>	Возвращает значение "истинно", если достигнут конец файла
<code>ferror()</code>	Возвращает значение "ложно", если обнаружена ошибка
<code>fread()</code>	Читает блок данных из потока
<code>fwrite()</code>	Пишет блок данных в поток
<code>rewind()</code>	Устанавливает указатель позиции файла на начало
<code>remove()</code>	Уничтожает файл

Чтобы объявить указатель на файл, используется оператор

```
FILE *fput;
```

Рассмотрим более подробно перечисленные выше функции.

Функция `fopen()` выполняет два действия: во-первых, открывает поток и связывает файл на диске с этим потоком; во-вторых, возвращает указатель, ассоциированный с этим файлом. Прототип функции

```
FILE *fopen(char *filename, char *mode);
```

где `mode` - это строка, содержащая режим открываемого файла. Возможные режимы открытия файлов перечислены ниже:

Режим	Действие
"r"	Открыть для чтения
"w"	Создать для записи
"a"	Открыть для добавления в существующий файл
"rb"	Открыть двоичный файл для чтения
"wb"	Открыть двоичный файл для записи
"ab"	Открыть двоичный файл для добавления
"r+"	Открыть файл для чтения и записи
"w+"	Создать файл для чтения и записи
"a+"	Открыть для добавления или создать для чтения и записи
"r+b"	Открыть текстовый файл для чтения и записи
"w+b"	Создать двоичный файл для чтения и записи
"a+b"	Открыть двоичный файл для добавления или создать для чтения и записи
"rt"	Открыть текстовый файл для чтения
"wt"	Создать текстовый файл для записи
"at"	Открыть текстовый файл для добавления
"r+t"	Открыть текстовый файл для чтения и записи
"w+t"	Создать текстовый файл для чтения и записи
"a+t"	Открыть текстовый файл для добавления или создать для чтения и записи

Если вы собираетесь открыть файл с именем `test` для записи, то достаточно написать

```
FILE *fp;
fp=fopen("test","w");
```

Однако рекомендуется использовать следующий способ открытия файла:

```
FILE *fp;
if((fp=fopen("test","w"))==NULL) {
    puts("Не могу открыть файл \n");
    exit(1);
}
```

Этот метод определяет ошибку при открытии файла. Константа `NULL` определена в `STDIO.H`. Функция `exit()`, которую мы использовали, имеет прототип в файле `STDLIB.H`

```
void exit(int val);
```

и прекращает выполнение программы, а величину `val` возвращает в операционную систему (вызывающую программу). При этом перед прекращением работы программы закрывает все открытые файлы, освобождает буферы, в частности выводя все необходимые сообщения на экран. Кроме того, существует функция `abort()` с прототипом

```
void abort(int val);
```

Эта функция немедленно прекращает выполнение программы без закрытия файлов и освобождения буферов. В поток `stderr` она направляет сообщение "abnormal program termination".

Если файл открыт для записи, то существующий файл уничтожается и создается новый файл. При открытии файла для чтения требуется, чтобы он существовал. В случае открытия для чтения и записи существующий файл не уничтожается, однако создается, если он не существует.

Запись функции в поток производится функцией `putc()` с прототипом

```
int putc(int ch, FILE *fptr);
```

Если операция была успешной, то возвращается записанный символ. В случае возникновения ошибки возвращается `EOF`.

Функция `getc()` считывает символ из потока, открытого для чтения функцией `fopen()`. Прототип функции `getc()` -

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

Исторически сложилось так, что `getc()` возвращает значение `int`. То же самое можно сказать про аргумент `ch` в описании функции `putc()`. Используется же в обоих случаях только младший байт. Функция возвращает символ `EOF`, если достигнут конец файла или произошла ошибка при чтении из файла. Чтобы прочитать текстовый файл, можно использовать конструкцию

```
ch=getc(fptr);
while(ch!=EOF) {ch=getc(fptr);}
```

Когда считывается двоичный файл, то определить наличие конца файла, так же как при чтении текстового файла, не удастся. Для определения конца текстового файла служит функция `feof()` с прототипом

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

Функция возвращает значение "истинно", если конец файла достигнут, и "нуль" в противном случае. Следующая конструкция читает двоичный файл до конца файла:

```
while(!feof(fptr)) { ch=getc(fptr); }
```

Для текстовых файлов эта конструкция также применима. Функция `fclose()`, объявленная в виде

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

возвращает "нуль", если операция закрытия файла была успешной. Другая величина означает ошибку. При успешной операции закрытия файла соответствующие данные из буфера считываются в файл, происходит освобождение блока управления файлом, ассоциированного с потоком, и файл становится доступным для дальнейшего использования.

Если произошла ошибка чтения или записи текстового файла, то соответствующая функция возвращает EOF. Чтобы определить, что же в действительности произошло, служит функция `ferror()` с прототипом

```
int ferror(FILE *fptr);
```

которая возвращает значение "истинно" при выполнении последней операции с файлами и значение "ложно" в противном случае. Функция `ferror()` должна быть выполнена непосредственно после каждой операции с файлами, иначе ее сообщение об ошибке может быть потеряно.

Функция `rewind()` устанавливает индикатор позиции файла на начало файла, определенного как аргумент функции, прототип этой функции имеет вид

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

Borland C++ определяет еще две функции буферизованного ввода /вывода: `putw()` и `getw()`. Эти функции не входят в стандарт ANSI языка C. Они используются для чтения и записи целых чисел. Эти функции работают точно так же, как `putc()` и `getc()`.

Стандарт ANSI языка C включает также функции `fread()` и `fwrite()`, которые используются для чтения и записи блоков данных:

```
unsigned fread(void *buf, int bytes, int c, FILE *fptr);  
unsigned fwrite(void *buf, int bytes, int c, FILE *fptr);
```

где `buf` - указатель на область памяти, откуда будет происходить обмен информацией; `c` - сколько единиц записи, каждая длиной `bytes` байтов будет считано (записано); `bytes` - длина каждой единицы записи в байтах; `fptr` - указатель на соответствующий файл.

Чтение и запись в файл необязательно делать последовательно, можно это делать непосредственно доступом к нужному элементу файла посредством функции `fseek()`, которая устанавливает указатель позиции файла в нужное место. Прототип этой функции -

```
int fseek(FILE *fptr, long numbytes, int origin);
```

здесь `fptr` - указатель на соответствующий файл; `numbytes` - количество байт от точки отсчета для установки текущей позиции указателя файла, `origin` - один из макросов, определенных в `STDIO.H`:

Точка отсчета	Макрос	Значение
Начало файла	SEEK_SET	0
Текущая позиция	SEEK_CUR	1
Конец файла	SEEK_END	2

Когда начинается выполнение программы, автоматически открываются 5 предопределенных потоков. Первые три из них - стандартный ввод (stdin), стандартный вывод (stdout) и стандартный поток ошибок (stderr). В обычной ситуации они связаны с консолью, однако могут быть перенаправлены на другой поток. Можно использовать stdin, stdout и stderr как указатели файлов во всех функциях, применяющих тип FILE.

Кроме того, Borland C++ открывает потоки stderr и stdaux, ассоциированные соответственно с принтером и последовательным портом компьютера. Эти потоки открываются и закрываются автоматически.

Стандарт ANSI включает также функции fprintf() и fscanf(), которые работают аналогично функциям printf() и scanf(), за тем исключением, что связаны с файлами на диске. Прототипы этих функций соответственно

```
int fprintf(FILE *fptr, const char *string,...);
```

и

```
int fscanf(FILE *fptr, const char *string,...);
```

здесь fptr - указатель на файл, возвращаемый функцией fopen().

Функция remove() уничтожает указанный файл. Прототип этой функции -

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

Функция возвращает значение 0 при успешной операции и ненулевое значение в противном случае.

Так как язык C связан с операционной системой UNIX, то в системе Borland C++ создана вторая система ввода/вывода. Эта система соответствует стандарту UNIX. Прототипы функций находятся в файле IO.H. Этими функциями являются:

```
read()      - читает буфер данных,
write()     - пишет в буфер данных,
open()      - открывает файл,
close()     - закрывает файл,
lseek()     - поиск определенного байта в файле,
unlink()    - уничтожает файл.
```

Описание этих функций можно найти в документации по cbcntvt Borland C++.

## УПРАВЛЕНИЕ ЭКРАНОМ В ТЕКСТОВОМ РЕЖИМЕ В MS DOS

Система Borland C++ обладает богатой библиотекой функций. Для хорошего оформления диалога пользователя с компьютером (программой) необходима развитая система функций управления работой экрана. Borland C++ предоставляет пользователю такую возможность. Пакет функций управления экраном делится на две части в соответствии с

возможностями компьютера. Первая - работа в текстовом режиме (text mode), вторая - работа в графическом режиме (graphics mode). Функции управления экраном не являются уникальными для Borland C++, они такие же, как в системах Turbo C v.2.0, Turbo C++. Поэтому мы дадим только обзор основных особенностей этих функций. Библиотека функций для работы с текстовым экраном, с заголовочным файлом CONIO.H и библиотека работы с графическим экраном с заголовочным файлом GRAPHICS.H не входят в стандарт языка C. Они существенно используют особенности операционной системы MS DOS, архитектуру IBM PC и графику BGI (Borland Graphics Interface).

### ОСНОВНЫЕ ФУНКЦИИ РАБОТЫ В ТЕКСТОВОМ РЕЖИМЕ

Все функции управления экраном в текстовом режиме имеют свои прототипы в заголовочном файле CONIO.H. Там же находятся некоторые константы и макросы, используемые этими функциями.

Большинство подпрограмм связано с окнами (windows), а не со всем экраном. Окно - это прямоугольная область экрана, которую программа использует для выдачи сообщений. Окно может занимать полный экран или быть настолько маленьким, что в него можно вывести только несколько символов.

Borland C++ позволяет устанавливать размер и местоположение окон на экране. После того как вы задали окно, подпрограммы, которые манипулируют текстом, используют не весь экран, а только заданное окно. К примеру, функция clrscr() очищает активное окно, а не весь экран. Все координаты оказываются связанными с активным окном. Для текстового экрана верхний левый угол окна имеет координаты (1,1). По умолчанию активным окном является весь экран.

Функция, которая создает окно заданного размера и определенного местоположения, имеет прототип

```
void window (int left, int top, int right, int bottom);
```

Если какая-либо координата задана неверно, то никакого действия не происходит. Если выполнение функции было успешным, то все ссылки на текущие координаты делаются относительно этого окна.

При создании окна с помощью функции window() координаты left, top, right, bottom являются абсолютными, т. е. рассматриваются относительно всего экрана. Поэтому вновь создаваемые окна могут перекрываться друг другом.

Одной из наиболее употребляемых функций является функция очистки активного окна

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

Активное окно очищается, а курсор устанавливается в левом верхнем углу окна.

Следующая функция - это функция позиционирования курсора. Ее прототип -

```
void gotoxy(int x,int y);
```



здесь *x* и *y* - координаты относительно активного окна. Если какая-то из координат выходит за границы области, то никакого действия не происходит, однако ошибка также не фиксируется.

#### Функции

```
int wherex();
int wherexy();
```

возвращают координаты *x* и *y* текущего положения курсора.

Другая полезная функция - `clrcol()`. Она очищает строку с текущей позиции курсора до правой границы окна. Особенно удобно с ее помощью освобождать место для ввода. Прототип имеет вид

```
void clrcol(void);
```

Следующие две функции уничтожают строку, в которой находится курсор, и вставляют пустую строку. Функция `delline()` с прототипом

```
void delline(void);
```

уничтожает строку, в которой находится курсор, а все строки, находящиеся под ней, поднимаются вверх. Функция

```
void insline (void);
```

вставляет пустую строку, а та строка, в которой находился курсор, и те строки, которые были ниже, опускаются на одну позицию.

Функции ввода/вывода, такие, как `printf()`, не связаны с оконным интерфейсом, поэтому в Borland C++ дополнительно включены функции ввода/вывода, ориентированные на работу с использованием окон. Такими функциями являются:

```
cprintf() - форматированный вывод в активное окно,
cputs()   - вывод строки в активное окно,
putch()   - вывод символа в активное окно,
getche()  - ввод символа из активного окна,
cgets()   - ввод строки из активного окна,
cscanf()  - форматированный ввод.
```

Эти функции работают почти так же, как соответствующие функции `printf()`, `gets()`, `puts()`, за тем исключением, что связаны с активным окном. Кроме того эти функции не являются перенаправляемыми. Функция `cputs()` не переводит курсор на новую строку, а функция `cprintf()` не разбивает символ `...'\n'` на `'\n\r'`.

Мы предполагаем, что установлен текущий видеорежим компьютера, который почти всегда следующий: текстовый режим 25 строк x 80 символов в строке. Существует несколько текстовых режимов. Установкой или заменой текстового режима управляет функция `textmode()` с прототипом

```
void textmode(int mode);
```

здесь аргумент `mode` может быть одной из следующих величин:



Макрос	Величина	Режим
BW40	0	40 символов x 25 строк, черно-белый
C40	1	40 символов x 25 строк, цветовой
BW80	2	80 символов x 25 строк, черно-белый
C80	3	40 символов x 25 строк, цветовой
MONO	7	80 символов x 25 строк, монохромный
LASTMODE	-1	Предыдущий режим был до замены
C4350	64	EGA 80 x 43 VGA 80 x 50, цветовой

Современная техника в большинстве своем предполагает возможность выдачи текста в цвете. Можно определить как цвет текста, так и цвет фона. Однако использовать цвет могут только специальные, связанные с оконным интерфейсом функции. Такие функции, как `printf()`, для этого не предназначены.

Функция `textcolor()` изменяет цвет выдаваемого текста. Ее прототип имеет вид

```
void textcolor(int color);
```

Аргумент `color` может принимать значения от 0 до 15. Каждое значение соответствует своему цвету. Кроме того, в файле `CONIO.H` определены макросы, соответствующие этим цветам. Они приведены ниже. Кроме того, `textcolor()` позволяет делать текст мигающим (`blink`):

Макрос	Код
BLACK	0
BLUE	1
GREEN	2
CYAN	3
RED	4
MAGENTA	5
BROWN	6
LIGHTGREY	7
DARKGREY	8
LIGHTBLUE	9
LIGHTGREEN	10
LIGHTCYAN	11
LIGHTRED	12
LIGHTMAGENTA	13
YELLOW	14
WHITE	15
BLINK	128

Изменение цвета текста оказывает воздействие только на вновь выводимый текст. Текст, уже находящийся на экране, своего цвета не меняет.

Для того чтобы сделать текст мигающим, надо применить побитовую операцию "OR" к цвету и величине BLINK. Например, чтобы выводить текст красными символами и так, чтобы они мигали, используем вызов функции

```
textcolor(RED|BLINK);
```

Функция `textbackground()` используется для установки цвета фона под вновь выводимым текстом. Прототип этой функции -

```
void textbackground(int color);
```

где величина `color` может принимать значения от нуля до шести, т. е. используются только первые 7 цветов из приведенной выше таблицы.

Функция `clrscr()` не только очищает активное окно, но и заполняет его цветом, заданным в функции `background()`.

Приведем пример использования некоторых из описанных выше функций.

```
#include <conio.h>
/* Пример 54. Использование функций работы с текстовым экраном */
main(void)
{
    register int ctext, cback;
    textmode(C80);
    for (ctext=BLUE; ctext<=WHITE; ctext++)
    {
        for (cback=BLACK; cback<= LIGHTGRAY; cback++);
        {
            textcolor(ctext);
            textbackground(cback);
            cprintf(" ТЕКСТ ");
        }
        cpintf("\n");
    }
    textcolor (WHITE|BLINK);
    textbackground(BLACK);
    cprintf(" КОНЕЦ ТЕКСТА ");
    textmode(LASTMODE);
    return(0);
}
```

Организация видеопамати, т. е. участка памяти, который отображается на экране в случае работы с текстовым экраном, весьма проста. Каждому символу соответствуют 2 байта. В одном из них хранится код символа, в другом - атрибут символа. Байт атрибута имеет следующий вид:

7	6	5	4	3	2	1	0
В	Ф	Ф	Ф	С	С	С	С

где СССС - 4-битовый двоичный код цвета символа; ФФФ - 3-битовый двоичный код цвета фона; В - признак мигания (1 - мигание включено, 0 - выключено).

Так как для цвета фона выделено 3 бита, то цветов может быть только 8, а для цвета символа, задаваемого четырьмя битами, - 16 цветов.

Более того, как мы знаем из физики, любой цвет формируется смешением красного (R-red), зеленого (G-green) и голубого (B-blue) цветов. В байте атрибута мы можем указать, какой бит за какой цвет отвечает:

7	6	5	4	3	2	1	0
B	R	G	B	C	R	G	B

При этом 3-й бит отвечает за яркость цвета символа. Если мы будем рассматривать 4 последних бита, то 0001 соответствует цвету BLUE (1), в то время как 1001 соответствует цвету LIGHTBLUE (9).

Соответственно 0000 - черный цвет (BLACK), 1111 - белый цвет (WHITE).

Функция, которая позволяет задавать атрибут, имеет прототип

```
void textattr(int newattr);
```

Если мы хотим задать атрибут, соответствующий мигающему красному символу на черном фоне, это можно сделать следующими способами:

```
textattr((BLACK<<4) + RED + BLINK);
```

или

```
textattr(0x84);
```

Двоичная запись шестнадцатеричного числа 0x84 имеет вид 10000100. Сравните его с битами атрибута.

Функция `highvideo()` устанавливает повышенную яркость символа, `lowvideo()` - стандартную яркость символа, `normvideo()` восстанавливает стандартные атрибуты (белые символы на черном фоне без мигания).

Функция

```
void _setcursor type(int cursortype);
```

устанавливает размер курсора, значения параметра могут быть следующими:

NOCURSOR	0	курсор не виден;
SOLIDCURSOR	1	курсор занимает все знакоместо;
NORMALCURSOR	2	курсор имеет нормальный вид.

Переменная `_wscroll` управляет прокруткой окна. Если присвоить `_wscroll` значение, равное нулю прокрутка в окне будет запрещена, если же присвоить значение, равное единице, то прокрутка будет разрешена.

В библиотеке предусмотрены функции, позволяющие сохранять, восстанавливать и перемещать содержимое части экрана. Этими функциями являются соответственно

```
int gettext(int left, int top, int right, int bottom, void * destin);
int puttext(int left, int top, int right, int bottom, void * source);
```

```
int movetext(int left, int top, int right, int bottom, int destleft, int desttop);
```

Область экрана задается координатами левого верхнего и правого нижнего углов, область памяти, в которую (из которой) переносится часть видеопамяти, задается указателями `destin` и `source`. В функции `movetext()` два последних аргумента указывают на координаты верхнего левого угла нового расположения области экрана.

Так как для каждого символа в видеопамяти отводится 2 байта, то при сохранении участка экрана надо предусмотреть буфер соответствующего размера.

```
#include <conio.h>
/* Пример 55. Использование gettext(), puttext() */
main(void)
{
    char buffer[288],i;
    char str[]="This is a string ! ";
    textbackground(BLACK);
    clrscr();
    window(5,5,20,10);
    textattr((GREEN<<4)+RED);
    clrscr();
    cputs("\n Hello, World !\n\n\r");
    textattr((GREEN<<4)+RED+BLINK);
    cputs("Press any key... ");
    getch();
    window(1,1,80,25);
    gettext(4, 4, 21, 11, buffer);
    textbackground(BLUE);
    textcolor(WHITE);
    for(i=1;i<23;i++) {
        gotoxy(1, i);
        cputs(str); cputs(str); cputs(str); cputs(str);
    }
    getch();
    puttext(24, 5, 41,12, buffer);
    getch();
    for(i=1;i<5;i++)
        movetext(22,4,32,8,10*i,17);
    getch();
    normvideo();
    return 0;
}
```

Переменная `directvideo` управляет тем, как выполняется вывод на консоль: непосредственно в оперативную память дисплея (`directvideo=1`) либо направляется туда через вызов BIOS (`directvideo=0`). По умолчанию значение `directvideo=1`. Работа непосредственно с памятью дисплея ускоряет ввод, но для этого требуется 100 %-ная совместимость машины с IBM PC.

Вывод через BIOS работает медленнее, но можно быть уверенным, что программа будет работать на любом IBM-совместимом компьютере.

### ВВЕДЕНИЕ В ГРАФИКУ Borland C++

Второй режим, который может устанавливаться на экране IBM-совместимого компьютера, - графический режим. Мы предполагаем, что на вашем компьютере установлен один из графических адаптеров и монитор, соответствующий этому адаптеру. Управление экраном в графическом режиме производится с помощью набора функций, прототипы которых находятся в заголовочном файле GRAPHICS.H. Там же объявлены константы и макросы. Файл GRAPHICS.H должен быть подключен с помощью директивы `#include` препроцессора языка C ко всем модулям, использующим графические подпрограммы.

Так же как и в текстовом режиме, все графические функции оперируют окнами. В терминологии Borland C++ окно называется `viewport`. Отличие графического окна от текстового состоит в том, что левый верхний угол окна имеет координаты (0, 0) а не (1, 1). По умолчанию графическое окно занимает весь экран.

Прежде чем использовать графические функции, необходимо установить видеоадаптер в графический режим. Для установки (инициализации) видеоадаптера служит функция `initgraph()`. Ее прототип -

```
void far initgraph(int far *driver, int far *mode, char far *path);
```

В состав графического пакета входят заголовочный файл GRAPHICS.H, библиотечный файл GRAPHICS.LIB, драйверы графических устройств (\*.BGI) и символные шрифты (\*.CHR).

Функция `initgraph()` считывает в память соответствующий драйвер, устанавливает видеорежим, соответствующий аргументу `mode`, и определяет маршрут к директории, в которой находится соответствующий драйвер \*.BGI. Если маршрут не указан, то предполагается, что этот файл расположен в текущей директории.

Заголовочный файл определяет макросы, соответствующие драйверам:

DETECT	0	Автоматическая установка режима наибольшего графического разрешения
CGA	1	
MCGA	2	
EGA	3	
EGA64	4	
EGAMONO	5	
IBM8514	6	
HERCMONO	7	
ATT400	8	

VGA 9  
PC3270 10  
CURRENT\_DRIVER -1

При использовании `initgraph()` можно указать или конкретный драйвер, или задать автоматическое определение (детектирование) типа видеоадаптера и выбора соответствующего драйвера уже во время выполнения программы (макрос `DETECT`). Это позволяет переносить без изменения программы на компьютеры с другими видеоадаптерами.

Значение `mode` должно быть одним из перечисленных ниже:

Драйвер	Значение	Разрешение	Палитра	Колич. страниц
CGAC0	0	320x200	0	1
CGAC1	1	320x200	1	1
CGAC2	2	320x200	2	1
CGAC3	3	320x200	3	1
CGAHI	4	640x200	1	
MCGAC0	0	320x200	0	1
MCGAC1	1	320x200	1	1
MCGAC2	2	320x200	2	1
MCGAC3	3	320x200	3	1
MCGAMED	4	640x200	1	
MCGAHI	5	640x480	1	
EGALO	0	640x200	16 цветов	4
EGAHI	1	640x350	16 цветов	2
EGA64LO	0	640x200	16 цветов	1
EGA64HI	1	640x350	4 цвета	1
EGAMONOH	0	640x350	1	4
HERCMONOH	0	720x348		2
ATT400C0	0	320x200	0	1
ATT400C1	1	320x200	1	1
ATT400C2	2	320x200	2	1
ATT400C3	3	320x200	3	1
ATT400MED	4	640x200		1
ATT400HI	5	640x400		1
VGALO	0	640x200	16 цветов	4
VGAMED	1	640x350	16 цветов	2
VGAHI	2	640x480	16 цветов	1
PC3270HI	0	720x350		1
IBM8514LO	0	640x480	256 цветов	
IBM8514HI	1	1024x768	256 цветов	

Чтобы выйти из графического режима и вернуться в текстовый режим, необходимо использовать функции

```
void far closegraph(void);
```

И

```
void far restorecrtmode(void);
```

Функция `closegraph()` используется, если программа дальше будет работать в текстовом режиме. Эта функция освобождает память, используемую графическими функциями, и устанавливает текстовый режим, который был до вызова функции `initgraph()`. Если программа завершает работу, то можно использовать функцию `restorecrtmode()`, которая устанавливает видеоадаптер в текстовый режим, который предшествовал первому вызову функции `initgraph()`.

Тип графического видеоадаптера, который установлен в вашем компьютере, определяет, какое количество цветов и какие цвета могут быть использованы в графическом режиме. Наибольшая разница существует между адаптерами CGA и EGA. Количество цветов приведено в предыдущей таблице.

Видеоадаптер CGA имеет 4 цвета в палитре и 4 палитры. Это значит, что на экране одновременно может быть 4 разных цвета. Цвета нумеруются от 0 до 3. Палитры также нумеруются от 0 до 3. Чтобы выбрать палитру, установите режим `CGACx`, где *x* - номер палитры. Цвет с номером 0 всегда совпадает с цветом фона. В качестве цвета фона могут использоваться 16 цветов с номерами от 0 до 15. Соответствие цвета и номера показано в таблице, приводимой ниже. В режиме CGAHI может быть только два цвета, один из которых черный цвет фона. Для цветов, так же как в текстовом режиме, определены макросы, приведенные в таблице.

В режиме EGA одновременно могут использоваться 16 цветов из 64 цветов, причем каждый из элементов палитры может быть задан пользователем. Палитра EGA по умолчанию соответствует цветам CGA, однако в файле `GRAPHICS.H` определены константы, которые содержат соответствующие цветам аппаратные значения. Приведем таблицу

CGA		EGA/VGA	
BLACK	0	EGA_BLACK	0
BLUE	1	EGA_BLUE	1
GREEN	2	EGA_GREEN	2
CYAN	3	EGA_CYAN	3
RED	4	EGA_RED	4
MAGENTA	5	EGA_MAGENTA	5
BROWN	6	EGA_LIGHTGRAY	7
LIGHTGRAY	7	EGA_BROWN	20
DARKGRAY	8	EGA_DARKGRAY	56
LIGHTBLUE	9	EGA_LIGHTBLUE	57
LIGHTGREEN	10	EGA_LIGHTGREEN	58
LIGHTCYAN	11	EGA_LIGHTCYAN	59
LIGHTRED	12	EGA_LIGHTRED	60
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61

YELLOW	14	EGA_YELLOW	62
WHITE	15	EGA_WHITE	63

Что касается цветов, то драйвер VGA работает фактически так же, как и драйвер EGA, только имеет более высокое разрешение.

Установка цвета фона производится функцией

```
void far setbkcolor(int color);
```

а изменение палитры - функцией

```
void far setpalette(int index, int color);
```

причем эта функция неприменима для видеоадаптера CGA за исключением цвета фона, который всегда имеет index, равный нулю.

Напомним, что графический экран представляет собой массив пикселей. Каждый пиксель соответствует одной точке на экране и может иметь свой цвет. Установить цвет пикселя в точке экрана с координатами (x, y) можно с помощью функции

```
void far putpixel(int x, int y, int color);
```

Основными "рисующими" функциями являются line() и circle(). Их прототипы -

```
void far line(int x, int y, int x1, int y1);
```

```
void far circle(int x, int y, int radius);
```

Функция line() чертит на экране прямую линию от точки с координатами (x, y) до точки с координатами (x1, y1) текущим цветом.

Функция circle() рисует на экране окружность с центром в точке с координатами (x, y) и радиусом radius (единица измерения - пиксель) также текущим цветом. По умолчанию текущий цвет устанавливается WHITE. Изменить текущий цвет, т. е. цвет, которым рисуются линии, можно обратившись к функции setcolor() с прототипом

```
void far setcolor(int color);
```

К другим "рисующим" функциям относятся

```
arc()           рисует дугу окружности;
```

```
drawpoly()     рисует контур многоугольника;
```

```
ellipse()      рисует эллипс;
```

```
lineto()       рисует линию из текущей точки в точку,
                задаваемую относительным расстоянием;
```

```
lineto()       рисует линию из текущей точки в точку с координатами (x, y);
```

```
moveto()       перемещает текущую точку в точку с координатами (x, y);
```

```
rectangle()    рисует прямоугольник;
```

```
setaspectratio() изменяет коэффициент сжатия, установленный по умолчанию;
```

```
setlinestyle() устанавливает ширину и стиль линии.
```

В качестве примера описания прототипа приведем прототип функции rectangle()

```
void far rectangle(int left, int top, int right, int bottom);
```



Для закрашивания (заполнения) замкнутого контура служит функция `floodfill()`, которая закрашивает область заданным цветом по заданному шаблону. Ее прототип -

```
void far floodfill(int x,int y, int bordecolor);
```

где `x` и `y` - координаты точки внутри контура, `bordecolor` - цвет контура. Цвет и шаблон заполнения устанавливаются функцией

```
void far setfillstyle(int pattern,int color);
```

Вид шаблона закрашивания и соответствующие ему макрос и значение (`pattern`) приведены ниже:

Макрос	Значение	Вид шаблона
EMPTY_FILL	0	Заполнение цветом фона
SOLID_FILL	1	Сплошное заданным цветом
LINE_FILL	2	Линиями _____
LSTLASH_FILL	3	Косыми линиями ///
SLASH_FILL	4	Яркими косыми линиями //
BKSLASH_FILL	5	Обратными косыми линиями \
LTBKSLASH_FILL	6	Яркими обратными косыми линиями \
HATCH_FILL	7	Светлая штриховка сеткой
XHATCH_FILL	8	Крестообразная штриховка
INTERLEAVE_FILL	9	Перекрестная штриховка
WIDE_DOT_FILL	10	Заполнение редкими точками
CLOSE_DOT_FILL	11	Заполнение частыми точками
USER_FILL	12	Шаблон заполнения, определяемый пользователем

Есть также набор функций, которые чертят контур и закрашивают область внутри контура:

`bar()` - заполненный прямоугольник,  
`bar3d()` - заполненный столбик,  
`fillellipse()` - заполненный эллипс,  
`fillpoly()` - заполненный многоугольник,  
`pieslice()` - заполненный сектор круга,  
`sector()` - заполненный эллиптический сектор.

Прототипы этих функций, их применение и особенности использования можно посмотреть с помощью HELP-системы оболочки Borland C++.

Перечислим еще некоторые функции графической библиотеки системы Borland C++:

`setfillpattern()` - задает шаблон определяемый пользователем;  
`getfillpattern()` - возвращает тип шаблона заполнения;  
`getfillsetting()` - возвращает информацию о шаблоне и цвете заполнения;  
`getlinesetting()` - возвращает информацию о текущем стиле, толщине и цвете линии;

`getpixel()` - сообщает о цвете пикселя в точке (x, y).

В начале мы упомянули о графических окнах, но предположили, что окно совпадает со всем экраном. Создать графическое окно можно используя функцию `setviewport()`. Ее прототип -

`void far setviewport(int left, int top, int right, int bottom, int flag);`

параметры `left`, `top`, `right` и `bottom` задают местоположение и размер окна в абсолютных координатах, т. е. координатах экрана. Параметр `flag` устанавливает режим выхода за границу окна. Если `flag` не нулевой, то происходит автоматическое прерывание выдачи при выходе за границу окна. В случае, когда `flag` равен нулю, может происходить вывод за границами окна. И в том и в другом случае ошибка не фиксируется.

Для работы с экраном, окнами и образами (`image`) служат следующие функции:

<code>cleardevice()</code>	- очищает активную страницу;
<code>setactivepage()</code>	- устанавливает номер активной страницы;
<code>setvisualpage()</code>	- устанавливает номер видимой страницы;
<code>clearviewport()</code>	- очищает активное окно;
<code>getviewsetting()</code>	- возвращает информацию об активном окне;
<code>getimage()</code>	- записывает образ в заданный участок памяти;
<code>imagesize()</code>	- определяет в байтах размер памяти, требуемый для хранения информации о прямоугольной области экрана;
<code>putimage()</code>	- помещает на экране ранее записанный в память образ.

Функция `cleardevice()` очищает весь экран, устанавливает текущей точкой левый верхний угол экрана, но оставляет неизменными все установки графического экрана: стиль линии, текущий цвет и т. д.

Функция `clearviewport()` очищает текущее окно и устанавливает текущую точку в левый верхний угол окна.

В зависимости от типа видеоадаптера и установленного видеорежима система может иметь от одной до четырех буферных страниц. Количество страниц было указано выше в таблице видеорежимов. Каждая из страниц может быть указана как активная, в которую происходит вывод, и визуальная (видимая), которая отображается на экране. Если они совпадают, то каждый вывод будет тут же отображаться на экране. По умолчанию активная и визуальная страницы совпадают.

Для создания движения образа по заданному шаблону на экране служат функции `getimage()`, `imagesize()` и `putimage()`: с помощью функции `getimage()` взять часть экранного образа, вызвать `imagesize()` для определения размера памяти, необходимой для хранения этого образа, а затем вернуть его на экран в любую желаемую позицию с помощью функции `putimage()`.

Наконец, для осуществления вывода текста в графическом режиме на экран используются функции

<code>outtext()</code>	- выводит строку на экран с текущей позиции;
<code>outtextxy()</code>	- выводит строку на экран с заданной позиции;

`settextjustify()` - устанавливает режим выравнивания текста;  
`setusercharsize()` - устанавливает шрифт, стиль и коэффициент увеличения текста;  
`textheight()` - возвращает высоту строки в пикселях;  
`textwidth()` - возвращает ширину строки в пикселях.

Во время работы программы могут возникать ошибки при выполнении графических операций. Например, указан недопустимый тип видеоадаптера. Для обработки ошибок, чтобы избежать аварийного прекращения работы программы, служат функции

`graphresult()` - возвращает код ошибки выполнения для последней графической операции;  
`grapherrormsg()` - возвращает строку с сообщением об ошибке по заданному коду ошибки.

Если ошибка произошла при вызове графической библиотечной функции (например, не задан шрифт, запрошенный функцией `settextstyle()`) устанавливается внутренний код ошибки. Функция `graphresult()` возвратит код ошибки, а вызов функции `grapherrormsg(graphresult())` выдаст сообщение об ошибке. Приведем фрагмент программы, использующей рекомендуемый способ инициализации графического режима:

```

{
  /* Запрос автоопределения максимально возможного
     режима работы видеоадаптера */
  int driver = DETECT, gmode, errorcode;
  /* инициализация графики */
  initgraph(&graphdriver, &gmode, "");
  /* получение результата инициализации */
  errorcode = graphresult();
  if(errorcode != grOk) /* если произошла ошибка */
  {
    printf("Ошибка: %s\n", grapherrormsg(errorcode));
    printf("Для останова нажмите любую клавишу\n");
    getch();
    exit(1); /* завершение работы программы */
  }
}

```

Приведем также некоторые примеры кодов и сообщений об ошибках выполнения графических операций.

Код	Макрос	Сообщение об ошибке
0	<code>grOK</code>	No error
1	<code>grNoInitGraph</code>	BGI graphics not installed (use <code>initgraph</code> )
...		
5	<code>grNoLoadMem</code>	Not enough memory to load driver

Приведем пример программы, использующей графическую библиотеку Borland C++:

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
/* Пример 56. */
/* использование графических функций BORLAND C++ */
main(void)
{
    int driver,mode,errorcode;
    register int i;
    driver=DETECT;
    initgraph(&driver,&mode,"");
    errorcode = graphresult();
    if(errorcode != grOk) /* если произошла ошибка */
    {
        printf("Ошибка:%s\n",grapherrormsg(errorcode));
        printf("Для останова нажмите любую клавишу\n");
        getch();
        exit(1); /* завершение работы программы */
    }
    rectangle(0,0,639,349);
    setcolor(RED);
    line(0,0,639,349); line(0,349,639,0);
    setfillstyle(5, GREEN);
    bar(50,50,300,300);
    setviewport(100,100,200,200,1);
    getch();
    clearviewport();
    for(i=3;i<83;i+=3) circle(50,50,i);
    getch(); restorecrtmode();
    return 0;
}
```

Еще одним примером использования графической библиотеки является пример построения графика функции. Мы рассмотрим пример построения графика функции  $R = \sin(C\phi)$ ; в псевдополярных координатах  $x = R \sin(A\phi)$ ,  $y = R \cos(B\phi)$ ;

Здесь  $A, B, C$  - целочисленные значения.

В случае  $A = B = 1$  мы получим график в полярных координатах.

Выбирая различные значения  $A, B$ , и  $C$  мы будем получать различные "замечательные картинки".

```
/* Пример 57. Замечательные картинки */
/* График в псевдополярных координатах */
#include<iostream.h>
#include<math.h>
#include<stdlib.h>
#include<conio.h>
#include<graphics.h>
```

```
#define PI      3.141593
void main(void) {
    int A, B, C, M;
    double step=0.01;
    int x, y, r;
    M=100; /* Масштабирующий множитель */
    printf("Введите целые числа:\nA = ");
    scanf("%d",&A);
    printf("B = ");
    scanf("%d",&B);
    printf("C = ");
    scanf("%d",&C);
    int gd=DETECT,gm; /* Инициализация графика */
    initgraph(&gd,&gm,"");
    setbkcolor(BLUE);
    cleardevice();
    setcolor(YELLOW);
    setlinestyle(SOLID_LINE,0,NORM_WIDTH);
    /* Построение графика */
    moveto(320,160);
    for (double fi=0;fi<2*PI,fi+=step){
        r=M*sin(C*fi);
        x=320+r*sin(A*fi);
        y=160+r*cos(B*fi);
        lineto(x,y);
    }
    getch();
    closegraph();
}
```

Еще один пример - использование функций, работающих с палитрами.

```
/* Пример 58. Окружности. Работа с палитрами */
#include<graphics.h>
#include<conio.h>
#include<stdio.h>
#include<dos.h>
void main(void) {
    int gd=DETECT,gm;
    initgraph(&gd,&gm,"F:\\BC31\\BGI");
    struct palettetype save,palette;
    getpalette(&save);
    getpalette(&palette);
    for(int i=0;i<10;i++) {
        setfillstyle(SOLID_FILL,6+i);
        setcolor(6+i);
        fillellipse(getmaxx()/2,getmaxy()/2,(10-i)*10,(10-i)*10);
    }
    while(!kbhit()) {
        int temp=palette.colors[6];
```

```

for(int i = 7; i < 16; i++) // Shift palette
    palette.colors[i-1] = palette.colors[i];
palette.colors[15] = temp;
setallpalette(&palette);
delay(100);
}
getch();
closegraph();
}

```

Как правило, в состав системы Borland C++ входят примеры использования возможностей системы, в том числе графических возможностей.

## 3 ЯЗЫК C++

---

### **C++ - ЯЗЫК ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ**

Язык C++ является надстройкой над языком C. И если язык C удобен для написания малых и средних по размерам программ, то все преимущества языка C++ проявляются на больших программах и проектах. Конечно, в рамках этой книги нам не удастся рассмотреть большие программы, многие примеры использования возможностей C++ будут модельными. Маленькая программа на языке C почти всегда короче, чем на языке C++, однако на начальном этапе надо применять язык C++ при решении небольших задач.

Прежде чем перейти к детальному изучению Borland C++ важно познакомиться с основными понятиями объектно-ориентированного программирования. Поэтому далее идет краткий обзор основных особенностей языка C++ как языка объектно-ориентированного программирования.

### **ЧТО ТАКОЕ ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**

Объектно-ориентированное программирование - это новый способ подхода к программированию. Такое программирование, взяв лучшие черты структурного программирования, дополняет его новыми идеями, которые переводят в новое качество подход к созданию программ.

Наиболее важное понятие языков объектно-ориентированного программирования - это понятие объекта (object). Объект - это логическая единица, которая содержит данные и правила (методы) обработки этих данных. В языке C++ в качестве таких правил обработки выступают функции, т. е. объект в Borland C++ объединяет в себе данные и функции, обрабатывающие эти данные. Внутри объекта данные и функции могут быть частными (приватными, private), защищенными (protected) и общими (public). Можно сказать, что объект - это переменная определенного пользователем типа. Объектно-ориентированные языки обладают четырьмя важнейшими характеристиками: инкапсуляцией (encapsulation), наследованием (inheritance), полиморфизмом (polymorphism) и абстракцией типов (abstraction).

Понятие инкапсуляции означает, что в качестве единицы целого рассматривается объединение некоторой группы данных и некоторой группы функций. Свойства объектов хранятся в структурах данных, напоминающих структуры языка C, а поведение объектов реализуется в виде функций, называемых функциями-членами (member function) объектов.

В объекте реализована защита данных: если данные или функции-члены объявлены приватными или защищенными, то к ним нет доступа извне. Зато данные и функции, объявленные общими, доступны любому внешнему объекту.

Объектно-ориентированное программирование поддерживает полиморфизм, означающий, что одно и то же имя может использоваться для логически связанных, но разных целей, т. е. имя определяет класс действий, которые в зависимости от типа данных могут существенно отличаться. Например, можно определить три типа переменных: целые, комплексные числа и векторы. Для каждого из них можно определить функцию `sum(x, y)` - сумму двух переменных, а можно сделать так, что для этих трех типов будет определена операция сложения  $x + y$ . В зависимости от того, какого типа будут переменные  $x$  и  $y$ , работать эта функция и операция сложения будут по-разному. Причем полиморфизм поддерживается Borland C++ и во время компиляции, и во время выполнения программы.

Наследование позволяет одним объектам приобретать атрибуты и свойства других объектов. Наследование позволяет строить иерархию объектов, переходя от более общего к частному, уточняя и конкретизируя объект.

### ОСОБЕННОСТИ ЯЗЫКА C++, НЕ СВЯЗАННЫЕ НАПРЯМУЮ С ОБЪЕКТНОЙ ОРИЕНТИРОВАННОСТЬЮ

Рассмотрим на примере простой программы некоторые черты языка C++.

// Это однострочный комментарий

```
#include <iostream.h> // Язык C++ имеет свою библиотеку ввода/вывода
#include <stdio.h> // Использовать библиотеку ввода/вывода тоже можно
```

// Пример 59. Особенности языка C++

```
main()
```

```
{
```

```
char str[80];
```

```
cout << " C++ хороший язык \n"; // Операция вывода << для каждого
```

```
//из встроенных типов работает как printf()
```

```
/* Можно использовать стиль комментариев языка C */
```

```
printf("Можно использовать функцию printf() \n");
```

```
// Ввод чисел, используя операцию >>
```

```
cout << " Введите число ";
```

```
cin >> i; // Операция ввода >> для встроенных
```

```
//типов работает по правилам scanf()
```

```
//тип аргумента определяется компилятором
```

```
//и не требует использования адреса переменной
```

```
cout << " Вы ввели число " << i << "\n";
```

```
// Ввод строки
```



```
cout << " Введите строку: ";
cin >> str;
// Печать введенной строки
cout << str;
float f=1.2345; // Переменную в языке C не обязательно
//определять в начале блока

cout << " переменная типа float f = "<< f << "\n";

return 0;
}
```

Как вы видите, эта программа выглядит иначе, чем программа на языке C. Во-первых, подключен заголовочный файл `<iostream.h>`, поддерживающий стиль ввода/вывода языка C++. Во-вторых, используются некоторые новые операторы, например оператор

```
cout << "C++ хороший язык \n";
```

В результате работы этого оператора на экран выводится сообщение: "C++ хороший язык" и курсор переводится на следующую строчку. В языке C++ операция `<<` имеет двойное значение. Ранее эта операция была определена как поразрядный сдвиг влево. Но когда эта операция используется, как в приведенном примере, то она является также операцией вывода. Такая операция называется перегружаемой. Ключевое слово `cout` - это поток стандартного ввода/вывода, аналогичный `stdin` языка C, по умолчанию связанный с экраном. Можно использовать `cout` и `<<` и для вывода на экран данных встроенного типа, включая символьные строки.

В языке C++ комментарии можно организовать двумя способами. Первый способ тот же, что и в языке C, т. е. заключить комментарий между парами символов `/*` и `*/`. Второй способ - использование пары символов `//`. Все, что находится после пары символов `//` до конца строки, является комментарием. Если комментарий короткий, то удобно использовать второй способ. Если же комментарий размещается на нескольких строках, то может быть целесообразно использовать классический способ.

Следующая строка,

```
cout << " Введите число ";
```

выводит приглашение пользователю ввести число. За ней следует строка с оператором ввода

```
cin >> i;
```

Идентификатор `cin` связан со стандартным потоком ввода, аналогичен потоку `stdin` и связан с клавиатурой по умолчанию. Операция `>>` в языке C++ также является перегружаемой. В зависимости от контекста это оператор сдвига вправо или оператор ввода. Используя `>>`, можно читать с клавиатуры переменные основных типов данных (включая строки символов). Можно для ввода/вывода использовать и библиотеки языка C. Необходимо

согласовать действие этих библиотек. Для этого существует функция `ios::sync_with_stdio()`.

Результатом работы оператора

```
cout << "Вы ввели число " << i << "\n";
```

будет сообщение (если  $i = 100$ ) "Вы ввели число 100", и курсор переместится в начало следующей строки. Операцию `<<` можно использовать подряд в одном операторе столько раз, сколько вы хотите.

Язык C++ имеет еще одну особенность. Если в языке C локальная переменная с тем же именем, какое было и у глобальной, закрывала видимость глобальной переменной, то в языке C++ глобальная переменная доступна в любом месте программы. Для этого надо использовать операцию `::`.

```
#include <iostream.h>
// Пример 60. Использование глобальной переменной
int i = 100; // Объявление глобальной переменной
main(void)
{
    int i = 5; // Локальная переменная

    cout << "Локальная переменная " << i << "\n";
    cout << "Глобальная переменная " << ::i << "\n";
    return 0;
}
```

## КОМПИЛЯЦИЯ ПРОГРАММ НА ЯЗЫКЕ C++

Как вы знаете, Borland C++ может компилировать программы, написанные и на языке C, и на языке C++. Если файл имеет расширение `.cpp`, то вызывается компилятор языка C++. Если расширение `.c`, то программа компилируется как C-программа. Однако в опциях интегрируемой среды Options|Compiler|C++options можно установить вызов компилятора с языка C++ вне зависимости от расширения файла. Таким образом, простейший путь указать Borland C++ компилировать программу как C++-программу - дать файлу расширение `.cpp`.

## ВВЕДЕНИЕ В ПОНЯТИЕ КЛАССА И ОБЪЕКТА

Одним из самых главных понятий языка C++ является понятие класса (class). В языке C++ для того, чтобы определить объект (object), надо сначала определить его форму с помощью ключевого слова `class`. Понятие класса напоминает понятие структуры. Рассмотрим пример, в котором определим класс `queue` (очередь), который будет затем использоваться для определения объектов. Очередь - это структура данных, для которой введены действия "поставить в очередь", "обслужить". При этом постановка в очередь осуществляется в конец очереди, а обслуживание происходит с начала очереди. Поэтому эту структуру данных называют еще FIFO (First

Input First Output). В нашем случае мы примем следующую модель: очередь состоит из целочисленного массива и двух целочисленных переменных - маркера начала очереди и маркер конца очереди. Поставить в очередь - это заполнить заданным значением очередной элемент массива с конца очереди. Обслужить - выдать значение очередного элемента с начала очереди. Очередь считается полной, если заполнен последний элемент массива. Очередь считается пустой, если маркеры начала и конца очереди указывают на один и тот же элемент массива.

```
// Объявим класс queue
class queue{
private: // режим доступа private к элементам класса

    int q[100];
    int sloc, rloc; // маркеры начала и конца очереди
public: // режим доступа public к следующим элементам класса
    void init(void); // Функции - члены класса
    void qput(int m);
    void qget(void);
};
```

Таким образом объявлен новый тип переменных - class queue.

Это объявление класса. Класс может содержать приватную часть (private) и общую часть (public). По умолчанию все элементы класса приватные, поэтому ключевое слово private можно опустить. В нашем примере приватные переменные - это переменные i, j и массив q[100]. Приватные элементы не могут использоваться никакими функциями, не являющимися членами класса. Это один из путей реализации принципа инкапсуляции - доступ к элементам данных может контролироваться объявлением их приватными. Можно также определить и приватные функции, которые могут вызываться только другими функциями - членами класса.

Для того чтобы объявить другую часть класса общей, т. е. сделать переменные и функции класса доступными из других частей программы, следует объявить их после ключевого слова public. Переменные и функции, объявленные общими, доступны для всех других функций программы, т. е. остальная часть программы имеет доступ к объекту через общие переменные и функции. Как правило, большинство данных объявляются приватными, а доступ к ним осуществляется через общие функции. Кроме режимов доступа private и public, в классе может быть режим доступа protected, который играет существенную роль при использовании механизма наследования классов, о чем речь пойдет ниже.

Ключевые слова private, protected и public могут встречаться в объявлении классов в любом порядке и любое количество раз. Например, можно объявить

```
class W{
    int i;
    int j;
```

```
public:
    void f1(void);
    void f2(void);
protected:
    int a;
public:
    int b;
};
```

В то же время хорошим стилем считается использование каждого из ключевых слов один раз.

Функции `init()`, `qput()` и `qget()` называются членами - функциями (member function) класса `queue`, так как они являются элементами класса `queue`.

Помните, что объект формирует связь между кодами и данными. Только те функции, которые объявлены в классе, имеют доступ к приватной части класса. Когда вы создали класс, вы можете определить объект (переменную) этого типа с использованием имени класса. Например, объявлением

```
queue obj1, obj2;
```

мы создали два объекта одного типа. Обратите внимание на то, что ключевое слово `class` при объявлении типа не используется.

Важно отметить, что классы - это не объекты (переменные), а шаблоны (типы) для создания объектов. Когда нужно, создается экземпляр класса, который называется объектом. Отношение между классом и объектом такое же, как между типом данных и переменной.

Можно создать объект и во время объявления класса, поместив имя объекта после закрывающей фигурной скобки, как это делали со структурами в языке C.

Объявление класса в общем виде следующее:

```
class имя_класса {
    private:
        приватные данные и функции
    protected:
        защищенные данные и функции
    public:
        общие данные и функции
} [список объектов];
```

Список объектов может отсутствовать.

При объявлении класса, как правило, используются прототипы функций - членов класса.

Все функции в C++ должны иметь прототипы.

Когда же требуется описать функцию - член класса, необходимо указать, к какому классу относится эта функция. Например, можно определить функцию `init()` класса `queue` следующим образом:

```
void queue::init(int i);
{
    sloc = rloc = 0;
}
```

Операция `::` называется операцией принадлежности (*scope resolution*). Она определяет, к какому классу принадлежит эта функция или, другими словами, к чьей сфере влияния принадлежит эта функция. Имя `queue::init` называется полным или квалифицированным именем функции - члена класса.

В языке C++ разные классы могут иметь функции с одинаковыми именами. Операция `::` и имя класса позволяют компилятору определить принадлежность функции.

Чтобы вызвать функцию - член класса в той части программы, которая не является частью класса, надо использовать имя объекта и операцию точка (`.`) доступа к члену (элементу) класса. Например, если объявлены два объекта `a` и `b` класса `queue` (`queue a, b;`), то для вызова функции `init()` для объекта класса `a` нужно написать

```
a.init();
```

Следует помнить, что `a` и `b` два разных объекта. Данные - члены этих объектов занимают в памяти разное место, однако функции - члены классов общие. Функции - члены класса для того, чтобы отличать один объект от другого, имеют неявный параметр, указатель на объект. Этот указатель называется `this`. В дальнейшем мы увидим, как указатель `this` используется в явном виде.

С другой стороны, функция - член класса может вызывать другую функцию - член того же класса или использовать данные - члены класса непосредственно, не используя операцию точка (`.`). Операцию точка надо использовать только тогда, когда функция - член класса вызывается в функциях, не являющихся членами класса (*nonmember functions*).

Следующая программа сводит воедино все, что было до сих пор сказано.

```
#include <iostream.h>
// Пример 61. Модель структуры данных "очередь"

//Объявляем класс queue
class queue{
    int q[10];
    int sloc, rloc;
public:
    void init(void);
    void qput(int);
    int qget(void);
```

```
};
```

```
void queue::init(void) //описание функций- членов класса queue
{
    rloc = sloc = 0;
}
```

```
int queue::qget(void)
{
    if (sloc==rloc) {
        cout << " Очередь пуста ";
        return 0; // Мы вынуждены вернуть какое-то значение
    }
    return q[rloc++];
}
```

```
void queue::qput(int i)
{
    if (sloc==10){
        cout << "Очередь полна";
        return;
    }
    q[sloc++]=i;
}
```

```
main(void) // Функция main()
{
    queue a, b; // Созданы два объекта
```

```
    a.init(); // Инициализация объектов
    b.init();
```

```
    a.qput(7); // Заполнение очереди a
    a.qput(9);
    a.qput(11);
```

```
    cout << a.qget() << " ";
    cout << a.qget() << " ";
    cout << a.qget() << " ";
    cout << a.qget() << "\n";
```

```
// Попробуйте заменить последние 4 оператора одним
```

```
// cout<<a.qget()<<" "<<a.qget()<<" "<<a.qget()<<" "<<a.qget()<<"\n";
```

```
// Проанализируйте результат

for (int i=0; i<12; i++) // Заполнение очереди b значением i^2
    b.qput(i * i);

for(i=0;i<12;i++)
    cout<<b.qget()<<" ";
cout<<"\n\n";

getch(); // Задержка результата работы на экране
return 0;
}
```

Особенностью этого примера, по сравнению с предыдущими, является тот факт, что функция `main()` стоит не на первом месте. Нет правила, обязывающего ставить `main()` первой функцией программы. Обычно в программах на языке C++ функции - члены классов определяются перед `main()`, и в дальнейшем мы будем придерживаться этого правила. На практике описание функций - членов класса как правило, выносится в отдельный файл.

## ПЕРЕГРУЖЕННЫЕ ФУНКЦИИ

Одним из путей реализации полиморфизма в языке C++ является перегрузка (overloading) функций. Две или более функции могут иметь одно и то же имя, но отличаться друг от друга количеством или типом параметров. Тогда говорят о перегруженных функциях (overload functions).

Пример перегруженных функций:

```
#include <iostream.h>
// Пример 62.
// Перегрузка функций

int sqr_it(int i); // Прототипы функций sqr_it(), отличающихся
double sqr_it(double d); // типом параметра
long sqr_it(long l);

main(void){
    int i = 7;
    double d = 1.1;
    long l = 20;
    cout << sqr_it(i) <<"\n"; // Вызов функций через переменные
    cout << sqr_it(d) <<"\n"; // Тип переменных известен
    cout << sqr_it(l) <<"\n";

    cout << sqr_it(10) <<"\n"; // Вызов функции с помощью констант
    cout << sqr_it(10.1)<<"\n";
    cout << sqr_it(70000) <<"\n";
    cout << sqr_it(10l) <<"\n"; // Типизированная через суффикс константа
}
```

```
unsigned u = 4;
cout << sqr_it(u); // Попытка вызвать функцию с типом параметра, для
cout << sqr_it(4u); // которого функция не описана, вызывает ошибку

return 0;
}

int sqr_it(int i)
{
    cout << " Функция использует целый аргумент ";
    return i*i;
}

double sqr_it(double d)
{
    cout << " Функция использует вещественный аргумент ";
    return d*d;
}

long sqr_it(long l)
{
    cout << " Функция использует аргумент типа long";
    return l*l;
}
```

Перегрузка функции позволяет использовать одно и то же имя для выполнения одинаковых или похожих действий с аргументами. Например, возводить в квадрат разные типы данных. Одна из причин важности и удобства перегрузки функций состоит в возможности помочь управлять комплексно. Что это значит? Большинство компиляторов в своей стандартной библиотеке языка C содержат функции `atoi()`, `atof()` и `atol()`. Эти функции преобразуют строку цифр во внутренний формат представления чисел типа соответственно `int`, `double` и `long double`. Каждая из них выполняет почти идентичные действия. Используются три разных имени для функций, выполняющих по сути одно и то же.

В языке C++ возможно использование одного имени `atoum()` для всех трех. Способ преобразования будет выбираться в зависимости от типа аргумента функции `atoum()`. Можно, конечно, использовать одно и то же имя для обозначения функций, выполняющих принципиально разные действия в зависимости от типа аргумента. Например, функция `sqr_it()` с целым аргументом возводит число в куб, а с аргументом типа `double` извлекает кубический корень. Но с точки зрения здравого смысла так делать не стоит.

Перегружаемые функции не могут отличаться только типом возвращаемого значения. В самом деле, если объявить функции



```
int sqr_it(long i);
```

И

```
long sqr_it(long i),
```

при вызове функции

```
sqr_it (70000);
```

перед компилятором встанет неразрешимая проблема выбора, так как при вызове функции тип возвращаемого значения никак не может быть указан.

## ПЕРЕГРУЗКА ОПЕРАЦИЙ

Второй способ реализации полиморфизма в языке C++ - это перегрузка операций. Например, операции << и >> в C++ имеют двойкий смысл: поразрядный сдвиг и оператор ввода/вывода на консоль. Причина этого в том, что в заголовочном файле `iostream.h` эти знаки операций перегружены. Когда перегружается знак операции, компилятор анализирует тип операндов и в зависимости от типа делает выбор. Более подробно перегрузка операций будет рассмотрена позднее.

## НАСЛЕДОВАНИЕ

Наследование - одна из важных черт языков объектно-ориентированного программирования. В C++ наследованием реализуется возможность объединять один класс с другим во время объявления второго класса.

Предположим, в примере с классом `queue` мы хотим добавить еще одну переменную `sum` - сумма целых чисел в очереди - и функцию `get_sum()`, вычисляющую эту сумму.

Мы имеем две возможности - переписать класс `queue` или построить новый класс `queue` на основе класса `queue` с помощью механизма наследования. Рассмотрим второй вариант.

Основная форма наследования -

```
class имя_наследующего_класса: режим_доступа наследуемый_класс{...};
```

В языке C++ принято называть наследуемый класс базовым классом (base class), наследующий класс - производным классом (derive class).

Здесь режим\_доступа - это одно из ключевых слов `private`, `public` или `protected`. Рассмотрим использование опции `public`. Эта опция означает, что все элементы типа `public` предка будут типа `public` для класса, который наследует его. Обратите внимание на то, что режим доступа `private` в классе `queue` заменен на `protected`. Тогда все функции - члены класса `queue1` имеют доступ к членам класса `queue` так, как если бы они были объявлены внутри класса `queue1`. Однако функции класса `queue1` не имеют доступа к приватной части класса `queue`. Далее приводится пример, иллюстрирующий понятие наследования.

```
#include <iostream.h>
```

```
// Пример 63. Использование наследования классов
```

```
class queue{ //Объявляем класс queue
protected:
int q[10];
int sloc, rloc;
public:
void init(void);
void qput(int i);
int qget(void);
};

class queue1: public queue { // Объявляем класс queue1
int sum;
public:
int get_sum(void);
void show_sum(void);
};

void queue::init(void) //описание функций - членов класса queue
{
rloc = sloc = 0;
}

int queue::qget(void)
{
if (sloc==rloc) {
cout << " Очередь пуста ";
return 0; // Мы вынуждены вернуть какое-то значение
}
return q[rloc++];
}

void queue::qput(int i)
{
if (sloc==10){
cout << "Очередь полна";
return;
}
q[sloc++]=i;
}

int queue1::get_sum(void) // Описание функций - членов класса queue1
{
sum = 0;
for(int i=rloc; i<sloc; i++)
sum+=q[i];
return sum;
}

void queue1::show_sum(void)
```

```

{
    cout << "Сумма очереди - " << sum << "\n";
}

main(void)
{
    queue1 obj;

    obj.init();
    for(int i=0; i<5; i++){
        obj.qput(100+i);
        obj.get_sum();
        obj.show_sum();
    }
    for(i=0; i<6; i++){
        obj.get_sum();
        obj.show_sum();
        obj.qget();
    }
    return 0;
}

```

## КОНСТРУКТОРЫ И ДЕКТРУКТОРЫ

Для большинства объектов естественно требовать, чтобы они были инициализированы до начала их использования. Примером тому является класс `queue`. Прежде чем объект класса `queue` будет использоваться, переменным `sloc` и `rloc` должно быть присвоено нулевое значение. Это было реализовано функцией `init()`. Так как необходимость инициализации объектов является общим требованием, то C++ предоставляет возможность делать это автоматически при создании объекта, т. е. при объявлении объекта. Эта автоматическая инициализация реализуется использованием функции, называемой конструктором (`constructor`).

Конструктор - это специальная функция, являющаяся членом класса и имеющая то же самое имя, что и класс. Например, класс `queue` можно было объявить так:

```

class queue{
    int q[100];
    int sloc, rloc;
public:
    queue(void); // конструктор
    void qput(int i);
    int qget(void);
};

```

Функция-конструктор не может иметь тип возвращаемого значения. Поэтому нет типа возвращаемого значения функции. Описание конструктора `queue()` можно сделать так:

```

queue queue(void) // конструктор класса queue
{
    sloc = rloc = 0;
    cout << "Очередь инициализирована \n";
}

```

Сообщение включено в эту функцию для иллюстрация того, что функция работает. В большинстве случаев конструктор никаких сообщений не выдает. Вызывается функция-конструктор в тот момент, когда создается объект, т. е. для объекта выделяется место в памяти. Для локального объекта это будет происходить при входе в блок, в котором есть объявление объекта. Для глобальных объектов конструктор вызывается один раз при создании объекта в начале работы программы. Нельзя вызывать функцию-конструктор в явном виде. Конструктор, как и другие функции, может иметь параметры. конструктор может быть перегруженной функцией, т. е. класс может иметь несколько конструкторов. Если в классе не объявлен ни один конструктор, компилятор сам создает функцию - конструктор класса.

Еще одна специальная функция класса - деструктор (destructor). Во многих случаях требуется, чтобы были произведены какие-либо действия при окончании работы объекта. Это может быть освобождение памяти, восстановление экрана, закрытие файлов и т. д. В C++ такой функцией и является деструктор. При создании объекта, при выделении памяти под объект вызывается конструктор, а при освобождении памяти из-под объекта, т. е. при прекращении действия объекта, выполняется деструктор.

Деструктор имеет такое же имя, как и имя класса, но перед ним ставится знак тильды (~). Деструктор, как и конструктор, не может иметь тип возвращаемого значения и не может иметь параметров. В отличие от конструктора деструктор может быть вызван явно.

Класс queue не нуждается в деструкторе, тем не менее включим эту функцию в качестве иллюстрации.

```

class queue{
    int q[100];
    int sloc, rloc;
public:
    queue(void), // конструктор
    ~queue(void), // деструктор, тип не объявляется
    void qput(int i),
    int qget(void);
}; // объявлен класс queue с конструктором и деструктором

queue queue(void)
{
    rloc = sloc = 0;
    cout << "Очередь инициализирована \n";
} // описание конструктора

```

```
queue::~~queue(void)
{
    cout << " Очередь разрушена \n ";
} // описание деструктора
```

Рассмотрим новую версию примера:

```
#include <iostream.h>
// Пример 64. Использование конструктора и деструктора класса
```

```
// объявление класса
```

```
class queue {
    int q[100];
    int sloc, rloc;
public:
    queue(void); // конструктор
    ~queue(void); // деструктор
    void qput(int i);
    int qget(void);
};
```

```
// описание функции-конструктора
```

```
queue::queue(void)
{
    sloc=rloc=0;
    cout << " Очередь инициализирована\n";
}
```

```
// описание функции-деструктора
```

```
queue::~~queue(void)
{
    cout << " Очередь разрушена \n";
}
```

```
void queue::qput(int i)
```

```
{
    if (sloc==100){
        cout << " Очередь полна\n ";
        return;
    }
```

```
    q[sloc++] = i;
```

```
}
int queue::qget(void)
```

```
{
    if (rloc==sloc) {
        cout << " Очередь пуста \n";
        return 0;
    }
    return q[rloc++];
}
```

```
main()
{
    queue a, b; // объявление двух объектов класса queue

    a.qput(10);
    b.qput(19);

    a.qput(20);
    b.qput(1);

    cout <<a.qget() << " ";
    cout <<a.qget() << " ";
    cout <<b.qget() << " ";
    cout <<b.qget() << "\n";

    return 0;
}
```

В результате работы этой программы получим:

```
Очередь инициализирована
Очередь инициализирована
10 20 19 1
Очередь разрушена
Очередь разрушена
```

## НОВЫЕ КЛЮЧЕВЫЕ СЛОВА C++

В дополнение к уже существующим ключевым словам языка C в язык C++ добавляются новые:

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

Рассмотрим более подробно понятие "class" в C++.

## КОНСТРУКТОР С ПАРАМЕТРАМИ

Когда объект создается, он может быть инициализирован функцией-конструктором. Однако часто бывает желательно или необходимо инициализировать элементы создаваемого объекта определенными конкретными значениями. Язык C++ предоставляет программисту такую возможность. Это достигается передачей значений через аргументы конструктора объекта.

Еще раз модифицируем класс queue, добавив в него новую переменную id и еще один конструктор. Конструктор без параметров обычно называется конструктором по умолчанию (default constructor), конструктор с параметрами называется конструктором инициализации (initialized constructor). Конструктор инициализации будет иметь один параметр, который будет

передаваться переменной `id`, эта переменная будет идентификатором области. Конструктор по умолчанию будет присваивать этой переменной значение 0.

Рассмотрим пример, опять используя класс `queue`, немного его изменив:

```
class queue {
    int q[100];
    int sloc, rloc;
    int id; // это значение будем определять
public:
    queue(void); // конструктор по умолчанию

    queue(int i); // конструктор инициализации

    ~queue(void); // деструктор
    void qput(int i);
    int qget(void);
};
```

Переменная `id` будет содержать значение параметра, идентифицирующего объект. Он будет определен при создании объекта типа `queue`. Перепишем конструктор по умолчанию `queue` в следующем виде:

```
queue::queue(void)
{
    rloc = sloc = 0;
    id = 0;
    cout << "Конструктор по умолчанию очереди " << id << "\n";
}
```

Конструктор инициализации напишем в следующем виде:

```
queue::queue(int i)
{
    rloc = sloc = 0;
    id = i;
    cout << "Конструктор инициализации очереди " << id << "\n";
}
```

Для передачи значение параметра при объявлении объекта в C++ существует два способа.

Первый способ:

```
queue a = queue(101);
```

Это объявление объекта `a` типа `queue` и передача значения 101 в этот объект. Это редко используемая форма, так как есть более короткий, в смысле записи, способ:

```
queue a(101);
```

В дальнейшем мы будем использовать именно этот способ. Общая форма передачи аргумента конструктора следующая:

```
class_type переменная(список_аргументов);
```

список аргументов - это отделенные запятыми фактические параметры, передаваемые функции конструктор.

```
# include <iostream h>
// Пример 65. Использование конструктора с параметром
// и перегруженных конструкторов

class queue {
    int q[100];
    int sloc, rloc;
    int id; // это значение будем определять
public:
    queue(void); // конструктор по умолчанию
    queue(int i); // конструктор инициализации
    ~queue(void); // деструктор
    void qput(int i);
    int qget(void);
};

queue::queue(void)
{
    rloc = sloc = 0;
    id = 0;
    cout << "Конструктор по умолчанию очереди " << id << "\n";
}

queue::queue(int i)
{
    rloc = sloc = 0;
    id = i;
    cout << "Конструктор инициализации очереди " << id << "\n";
}

queue::~~queue(void)
{
    cout << "Очередь " << id << " разрушена \n";
}

void queue::qput(int i)
{
    if (sloc == 100) {
        cout << " Очередь полна\n";
        return;
    }
    q[sloc++] = i;
}

int queue::qget(void)
{

```



```

if (rloc==sloc){
cout << " Очередь пуста\n ";
return 0;
}
return q[rloc++];
}

main()
{
queue a = queue(101), // Объявление объекта с инициализацией
b (102), // Объявление другого объекта
c; // Объявление объекта без передачи
    // параметра

a.qput(10);
b.qput(19);
a.qput(20);
b.qput(1);
for(int l=0; l<3; l++)
c.qput(200+l);

cout << a.qget() << " ";
cout << a.qget() << " ";
cout << b.qget() << " ";
cout << b.qget() << "\n";

for(l=0; l<3; l++)
cout << c.qget() << " ";
cout << "\n";

return 0;
}

```

Результатом работы программы должно быть следующее:

```

Конструктор инициализации очереди 101
Конструктор инициализации очереди 102
Конструктор инициализации очереди 0
10 20 19 1
200 201 202
Очередь 0 разрушена
Очередь 102 разрушена
Очередь 101 разрушена

```

Конструктор может иметь не только один аргумент, как было в примере. Аргументов может быть несколько, причем разного типа. Например,

```

#include <iostream.h>
// Пример 66. Конструктор с несколькими параметрами
class cl
{

```

```

int i;
float j;
public:
    cl (int a, float b);
    void show (void);
};

cl::cl(int a, float b)
{
    i = a; j = b;
}

void cl::show(void)
{
    cout << "i= " << i << " j=" << j << "\n";
}

main(void)
{
    cl x(1, 2.1), y(3, 4), z(100, 101.1);

    x.show();
    y.show();
    z.show();
    return 0;
}

```

## ДРУЖЕСТВЕННЫЕ ФУНКЦИИ

Одним из важных принципов языка C++ является возможность защиты данных от несанкционированного использования. Реализуется этот принцип, как мы видели, благодаря режиму доступа к членам класса. Члены класса с режимом доступа `private` могут использовать только функции - члены этого класса. Однако возникают ситуации, когда необходимо, чтобы к приватной части класса имела доступ функция, не являющаяся членом этого класса. Это можно сделать при объявлении класса, объявив эту функцию, используя ключевое слово `friend`.

```

class Cl{
    ...
public:
    friend void frd(void);
    ...
}

```

Функция `frd` объявлена как дружественная функция (friend function) класса `Cl`.

Причиной введения дружественных функций явилась ситуация, когда одна и та же функция должна использовать приватные элементы двух или

более разных классов. Рассмотрим пример объявления двух классов, в каждом из которых объявлена дружественная функция `int same_color(line l, box b)`, не являющаяся членом ни первого, ни второго класса. Объявление дружественной функции ни в коем случае не является нарушением принципа инкапсуляции данных, ибо сам класс разрешает доступ дружественной функции к приватной части класса.

```
class box{
    int color;
    int upx, upy;
    int lowx, lowy;
public:
    friend int same_color(line l, box b);
    void set_color(int c);
    void define_box(int x1, int y1, int x2, int y2);
    void show_box(void);
};

class line{
    int color;
    int startx, starty;
    int len;
public:
    friend int same_color (line l, box b);
    void set_color (int c);
    void define_line (int x, int y, int l);
    void show_line (void);
};
```

Дружественная функция в нашем примере должна сравнивать значение переменной `color` в каждом из классов и возвращать значение 1, если их значения одинаковы, и 0 в противном случае.

```
#include <iostream.h>
#include <conio.h>
// Пример 67. Использование дружественных функций

class line; // Предварительное объявление класса

class box{
    int color;
    int upx, upy;
    int lowx, lowy;
public:
    friend int same_color(line l, box b);
    void set_color(int c);
    void define_box(int x1, int y1, int x2, int y2);
    void show_box(void);
};
```

```
class line{
    int color;
    int startx, starty;
    int len;
public:
    friend int same_color (line l, box b);
    void set_color (int c);
    void define_line (int x, int y, int l);
    void show_line (void);
};

// Описание функций членов класса box
void box:: set_color (int c)
{
    color = c;
}

void box:: define_box (int x1, int y1, int x2, int y2)
{
    upx = x1; lowx = x2;
    upy = y1; lowy = y2;
}

void box:: show_box (void)
{
    window(upx, upy, lowx, lowy);
    textbackground(color);
    clrscr();
    textbackground(BLACK);
    window(1, 1, 80, 25);
}

//Описание функций членов класса line
void line:: set_color (int c)
{
    color = c;
}

void line:: define_line (int x, int y, int l)
{
    startx = x; starty = y; len = l;
}

void line:: show_line (void)
{
    textcolor(color);
    gotoxy(startx, starty);
    for(int k = 0; k< len; k++)
        cprintf("%c", '-');
```

```
    textcolor(WHITE);
}

// Описание дружественной функции
int same_color (line l, box b)
{
    if (l.color == b.color)
        return 1;
    return 0;
}

main (void)
{
    line l;
    box b;

    textbackground(BLACK);
    clrscr();

    b.define_box(5, 5, 25, 10);
    b.set_color(RED);
    l.define_line(5, 15, 30);
    l.set_color (BLUE);
    b.show_box();
    l.show_line();

    gotoxy(1, 1);
    if (same_color(l, b))
        cputs(" Same colors \n");
    else
        cputs(" Different colors \n");
    getch();

    b.define_box(45, 6, 70, 11);
    b.set_color(GREEN);
    l.define_line(45, 16, 30);
    l.set_color (2);
    b.show_box();
    l.show_line();

    gotoxy(41, 2);
    if (same_color(l, b))
        cputs(" Same colors \n ");
    else
        cputs(" Different colors \n ");
    getch();

    return 0;
}
```

Обратите внимание на предварительное (forward) объявление класса line

```
class line;
```

в начале программы. Это объявление похоже на объявление прототипа функций. Класс line используется в классе box до того, как объявлен шаблон класса line, поэтому мы обязаны сделать такое объявление.

Дружественной функцией может быть не только внешняя функция, но и функция - член другого класса. В предыдущем примере можно было объявить функцию same\_color() функцией - членом класса box, а в классе line - дружественной функцией:

```
class line;
class box{
    int color;
    int upx, upy;
    int lowx, lowy;
public:
    int same_color(line l);
    void set_color(int c);
    void define_box(int x1, int y1, int x2, int y2);
    void show_box(void);
};
class line{
    int color;
    int startx, starty;
    int len;
public:
    friend int box::same_color (line l);
    void set_color (int c);
    void define_line (int x, int y, int l);
    void show_line (void);
};
```

Обратите внимание на то, что в объявлении класса line указано полное имя функции box::same\_color(). Кроме того, можно не указывать в качестве аргумента объект класса box. Выглядеть новая функция box::same\_color() может следующим образом:

```
// Описание дружественной функции
int box::same_color (line l)
{
    if (l.color == color)
        return 1;
    return 0;
}
```

Читателю будет нетрудно переписать пример 67 для этого случая.

## ДРУЖЕСТВЕННЫЕ КЛАССЫ

При объявлении класса можно объявить сразу все функции - члены другого класса дружественными одним объявлением:

```
class X {...};
class Y {
    //...
    friend class X;
    //...
};
```

Объявление класса дружественным предполагает, что приватные и защищенные члены класса Y могут использоваться в классе X.

## АРГУМЕНТЫ ФУНКЦИИ, ЗАДАВАЕМЫЕ ПО УМОЛЧАНИЮ

Язык C++ предоставляет пользователю возможность устанавливать значения параметров функции по умолчанию, если при вызове функции аргументы не были указаны.

Пусть функция f() имеет единственный целый аргумент и его значение по умолчанию должно равняться 10. Описание функции может быть следующим:

```
void f(int i = 10)
{
    ...
}
```

Вызвать функцию, определенную таким образом, можно двумя способами:

```
f(1); // передается значение аргумента i, равное 1
f();  // передается значение i по умолчанию, равное 10
```

Чтобы лучше понять удобство установки аргументов по умолчанию, приведем пример полезной функции, отсутствующей в библиотеке Borland C++. Эта функция при работе с текстовым экраном выводит указанную строку начиная с заданного места экрана. Назовем ее stringxy():

```
void stringxy(char *str, int x = -1, int y = -1)
{
    if (x == -1) x = wherex();
    if (y == -1) y = wherey();
    gotoxy(x, y);
    cout << str;
}
```

Эта функция работает в текстовом режиме экрана. Функции wherex() и wherey() являются библиотечными функциями Borland C++, их прототипы находятся в файле conio.h, они возвращают текущие координаты курсора x

и у соответственно. Использование функции `stringxy()` приведено в следующем примере:

```
#include <iostream.h>
#include <conio.h>
// Пример 68. Аргументы функции, задаваемые по умолчанию
// Прототип функции с аргументами, задаваемыми по умолчанию
void strtoutxy(char *str, int x=1, int y=-1);

main(void)
{
    stringxy(" Это линия 11\n", 1, 11);
    getch();
    stringxy(" Продолжение на той же строке\n ");
    getch();
    stringxy(" Линия 13\n", 20, 13);
    getch();
    stringxy(" С позиции 60\n", 60);
    getch();
    return 0;
}
// Описание функции с аргументами, задаваемыми по умолчанию
// Обратите внимание на отличие заголовка функции от прототипа
// функции
void stringxy(char *str, int x, int y)
{
    if (x==-1) x=wherex();
    if (y==-1) y=wherey();
    gotoxy(x, y);
    cout << str;
}
```

Обратим внимание на то, что при вызове функции `stringxy()` из функции `main()` указано разное количество аргументов. Это допускается реализацией языка C++. Если определены все три параметра, то функция работает естественным образом. Если определены два параметра, то значение `y` устанавливается по умолчанию равным `-1`, а в результате работы функции устанавливается равным значению координаты `y` курсора. Если указан один параметр, то `x` и `y` будут определены по умолчанию. Можно ли задать `y` в явном виде, а `x` по умолчанию?

`stringxy(" Линия 13\n", , 13);` // Неправильный вызов

Нет, нельзя. Поэтому все аргументы, задаваемые по умолчанию, должны находиться после аргументов, значения которых не заданы по умолчанию.

Важно помнить, что все параметры, которым присваиваются значения по умолчанию, должны быть правее тех параметров, значения которых по умолчанию не устанавливаются.

Далее приведены два примера неправильного объявления функций:



```
void stringxy(char *str, x=-1, y); // Неправильное объявление
void stringxy(x=-1, y=-1, char*str); // Неправильное объявление
```

При использовании функции с аргументами, задаваемыми по умолчанию, и перегруженных функций одновременно необходимо следить за корректностью их использования.

При объявлении

```
void f(void);
void f(int i = 0);
```

и вызове

```
f();
```

компилятор выдаст ошибку, так как появляется неоднозначность выбора функции.

Устанавливать значения параметров по умолчанию можно и для конструктора класса. Вернемся к конструктору класса `queue`. То, что в примере 66 мы делали объявлением двух конструкторов (перегрузкой конструкторов), можно сделать с помощью одного конструктора, с параметром, задаваемым по умолчанию:

```
queue::queue(int i=0)
{
    floc=sloc=0;
    id=i;
    cout << "Очередь "<< id<< " инициализирована\n";
}
```

При таком описании конструктора можно объявлять объекты класса `queue` следующим образом:

```
queue a(5), b;
```

Значение `id` для первого объекта (a) будет равно пяти, для второго (b) будет равно нулю.

Использовать начальную установку параметров следует корректно. Целесообразность начальной установки по умолчанию зависит от того, как часто надо устанавливать значение по умолчанию. Если в подавляющем большинстве случаев требуется одно и то же значение, то установка по умолчанию имеет смысл.

## СТРУКТУРЫ И КЛАССЫ

Структуры в языке C++ обладают теми же возможностями, что и классы. В C++ классы и структуры, за одним исключением, могут быть взаимозаменяемы. Структура может включать данные и правила обработки этих данных. Различие в языке C++ структур и классов состоит в том, что члены класса по умолчанию имеют режим доступа `private`, а в структуре члены

структуры по умолчанию `public`. За этим исключением объявление структуры и класса одинаково.

Можно вместо класса `queue` объявить структуру `queue`:

```
struct queue {
    queue(void); // конструктор по умолчанию

    queue(int i); // конструктор инициализации

    ~queue(void); // деструктор
    void qput(int i);
    int qget(void);
private:

    int q[100];
    int sloc, rloc;
    int id; // это значение будем определять
};
```

Объявление объекта структуры `queue` производится так же, как и для класса:

```
queue obj(1), obj2;
```

Можно переписать пример 65 для случая структуры и убедиться, что никаких изменений в работе программы не произойдет.

## ОБЪЕДИНЕНИЯ И КЛАССЫ

Объединения (`union`) и классы также имеют много общего. По сути объединение - это структура, в которой для всех элементов выделено общее место в памяти. Объединение может иметь конструктор, деструктор и дружественные функции. Все члены объединения по умолчанию `public`. Ключевые слова `public`, `private` и `protected` не могут использоваться в объединениях. Кроме того, объединения не могут участвовать в механизме наследования ни как базовый, ни как производный тип. Если требуется использовать объединение в стиле языка C, то это можно делать без каких-либо ограничений.

В языке C++ разрешены так называемые безымянные, или неименованные (`anonymous`), объединения:

```
union { список_членов };
```

Неименованные объединения определяют объект, а не тип. Имена членов неименованного объединения должны отличаться от других имен из области действия, эти имена используются непосредственно, т. е. без операции точка. Но, поскольку они являются членами объединения, для них выделяется в памяти одно и то же место. Они имеют один и тот же адрес.

```
#include <iostream.h>
```

```
// Пример 69. Использование неименованных объединений
```

```
main()
{
    union {
        long l;
        unsigned char ch[4];
    };

    cout << "Введите число типа long: ";
    cin >> l;
    cout << "Младший байт: " << int(ch[0]) << "\n";
    cout << "Второй байт: " << int(ch[1]) << "\n";
    cout << "Третий байт: " << int(ch[2]) << "\n";
    cout << "Старший байт: " << int(ch[3]) << "\n\n";

    return 0;
}
```

Программа этого примера "разбирает" число типа long по байтам и печатает целые значения этих байтов. Используется основная идея применения объединений - трактовка одного и того же места в памяти как переменные разного типа.

### ПОДСТАВЛЯЕМЫЕ (INLINE) ФУНКЦИИ

C++ имеет еще одну важную черту, которой нет в языке C. Это подставляемые функции (inline function). Подставляемые функции не вызываются как обычные функции языка C++. В машинный код программы вставляется код, соответствующий этой функции. Это как бы параметризованная макроподстановка, но уже на процессе компиляции.

Существует два способа создания inline-функций. Первый - используется модификатор inline. Например,

```
inline int f(void)
{
    ...
}
```

т. е. перед объявлением функции ставится модификатор inline. Он предшествует всем другим модификаторам. Причина введения таких функций - повышение эффективности работы программы. Каждый раз, когда происходит вызов функции, выполняется несколько действий, включая пересылку параметров в стек и возврат значения функции. Требуется несколько тактов процессора (может быть достаточно много) для выполнения этих действий. Когда же используется inline-функция, на вызов такой функции время не тратится, соответственно возрастает скорость выполнения программы. Но возрастает и размер кода программы, особенно если функция достаточно большая. Таким образом, лучше использовать inline-функцию тогда, когда функция очень маленькая. Большие по размеру функции рекомендуется вызывать обычным образом.

Рассмотрим пример:

```
#include <iostream.h>
```

```
// Пример 70.
```

```
class cl{
    int i;
public:
    int get_i(void);
    void put_i(int j);
};
inline int cl::get_i(void)
{
    return i;
}
inline void cl::put_i(int i)
{
    i=j;
}
main(void)
{
    cl s;
    s.put_i(10);
    cout << s.get_i();
    return 0;
}
```

Важно помнить, что для компилятора `inline` это требование, а не команда создания `inline`-кода. Есть различные ситуации, препятствующие созданию `inline`-кода. Это происходит тогда, когда функция содержит циклы, операторы `switch`, `goto` или возвращает значение более одного раза. `Inline`-функции не могут быть рекурсивными или содержать статические (`static`) переменные.

Функция, имеющая модификатор `inline`, может быть как функцией-членом какого-либо класса, так и обычной функцией.

Другой способ создания подставляемых функций в языке C++ состоит в описании функции внутри объявления шаблона класса. Любая функция, определенная внутри шаблона класса, автоматически делается `inline`-функцией. Поэтому нет необходимости предварять ее определение ключевым словом `inline`.

Например:

```
#include <iostream.h>
```

```
// Пример 71.
```

```
class cl{
    int i;
public:
```

```

    int get_i(void){ return i; }
    void put_i(int j){ i=j; }
};

    main(void)
{
    cl s;
    s.put_i(10);
    cout << s.get_i();

    return 0;
}

```

Этот пример написан в стиле языка C++. В C++ короткие функции обычно определяются, как в этом примере, внутри класса.

## НАСЛЕДОВАНИЕ КЛАССОВ

Обсудим более детально понятие наследования классов. Начнем с терминологии. Класс, который наследуется, называется базовым классом (base class). Класс, который является наследником, называется производным классом (derived class). Иногда такой класс называют потомком (child class). В языке C++ приняты термины "базовый (base) класс" и "производный (derived) класс".

Напомним, что в языке C++ элементы класса могут быть трех режимов доступа: общие (public), приватные (private) и защищенные (protected).

Общие элементы могут быть доступны другим функциям программы. Приватные элементы могут быть доступны только функциям-членам или дружественным (friend) функциям. Защищенные элементы также могут быть доступны только функциям-членам или дружественным функциям.

Когда один класс наследует другой класс, все приватные элементы базового класса недоступны для производного класса. Например:

```

class X{ // Базовый класс X
    int i;
    int j;
public:
    void get_ij(void);
    void put_ij(void);
};
class Y:public X{ // Производный класс Y
    int k;
public:
    int get_k(void);
    void make_k(void);
};

```

В этом примере X - базовый класс, Y - производный класс. Функции - члены класса Y могут использовать общие функции get\_i() и put\_i(), но

не могут использовать `i` и `j`, так как они приватные для `X` и, соответственно, недоступны для функций `get_k()`, `put_k()` класса `Y`.

Можно обеспечить доступ членов-функций класса `Y` к элементам `i` и `j` класса `X`. Для этого в классе `X` надо объявить их защищенными - `protected`:

```
class X{
protected:
    int i;
    int j;
public:
    void get_ij(void);
    void put_ij(void);
};
class Y:public X{
    int k;
public:
    int get_k(void);
    void make_k(void);
};
```

Теперь члены класса `Y` имеют доступ к членам `i`, `j` класса `X`. В то же время `i`, `j` остаются недоступными для остальной части программы. Доступ осуществляется к элементам, объявленным защищенными или общими, но не наследуется для приватных элементов.

Основная форма наследования -

```
class имя_производного_класса : режим_доступа имя_базового_класса
{
    ...
};
```

Здесь режим доступа может быть `public`, `protected` или `private`. (Режим доступа может отсутствовать, в этом случае предполагается, что режим доступа `public`, если производный класс структура, и `private`, если производный класс `class`.)

Если режим доступа `public`, то все общие и защищенные члены базового класса остаются общими и защищенными членами производного класса. Если режим доступа `private`, то все общие и защищенные элементы базового класса становятся приватными элементами производного класса. Если режим доступа `protected`, то все общие и защищенные члены базового класса становятся защищенными членами производного класса. Сказанное выше иллюстрируется следующей схемой.

## Режим доступа

к элементу в базовом классе	при наследовании класса	к элементу в производном классе
private	public ----->	Недоступен
protected		protected
public		public
private	protected ----->	Недоступен
protected		protected
public		protected
private	private ----->	Недоступен
protected		private
public		private

Чтобы проиллюстрировать передачу режима доступа к переменным при наследовании, рассмотрим все это на модельном примере.

```
#include <iostream.h>
```

```
// Пример 72. Наследование режима доступа
```

```
class X {
protected:
    int i;
    int j;
public:
    int get_ij (void);
    void put_ij (void);
};
```

```
// i, j класса X стали protected членами в классе Y
```

```
class Y: public X {
    int k;
public:
    int get_k(void);
    void make_k(void);
};
```

```
// Z имеет доступ к переменным i, j класса X
```

```
// Z не имеет доступа к k класса Y
```

```
class Z: public Y{
public:
    void f(void);
};
```

```
void X:: get_ij(void)
```

```
{
```

```
    cout << "Введите два числа";
```

```
cin >> i >> j;
}
void X::put_ij(void)
{
    cout << "i=" << i << " j=" << j << "\n";
}
int Y::get_k(void)
{
    return k;
}
void Y::make_k(void)
{
    k = i * j;
}
void Z::f(void)
{
    i = 2; j = 3;
}
```

```
main()
{
    Y var;
    Z var2;

    var.get_ij();
    var.put_ij();
    var.make_k();
    cout << var.get_k();
    cout << "\n";

    var2.f();
    var2.put_ij();
    var2.make_k();
    cout << var.get_k();
    cout << "\n";
    return 0;
}
```

Если мы заменим режим доступа при наследовании класса X на `private`, то функция `void Z::f(void)` не будет иметь право доступа к переменным `i, j`.

Если же мы заменим режима доступа при наследовании класса Z на `private`, не изменяя режим доступа при наследовании класса X, то действие примера по сравнению с начальным не изменится.

### КОНСТРУКТОРЫ С ПАРАМЕТРАМИ ПРИ НАСЛЕДОВАНИИ

Рассмотрим еще один модельный пример. Класс X задает переменную `x`, класс Y задает переменную `y`, класс Z содержит переменную `z`, которая



будет содержать произведение  $x \cdot y$ . Каждый из классов содержит функцию `void show(void)`. В примере продемонстрировано использование функций, имеющих одинаковые имена в базовом и производном классах. Обе эти функции доступны, так как у них различаются полные имена, соответственно `Z::show()`, `X::show()` и `Y::show()`. На этом примере можно экспериментировать, изменяя режим доступа к переменным `x` и `y`, режим доступа при наследовании классов.

```
#include <iostream.h>

// Пример 73. Простое наследование. Конструкторы с параметрами.
class X {
protected:
    int x;
public:
    X (int i); // Конструктор с параметром
    ~X (void); // Деструктор
    void put_x(int i) { x = i; }
    int get_x(void) { return x; }
    void show (void);
};

class Y: public X {
protected:
    int y;
public:
    Y (int i, int j); // Конструктор с параметром
    ~Y (void); // Деструктор
    void put_y(int i) { y = i; }
    int get_y(void) { return y; }
    void show (void);
};

class Z: public Y {
protected:
    int z;
public:
    Z (int i, int j); // Конструктор с параметром
    ~Z (void); // Деструктор
    void make_z (void);
    void show (void);
};

X::X (int i)
{
    x = i;
    cout << "Конструктор X \n";
}

X::~X (void)
{
    cout << "Деструктор X \n";
}
```

```
void X:: show(void)
{
    cout << " x = " << x << "\n";
}
Y:: Y (int i, int j): X(i) // Конструктор класса Y передает значение
    // конструктору класса X, поэтому он должен
    // иметь два параметра
{
    y = j;
    cout << "Конструктор Y \n";
}
Y::~~ Y (void)
{
    cout << "Деструктор Y \n";
}
void Y:: show(void)
{
    cout << " x = " << x << " y = " << y << "\n";
}
Z:: Z (int i, int j): Y(i, j) // Конструктор класса Z передает значения
    // своих параметров конструктору Y
{
    cout << "Конструктор Z \n";
}
Z::~~ Z (void)
{
    cout << "Деструктор Z \n";
}
void Z:: make_z (void)
{
    z=x * y;
}
void Z:: show(void)
{
    cout << z << " = " << x << " * " << y << "; \n";
}

main(void)
{
    Z zobj (3, 5); // Создание и инициализация объекта

    zobj.make_z();
    zobj.show();
    zobj.X::show(); // Обратите внимание на вызов функций
    zobj.Y::show(); // show() из разных классов

    zobj.put_x(7); // Задание новых значений переменных x и y
    zobj.put_y(9);
    zobj.make_z();
}
```

```
zobj.show();
zobj X.show();
zobj Y.show();
```

```
return 0;
}
```

Деструкторы классов мы ввели для иллюстрации их вызовов при наследовании классов.

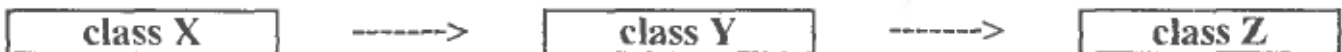
По этой же причине конструкторы классов выдают сообщения.

Обратите внимание на передачу параметров конструктора производного класса конструктору базового класса:

```
Y::Y(int i, int j): X(i) { //... }
```

Попытка просто изменить режим доступа при наследовании классов на `private`, вместо `public` без каких-либо других изменений, вызовет ошибки при компиляции. Для того чтобы изменить программу так, чтобы при этом режиме наследования программа работала правильно, необходимо использовать функции `get_x()` и `get_y()`, которые в предыдущем примере не использовались. Попробуйте заменить режим доступа к переменным `x` и `y` на `private`.

В этом примере использована следующая схема наследования:



При этом классы `X` и `Y` по своей сути равноправны, но из-за реализации простого наследования класс `Y` существенно отличается от класса `X`, хотя бы числом параметров конструктора.

Избавиться от неравноправия классов можно используя механизм множественного наследования.

## МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

До сих пор у нас каждый производный класс содержал один базовый. Язык C++ позволяет наследование не только от одного, а одновременно от нескольких классов. Форма наследования в этом случае следующая:

```
class имя-производного-класса: список базовых классов{
//...
};
```

Список базовых классов содержит перечисленные через запятую базовые классы с соответствующими режимами доступа к каждому из базовых классов, например:

```
class Z: public U, public V, private W {
//...
}
```

Пример 73 можно переписать в более естественном для него виде с множественным наследованием, когда классы X и Y будут равноправны. Такое наследование будет реализовать следующую схему:



```
#include <iostream.h>
```

```
// Пример 74. Множественное наследование.
```

```
// Конструкторы с параметрами.
```

```

class X {
protected:
    int x;
public:
    X (int i); // Конструктор с параметром
    ~X (void); // Деструктор
    void put_x(int i) { x = i; }
    int get_x(void) { return x; }
    void show (void);
};

class Y {
protected:
    int y;
public:
    Y (int i ); // Конструктор с параметром
    ~Y (void); // Деструктор
    void put_y(int i) { y = i; }
    int get_y(void) { return y; }
    void show (void);
};

class Z: public X, public Y {
protected:
    int z;
public:
    Z (int i, int j); // Конструктор с параметром
    ~Z (void); // Деструктор
    void make_z (void);
    void show (void);
};

X::X (int i)
{
    x = i;
    cout << "Конструктор X \n";
}

X::~~ X (void)
{
    
```

```

    cout << "Деструктор X \n";
}
void X:: show(void)
{
    cout << " x = " << x << "\n";
}
Y:: Y (int i )
{
    y = i;
    cout << "Конструктор Y \n";
}
Y::~ Y (void)
{
    cout << "Деструктор Y \n";
}
void Y:: show(void)
{
    cout << " y = " << y << "\n";
}
Z:: Z (int i, int j): Y( j), X(j) // Конструктор класса Z передает
    // значения своих параметров
    // конструкторам X и Y
{
    cout << "Конструктор Z \n";
}
Z::~ Z (void)
{
    cout << "Деструктор Z \n";
}
void Z:: make_z (void)
{
    z=x * y;
}
void Z:: show(void)
{
    cout << z << " = " << x << " * " << y << "; \n";
}

main(void)
{
    Z zobj (3, 5); // Создание и инициализация объекта

    zobj.make_z();
    zobj.show();
    zobj.X::show(); // Обратите внимание на вызов функций
    zobj.Y::show(); // show() из разных классов

    zobj.put_x(7); // Задание новых значений переменных x и y
    zobj.put_y(9);
}

```

```
zobj make_z();  
zobj show();  
zobj X::show();  
zobj Y::show();
```

```
return 0;  
}
```

Использование конструкторов, выдающих сообщение, позволяет убедиться в том, что базовые классы создаются в том порядке, в котором они перечислены в списке базовых классов при объявлении производного класса Z.

Пока конструкторы базовых классов не имеют аргументов, то производный класс может не иметь функцию-конструктор. Если же конструктор базового класса имеет один или несколько аргументов, каждый производный класс обязан иметь конструктор. Чтобы передать аргументы в базовый класс, нужно определить их после объявления конструктора базового класса так, как указано в основной форме:

```
конструктор-производного-класса(список аргументов):  
    базовый-класс1(список аргументов),
```

```
    ...,  
    базовый-классN(список аргументов)  
{  
    ...  
}
```

Здесь базовый-класс1... базовый-классN - имена конструкторов базовых классов, которые наследуются производным классом. Двосточие отделяет имя конструктора производного класса от списка аргументов базового класса. Список аргументов, ассоциированный с базовым классом, может состоять из констант, глобальных параметров и/или параметров для конструктора производного класса. Так как объект инициализируется во время выполнения программы, можно в качестве параметров использовать переменные.

## ПЕРЕДАЧА ОБЪЕКТОВ КАК АРГУМЕНТОВ ФУНКЦИЙ

Объект передается функции тем же способом, каким передаются и переменные другого типа. В C++, как и в языке C, объекты передаются функции по значению. Это значит, что копия объекта передается в функцию и, таким образом, любые изменения, происходящие с элементами объекта внутри функции, не изменяют его самого.

```
#include <iostream.h>  
// Пример 75. Передача объектов как аргументов функции  
class OBJ{  
    int i;
```

```

public:
    void set_i (int x){ i=x; }
    void out_i (void){ cout << i << " "; }
};

void f(OBJ x); // Прототип функции

main(void)
{
    OBJ A;

    A.set_i (10);
    f(A);
    A.out_i ( ); // Значение i не изменилось

    return 0;
}

void f (OBJ x)
{
    x.out_i ( ); // Вывод x=10
    x.set_i (100); // Изменение x
    x.out_i ( ); // Вывод x=100
}

```

Можно передать параметр объекта и по ссылке, т. е. передать адрес объекта. Тогда изменения, произведенные функцией в объекте, могут изменить этот объект. При передаче в качестве параметра функции по значению объекта некоторого класса, как и при передаче в качестве параметра простой переменной, произойдет создание копии объекта, тем самым вызовется конструктор, а затем и деструктор. Это может привести к побочным эффектам, особенно если конструктор динамически выделяет память. В языке C++ вводится понятие ссылочной переменной, при использовании которой в качестве параметра не происходит создания копии объекта. В то же время возможно изменить значение данных - членов этого объекта. Об использовании ссылочных переменных речь пойдет выше.

## МАССИВЫ ОБЪЕКТОВ

Можно создать массив объектов совершенно так же, как и массив элементов другого типа данных. Точно так же можно создать и многомерные массивы. Рассмотрим пример использования массива объектов.

```

#include <iostream.h>

// Пример 76. Использование массива объекта

enum disp_type{mono, cga, ega, vga };

```

```
class display {
    int colors;           // число цветов
    enum disp_type dt;    // тип дисплея
public:
    void set_colors (int num) { colors = num; }
    int get_colors( ) { return colors; }
    void set_type (enum disp_type t) { dt=t; }
    enum disp_type get_type(void) { return dt; }
};
char names[4][5] = {
    "mono",
    "cga",
    "ega",
    "vga"
};

main(void)
{
    display monitor[4]; // Объявление массива объектов
    int i;

    monitor[0].set_type(mono);
    monitor[0].set_colors(1);
    monitor[1].set_type(cga);
    monitor[1].set_colors(4);
    monitor[2].set_type(ega);
    monitor[2].set_colors(16);
    monitor[3].set_type(vga);
    monitor[3].set_colors(16);
    for (i=0; i<=3; i++) {           // Использование элементов
        // массива объектов
        cout << names[monitor[i].get_type( )] << " ";
        cout << " has " << monitor[i].get_colors( );
        cout << " colors \n";
    }
    return 0;
}
```

Результатом работы программы будет следующее:

```
mono has 1 colors
cga  has  4 colors
ega  has 16 colors
vga  has 16 colors
```

## УКАЗАТЕЛЬ НА ОБЪЕКТ

Как и на другие типы данных, на объект можно ссылаться через указатель. Чтобы получить доступ к элементу конкретного объекта, используется операция точка. Чтобы получить доступ к элементу объекта, когда исполь-



зуется указатель на объект, используется операция стрелка. Использование операций точка и стрелка аналогично использованию этих операций для структур и объединений в языке C.

```
class p_example{
//...
};
main(void)
{
    p_exsample ob, *p;
                                // ob - объект
                                // p - указатель на объект
    ob.set_num(1);              // прямой доступ
    p=&ob;                       // присвоение адреса
    p->set_num(2);               // доступ через указатель
    return 0;
}
```

## ПЕРЕГРУЗКА ФУНКЦИЙ И ОПЕРАЦИЙ

Мы уже обсуждали перегрузку функций в языке C++. Кроме того, язык C++ позволяет также перегрузить и существующие операции. Когда мы говорим о перегрузке операций, мы имеем в виду знак операции, придание ему нового смысла. Рассмотрим эти возможности подробнее.

### ПЕРЕГРУЗКА КОНСТРУКТОРОВ

Функции-конструкторы играют специфическую роль в языке C++. Тем не менее они могут быть перегружены так же, как и другие функции. Перегрузка конструкторов класса осуществляется просто объявлением конструкторов с различным набором параметров.

Мы уже видели ранее класс с двумя конструкторами. Проиллюстрируем перегрузку конструкторов еще одним примером.

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
// Пример 77. Перегрузка конструкторов
class timer{
    int seconds;
public:
    // Конструктор по умолчанию
    timer(void) { seconds = 5;}
    // Время задается строкой
    timer(char *t) { seconds=atoi(t);}
    // Задание секунд целым числом
    timer(int t){ seconds=t;}
    // Задание минутами и секундами
    timer(int min, int sec){ seconds=60*min + sec;}
```

```
void run(void);
},

void timer_run(void)
{
    clock_t t1, t2;
    t1 = t2 = clock( ) / CLK_TCK;
    while (seconds) {
        if (t1 / CLK_TCK + 1 <= (t2 = clock( )) / CLK_TCK) {
            seconds--;
            t1 = t2;
        }
    }
    cout << "\a"; // Сигнал
}

main(void)
{
    timer a(10), b("20"), c (1, 10), d;
    cout<< "Подождите 10 секунд ...\n";
    a.run( ); // 10 секунд
    cout<< "Подождите 20 секунд ...\n";
    b.run( ); // 20 секунд
    cout<< "Подождите 1 минуту 10 секунд ...\n";
    c.run( ); // 1 минута 10 секунд
    cout<< "Подождите 5 секунд ...\n";
    d.run( );
    return 0;
}
```

Эта программа использует функцию `clock( )` системы Borland C++. Эта функция возвращает число тиков системных часов, прошедших после начала выполнения программы. Макрос `CLK_TCK` является числом тиков в секунду. И функция, и макрос объявлены в заголовочном файле `time.h`. Количество тиков, деленное на `CLK_TCK`, дает число секунд.

### ДИНАМИЧЕСКАЯ ИНИЦИАЛИЗАЦИЯ И ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

В языке C есть требование, чтобы все локальные переменные были объявлены в начале блока. После появления оператора в блоке уже нельзя объявить переменную в этом блоке. C++ снимает это ограничение, т. е. можно объявить локальную переменную в любом месте блока, но до ее первого использования. Объявление переменной непосредственно перед ее первым использованием может помочь избежать побочных эффектов.

В языке C++ и локальные, и глобальные переменные могут инициализироваться во время выполнения программы. Язык C требует, чтобы переменная инициализировалась константой. В языке C++ переменная может быть инициализирована с помощью некоторого выражения во время выполнения программы.

Так же как и другие переменные, объекты могут инициализироваться динамически во время их создания. Это позволяет создавать точно тот объект, который нужен, используя информацию, которая становится известной только во время выполнения программы. Для иллюстрации динамической инициализации используем программу таймер, рассмотренную ранее. В первом использовании этого примера все объекты инициализировались константами. В предлагаемом примере инициализация проводится во время выполнения программы. Инициализируются два объекта `b` и `c`:

```
#include <iostream.h>
#include <stdlib.h>
#include <time.h>
// Пример 78. Динамическая инициализация объектов

class timer{
    int seconds;
public:
    // Конструктор по умолчанию
    timer(void) { seconds = 5;}

    // время задается строкой
    timer(char *t) { seconds=atoi(t);}
    // задание секунд целым числом
    timer(int t){ seconds=t;}
    // задание времени минутами и секундами
    timer(int min, int sec){ seconds=60*min+sec;}
    void run(void);
};

void timer::run(void)
{
    clock_t t1, t2;
    t1=t2=clock( )/CLK_TCK;
    while(seconds){
        if (t1/CLK_TCK+1<= (t2=clock( ))/CLK_TCK){
            seconds--;
            t1=t2;
        }
    }
    cout << "\a"; // Сигнал
}

main(void)
{
    timer a(10) // инициализация во время компиляции
    a.run( );
    cout << "Введите количество секунд:";
    char str[80];
    cin >> str;
    timer b(str); // Инициализация во время выполнения
```

```
b.run( );  
cout << "Введите минуты и секунды:";  
int min, sec;  
cin >> min >> sec;  
timer c(min, sec); // Инициализация во время выполнения  
c.run( );  
timer d;  
  
return 0;  
}
```

## КЛЮЧЕВОЕ СЛОВО THIS

При вызове функции - члена класса передается еще один неявный параметр - указатель на объект класса, который вызывает данную функцию. Этот указатель называется `this`.

Ключевое слово `this` используется, в частности, при перегрузке операций.

Как известно, функции - члены класса могут иметь доступ к `private` - членам их класса. Например,

```
class cl{  
    int i;  
  
};
```

Для данного класса функция - член класса может иметь оператор присваивания `i=10`; В действительности этот оператор является короткой записью оператора

```
this->i=10;
```

Чтобы понять, как указатель `this` работает, рассмотрим следующую короткую программу:

```
#include <iostream.h>  
    // Пример 79. Использование указателя this  
  
class cl{  
    int i;  
public:  
    void load_i (int val){ this->i=val; } // аналогично i=val;  
    int get_i (void){ return this->i; }   // аналогично return i;  
};  
void main(void)  
{  
    cl c;  
    c.load_i(100);  
    cout << c.get_i( );  
}
```

Программа выводит число 100.

## ПЕРЕГРУЗКА ОПЕРАЦИЙ

За небольшим исключением большинству операций языка C++ может быть придано специальное значение относительно вновь определенных классов. Для встроенных типов данных значение операции изменить нельзя. Например, класс, который определяет список указателей, может использовать операцию `+` для добавления объекта в список. Когда операция перегружена, ни одно из ее исходных значений не теряется. Просто вводится новая операция относительно нового конкретного класса.

Чтобы перегрузить операцию, надо определить, что эта операция значит относительно класса, к которому она будет применяться. Для этого создается специальная функция операции (*operator function*), которая определяет действие этой операции.

Основная форма функции-операции, являющейся функцией - членом класса, -

```
тип имя-класса::операция #(список аргументов)
{
    // операторы, определяющие действие
}
```

Здесь тип - тип возвращаемого значения, # - конкретный знак операции. Часто возвращаемое значение того же типа, что и класс, хотя возможен и другой тип этого значения. Функция операции должна быть или членом класса или дружественной функцией. Небольшие отличия в случае члена класса или дружественной функции имеются.

Рассмотрим на примере как реализуется перегрузка операций. Рассмотрим программу, которая перегружает операции `+` и `=` относительно класса `vector`. Класс `vector` определяет трехмерный вектор в евклидовом пространстве. Операция сложения двух векторов выполняется как сложение соответствующих координат:

```
#include <iostream.h>
// Пример 80. Перегрузка операций + и =

class vector{
    int x, y, z; // Три координаты
public:
    vector operator+ (vector t);
    vector operator= (vector t);
    void show(void);
    void assign (int mx, int my, int mz);
};
// Перегрузка операции +
vector vector::operator+ (vector t)
{
    vector temp;

    temp.x=x+t.x;
```

```

    temp.y=y+t.y;
    temp.z=z+t.z;
    return temp;
}
// Перегрузка операции присваивания =
vector operator = (vector t)
{
    x=t.x;
    y=t.y;
    z=t.z;
    return *this; // Использовали this
}

void vector::show (void)
{
    cout << x <<" ";
    cout << y <<" ";
    cout << z <<" \n";
}

void vector::assign(int mx, int my, int mz)
{
    x = mx;
    y = my;
    z = mz;
}

main(void)
{
    vector a, b, c;
    a.assign(1, 2, 3);
    b.assign(10, 10, 10);
    a.show( );
    b.show( );
    c=a+b; // Использование перегруженных операторов
    c.show( );
    c = a + b + c;
    c.show( );
    c = b = a;
    c.show ( );
    b.show ( );
    return 0;
}

```

Как видно, каждая функция операции имеет только один параметр, в то время как сами функции определяют бинарные операции. Другой аргумент неявно передается с использованием `this`-указателя. Строка `temp.x=x+t.x`; аналогична строке `temp.x=this->x+t.x`, т. е. `x` ссылается на `this->x`. Здесь `this` ассоциируется с объектом, предшествующим знаку операции. Объект справа знака операции передается как параметр функции.

Если используются функции-члены, то не нужны параметры для унарных операций и требуется лишь один параметр для бинарных операций. Во всех случаях объект, активизирующий операцию, передается в функцию-операцию неявно через указатель `this`.

В этом примере важно то, что возвращаемое значение имеет тип `vector`. Это позволяет использовать выражение типа `a+b+c`. Перегруженная операция `+` не изменяет значения своих операндов. В то же время перегруженная операция `=` модифицирует операнд, стоящий слева.

Можно также перегрузить операции `++` и `--`. Это унарные операции. В предыдущем примере в определение класса можно вставить объявление функции:

```
vector operator++(void);
```

и описать функцию-операцию:

```
vector vector::operator++(void)
{
    x++;
    y++;
    z++;
    return *this;
}
```

В то же самое время префиксная операция

```
++obj;
```

даст тот же результат, хотя этот знак операции должен был бы быть связан с левым операндом.

```
#include <iostream.h>
```

```
// Пример 81. Перегрузка операций ++
```

```
class vector{
    int x, y, z; // Три координаты
public:
    vector operator+ (vector t);
    vector operator= (vector t);
    vector operator++ (void);
    void show(void);
    void assign (int mx, int my, int mz);
};
// Перегрузка операции +
vector vector::operator+ (vector t)
{
    vector temp;

    temp.x = x + t.x;
    temp.y = y + t.y;
```

```
    temp.z=z+t.z;
    return temp;
}
// Перегрузка операции присваивания =
vector vector::operator = (vector t)
{
    x=t.x;
    y=t.y;
    z=t.z;
    return *this; // Использовали this
}

vector vector::operator++(void)
{
    x++;
    y++;
    z++;
    return *this;
}

void vector::show (void)
{
    cout << x <<" ";
    cout << y <<" ";
    cout << z <<" \n";
}

void vector::assign(int mx, int my, int mz)
{
    x = mx;
    y = my;
    z = mz;
}

main(void)
{
    vector a, b, c;
    a.assign(1, 2, 3);
    b.assign(10, 10, 10);
    a.show( );
    b.show( );
    c=a+b; // Использование перегруженных операторов
    c.show( );
    c = a + b + c;
    c.show( );
    c = b = a;
    c.show( );
    b.show( );
    c++; // Использование постфиксной операции
```



```

c.show( );
    ++c;    // Использование префиксной операции
c.show( );
return 0;
}

```

Сначала в C++ не было возможности отличить префиксную операцию от постфиксной. К настоящему времени появилась возможность отличать префиксную операцию ++ от постфиксной операции. Объявление этих функций-операций почти одинаково, но в постфиксной операции добавляется фиктивный целочисленный указатель при описании функции-операции (этот фиктивный аргумент никак не используется и не указывается при вызове).

```

vector operator++(void); // префиксная операция
vector operator++(int);  // постфиксная операция

```

и описать функцию-операцию для префиксной операции:

```

vector vector::operator++(void)
{
    x++;
    y++;
    z++;
    return *this;
}

```

и функцию-операцию для постфиксной операции:

```

vector vector::operator++(int)
{
    vector tmp;
    tmp = *this;

    x++; y++; z++;
    return tmp;
}

```

Данная реализация постфиксной операции содержит важное качество постфиксной операции ++, а именно она увеличивает значение координат вектора на единицу, но возвращает старое значение объекта.

Можно добавить постфиксную операцию ++ в пример 80.

## ДРУЖЕСТВЕННЫЕ ФУНКЦИИ-ОПЕРАЦИИ

Функции-операции могут быть членами класса или дружественными функциями класса. Функции-операции - члены класса рассмотрены в предыдущем параграфе. Отличие дружественных функций состоит в том, что для них не используется указатель this. При объявлении дружественной функции-оператора должны передаваться два аргумента для бинарных операций и один для унарных операций. Дружественными функциями не могут перегружаться операции =, ( ), [], ->.

Сначала проиллюстрируем перегрузку операции  $+$  для класса вектор с помощью дружественной операции. Объявление класса будет выглядеть так:

```
class vector{
    int x, y, z; // Три координаты
public:
    friend vector operator + (vector t, vector t1);
    vector operator= (vector t);
    void show(void);
    void assign (int mx, int my, int mz);
};
```

а описание функции так:

```
vector operator+(vector t1, vector t2)
{
    vector temp;
    temp.x=t1.x+t2.x;
    temp.y=t1.y+t2.y;
    temp.z=t1.z+t2.z;
    return temp;
}
```

Во многих случаях преимуществ использования дружественных функций перед функциями-членами при перегрузке нет. Однако есть ситуация, в которой использование дружественных функций обязательно. Перегрузку операции умножения объекта типа `vector` на целое можно записать и в виде функции-члена, и в виде дружественной функции. В то же время операцию умножения целое на `vector` можно определить только через дружественную функцию.

```
#include <iostream.h>
// Пример 82.
```

```
class CL{
public:
    int c;
    CL operator = (int i);
    friend CL operator+ (CL ob, int i);
    friend CL operator+ (int i, CL ob);
};
CL CL::operator=(int i)
{
    count = i;
    return *this;
}
CL operator+ (CL ob, int i);
{
    CL temp;
```

```

    temp.c = ob.c + i;
    return temp;
}
CL operator+(int i, CL ob)
{
    CL temp;
    temp.c = ob.c + i;
    return temp;
}
main(void)
{
    CL a;

    a = 10;
    cout << a.c << " ";
    a = a + 10;
    cout << a.c << " ";
    a = 15 + a;
    cout << a.c << "\n";
    return 0;
}

```

Прежде чем перейти к перегрузке унарных операций с помощью дружественных функций, необходимо познакомиться с еще одной особенностью языка C++, а именно ссылочными переменными (references).

## Ссылки

По умолчанию C и C++ передают аргументы функции, используя вызов по значению (call by value). Передача параметра по значению влечет за собой создание копии аргумента. Эта копия используется функцией и может быть изменена, но исходное значение аргумента при этом не меняется. Когда необходимо, чтобы функция изменяла значение аргумента, то параметры объявляются как указатели с использованием символа \*.

Функция `swap()` должна менять значения своих аргументов.

```

void swap(int i, int j)
{
    int tmp;
    tmp = i;
    i = j;
    j = tmp;
}

```

с вызовом

```
swap(i, j);
```

не вызывает ошибки компиляции, но и не решает проблемы, так как она не меняет местами значения параметров.

```
void swap1(int* i, int* j)
{ int tmp;
  tmp= *i;
  *i=*j;
  *j=tmp;
}
```

с вызовом

```
swap(&i, &j);
```

решает проблему, но вызов необходим с указанием адреса переменных.

При определении функции использовали объявление

```
void swap(int *a, int *b);
```

При вызове этой функции использовались указатели на аргументы. Это способ вызова по ссылке, используемый в языке C. Однако в языке C++ имеется возможность сообщить компилятору о необходимости генерировать вызов по ссылке другим способом. Это достигается использованием перед именем параметра в объявлении функции символа &. Такой параметр называется ссылкой (reference parameter). Рассмотрим пример определения функции, имеющей один ссылочный параметр.

Чтобы понять работу ссылочного параметра, перепишем функцию swap( ) предыдущего примера с использованием ссылочного параметра:

```
void swap2(int &a, int& b)
{
  int t;
  t=a;
  a=b;
  b=t;
}
```

с вызовом

```
swap2(a, b);
```

как и в первом случае.

Все три вызова присутствуют в программе:

```
#include <iostream.h>
// Пример 83. Использование ссылочных параметров

void swap (int a, int b);    // Неработающая версия
void swap1(int *a, int *b);  // Использование указателей
void swap2(int &a, int &b);   // Использование ссылок

main(void)
{
  int a=7, b=9;
  cout << "До вызова функции swap( )\n ";
  cout << " a= " << a << " b= " << b << "\n";
  swap(a, b);
}
```

```

cout << "После вызова функции swap( ) до вызова swap1( )\n";
cout << " a= " << a << " b= " << b << "\n";
swap1(&a, &b);
cout << "После вызова функции swap1( ) до вызова swap2( )\n";
cout << " a= " << a << " b= " << b << "\n";
swap2(a, b);
cout << "После вызова функции swap2( )\n";
cout << " a= " << a << " b= " << b << "\n";
return 0;
}
void swap(int i, int j)
{ int tmp;
  tmp = i;
  i = j;
  j = tmp;
}
void swap1(int *i, int *j)
{
  int tmp;
  tmp = *i;
  *i = *j;
  *j = tmp;
}
void swap2(int &i, int& j)
{
  int tmp;
  tmp = i;
  i = j;
  j = tmp;
}

```

Некоторые программисты символ & связывают с типом, а не с переменной. Например, используется такой способ задания прототипа функции:

```
void swap (int& a, int& b);
```

Более того, иногда при объявлении указателя символ \* связывают с типом, а не с переменной, например как показано ниже:

```
float* p;
```

Однако такой способ объявления не следует использовать для объявления списка переменных. Объявление

```
int* x, y;
```

означает, что переменная x объявлена как указатель, а переменная y - как переменная целого типа! Это замечание важно для чтения программ на языке C++. В дальнейшем будет использоваться традиционная форма объявления переменных.

Возможно также объявление ссылочных переменных, когда они не являются параметрами функций (непараметрические ссылки). Однако это не

рекомендуется делать, так как создание таких переменных ведет к путанице и деструктуризации программы. Непараметрическая ссылка иногда называется независимой (independent) или автономной ссылкой. Независимая ссылка должна быть инициализирована при ее объявлении. Это означает, что ей должен быть присвоен адрес переменной, объявленной до нее. Если это было сделано, то ссылочная переменная может быть использована в программе везде, где может быть использована и переменная, и ссылка. Фактически это означает, что между переменной и ссылкой нет различия, ссылка является как бы псевдонимом для переменной. Для примера, рассмотрим следующую программу:

```
#include <iostream.h>
// Пример 84. Использование непараметрической ссылки
main( )
{
    int j, k;
    int &i=j; // Объявление и инициализация ссылки
    j = 10;
    cout << j << " " << i; // Вывод: 10 10
    k = 121;
    i = k; // В j копируется значение k, а не адрес
    cout << "\n" << j << "\n"; // Вывод: 121
    return 0;
}
```

Когда выполняется оператор `i = k`, то значение `k` копируется в `j` (указанному `i`), а не ее адрес. Есть некоторые ограничения на использование непараметрических ссылок: нельзя ссылаться на ссылочную переменную и нельзя создать указатель на ссылку. Можно использовать независимую ссылку на константу:

```
int &i=100;
```

В этом случае `i` ссылается на место, где хранится константа.

#### ИСПОЛЬЗОВАНИЕ ССЫЛОЧНЫХ ПЕРЕМЕННЫХ ДЛЯ ПЕРЕГРУЗКИ УНАРНЫХ ОПЕРАЦИЙ

Операция `++` в программе не была перегружена дружественной функцией, так как перегрузка требовала использования ссылки. Теперь перегрузку можно осуществить следующим образом:

```
friend vector operator++(vector &opl)
{
    opl.x++;
    opl.y++;
    opl.z++;
    return opl;
}
```

Теперь в `main( )` можно использовать `s++`, где `s` - объект типа `vector`.

При использовании ссылочного параметра компилятор знает заранее, что он должен передать адрес при вызове функции.

## ПЕРЕГРУЗКА ОПЕРАЦИИ ИНДЕКСАЦИИ [ ]

Операция индексации рассматривается как бинарная операция, где первый операнд - объект класса, а второй операнд - целочисленный индекс. Перегрузка операции индексации позволяет работать с элементами объекта `vector` как с элементами массива. Для использования операции индексации класс `vector` придется немного изменить.

```
#include <iostream.h>
```

```
// Пример 85. Перегрузка операции индексации
```

```
class vector {
    int v[3];
public:
    vector (int a=0, int b=0, int c=0){ v[0]=a; v[1]=b; v[2]=c; } // Конструктор
    vector operator + (vector t);
    vector operator = (vector t);
    int& operator [] (int i);
    void show(void);
};

vector vector::operator+ (vector t)
{
    vector tmp;
    for(int i=0; i<3; i++)
        tmp.v[i] = t.v[i] + v[i];
    return tmp;
}

vector vector::operator= (vector t)
{
    vector tmp;
    for(int i=0; i<3; i++)
        v[i] = t.v[i];
    return *this;
}

int& vector::operator [] (int i)
{
    return v[i];
}

void vector::show(void)
{
    for (int i = 0; i<3; i++)
        cout << v[i] << " ";
    cout << "\n";
}

main (void)
{
```

```

vector v1(1, 2, 3), v2(10, 11, 12), v3;

v3 = v1 + v2;
for (int i=0; i<3; i++)
    cout << v3[i] << " ";
cout << "\n";
v3.show();

// Перегрузка операции [] с возвращением
// ссылки позволяет написать
v1[0] = 100;
// v1[i] является вызовом функции в левой части операции присваивания
// В левой части стоит функция и величина lvalue одновременно!
v1[1] = 201;
v1[2] = 302;
v1.show();
return 0;
}

```

## ИСПОЛЬЗОВАНИЕ ВИРТУАЛЬНЫХ ФУНКЦИЙ

Понятие полиморфизма является очень важным в объектно-ориентированном программировании. В приложении к языку C++ полиморфизм - это термин, используемый для описания процесса, при котором различные реализации функций могут быть доступны с использованием одного имени. По этой причине полиморфизм иногда характеризуется одной фразой - "один интерфейс, много методов". Это означает, что основной класс операций может быть оформлен в одном стиле, хотя конкретные действия могут быть различны. В C++ полиморфизм поддерживается и во время компиляции, и во время выполнения программы. Перегрузка функций и операций - это пример полиморфизма во время компиляции. Но в C++ поддерживается полиморфизм и во время выполнения программы. Это достигается использованием указателей на базовые классы и виртуальных функций.

### УКАЗАТЕЛИ НА ПРОИЗВОДНЫЕ ТИПЫ

Указатели на базовый тип и на производный тип зависимы. Пусть имеем базовый тип B\_class и производный от B\_class тип D\_class. В языке C++ всякий указатель, объявленный как указатель на B\_class, может быть также указателем на D\_class:

```

B_class *p; // указатель на объект типа B_class
B_class B_ob; // объект типа B_class
D_class D_ob; // объект типа D_class

```

После этого можно использовать следующие операции:

```

p=&B_ob; // Указатель на объект типа B_class
p=&D_ob; // Указатель на объект типа D_class

```



Все элементы класса `D_class`, наследуемые от класса `B_class`, могут быть доступны через использование указателя `p`. Однако на элементы, объявленные в `D_class`, нельзя ссылаться, используя `p`. Если требуется иметь доступ к элементам, объявленным в производном классе, используя указатель на базовый класс, надо привести его к указателю на производный тип. Например, можно сделать так:

```
((D_class *)p)->f();
```

Здесь функция `f()` — член класса `D_class`. Внешние круглые скобки необходимы. Хотя указатель на базовый класс может быть использован как указатель на производный класс, обратное неверно: нельзя использовать указатель на производный класс для присваивания ему адреса объекта базового класса. И наконец, указатель увеличивается и уменьшается при операциях `++` и `--` относительно его базового типа. Когда указатель на базовый класс указывает на производный класс, увеличение указателя не делает его указывающим на следующий элемент производного класса.

Рассмотрим модельный пример использования указателей на базовый класс. При этом каждый класс будет содержать функцию `void show(void)`, свою в каждом классе.

```
#include <iostream.h>
// Пример 86. Использование указателя на базовый класс

class Base {
public:
    void show(void)
    { cout << "In Base class\n"; }
};
class Derive: public Base {
public:
    void show(void)
    { cout << "In Derive class\n"; }
};
class Derive1: public Derive {
public:
    void show(void)
    { cout << "In Derive1 class\n"; }
};
class Derive2: public Derive1 {
public:
    void show(void)
    { cout << "In Derive2 class\n"; }
};

main(void)
{
    Base    bobj, *pb;
    Derive  dobj, *pd;
```

```

Derive1 d1obj, *pd1,
Derive2 d2obj, *pd2;

pb = &bobj;
bobj.show( ); // Вызов функции show( ) класса Base
pb -> show( ); // Вызов функции show( ) класса Base

pd = &dobj;
dobj.show( ); // Вызов функции show( ) класса Derive
pd -> show( ); // Вызов функции show( ) класса Derive

pd1 = &d1obj;
d1obj.show( ); // Вызов функции show( ) класса Derive1
pd1 -> show( ); // Вызов функции show( ) класса Derive1

pd2 = &d2obj;
d2obj.show( ); // Вызов функции show( ) класса Derive2
pd2 -> show( ); // Вызов функции show( ) класса Derive2

pb = &dobj; // Указателю на базовый класс присвоен
// адрес производного класса Derive
pb -> show( ); // Вызов функции show( ) класса Base!

pb = &d1obj; // Указателю на базовый класс присвоен
// адрес производного класса Derive1
pb -> show( ); // Вызов функции show( ) класса Base!

((Derive1 *)pb) -> show( ); // Вызов функции show( ) класса Derive1

pd1 = &d2obj; // Указателю на базовый класс Derive1 присвоен
// адрес производного класса Derive2
pd1 -> show( ); // Вызов функции show( ) класса Derive1

pd1 -> Base::show( ); // Вызов функции show( ) класса Base!

pd1 -> Derive::show( ); // Вызов функции show( ) класса Derive

pd1 -> show( ); // Вызов функции show( ) класса Derive1

((Derive2 *)pd1) -> show( ); // Вызов функции show( ) класса Derive2

return 0;
}

```

Если мы прокомментируем функцию `show( )` в классе `Derive1`, т. е. в классе `Derive1` не будет функции `show( )`, результат работы программы изменится. Попробуйте предугадать результат.

## ВИРТУАЛЬНЫЕ ФУНКЦИИ

Полиморфизм во время выполнения программы поддерживается использованием производных типов и виртуальных функций. Виртуальные функции – это функции, которые объявляются с использованием ключевого слова `virtual` в базовом классе и переопределяются (`override`) в одном или нескольких производных классах. При этом прототипы функций в разных классах одинаковы. Если типы функций различны, то механизм виртуальности для них не включается. Если функции, объявленные виртуальными, отличаются только типом возвращаемого значения, это является ошибкой.

Особенность использования виртуальных функций состоит в том, что при вызове функции, объявленной виртуальной, через указатель на базовый тип, во время выполнения программы определяется, какая виртуальная функция будет вызвана, в зависимости от того, на объект какого класса будет указывать указатель. Таким образом, когда указателю базового класса присвоены адреса объектов различных производных классов, выполняются различные версии виртуальных функций.

Виртуальная функция объявляется как виртуальная внутри базового класса, когда перед ее объявлением стоит ключевое слово `virtual`. При переопределении функции в производном классе слово `virtual` уже не добавляется. Хотя, если это сделать, ошибки не будет. Первый пример использования виртуальных функций дает программа, являющаяся модификацией примера 86 предыдущего параграфа:

```
#include <iostream.h>
// Пример 87. Использование виртуальных функций

class Base {
public:
    virtual void show(void)
    { cout << "In Base class\n"; }
};
class Derive: public Base {
public:
    void show(void)
    { cout << "In Derive class\n"; }
};
class Derive1: public Derive {
public:
    void show(void)
    { cout << "In Derive1 class\n"; }
};
class Derive2: public Derive1 {
public:
    void show(void)
    { cout << "In Derive2 class\n"; }
};
```

```
main(void)
{
    Base    bobj, *pb;
    Derive  dobj, *pd;
    Derive1 d1obj, *pd1;
    Derive2 d2obj, *pd2;

    pb = &bobj;
    pb -> show( ); // Вызов функции show( ) класса Base

    pd = &dobj;
    pd -> show( ); // Вызов функции show( ) класса Derive
    // виртуальность функции ни при чем
    pd1 = &d1obj;
    pd1 -> show( ); // Вызов функции show( ) класса Derive1

    pd2 = &d2obj;
    pd2 -> show( ); // Вызов функции show( ) класса Derive2

    pb = &dobj; // Указателю на базовый класс присвоен
    // адрес производного класса Derive
    pb -> show( ); // Вызов функции show( ) класса Derive
    // используется механизм виртуальных функций
    pb = &d1obj; // Указателю на базовый класс присвоен
    // адрес производного класса Derive1
    pb -> show( ); // Вызов функции show( ) класса Derive1

    pd1 = &d2obj; // Указателю на базовый класс Derive1 присвоен
    // адрес производного класса Derive2
    pd1 -> show( ); // Вызов функции show( ) класса Derive2
    // Работает механизм виртуальных функций
    pd1 -> Base::show( ); // Явный вызов функции show( ) класса Base

    pd1 -> Derive::show( ); // Явный вызов функции show( ) класса Derive

    ((Derive2 *)pd1) -> show( ); // Явный вызов функции show( )
    //класса Derive2

    return 0;
}
```

Так, как и в предыдущем примере, можно закомментировать функцию `show( )` в классе `Derive1`.

В одном или нескольких производных класса переопределение виртуальной функции может отсутствовать. При этом механизм виртуальной функции сохраняется и вызывается функция базового класса, ближайшего к тому, где функция не переопределена.

Виртуальная функция должна быть членом класса. Она не может быть дружественной для класса, в котором она определена. Однако виртуальная функция может быть "другом" другого класса.

Вследствие запретов и различий между перегрузкой обычных функций и перегрузкой виртуальных функций для описания переопределения виртуальных функций используется термин "замещение" (overriding).

Функция, объявленная виртуальной, остается таковой, как бы много производных классов не было подстроено. Если в предыдущем примере класс `Derive1` будет производным классом для `Derive`, а не для `Base`, функция `who()` в классе `Derive1` остается виртуальной.

Если в производном классе функция не замещает виртуальную, так как она не объявлена или имеет другой прототип, то вызывается функция базового класса.

Следует помнить, что наследственные характеристики являются иерархическими, т. е. если, например, класс `Derive1` является производным классом от `Derive`, но не от `Base`, а класс `Derive` является производным классом от `Base` и функция `who()` виртуальная и определена в `Base` и `Derive`, но не определена в `Derive1`, то при попытке вызвать функцию `who()` по указателю базового класса на `Derive1` вызывается функция, объявленная в `Derive`, а не в `Base`.

Виртуальные функции удобны для использования по следующим причинам. Базовый класс задает основной интерфейс, который будут иметь производные классы. Но производные классы задают свой метод. Вот почему фраза "один интерфейс, много методов" часто используется для описания полиморфизма.

Объектно-ориентированное программирование позволяет программисту создавать сложные программы. Если производные классы строятся корректно и известно, что все объекты, начиная от базового класса, доступны посредством одного и того же основного способа, даже когда конкретные действия варьируются от одного производного класса к другому, это означает, что надо помнить только интерфейс. Более того, отделение интерфейса и наполнения функций позволяют создавать классы библиотек (class libraries).

Вызов виртуальной функции обычно реализуется как не прямой вызов по таблице виртуальных функций класса. Эта таблица создается компилятором во время компиляции, а связывание происходит во время выполнения. Иногда это называется поздним связыванием (late binding).

Еще одним, более приземленным, примером использования виртуальных функций является следующий. Введем класс `figure`, который описывает плоскую фигуру, для вычисления площади которой достаточно двух измерений. В этом классе описана виртуальная функция `show_area()`, печатающая значение площади фигуры. На основе этого класса строятся другие классы - `triangle`, `rectangle`, `circle`, для которых определена конкретная формула вычисления площади фигуры.

```
#include <iostream.h>
// Пример 88. Использование виртуальных функций

class figure{
protected:
    double x, y;
public:
    void set_dim(double i, double j=0) {
        x=i;
        y=j;
    }
    virtual void show_area( ) {
        cout << "Площадь не определена ";
        cout << "для этого класса \n";
    }
};

class triangle: public figure {
public:
    void show_area( ) {
        cout << "Треугольник с высотой " << x << " и основанием " << y;
        cout << " имеет площадь " << x * 0.5 * y << "\n";
    }
};

class rectangle: public figure{
public:
    void show_area( ) {
        cout << " Прямоугольник со сторонами " << x << " и " << y;
        cout << " имеет площадь " << x * y << "\n";
    }
};

class circle: public figure {
public:
    void show_area( ) {
        cout << " Круг с радиусом " << x;
        cout << " имеет площадь " << 3.14159 * x * x << "\n";
    }
};

main(void)
{
    figure f, *p; // объявление указателя на базовый тип
    triangle t; // создание объекта производного типа
    rectangle s;
    circle c;

    p = &f;
    p -> set_dim(1, 2);
    p -> show_area( );
}
```

```

p = &t;
p -> set_dim(3, 4);
p -> show_area( );

p = &s;
p -> set_dim(5, 6);
p -> show_area( );

p = &c;
p -> set_dim(8);
p -> show_area( );
return 0;
}

```

Обратите внимание на одинаковость вызова

```

p -> set_dim(1, 2);
p -> show_area( );

```

в трех случаях. Еще обратите внимание на то, что для класса `circle` требуется задать всего один размер - радиус. Поэтому функция `set_dim( )` имеет второй параметр, задаваемый по умолчанию. Хотя и в этом случае можно было вызвать функцию `p -> set_dim(1, 2)`, однако второй параметр просто не использовался бы.

#### ЧИСТЫЕ ВИРТУАЛЬНЫЕ ФУНКЦИИ И АБСТРАКТНЫЕ ТИПЫ

Когда виртуальные функции вызываются из производного класса, но не замещаются, то вызывается соответствующая функция базового типа. Однако во многих функциях нет смыслового определения виртуальной функции в базовом классе.

Например, в базовом классе `figure` предыдущего примера для функции `show_area( )` используется просто "заглушка". При создании библиотек классов для виртуальных функций еще неизвестно, будет ли смысловое значение в контексте базового класса. Существует два способа справиться с этой проблемой.

Первый способ - это выдать предупреждающее сообщение. Это может быть полезно в отдельных, но далеко не во всех случаях. Например, может быть виртуальная функция, которая должна быть определена в производном классе и иметь некоторое значение.

Рассмотрим класс `figure`. Площадь в общем случае не определена.

Другим решением этой проблемы в языке C++ являются чистые (pure) виртуальные функции.

Чистые виртуальные функции - это функции, объявленные в базовом классе как виртуальные, но не имеющие описания в базовом классе. Так, производный тип должен определить свою собственную версию - нельзя просто использовать версию, определенную в базовом классе.

Для объявления чисто виртуальной функции, используется следующая основная форма:

```
virtual тип имя-функции(список параметров) = 0;
```

тип - возвращаемый тип функции; имя-функции - это имя функции; = 0 - признак чистой виртуальной функции. В новой версии предыдущей программы в классе `figure` функция `show_area()` можно объявить как чистую виртуальную функцию:

```
#include <iostream.h>
```

```
// Пример 89. Использование чистой виртуальной функции
```

```
class figure{
protected:
    double x, y;
public:
    void set_dim(double i, double j=0) {
        x=i;
        y=j;
    }
    virtual void show_area() = 0;
};
class triangle: public figure {
public:
    void show_area() {
        cout << "Треугольник с высотой " << x << " и основанием " << y;
        cout << " имеет площадь " << x * 0.5 * y << "\n";
    }
};
class rectangle: public figure{
public:
    void show_area() {
        cout << " Прямоугольник со сторонами " << x << " и " << y;
        cout << " имеет площадь " << x * y << "\n";
    }
};
class circle: public figure {
public:
    void show_area() {
        cout << " Круг с радиусом " << x;
        cout << " имеет площадь " << 3.14159 * x * x << "\n";
    }
};

main(void)
{
    figure *p; // объявление указателя на базовый тип
               // создать объект нельзя!
    triangle t; // создание объекта производного типа
```



```

rectangle s;
circle c;
p = &t;
p -> set_dim(3, 4);
p -> show_area( );
p = &s;
p -> set_dim(5, 6);
p -> show_area( );
p = &c;
p -> set_dim(8);
p -> show_area( );
return 0;
}

```

Определив функцию `show_area()` как чисто виртуальную, вы требуете от всех производных классов определить свое собственное наполнение. Или объявить эту функцию как чистую виртуальную функцию. Если в производном классе это не сделано, то компилятор C++ сообщит об ошибке.

Если класс имеет по крайней мере одну чисто виртуальную функцию, то говорят, что этот класс абстрактный (`abstract class`). Абстрактные классы имеют одну важную черту: может не быть объекта этого класса. Абстрактный класс должен использоваться только как базовый класс, от которого наследуются другие производные классы.

Причина, по которой абстрактные классы не могут использоваться для объявления объекта, состоит в том, что одна или более функция не имеет определения. Однако, даже если базовый класс абстрактный, можно создать указатель на объект базового класса и применить его для использования механизма виртуальных функций.

## ПРОИЗВОДНЫЕ КЛАССЫ И ИХ КОНСТРУКТОРЫ И ДЕКТРУКТОРЫ

При объявлении производных классов возникает еще одна проблема. Каждый класс, как базовый, так и производный, может иметь конструктор и деструктор. В каком порядке выполняются эти функции? Этот же вопрос возникает в случае множественного наследования. Начнем с простейшего случая.

Рассмотрим пример класса с конструктором.

```

#include <iostream.h>
// Пример 90. Наследование классов с конструкторами
//                и деструкторами
class Base{
public:
    Base( ) { cout << "Конструктор класса Base \n";}
    ~Base( ) { cout << "Деструктор класса Base \n";}
};
class Derive1:public Base{

```

```
public:
    Derive1( )
    {
        cout<< "Конструктор производного класса Derive1\n";
    }
    ~Derive1( )
    {
        cout<< "Деструктор производного класса Derive1\n";
    }
};
class Derive2: public Derive1{
public:
    Derive2( )
    {
        cout << "Конструктор производного класса Derive2\n";
    }
    ~Derive2( )
    {
        cout << "Деструктор производного класса Derive2\n";
    }
};
main( )
{
    Derive1 d1;
    cout << "\n";
    Derive2 d2;
    cout << "\n";
    return 0;
}
```

Программа не делает ничего, кроме создания объектов типа Derive1 и Derive2. Зато функции-конструкторы выполняются при каждом создании объекта. Причем вывод будет следующим:

```
Конструктор класса Base
Конструктор производного класса Derive1

Конструктор Класса Base
Конструктор производного класса Derive1
Конструктор производного класса Derive2

Конструктор производного класса Derive2
Конструктор производного класса Derive1
Конструктор Класса Base
```

Таким образом, при создании объекта производного типа выполняются последовательно, начиная с класса Base, конструкторы всех классов-прародителей.

Это естественный порядок выполнения. Для создания производного класса необходима инициализация базового класса, так что конструктор базового класса должен выполняться.

С другой стороны, деструктор в производном классе должен выполняться прежде выполнения деструктора базового класса. Если деструктор базового класса выполнится раньше, то деструктор производного класса вообще не сможет выполниться. Поэтому деструкторы вызываются в порядке, обратном вызову конструкторов.

### ПОРЯДОК ВЫЗОВА КОНСТРУКТОРОВ И ДЕКТРУКТОРОВ ПРИ МНОЖЕСТВЕННОМ НАСЛЕДОВАНИИ

Как мы видели, в языке C++ разрешено при создании производного класса пользоваться несколькими базовыми классами. При объявлении производного класса базовые классы перечисляются через запятую. Причем при создании объекта конструкторы выполняются в порядке следования базовых классов слева направо.

Можно опять модифицировать предыдущий пример следующим образом:

```
#include <iostream.h>
// Пример 91
class Base1 {
public:
    Base1( ){cout<<"Конструктор Base1\n";}
    ~Base1( ){cout<<"Деструктор Base1\n"; }
};
class Base2 {
public:
    Base2( ){cout<<"Конструктор Base2\n";}
    ~Base2( ){cout<<"Деструктор Base2\n"; }
};
class Derive: public Base1, public Base2 {
public:
    Derive( ){cout<<"Конструктор Derive\n";}
    ~Derive( ){cout<<"Деструктор Derive\n"; }
};
main (void)
{
    Base1 b1;
    cout <<"\n";
    Base2 b2;
    cout <<"\n";
    Derive d;
    cout <<"\n";
    return 0;
};
```

Результат работы программы показывает, в каком порядке выполняются конструкторы и деструкторы. Вы увидите, что деструкторы выполняются в порядке, обратном по отношению к конструкторам.

## ВИРТУАЛЬНЫЕ БАЗОВЫЕ КЛАССЫ

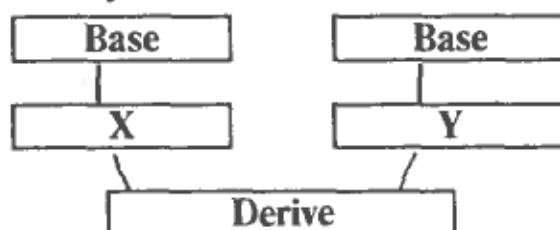
При множественном наследовании базовый класс не может быть задан в производном классе более одного раза:

```
class Der: Base, Base { ... }; // Ошибка!
```

В то же время базовый класс может быть передан производному классу более одного раза косвенно:

```
class X: public Base{...};
class Y: public Base{...};
class Derive: public X, public Y{...};
```

Этот пример соответствует схеме наследования

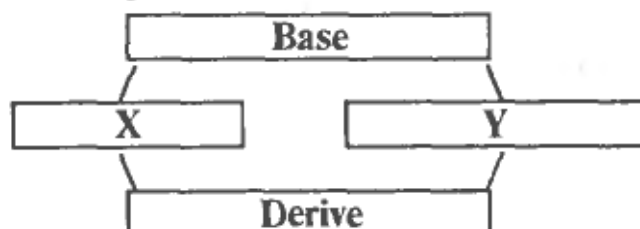


Каждый объект класса Derive будет иметь два подобъекта класса Base. Чтобы избежать неоднозначности при обращении к членам базового объекта Base, можно объявить этот базовый класс виртуальным. Для этого используется то же зарезервированное слово `virtual`, что и при объявлении виртуальных функций.

```
class X: virtual public Base{...};
class Y: virtual public Base{...};
class Derive: public X, public Y{...};
```

Теперь класс Derive имеет только один подобъект класса Base.

Этот пример соответствует схеме наследования



## ОПЕРАЦИИ ДИНАМИЧЕСКОГО ВЫДЕЛЕНИЯ ПАМЯТИ NEW И DELETE

В языке C для динамического выделения памяти под переменную и освобождения памяти из-под ненужной переменной служат соответственно функции `malloc()` и `free()`, а для определения размера необходимой памяти - операция `sizeof`.

В языке C++ предусмотрены две операции - new и delete. Основная форма их использования следующая:

```
pointer_var = new var_type;
delete pointer_var;
```

здесь pointer\_var указатель - типа var\_type.

Операция new выделяет соответствующее место для переменной в соответствующей области памяти и возвращает адрес выделенного места. Если по каким-либо причинам память не может быть выделена, операция new возвращает нулевой указатель (NULL).

Операция delete освобождает соответствующую память, на которую указывает pointer\_var. Всегда необходимо проверять, была ли попытка успешной.

Удобство использования операции new состоит, в частности, в том, что операция сама определяет размер переменной var\_type автоматически и возвращает указатель, уже преобразованный к этому типу. Если нам необходимо выделить память под массив переменных, это можно сделать следующим образом:

```
pointer_var = new var_type[SIZE];
delete [SIZE] pointer_var;
```

При освобождении памяти из-под массива возможно использование

```
delete pointer_var;
```

без указания размера массива, ибо в процессе работы программы сохраняется информация о размере выделенной памяти.

Пример использования этих операций:

```
#include <iostream.h>
// Пример 92. Простое использование new и delete.
```

```
main (void)
{
    int *p;
    p=new int; // Выделяем память по целое
    if(!p) {
        cout << "Памяти недостаточно \n";
        return 1;
    }
    *p=20;
    cout << p << "\n";
    delete p; // Освобождаем память

    return 0;
}
```

Иллюстрация работы с динамически выделенной памятью под массив:

```
#include <iostream.h>
// Пример 93. Динамическое выделение памяти под массив.
```

```
main (void)
{
    int *p;
    unsigned int size;

    cout << "Введите размер массива: ";
    cin >> size;

    p = new int[size]; // Выделяем память под массив целых
    if(!p) {
        cout << "Недостаточно памяти \n";
        return 1;
    }
    for(int i=0; i < size; i++)
        p[i] = i * i;
    // Использование *p++ = i*i; неудачно, так как при освобождении
    // памяти мы укажем измененное значение указателя.
    int *q = p;
    for(i = 0; i < size; i++)
        cout << *q++ << " "; // Это уже корректно
    cout << "\n ";
    delete p; // Освобождаем память
    return 0;
}
```

Можно выделить динамически память под объект любого определенного к этому моменту типа. Динамически выделять память под простые переменные невыгодно, так как память расходуется не только под переменные, но и под информацию о выделении памяти. Динамическое выделение памяти целесообразно для выделения памяти под большие объекты, массивы, особенно массивы неизвестного заранее размера. Часто динамическое выделение памяти используется в конструкторах класса, членом которого является массив. При этом для выделения памяти под объект также может использоваться динамическое выделение памяти. Вспомним пример класса `queue`, содержащего массив `q[]`. В приведенном выше примере массив имел фиксированный размер. Теперь рассмотрим класс `queue` с переменным размером массива.

```
#include <iostream.h>
// Пример 94. Использование new в конструкторе класса

class queue {
    int *q;
    int sloc, rloc;
    unsigned size;
public:
    queue(int sz); // конструктор
    ~queue(void); // деструктор
```

```

        void qput(int i);
        int  qget(void);
    },
    // описание функции-конструктора
    queue::queue(int sz)
    {
size = sz;
if (!(q = new int[ size])) {
    cout << "Недостаточно памяти \n";
    return;
}
sloc = rloc = 0;
cout << " Очередь размера " << size << "инициализирована\n";
}
    // описание функции-деструктора
    queue::~~queue(void)
    {
delete q;
    cout << " Очередь разрушена \n";
}
    void queue::qput(int i)
    {
        if (sloc == size) {
            cout << " Очередь полна\n ";
            return;
        }

        q[sloc++ ] = i;
    }
    int queue::qget(void)
    {
        if (rloc == sloc) {
            cout << " Очередь пуста \n";
            return 0;
        }
        return q[rloc++ ];
    }
}

main( )
{
    queue a(5), b(100); // объявление двух объектов класса queue
    // с массивами разных размеров

    a.qput(10);
    b.qput(19);

    a.qput(20);
    b.qput(1);
}

```

```

cout << a.qget( ) << " ";
cout << a.qget( ) << " ";
cout << b.qget( ) << " ";
cout << b.qget( ) << "\n";

queue *pq;
int s = 20;
pq = new queue(s); // Динамическое создание объекта
// Обратите внимание на передачу параметра конструктору
// Именно сейчас вызывается конструктор queue
if (!pq) {
    cout << "Недостаточно памяти \n";
    return 0;
}
else
    cout << " Объект класса queue создан \n ";
    for (int i = 0; i < s; i++)
        pq->qput(2*i+1); // заполнение очереди
    for (i = 0; i < s; i++) // Просмотр элементов
        cout << pq->qget( ) << " ";
        cout << "\n";
    delete pq;
return 0;
}

```

Ситуация, когда объект создается динамически и конструктор динамически выделяет память, является типичной в использовании C++.

## ВИРТУАЛЬНЫЕ ДЕКТРУКТОРЫ

Не менее типичной является ситуация, когда динамически создается объект производного класса, а указатель используется базового класса. Например:

```

#include <iostream.h>
// Пример 95. Динамическое выделение памяти
// через указатель на базовый класс
class Base{
public:
    Base( ) { cout << "Конструктор класса Base \n";}
    ~Base( ) { cout << "Дектруктор класса Base \n";}
};
class Derive1:public Base{
public:
    Derive1( )
    {
        cout<< "Конструктор производного класса Derive1\n";
    }
}

```



```

    }
    ~Derive1( )
    {
        cout<< "Деструктор производного класса Derive1\n";
    }

};
class Derive2 public Derive1{
public:
    Derive2( )
    {
        cout << "Конструктор производного класса Derive2\n";
    }
    ~Derive2( )
    {
        cout << "Деструктор производного класса Derive2\n";
    }
};
main( )
{
    Base *pb = new Derive2;
    if (!pb) {
        cout << "Недостаточно памяти \n";
        return 1;
    }

    cout << "\n";
    delete pb;
    return 0;
}

```

Результатом работы программы будет следующее:

```

Конструктор класса Base
Конструктор производного класса Derive1
Конструктор производного класса Derive2
Деструктор класса Base

```

Обратите внимание на то, что было вызвано три конструктора и всего один деструктор. При удалении объекта через указатель на базовый класс вызывается лишь деструктор базового класса. Если бы в конструкторах выделялась динамическая память при создании объекта, эта память при разрушении объекта не освобождалась бы корректно. Эта проблема решается введением понятия “виртуальный деструктор”. Если при объявлении деструктора базового класса он объявляется как виртуальный, то все конструкторы производных классов также являются виртуальными. При разрушении объекта с помощью операции delete через указатель на базовый класс будут корректно вызваны деструкторы всех классов.

```
#include <iostream.h>
// Пример 96. Динамическое выделение памяти
// через указатель на базовый класс
class Base{
public:
    Base( ) { cout << "Конструктор класса Base \n";}
    virtual ~Base( ) { cout << "Деструктор класса Base \n";}

};
class Derive1:public Base{
public:
    Derive1( )
    {
        cout<< "Конструктор производного класса Derive1\n";
    }
    ~Derive1( )
    {
        cout<< "Деструктор производного класса Derive1\n";
    }

};
class Derive2: public Derive1{
public:
    Derive2( )
    {
        cout << "Конструктор производного класса Derive2\n";
    }
    ~Derive2( )
    {
        cout << "Деструктор производного класса Derive2\n";
    }

};

main( )
{
    Base *pb = new Derive2;
    if (!pb) {
        cout << "Недостаточно памяти \n";
        return 1;
    }
    cout << "\n";
    delete pb;
    return 0;
}
```

Теперь результатом работы программы будет следующее:

```
Конструктор класса Base
Конструктор производного класса Derive1
Конструктор производного класса Derive2
```

Деструктор производного класса Derive2  
Деструктор производного класса Derive1  
Деструктор класса Base

Вызываются все деструкторы классов, и в нужном порядке.

## ШАБЛОНЫ КЛАССОВ И ФУНКЦИЙ

В языке C++ предусмотрена еще одна реализация полиморфизма - шаблоны функций (Function Templates) и шаблоны классов (Class Templates). В первом компиляторе языка C++ фирмы Borland - Turbo C++ v. 1.0 механизм шаблонов еще не был реализован. Во всех современных компиляторах шаблоны классов и функций уже реализованы. Начнем с шаблонов функций.

### Шаблоны функций

При перегрузке функций мы рассматривали семейство функций `sqr_it( )` с различными типами аргументов. Напомним, что эти функции возвращали квадрат аргумента, причем тип возвращаемого значения совпадал с типом аргумента. Для каждого типа мы описывали свое тело функции, причем отличались функции только типом аргумента и типом возвращаемого значения. Шаблоны функций как раз позволяют использовать в качестве аргумента тип переменной.

```
template < class T >
T sqr_it (T x)
{
    return x*x;
}
```

Любой тип данных, а не только определенный как класс может использоваться при применении этих функций. Конечно, над типом, для которого будет вызываться функция, определенная шаблоном, должны быть определены операции над переменными типа T, которые используются в теле функции. Шаблонные функции могут использоваться совместно с функциями, определенными обычным образом, с тем же именем. В какой момент компилятор генерирует код функции, определенной шаблоном, соответствующий данному типу? При компиляции программы, встретив вызов какой-либо функции, компилятор ищет:

- точного определения обычной функции, ее вызов вставляется в тело программы;
- шаблон функции, из которого она может быть сгенерирована; если шаблон найден, она создается и вызывается;
- применения обычной техники совместного вызова “обычной” функции с другим типом параметра; если функция определяется однозначно, она вызывается.

Пример использования шаблонов:

```
#include <iostream.h>
// Пример 97. Использование шаблонов функций
template < class T >
T sqr_it (T x)
{
    return x * x;
}; // Наличие точки с запятой необязательно

main (void)
{
    int i = 10;
    float f = 1.1;
    long l = 2345;
    long double ld = 123.1e+123;

    cout << " int " << sqr_it (i) << " \n";
    cout << " float " << sqr_it (f) << " \n";
    cout << " long " << sqr_it (l) << " \n";
    cout << " long double " << sqr_it (ld) << " \n";

    return 0;
}
```

В шаблоне функции может использоваться необязательно один тип как параметр шаблона. Например, семейство функций `max (a, b)`; с различными типами аргументов может быть определено следующим образом:

```
template < class T1, class T2 >
T1 max (T1 x, T2 y)
{
    return (x > y)? x: y;
}
```

Конечно, возникает вопрос, почему возвращаемый тип `T1`, а не тип `T2`? В C++ нет механизма изменения типа возвращаемого значения, поэтому надо было сделать какой-либо выбор. Для того чтобы эти функции возвращали результат без потери значения, при вызове функции желательно переменную или константу “старшего” типа использовать в качестве первого аргумента.

```
#include <iostream.h>
// Пример 98. Использование шаблонов с двумя типами-параметрами

template < class T1, class T2 >
T1 maximum (T1 x, T2 y)
{
    if (x >= y) return x;
    return y;
}
```

```

main (void)
{
    int i = 5;
    double d = 3.2345;
    long l = 123456;
    long double ld = 1.2e123;

    cout << " максимум из " << d << " и " << i << " равен "
          << maximum (d, i) << "\n";
    cout << " максимум из " << i << " и " << d << " равен "
          << maximum (i, d) << "\n";
    cout << " максимум из " << ld << " и " << l << " равен "
          << maximum (ld, l) << "\n";

    // В двух следующих вызовах будет выдаваться неверный результат,
    // это не связано с шаблоном, это связано с преобразованием типов
    cout << " максимум из " << i << " и " << l << " равен "
          << maximum (i, l) << "\n";
    cout << " максимум из " << l << " и " << 5.5 << " равен "
          << maximum (l, 5.5) << "\n";
    // А в этом вызове результат правильный
    cout << " максимум из " << 5.5 << " и " << l << " равен "
          << maximum (5.5, l) << "\n";

    return 0;
}

```

### ШАБЛОНЫ КЛАССОВ

Шаблоны классов позволяют построить отдельные классы аналогично шаблону функций. Шаблон класса задает параметризованный тип. Имя шаблона класса должно быть уникальным и не может относиться к какому-либо шаблону, классу, функции, объекту и т. д.

В качестве параметра при задании шаблона класса может использоваться не только тип, но и другие параметры, например целочисленные значения. Правда, эти параметры должны быть константными выражениями.

Еще раз преобразуем класс queue, задав этот класс шаблоном. Параметрами шаблона будет тип массива, входящего в класс queue, и размер size этого массива.

```

#include <iostream.h>
// Пример 99. Использование шаблона класса
template <class T, int size>
class queue {
    T *q;
    int sloc, rloc;
public:
    queue(void); // конструктор

```

```

    ~queue(void); // деструктор
    void qput(T i);
    T qget(void);
};
// описание функции-конструктора
template <class T, int size> // Обратите внимание на синтаксис
queue<T, size>::queue(void) // объявления членов функций шаблона класса
{
    if (!(q = new T[ size])) {
        cout << "Недостаточно памяти \n";
        return;
    }
    sloc = rloc = 0;
    cout << " Очередь размера "<< size << " инициализирована\n";
}
// описание функции-деструктора
template <class T, int size>
queue<T, size>::~~queue(void)
{
    delete q;
    cout << " Очередь разрушена \n";
}
template <class T, int size>
void queue<T, size>::qput(T i)
{
    if (sloc == size) {
        cout << " Очередь полна\n ";
        return;
    }

    q[sloc++] = i;
}
template <class T, int size>
T queue<T, size>::qget(void)
{
    if (rloc == sloc) {
        cout << " Очередь пуста \n";
        return 0;
    }
    return q[rloc++];
}
//
int main( )
{
    queue<int, 5> a;
    queue<double, 200> b;
    // объявление двух объектов класса queue
    // с массивами разных типов и разных размеров

```

```

a.qput(10);
b.qput(1.129);

a.qput(23);
b.qput(5.555);
cout << a.qget( ) << " ";
cout << a.qget( ) << " ";
cout << b.qget( ) << " ";
cout << b.qget( ) << "\n";
const int s = 10;
queue<long double, s> *pq;
pq = new queue<long double, s>; // Динамическое создание объекта
    // Именно сейчас вызывается конструктор queue
if (! pq) {
    cout << "Недостаточно памяти \n";
    return 0;
}
else
    cout << " Объект класса queue создан \n ";
for (int i = 0; i < s; i++)
    pq->qput(i/2.0+i); // заполнение очереди
for (i = 0; i < s; i++) // Просмотр элементов
    cout << pq->qget( ) << " ";
    cout << "\n";
delete pq;
return 0;
}

```

## СТАТИЧЕСКИЕ ЧЛЕНЫ КЛАССА

Некоторые члены класса могут быть объявлены с модификатором класса памяти `static`. Такие члены класса называются статическими членами класса. Статические члены - данные класса являются общими для всех объектов данного класса. Изменив значение статического члена класса в одном объекте, мы получим изменившееся значение во всех других объектах класса. Объявление статических членов - данных класса внутри объявления класса не является описанием, т. е. под эти данные не выделяется память. Описание данных должно быть где-то еще. Тем самым все объекты класса ссылаются на одно и то же место в памяти. Статические члены - данные класса можно использовать для подсчета количества созданных объектов класса или существующих в данный момент объектов класса.

Функции - члены класса также могут быть объявлены статическими. Статические функции - члены класса не получают указатель `this`, соответственно эти функции не могут обращаться к нестатическим членам класса. К статическим членам класса статические функции - члены класса могут обращаться посредством операций точка или `->`. Статическая функция - член класса не может быть виртуальной. К статическим данным - членам класса и статическим функциям - членам класса можно обращаться, даже если не соз-

дано ни одного объекта данного класса, надо только использовать полное имя члена класса. Если функция `a()` является статической функцией - членом класса `cl`, можно вызвать эту функцию:

```
cl::a( );
```

Приведем модельный пример использования статических членов класса для подсчета числа существующих и созданных объектов класса.

```
#include <iostream.h>
```

```
// Пример 100. Использование статических членов класса.
```

```
class st {
    static int count1;
    static int count2;
public:
    static void show_count(void);
    st(void); // Конструктор
    ~st(void); // Деструктор
};

st::st(void)
{
    count1 ++;
    count2 ++;
}

st::~~st(void)
{
    count2--;
}

void st::show_count(void)
{
    cout<< "Создано объектов: " << count1 << "\n";
    cout<< "Существует объектов: " << count2 << "\n\n";
}

int st::count1;
int st::count2; // Описание переменных

main(void)
{
    st::show_count( ); // Вызов функции до создания объектов
    st a, b, c, *p;
    a.show_count( );
    {
        st x, y, z;
        st::show_count( ); // эти два вызова дадут
        z.show_count( ); // одинаковый результат
    }
    p=new st;
    st::show_count( );
    delete p;
    st::show_count( );
    return 0;
}
```



## ЛОКАЛЬНЫЕ КЛАССЫ

Класс может быть объявлен внутри функции. Такой класс называется локальным классом (local class). Функция, в которой объявлен локальный класс, не имеет специального доступа к членам локального класса. Локальный класс не может иметь статических членов - данных.

Объект локального класса может быть создан только внутри функции, в области действия объявления класса. Все функции-члены локального класса должны быть объявлены внутри объявления класса, т. е. должны быть подставляемыми функциями.

```
#include <iostream.h>
// Пример 101. Локальные классы.
void f(void);
main(void)
{
    f();
    return 0;
}
void f (void)
{
    class local_class{
        int who;
    public:
        local_class(int a)
        {
            who = a;
            cout << "Конструктор локального класса " << who << "\n";
        }
        ~local_class(void)
        {
            cout << "Деструктор локального класса " << who << "\n";
        }
    } obj1(1), obj2(2);
}
```

## ВЛОЖЕННЫЕ КЛАССЫ

При объявлении одного класса внутри его объявления может быть объявлен другой класс или классы. Такой класс называется вложенным (nested). Вложенный класс находится в области действия объемлющего класса, соответственно объекты этого класса могут использоваться как члены этого класса или в функциях - членах класса. Функции - члены класса и статические члены вложенного класса могут быть описаны в глобальной области действия.

Модельный пример использования вложенных классов:

```
#include <iostream h>
```

```
// Пример 102. Вложенные классы
```

```
class Cl{
    class nested_class{
        int who;
    public:
        nested_class(int a);
        ~nested_class(void);
    };
public:
    Cl(int b)
    {
        nested_class obj(b*b);
        cout<< "Конструктор класса Cl\n";
    }
    ~Cl(void)
    {
        cout<< "Деструктор класса Cl\n";
    }
};

Cl::nested_class::nested_class(int a)
{
    who = a;
    cout << "Конструктор вложенного класса "<< who <<"\n";
}

Cl::nested_class::~~nested_class(void)
{
    cout << "Деструктор вложенного класса "<< who <<"\n";
}

main(void)
{
    Cl cl_obj(3);
    return 0;
}
```

## ИСПОЛЬЗОВАНИЕ БИБЛИОТЕКИ ВВОДА/ВЫВОДА ЯЗЫКА C++

Система Turbo C++ содержит и позволяет использовать библиотеки ввода/вывода языка C. Вместе с этим она содержит собственную библиотеку ввода/вывода. Причина появления собственной библиотеки ввода/вывода в том, что библиотеки языка C не позволяют выводить определенные пользователем объекты так же, как и простые переменные.

Библиотека ввода/вывода языка C++ позволяет нам перегрузить операции >> и << так, чтобы можно было распознать тип объекта и вывести объект этого класса.

## Потоки языка C++

Система ввода/вывода C++, так же как и в языке C, оперирует потоками (streams). В начале выполнения программы автоматически открывается 4 predetermined потока: cin, cout, cerr и clog. Поток cin связан со стандартным вводом, поток cout - со стандартным выводом. Потоки cerr и clog также связаны со стандартным выводом. Поток cerr небуферизованный, т. е. вызывает немедленный вывод. Поток clog буферизован, и вывод происходит только после того, как наполнится буфер. Оба этих потока используются для вывода сообщения об ошибках.

По умолчанию поток cin связан с клавиатурой, cout - с дисплеем, но они могут быть перенаправлены на другие устройства или на файловую систему.

В заголовочном файле IOSTREAM.H определены классы, относящиеся к потокам. Эти классы образуют иерархическую систему.

Класс нижнего уровня называется streambuf. Он обеспечивает основные операции по неформатированному выводу. Следующий класс называется ios. Он обеспечивает поддержку форматированного ввода/вывода и используется для построения трех следующих классов: istream, ostream и iostream. Istream позволяет создавать поток ввода, ostream - поток вывода. Класс iostream может создавать поток, предназначенный как для ввода, так и для вывода.

## ПЕРЕГРУЗКА ОПЕРАЦИЙ ВВОДА/ВЫВОДА. ИНСЕРТОРЫ И ЭКСТРАКТОРЫ

Для организации ввода/вывода данных, связанных с классом, строится специальная функция - член класса, целью которой является ввод и вывод данных этого класса. Достигается это путем перегрузки операций << и >>.

Операция << в англоязычной литературе обычно называется inserting (вставка данных в поток), а операция >> - extracting (извлечение данных из потока).

Функции - члены класса, обеспечивающие перегрузку операции <<, мы будем называть инсертором (insertor), а функцию, соответствующую перегрузке операции >>, будем называть экстрактором (extractor).

В качестве примера создания функции инсертора рассмотрим класс vector:

```
class vector{
public:
    int x, y, z;
    vector (int a, int b, int c){x=a, y=b, z=c}
};
```

Чтобы создать функцию-инсертор для этого класса, мы должны определить операцию ввода относительно этого класса, т. е. перегрузить операцию >>.

```
// vector инсертор
ostream &operator << (ostream &stream, "vector obj)
```

```

{
    stream << obj.x << " ";
    stream << obj.y << " ";
    stream << obj.z << "\n";
    return stream;
};

```

Эта функция содержит черты, общие для всех инсерторов.

Возвращаемый тип этой функции - ostream. Это необходимое условие для того, чтобы можно было использовать инсертор этого типа в одном потоке вывода.

Функция имеет два параметра: ссылку на поток, который находится слева от знака операции <<, и объект, который находится справа от знака операции, данные этого объекта и будут выводиться.

Затем следует выдача трех значений членов объектов класса vector и возвращается поток stream. Возвращение потока обязательно.

Продемонстрируем работу инсертора на примере.

```

#include <iostream.h>
// Пример 103. Перегрузка операции вывода.
class vector{
public:
    int x, y, z;
    vector (int a, int b, int c) { x=a; y=b; z=c; }
};
// vector инсертор
ostream& operator << (ostream &stream, vector obj)
{
    stream << obj.x << " ";
    stream << obj.y << " ";
    stream << obj.z << "\n";
    return stream;
}

main(void)
{
    vector a(1, 2, 3), b(3, 4, 5);

    cout << a << "\n" << b << "\n";
    return 0;
}

```

Общая форма функции инсертора следующая:

```

ostream &operator <<(ostream &stream, class_type obj)
{
    // тело программы
    return stream;
}

```

Мы используем вариант

```
stream << obj.x << " ";
a не
cout << obj.x << " ";
```

так как второй вариант жестко связан с потоком cout, а первый вариант может быть применен к любому потоку.

В примере 103 данные - члены класса vector объявлены как public. Функция-инсертор не может быть членом класса. Для того чтобы она имела доступ к приватным элементам класса, мы должны объявить ее "другом" класса. Предыдущий пример в этом случае примет вид

```
#include <iostream.h>
// Пример 104. Перегрузка операции вывода.
// с помощью дружественной функции
class vector{
    int x, y, z;
public:
    vector (int a, int b, int c) { x=a; y=b; z=c; }
    friend ostream& operator << (ostream &stream, vector obj);
};
// vector инсертор
ostream& operator << (ostream &stream, vector obj)
{
    stream << obj.x << " ";
    stream << obj.y << " ";
    stream << obj.z << "\n";
    return stream;
}

main(void)
{
    vector a(1, 2, 3), b(3, 4, 5);
    cout << a << "\n" << b << "\n";
    return 0;
}
```

Для перегрузки функции >> для класса vector можно использовать следующую функцию-экстрактор:

```
istream &operator >> (istream &stream, vector obj)
{
    cout << "Enter x, y, z: ";
    stream >> obj.x >> obj.y >> obj.z;
    return stream;
}
```

Так же как и инсертор, функция-экстрактор не может быть членом класса, а поэтому чаще всего будет дружественной функцией класса.

Еще раз модифицируем пример с классом vector, построив для него экстрактор.

```
#include <iostream.h>
```

```
// Пример 105. Перегрузка операций ввода/вывода.
```

```
class vector{
    int x, y, z;
public:
    vector (int a, int b, int c) { x=a; y=b; z=c; }
    friend ostream& operator << (ostream &stream, vector obj);
    friend istream& operator >> (istream &stream, vector &obj);
};
// vector инсептор
ostream& operator << (ostream &stream, vector obj)
{
    stream << obj.x << ", ";
    stream << obj.y << ", ";
    stream << obj.z << "\n";
    return stream;
}
istream &operator >> (istream &stream, vector &obj)
{
    cout << "Enter x, y, z: ";
    stream >> obj.x >> obj.y >> obj.z;
    return stream;
}
```

```
main(void)
{
    vector a(1, 2, 3);
    cout << a;
    cin >> a;
    cout << a;
    return 0;
}
```

### ФОРМАТИРОВАННЫЙ ВВОД/ВЫВОД

Для того чтобы организовать форматированный ввод и вывод, аналогичный тому, что предоставляют пользователю функции `printf()` и `scanf()`, языке C++ используются два способа.

Первый состоит в применении функций членов - класса `ios`.

При втором способе употребляется специальный вид функций, называемых манипуляторами (`manipulator`).

### ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ - ЧЛЕНОВ КЛАССА `IOS`

Состояние правил форматного вывода в поток определяется состоянием флагов форматирования потока. В заголовочном файле `iostream.h` определено следующее перечисление, задающее флаги форматирования:

```
enum {
skipws      = 0x0001    отбрасывание пробелов
left        = 0x0002    выравнивание по левому краю
right       = 0x0004    выравнивание по правому краю
internal    = 0x0008    заполнение пустых позиций
dec         = 0x0010    выдача в десятичном формате
oct         = 0x0020    выдача в восьмеричном формате
hex         = 0x0040    выдача в шестнадцатеричном формате
showbase    = 0x0080    выдача основания системы счисления
showpoint   = 0x0100    выдача позиции точки .
uppercase   = 0x0200    выдача в формате xx.xxxx Eхх
showpos     = 0x0400    выдача знака у положительного числа
scientific  = 0x0800    выдача в форме с фиксированной точкой
fixed       = 0x1000    выдача в форме с плавающей точкой
unibuf      = 0x2000    улучшенная выдача
stdio       = 0x4000    освобождение потока
};
```

Формат соответствует целому типа `long`. Изменить состояние флагов можно функцией

```
long setf(long flags).
```

Чтобы установить флаг `showbase` в режим включено (`on`), используем оператор

```
stream.setf(ios::showbase);
```

здесь `stream` - тот конкретный поток, на который мы воздействуем.

Для установки флагов можно использовать побитовые операции:

```
cout.setf(ios::left|ios::hex);
```

Функция `unsetf(long flags)` возвращает флаг в состояние выключено(`off`).

Функция `long flags(void)` возвращает текущее состояние флагов.

Функция `int width(int len)` возвращает текущую ширину поля выдачи и устанавливает ее ширину `len`.

Функция `char fill(char ch)` возвращает текущий символ заполнения и устанавливает новый символ `ch` заполнения.

Функция `int precision(int num)` возвращает текущее значение десятичных знаков после точки и устанавливает новое значение.

Использование этого способа управления форматным выводом иллюстрирует следующий пример:

```
#include <iostream.h>
#include <iomanip.h>
// Пример 106. Использование флагов форматирования.

main(void)
{
```

```
long fl;
fl = cout.flags();
cout << "Исходное состояние флагов: " << fl << "\n";

cout.setf(ios::showpos);
cout.setf(ios::scientific);
cout << 123 << " " << 1.2345678 << "\n";

cout.setf(ios::hex | ios::showbase);
cout.unsetf(ios::showpos);
cout.width(20);
cout.precision(10);
cout << 123 << " " << 123.456 << " " << 1.2345678 << "\n";
cout << "Новое состояние флагов: " << cout.flags() << "\n";
cout.flags(fl); // Восстанавливаем исходное состояние
cout << " После восстановления исходного состояния флагов: \n";
cout << 123 << " " << 123.456 << 1.2345678 << "\n";

return 0;
}
```

### ИСПОЛЬЗОВАНИЕ МАНИПУЛЯТОРОВ

Для управления форматом выдачи из потока можно использовать специальные функции, называемые манипуляторами. Стандартными манипуляторами, доступ к которым можно получить, подключив файл `iomanip.h`, являются:

<i>Манипулятор</i>	<i>Действие манипулятора</i>
<code>dec</code>	Десятичный формат
<code>endl</code>	Вывод символа '\n' и освобождение буфера
<code>ends</code>	Вывод NULL
<code>flush</code>	Освободить поток
<code>hex</code>	Шестнадцатеричный формат числа
<code>resetiosflags(long f)</code>	Отключить флаги, определенные f
<code>setbase(int base)</code>	Установить основание системы счисления
<code>setfill(char ch)</code>	Установить символ заполнения
<code>setiosflags(long f)</code>	Установить режим по флагам, указанным f
<code>setprecision(int p)</code>	Установить число цифр после десятичной точки
<code>setw(int w)</code>	Установить ширину поля выдачи
<code>ws</code>	Режим пропуска символов пробела

Использование стандартных манипуляторов можно продемонстрировать следующим примером:

```
#include <iostream.h>
#include <iomanip.h>
// Пример 107. Использование манипуляторов.
```



```

main(void)
- {
    cout << setprecision(2) << 100.5375 << endl;
    cout << setw(20) << "MANIPULATORS /n";
    return 0;
}

```

### СОЗДАНИЕ МАНИПУЛЯТОРОВ

Можно создать свою собственную функцию-манипулятор. Мы покажем, как сделать функцию-манипулятор без параметров. Форма объявления функции - манипулятора функции следующая:

```

ostream &manip_name(ostream &stream)
{
    // Коды программы
    return stream;
}

```

Это манипулятор вывода. Для создания функции - манипулятора ввода надо заменить поток `ostream` на `istream`. Пример создания манипулятора с именем `left10`:

```

#include <iostream.h>
#include <iomanip.h>
// Пример 108. Создание манипуляторов

ostream& left10(ostream &stream)
{
    stream.setf(ios::left);
    stream << setw(10);
    return stream;
}

main (void)
{
    cout << 12 << left10 << 15 << 17 << "\n";
    return 0;
}

```

Использование манипуляторов может быть полезно для вывода на устройство, для которого нет predefined манипуляторов. Таким устройством может быть, например плоттер.

### РАБОТА С ФАЙЛАМИ В ЯЗЫКЕ C++

Для организации обмена с файлами к программе надо подключить файл `FSTREAM.H`. В C++ файл открывается присоединением к потоку. Если поток объявлен объектом класса `ifstream`, то он открыт для ввода, если же он присоединен к объекту класса `ofstream`, то он открыт для вывода.

Поток, предназначенный и для вывода, и для ввода, должен быть объектом класса `fstream`. Например:

```
ifstream input_from;
ofstream out;
fstream in_out_stream;
```

Если поток уже создан, то он может быть ассоциирован с файлом с помощью функции `open()`. Ее прототип

```
void open (char * filename, int mode, int access);
```

здесь `filename` - имя файла; `mode` - режим, определяющий, как открыть файл; `access` - режим доступа файла.

Режим доступа файла соответствует аналогичным режимам DOS:

Режим		Макрос
0	Открытый доступ	NORMAL
1	Только для чтения	READ_ONLY
2	Скрытый файл	HIDDEN
4	Системный файл	SYSTEM
8	Архивный файл	ARCHIVE

Эти режимы могут комбинироваться.

Значения режима открытия файла также могут комбинироваться. В файле `FSTREAM.H` определены следующие величины:

<code>ios::app</code>	- файл открыть в режиме добавления;
<code>ios::ate</code>	- маркер позиции файла установить на конец файла;
<code>ios::in</code>	- файл открыть для ввода;
<code>ios::nocreate</code>	- если файл не существует, то не создавать новый;
<code>ios::out</code>	- файл открыт для вывода;
<code>ios::trunc</code>	- если файл уже существует, то он очищается, маркер - в начале файла;
<code>ios::noreplace</code>	- прекратить выполнение, если файл уже существует;
<code>ios::text</code>	- текстовый файл;
<code>ios::binary</code>	- двоичный файл.

В случае объявления потока `ifstream` или `ofstream` автоматически устанавливаются соответственно режимы `ios::in` и `ios::out`.

При вызове функции `open()` указывать все параметры необязательно.

ВЫЗОВЫ

```
ofstream out;
out.open("myfile", ios::out, 0)
```

и вызов функции

```
ofstream out;
out.open("myfile")
```

эквивалентны. При открытии файла для ввода и вывода необходимо указать оба режима работы:

```
fstream m_in_out;
m_in_out.open("myfile", ios::in|ios::out);
```

В случае успешной операции открытия файла значение переменной типа `fstream` равно нулю.

Чтобы закрыть файл, используется функция - член класса `close()`. Например,

```
m_in_out.close();
```

Ее прототип

```
void close(void).
```

Пример открытия файла для вывода:

```
ofstream m_out;
m_out.open("myfile.txt", ios::out, 1);
if (!m_out) {
    cout << "Не могу открыть файл \n";
    return 1;
}
```

Конструкторы классов `ifstream`, `ofstream` и `fstream` имеют тот же набор документов, что и функция `open()`. Поэтому первые два оператора предыдущего примера можно записать одним оператором:

```
ofstream m_in_out("myfile.txt", ios::out, 1);
```

### ОБРАБОТКА ТЕКСТОВЫХ ФАЙЛОВ

Для ввода/вывода в текстовый файл можно непосредственно применять операции `<<` и `>>`. Например, вывод в файл `myfile.txt` :

```
#include <iostream.h>
#include <fstream.h>
// Пример 109.
main()
{
    ofstream out("myfile.txt");
    if(!out){
        cout << "Не могу открыть файл \n";
        return 1;
    }
    out << 10 << " " << 1992.2 << "\n";
    out << "Строка вводится очень просто";

    out.close();
    return 0;
}
```

Для чтения из файла можно использовать следующую программу:

```
#include <iostream.h>
#include <fstream.h>
```

// Пример 110. Чтение текстового файла

```
main()
{
    char ch;
    int i;
    float f;
    char str[80];

    ifstream in("myfile.");
    if(!in){
        cout << "Не могу открыть файл \n";
        return 1;
    }
    in >> i;
    in >> f;
    in >> ch;
    in >> str;
    cout << i << " " << f << " " << ch << "\n";
    cout << str << "\n";
    in.close();
    return 0;
}
```

### Двоичный ввод/вывод

Есть два способа работы с двоичными файлами. Первый состоит в использовании для ввода/вывода байта функций - членов соответствующего класса `put()` и `get()`; наиболее часто используемая форма этих функций следующая:

```
istream &get(char &ch)
ostream &put(char ch);
```

Функция `get()` читает простой символ из потока и присваивает значение переменной `ch`. Функция `put()` записывает символ `ch` в поток. Программа копирования файла на экран с использованием `get()` имеет следующий вид:

```
#include <iostream.h>
#include <fstream.h>
// Пример 111. Копирование файла на экран
```

```
main(int argc, char *argv[])
{
    char ch;
    if (argc != 2){
        cout << " Usage: PR <filename> \n";
        return 1;
    }
    ifstream in(argv[1]);

    if (!in) {
```

```

        cout << "Не могу открыть файл \n";
        return 2;
    }

    while (in.getch())
        cout << ch;
    in.close();
    return 0;
}

```

Когда поток `in` достигает конца файла, значение переменной `in` становится нулевым.

Следующая программа копирует строку в файл используя функцию `put()`.

```

#include <iostream.h>
#include <fstream.h>
// Пример 112. Копирование строки в файл

main(void)
{
    char *p = " Hello, World !";
    ofstream out("test");

    if (!out) {
        cout << "Не могу открыть файл \n";
        return 1;
    }

    while (*p) out.put(*p++);
    out.close();
    return 0;
}

```

Для быстрой обработки двоичных файлов можно использовать функции `read()` и `write()` с прототипами:

```

istream &read(unsigned char *buf, int num);
ostream &write(unsigned char *buf, int num);

```

Функция `read()` читает из потока `num` байт и помещает их в буфер `buf`.

Функция `write()` записывает в поток `num` байт из буфера `buf`.

```

#include <iostream.h>
#include <fstream.h>
// Пример 113. Использование функций read() и write()

```

```

main(void)
{
    float numbers [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
    int i;

    ofstream out("test");
    if (!out) {

```

```

        cout << "Не могу открыть файл \n";
        return 1;
    }

    out.write((unsigned char *) numbers, sizeof numbers);
    out.close();

    for (i = 0; i < 10; i++)
        numbers[i] = 0;

    ifstream in("test");
    if (!in) {
        cout << "Не могу открыть файл \n";
        return 1;
    }
    in.read((unsigned char *) numbers, sizeof numbers);
    for (i = 0; i < 10; i++)
        cout << numbers[i] << " ";
    cout << "\n";
    cout << "Было прочитано " << in.gcount() << " символов\n";
    in.close();
    return 0;
}

```

Функция `gcount()` возвращает количество прочитанных байт последним вызовом функции `read()`.

Для обнаружения конца файла можно использовать функцию `eof()`. Функция `eof()` возвращает ненулевое значение, когда маркер файла достигает конца файла.

Для организации прямого доступа к элементам файла служат функции

```

istream &seekg(streamoff offset, seek_dir origin);
ostream &seekp(streamoff offset, seek_dir origin);

```

Тип `seek_dir` перечислимый и переменная этого типа могут иметь следующие значения:

<code>ios::beg,</code>	- начало файла,
<code>ios::loc,</code>	- текущая позиция маркера файла,
<code>ios::end</code>	- конец файла.

Примером использования прямого доступа может служить программа, копирующая содержимое файла на экран начиная с указанного байта от начала файла:

```

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
// Пример 114. Использование функций прямого доступа.
main (int argc, char *argv[])
{
    char ch;

```

```
if (argc!=3){  
    cout << "Usage: PR <filename> <start location> \n";  
    return 1;  
}  
ifstream in(argv[1]);  
if(!in) {  
    cout <<"Cannot open file\n";  
    return 1;  
}  
in.seekg(atoi(argv[2]), ios::beg);  
while(in.get(ch))  
    cout << ch;  
return 0;  
}
```

Определить текущее положение маркера файла можно используя функции

```
streampos tellg();  
streampos tellp();
```

Заканчивая рассмотрение возможностей библиотеки ввода/вывода в языке C++, надо заметить, что в этой книге мы рассмотрели только самую вершину айсберга - только небольшую часть библиотечных функций языков C и C++ и совсем не затрагивали вопросы использования библиотек классов, таких, как Turbo Vision, Object Windows, MFC и др. Это тема отдельной книги, и не одной.

За рамками этой книги остались такие темы, как обработка исключительных ситуаций, которая не поддерживается компилятором Borland C++ 3.1, и некоторые другие.

В заключение приведем несколько примеров использования языка C++ для решения задач.

## 1. ПРИМЕРЫ ИСПОЛЬЗОВАНИЯ ЯЗЫКА C++

В качестве первого примера представлена программа, создающая в текстовом режиме простейший оконный интерфейс с использованием наследования от простого окна до окна ввода или окна вывода текста с прокруткой. Программа составлена и любезно предоставлена нам О. В. Лазаренко.)

```
#include <conio.h>
#include <dos.h>

class _window
{
private:
    void *ptr;
protected:
    int left,up,right,down;
    int color;
public:
    _window(int l=1,int u=1,int r=25,int d=15,int c=7);
    ~_window(void);
    virtual void title(void);
    virtual void move(int a,int b);
    void show(void);

    void hide(void);
};

_window::_window(int l,int u,int r,int d,int c)
{
    left = l; up = u; right = r; down = d; color = c;
    ptr = (void *)calloc((right-left+1)*(down-up+1),2*sizeof(char));
}

_window::~~_window(void)
{
    free(ptr);
}

void _window::title(void)
{
    gotoxy((left+right)/2-4,up);
    textattr(color);
    cputs("_WINDOW_");
}
```



```

void _window::show(void)
{
    int i,j;
    gettext(left,up,right,down,ptr);
    textattr(color);
    for(i=up;i<=down;i++)
    {
        gotoxy(left,i);
        for(j=left;j<=right;j++)
            cputs(" ");
    }
    for(i=left+1;i<right;i++)
    {
        gotoxy(i,up); cputs("H");
        gotoxy(i,down); cputs("H");
    }
    for(i=up+1;i<down;i++)
    {
        gotoxy(left,i); cputs("e");
        gotoxy(right,i); cputs("e");
    }
    gotoxy(left,up); cputs("Й");
    gotoxy(right,up); cputs("»");
    gotoxy(left,down); cputs("И");
    gotoxy(right,down); cputs("");
    title();
    gotoxy(left+1,up+1);
}

```

```

void _window::move(int a,int b)
{
    hide();
    left += a; right += a;
    up += b; down += b;
    show();
}

```

```

void _window::hide(void)
{
    puttext(left,up,right,down,ptr);
    gotoxy(1,1);
}

```

```

//----- Next Class -----
// O_WINDOW <-- _WINDOW
// (output window)
//-----

```

```

class o_window : public _window
{
    void *ptr;
    void text_hide(void) { gettext(left,up,right,down,ptr); }
    void text_show(void) { puttext(left,up,right,down,ptr); }
protected:
    int title_color;
    int txt_x,txt_y;
public:
    o_window(int l,int u,int r,int d,int c,int ct = 0x07);
    ~o_window();
    virtual void title(void);
    virtual void move(int a,int b);
    void put_text(char *s);
    void clr_window(void);
};

o_window::o_window(int l,int u,int r,int d,int c,int ct) : _window(l,u,r,d,c)
{
    txt_x = txt_y = 1;
    title_color = ct;
    ptr = (void *)calloc((right-left+1)*(down-up+1),2*sizeof(char));
}

o_window::~o_window()
{
    free(ptr);
}

void o_window::title(void)
{
    gotoxy((left+right)/2-6,up);
    textattr(title_color & 0x0F | 0x10);
    cputs(" OUT_WINDOW ");
}

void o_window::put_text(char *s)
{
    window(left+1,up+1,right-1,down-1);
    gotoxy(txt_x,txt_y);
    textattr(color);
    cputs(s);
    txt_x = wherex();
    txt_y = wherey();
    window(1,1,80,25);
    gotoxy(left+txt_x,up+txt_y);
}

```

```

void o_window::clr_window()
{
    int i,j;
    textattr(color);
    for(i=up+1;i<down;i++)
    {
        gotoxy(left+1,i);
        for(j=left+1;j<right;j++)
            cputs(" ");
    }
    txt_x = txt_y = 1;
    gotoxy(left+1,up+1);
}

```

```

void o_window::move(int a,int b)

```

```

{
    text_hide();
    hide();
    left += a; right += a;
    up += b; down += b;
    show();
    text_show();
}

```

```

//----- Next Class -----
// IO_WINDOW <- O_WINDOW
// (input/output window)
//-----

```

```

class io_window : public o_window

```

```

{
    int input_color;
public:
    io_window(int l,int u,int r,int d,int c,int ct,int ic = 0x07);
    void title(void);
    char *get_text(void);
    char *get_hidden_text(void);
};

```

```

io_window::io_window(int l,int u,int r,int d,int c,int ct,int ic)
: o_window(l,u,r,d,c,ct)

```

```

{
    input_color = ic;
}

```

```

void io_window::title(void)

```

```

{
    gotoxy((left+right)/2-8,up);
    textattr(title_color & 0x0F | 0x40);
}

```

```

    cputs(" IN/OUT_WINDOW ");
}

char *io_window::get_text(void)
{
    static char buffer[80];
    window(left + 1, up + 1, right - 1, down - 1);
    gotoxy(txt_x, txt_y);
    textattr(input_color & 0x0F | color & 0xF0);
    cscanf("%s", buffer);
    txt_x = wherex();
    txt_y = wherey();
    window(1, 1, 80, 25);
    textattr(color);
    gotoxy(left + txt_x, up + txt_y);
    return buffer;
}

char *io_window::get_hidden_text(void)
{
    static char buffer[80];
    window(left + 1, up + 1, right - 1, down - 1);
    gotoxy(txt_x, txt_y);
    textattr(color & 0xF0 | color >> 4);
    cscanf("%s", buffer);
    window(1, 1, 80, 25);
    textattr(color);
    gotoxy(left + txt_x, up + txt_y);
    return buffer;
}

//===== MAIN =====

main(void)
{
    char *p;
    o_window a(2, 2, 20, 12, 0x5F, 0xE);
    io_window b(7, 7, 70, 14, 0x1E, 0x3, 0xC);

    a.show();
    getch();
    a.put_text("This is normaly: ");
    getch();
    a.put_text("Test text");
    getch();
    a.put_text("Some test text");
    getch();
    b.show();
}

```

```

    getch();
    b.put_text("Enter a string: ");
    p = b.get_text();
    b.put_text("\nYour string: ");
    b.put_text(p);
    getch();
    getch();
    b.put_text("\nEnter password :");
    p = b.get_hidden_text();
    if(*p == 'u')
        b.put_text("Ok");
    else
        b.put_text("Error");
    getch();
    getch();
    b.hide();
    getch();
    a.clr_window();
    getch();
    a.hide();
    getch();
}

```

В качестве второго примера рассмотрим программу, которая строит график функции, задаваемой строкой. Нетривиальной частью этой программы является разбор выражения.

Построение собственно графика функции проблемы не составляет.

```

#include<stdio.h>
#include<string.h>
#include<ctype.h>
#include<math.h>
#include<stdlib.h>
#include<conio.h>
#include<graphics.h>

#define NUMBER 0
#define OPERATION 1
#define ARGUMENT 2
#define FUNCTION 3
#define END 4
#define PI 3.141593

inline void error(char *s) {
    printf("\n\nError %s\n\n",s);
    exit(1);
}

class TokenString {
    char *ptr;

```

```

char str[81];
public :
TokenString(char *s) { strcpy(ptr=str,s); }
void reset(void) { ptr=str; }
void gettoken(void);
int type;
char t[81];
double val;
};

void TokenString::gettoken(void) {
    while(isspace(*ptr))
        ptr++;
    if(!*ptr) {
        *t=0;
        type=END;
    }
    else if(strchr("+ - /()", *ptr)) {
        t[0]=*ptr++;
        t[1]=0;
        type=OPERATION;
    }
    else if(toupper(*ptr)=='X') {
        t[0]=*ptr++;
        t[1]=0;
        type=ARGUMENT;
    }
    else if(isdigit(*ptr)) {
        int p=0;
        char *temp=t;
        while(isdigit(*ptr)||(*ptr=='.'&&!p)) {
            if(*ptr=='.')
                p=1;
            *temp++=*ptr++;
        }
        *temp=0;
        val=atof(t);
        type=NUMBER;
    }
    else if(isalpha(*ptr)) {
        char *temp=t;
        while(isalpha(*ptr)||isdigit(*ptr))
            *temp++=*ptr++;
        *temp=0;
        type=FUNCTION;
    }
    else {
        char s[41];
        sprintf(s,"Bad symbol '%c' %x \n",*ptr,*ptr);
    }
}

```

```
    error(s);
}
}

class Function : private TokenString {
    double x;
    void primitive(double&);
    void level1(double&);
    void level2(double&);
    void level3(double&);
    void level4(double&);
    void level5(double&);
public :
    Function(char *f) : TokenString(f) {}
    double get(double);
};

double Function::get(double a) {
    x=a;
    reset();
    gettoken();
    if(type == END)
        error("No function");
    double result;
    level1(result);
    return result;
}

void Function::level1(double &result) {
    double temp;
    char op;
    level2(result);
    while((op = 't') == '+' || op == '-') {
        gettoken();
        level2(temp);
        if(op == '+')
            result += temp;
        else
            result -= temp;
    }
}

void Function::level2(double &result) {
    double temp;
    char op;
    level3(result);
    while((op = 't') == '*' || op == '/') {
        gettoken();
        level3(temp);
```

```

    if(op == '*')
        result* = temp;
    else
        result/= temp;
}
}

void Function::level3(double &result) {
    int f=0;
    char str[81];
    if(type == FUNCTION) {
        f = 1;
        strcpy(str,t);
        gettoken();
    }
    level4(result);
    if(f) {
       strupr(str);
        if(!strcmp("SIN",str))
            result = sin(result);
        else if(!strcmp("COS",str))
            result = cos(result);
        else if(!strcmp("TG",str))
            result = sin(result)/cos(result);
        else if(!strcmp("ABS",str))
            result = fabs(result);
        else error("Unknown function");
    }
}

void Function::level4(double &result) {
    int sign = 0;
    if('t' == '-') {
        sign = 1;
        gettoken();
    }
    level5(result);
    if(sign)
        result = -result;
}

void Function::level5(double &result) {
    if('t' == '(') {
        gettoken();
        level1(result);
        if('t' == ')')
            error("No "" """);
        gettoken();
    }
}

```



```

    else
        primitive(result)
}

void Function::primitive(double &result) {
    switch(type) {
        case NUMBER :
            result = val;
            gettoken();
            break;
        case ARGUMENT :
            result = x;
            gettoken();
            break;
        default :
            error("Number or X needed");
    }
}

void main(void) {
    char str[81];
    printf("\nPI=%lf\n",PI);
    gets(str);
    Function f(str);

    double xs,ys;
    printf("XSIZE=");
    scanf("%lf",&xs);
    printf("YSIZE=");
    scanf("%lf",&ys);
    xs=fabs(xs);
    ys=fabs(ys);
    int n;
    double x1=-xs/2,y1=-ys/2,x2=xs/2,y2=ys/2;
    printf("N=");
    scanf("%d",&n);

    int gd=DETECT,gm;
    initgraph(&gd,&gm,"c:\\tcplus\\bgi");
    double xd=x2-x1,xstep=xd/double(n);
    double sxstep=(x2-x1)/double(getmaxx()+1);
    double systep=(y2-y1)/double(getmaxy()+1);
    setbkcolor(BLUE);
    cleardevice();

    setcolor(WHITE);
    setlinestyle(DOTTED_LINE,0,NORM_WIDTH);
    int xx=int(-x1/sxstep);
    line(xx,0,xx,getmaxy());

```

```

int yy=int(-y1/systep);
line(0,yy,getmaxx(),yy);

setcolor(YELLOW);
setlinestyle(SOLID_LINE,0,NORM_WIDTH);
double temp=x1;
moveto(0,int((y2-f.get(temp))/systep));
for(int i=1;i<=getmaxx();i++) {
    temp+=xstep;
    double val=f.get(temp);
    lineto(int((temp-x1)/sxstep),int((y2-val)/systep));
}
getch();
closegraph();
}

```

И, наконец, последний пример, называющийся "резидентный калькулятор" помещен в эту книгу с расчетом на будущее.

В этом примере используются некоторые понятия и приемы, о которых в книге не шла. Это ассемблерная вставка, создание резидентных юграмм, опять, как и в третьем примере, разбор выражения, прямая работа с видеопамятью в текстовом режиме.

```

#include<stdlib.h>
#include<ctype.h>
#include<conio.h>
#include<bios.h>
#include<dos.h>
// Резидентный калькулятор

```

```

// Константы для рисования линий.

```

```

const char chLeftTop = 218;
const char chRightBottom = 217;
const char chLeftBottom = 192;
const char chRightTop = 191;
const char chVertical = 179;
const char chHorizontal = 196;

```

```

// class Screen - Прямой доступ в видеопамять в текстовом режиме

```

```

class Screen {
    char far *vbase;    // Адрес начала видеопамяти
public:
    Screen();
    int getVideoMode();
    void putChar(int x,int y,char ch,int attr);
    void putString(int x,int y,const char *str,int attr,int width=-1);
    void putBorder(int left,int top,int right,int bottom,int attr);
    void fillText(int left,int top,int right,int bottom,int ch=-1,int attr=-1);
    void putText(int left,int top,int right,int bottom,const void *source);

```

```

    void getText(int left,int top,int right,int bottom,void *dest);
} screen;

// Устанавливает адрес начала видеопамати. В байте по адресу 0x0000:0x0447
// находится номер текущего видеорежима. Если он равен семи, установлен
// монохромный видеоадаптер и видеопамать начинается с адреса 0xB000:0x0000.
// Во всех остальных случаях видеопамать начинается с адреса 0xB800:0x0000.

Screen::Screen() {
    unsigned char far *videoMode=(unsigned char far *)0x447;
    if(*videoMode==7)
        vbase=(char far *)0xB0000000;
    else
        vbase=(char far *)0xB8000000;
}

// Возвращает номер текущего видеорежима. Он находится в байте
// по адресу 0x0000:0x0447

int Screen::getVideoMode() {
    return *((unsigned char *)0x447);
}

// Выводит на экран символ

void Screen::putChar(int x,int y,char ch,int attr) {
    char far *p=vbase+(x-1)*2+(y-1)*160;
    *p++=ch;
    *p++=attr;
}

// Выводит на экран строку символов

void Screen::putString(int x,int y,const char *string,int attr,int width) {
    char far *p=vbase+(x-1)*2+(y-1)*160;
    for(int i=0;string[i];i++) {
        *p++=string[i];
        *p++=attr;
    }
    for(;i<width;i++) {
        *p++=' ';
        *p++=attr;
    }
}

// Рисует на экране одинарную рамку из символов псевдографики

void Screen::putBorder(int left,int top,int right,int bottom,int attr) {
    putChar(left,top,chLeftTop,attr);

```

```

for(int x=left+1;x<right;x++)
    putChar(x,top,chHorizontal,attr);
putChar(right,top,chRightTop,attr);
for(int y=top+1;y<bottom;y++) {
    putChar(left,y,chVertical,attr);
    putChar(right,y,chVertical,attr);
}
putChar(left,bottom,chLeftBottom,attr);
for(x=left+1;x<right;x++)
    putChar(x,bottom,chHorizontal,attr);
putChar(right,bottom,chRightBottom,attr);
}

// Заполняет прямоугольник на экране заданным символом с заданным атрибутом

void Screen::fillText(int left,int top,int right,int bottom,int ch,int attr) {
    for(int y=top;y<=bottom;y++) {
        char far *p=vbase+(left-1)*2+(y-1)*160;
        for(int x=left;x<=right;x++,p+=2) {
            if(ch!=-1)
                p[0]=ch;
            if(attr!=-1)
                p[1]=attr;
        }
    }
}

// Выводит на экран прямоугольную область (сохраненную ранее
// вызовом getText)

void Screen::putText(int left,int top,int right,int bottom,const void *source) {
    const unsigned *buffer=(const unsigned *)source;
    for(int y=top;y<=bottom;y++) {
        unsigned far *p=(unsigned far *)vbase+left-1+(y-1)*80;
        for(int x=left;x<=right;x++)
            *p++=*buffer++;
    }
}

// Сохраняет в памяти по адресу dest содержимое прямоугольной области экрана

void Screen::getText(int left,int top,int right,int bottom,void *dest) {
    unsigned *buffer=(unsigned *)dest;
    for(int y=top;y<=bottom;y++) {
        unsigned far *p=(unsigned far *)vbase+left-1+(y-1)*80;
        for(int x=left;x<=right;x++)
            *buffer++=*p++;
    }
};

```

```

// Пропускает пробелы в строке

#define skipspaces(p) while(isspace(*p)) p++;

// class Expression - арифметическое выражение

class Expression {
protected :
    const char *text; // Текст арифметического выражения
    const char *p;    // Указатель на следующий символ при разборе выражения
    int error;        // Признак ошибки
public :
    Expression(const char *text) : text(text), error(0)
    { }
    int wasError()
    { return error; }
    int getResult();
protected :
    int level1();
    int level2();
    int level3();
};

// Вычисление выражения методом рекурсивного спуска

int Expression::getResult() {
    error=0;
    p=text;
    int result=level1();
    if(!error)
        error=(*p!=0);
    return result;
}

// Вычисление суммы и разности

int Expression::level1() {
    int result=level2();
    while(!error) {
        skipspaces(p);
        if(*p=='+') {
            p++;
            result+=level2();
        } else if(*p=='-') {
            p++;
            result-=level2();
        } else break;
    }
}

```

```
    return result;  
}
```

```
// Вычисление произведения частного
```

```
int Expression::level2() {  
    int result = level3();  
    while(!error) {  
        skipspaces(p);  
        if(*p == '"') {  
            p++;  
            result = level3();  
        } else if(*p == '/') {  
            p++;  
            int divider = level3();  
            if(!divider)  
                error = 1;  
            else  
                result /= divider;  
        } else break;  
    }  
    return result;  
}
```

```
// Обработка чисел и выражений в скобках.
```

```
int Expression::level3() {  
    skipspaces(p);  
    if(*p == '(') {  
        p++;  
        int result = level1();  
        skipspaces(p);  
        if(*p++ != ')')  
            error = 1;  
        return result;  
    } else if(isdigit(*p)) {  
        int result = 0;  
        while(isdigit(*p))  
            result = result * 10 + *p++ - '0';  
        return result;  
    } else {  
        error = 1;  
        return 0;  
    }  
}
```

```
// Рисует окно и вычисляет выражения
```

```
void runCalculator() {
```

```

const int left = 10;
const int top = 8;
const int right = 69;
const int bottom = 17;
static char buffer[(right-left+1)*(bottom-top+1)*2];
// Нарисовать окно
screen.getText(left,top,right,bottom,buffer);
screen.fillText(left+2,top+1,right,bottom,-1,0x07);
screen.putBorder(left,top,right-2,bottom-1,0x1F);
screen.fillText(left+1,top+1,right-3,bottom-2,' ',0x1F);
screen.putString(left+2,top," Evaluate ",0x1F);
screen.putString(left+2,top+2,"Expression",0x1F);
screen.putString(left+2,top+3,"",0x2F,right-left-5);
screen.putString(left+2,top+5,"Result",0x1F);
screen.putString(left+2,top+6,"",0x2F,right-left-5);
gotoxy(left+2,top+3);
// Основной цикл работы калькулятора
static char text[right-left-5];
for(int x=0;;) {
    int key=getch();
    if(key==27)
        break;
    else if(key==13) {
        text[x]=0;
        Expression expr(text);
        int result=expr.getResult();
        if(expr.wasError())
            screen.putString(left+2,top+6,"Error in expression.",0x2F,right-left-5);
        else {
            static char answer[7];
            screen.putString(left+2,top+6,itoa(result,answer,10),
                            0x2F,right-left-5);
        }
    } else if(key==8 && x) {
        x--;
        screen.putChar(left+2+x,top+3,' ',0x2F);
        gotoxy(left+2+x,top+3);
    } else if(key>=32 && key<=255 && x<right-left-6) {
        text[x]=key;
        screen.putChar(left+2+x,top+3,key,0x2F);
        x++;
        gotoxy(left+2+x,top+3);
    }
}
screen.putText(left,top,right,bottom,buffer);
}

unsigned stackSeg,stackOfs; // Стек нашей программы
unsigned saveStackSeg,saveStackOfs; // Стек прерванной программы

```

```

int saveX, saveY; // Координаты курсора

void interrupt (*old9)(...); // Указатель на старый обработчик прерывания9h
// Новый обработчик прерывания от клавиатуры
void interrupt new9(...) {
    unsigned char scanCode = inportb(0x60);
    (*old9)(); // Вызываем старый обработчик - Он разблокирует клавиатуру
    // и контроллер прерываний
    if(scanCode == 62 && bioskey(2)&0xC) { // CTRL-ALT-F4
        static int inside = 0;
        if(!inside) {
            asm {
cli
mov saveStackSeg, ss
mov saveStackOfs, sp
mov ss, stackSeg
mov sp, stackOfs
sti
            }
            inside++;
            saveX = whereX();
            saveY = whereY();
            bioskey(0); // Считаем нажатую клавишу F4
            runCalculator();
            gotoxy(saveX, saveY);
            inside--;
            asm {
cli
mov ss, saveStackSeg
mov sp, saveStackOfs
            }
        }
    }
}

void main(void) {
    printf("Resident Calculator. Press CTRL-ALT-F4 to activate.\n\r");
    asm {
        cli
        mov stackSeg, ss // Сохраним указатель стека для
        mov stackOfs, sp // использования в обработчике прерывания.
        sti
    }
    old9 = getvect(9);
    setvect(9, new9);
    unsigned size = stackSeg + (stackOfs/16) + 1 - _psp; // Размер программы в
    параграфах
    keep(0, size); // Оставляем программу резидентной
}

```



Приемы, использованные в этом примере, будут рассматриваться в дальнейших книгах. Их можно найти, например, в книгах из серии “Библиотека системного программиста”, которая издается Диалог-МИФИ.

## 2. ИСПОЛЬЗОВАНИЕ ВСТРОЕННОГО ОТЛАДЧИКА

Система Borland C++ 3.1 имеет в интегрированной среде встроенный отладчик исходных текстов. Для того чтобы можно было пользоваться встроенным отладчиком, надо быть уверенным в том, что программа компилируется с включенными соответствующими опциями. Чтобы отладчик работал, надо включить отладочную информацию в выполняемый файл. Это же необходимо для того, чтобы воспользоваться внешним отладчиком, например таким, как Turbo Debugger. Опция, управляющая включением отладочной информации, находится в меню Options|Debugger|Source Debugging:



По умолчанию эта опция включена.

Отладчик исходных текстов, в отличие от традиционных, не требует дизассемблирования, а автоматически включает оригинальный исходный текст программы в выполняемый файл. Отладчик связывает скомпилированный объектный код с каждой строкой исходного текста программы.

Отладчик предоставляет пользователю много возможностей. Можно управлять выполнением программы установкой точек прерывания в исходном тексте. Можно выполнять программу пошагово, наблюдая за изменением значений переменных и выражений.

### ОСНОВЫ ИСПОЛЬЗОВАНИЯ ОТЛАДЧИКА

В этом разделе обсуждаются наиболее общие команды отладчика. Прежде чем начинать изучать возможности отладчика, необходимо иметь программу. В качестве демонстрационной программы предлагаем использовать следующую простую программу на языке C++:

```
#include <iostream.h>
// Пример программы для демонстрации возможностей отладчика

void sqr_it ( int n );

main ( void )
{
```

```

int i;
for ( i=0; i<10; i++ ) {
    cout<< "i= "<< i << " ";
    sqr_it ( i );
}
return 0;
}
void sqr_it ( int n )
{
    cout << n*n << "\n";
}

```

После того как программа введена, откомпилируйте ее и запустите на выполнение. Программа должна напечатать значения от 0 до 9 и квадраты их чисел.

### ПОШАГОВОЕ ВЫПОЛНЕНИЕ ПРОГРАММЫ

Пошаговое выполнение программы - это процесс выполнения одного оператора (точнее, одной строки программы) за один шаг.

Для пошагового выполнения программы с использованием встроенного плагина системы Borland C++ 3.1 нажмите клавишу F7. Заметим, что юбка программы, содержащая main (void) в окне редактирования, будет делена. Это начало выполнения программы. Заметим также, что строки

```
#include<iostream.h>
```

```
void sqr_it(int n);
```

будут пропущены, так как директивы препроцессора и объявление про-  
типа функции не генерируют кода и автоматически пропускаются отлад-  
ком. Это же касается операторов объявления переменных. Нажатие кла-  
виши F7 эквивалентно выбору пункта меню Run | Trace into. Нажмите F7  
сколько раз. Выделенная строка передвигается от строки к строке. Выде-  
ется та строка, которая будет выполняться при следующем шаге. Когда  
мы достигнем строки

```
cout<< "i= "<< i << " ";
```

при следующем нажатии F7 откроется окно с файлом iostream.h. Выде-  
нная строка будет находиться в функции, которая перегружает операцию  
<. Если бы мы использовали функцию printf() для вывода значений i, мы  
ючили бы эту строку за один шаг. Это связано с тем, что функция printf()  
содержит отладочной информации, в то время как функция - операция  
перегрузки << содержит отладочную информацию (она компилируется при  
осмотре подключаемого файла iostream.h). Когда встретится вызов функ-  
ции sqr\_it(), выделенная строка передвинется в тело функции. Возможно  
исполнение только одной функции, выполняя вызов функции как опера-  
ра за один шаг. Это достигается нажатием клавиши F8 (Run|Step over).  
ажатие клавиш F8 и F7 можно комбинировать в любом порядке.

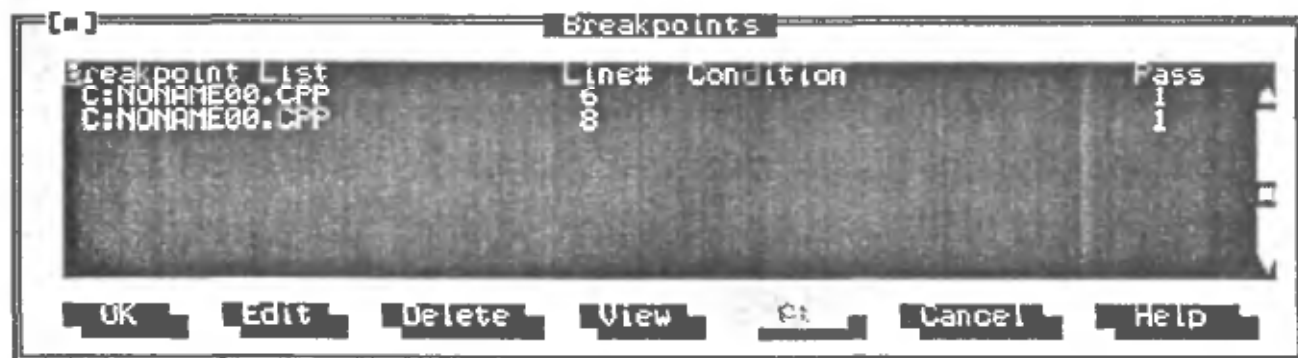
Установка точек прерывания. При всей пользе пошагового выполнения программы это может быть довольно длительным процессом в большой программе или при наличии циклов, особенно если место программы, которое надо отладить, находится далеко от начала программы. В отладчике предусмотрена возможность двигаться по программе большими шагами. Первая возможность - выполнение программы до строки, в которой находится курсор. Эту возможность можно реализовать, установив курсор в требуемую строку и нажав F4 или выбрав пункт меню Run|Go to cursor.

Вторая возможность - установить точку прерывания (breakpoint). Чтобы установить точку прерывания, переместите курсор в окне редактирования в ту строку, в которой хотите приостановить выполнение программы. Затем выберите пункт меню Debug|Toggle breakpoint. Эквивалентная комбинация клавиш - Ctrl-F8. Строка будет выделена ярким цветом. Этот пункт меню работает в триггерном режиме (переключательном режиме), повторный выбор пункта меню отменит точку прерывания, в которой находится курсор.

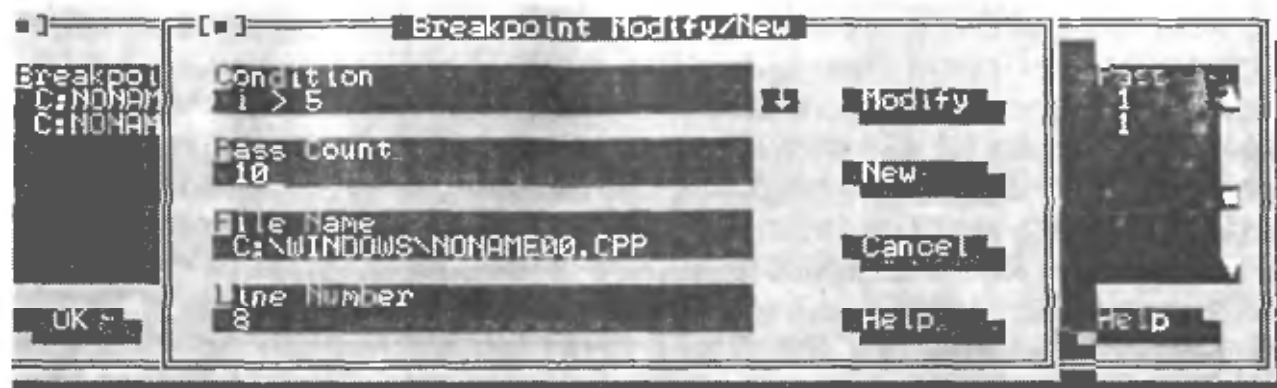
В программе можно установить несколько точек прерывания.

Если установлена одна или несколько точек прерывания, начните выполнение программы (Run|Run или Ctrl-F9). Выполнение программы остановится на первой встретившейся точке прерывания. Операторы в строке, в которой находится точка прерывания, выполняться не будут. Продолжить выполнение программы можно в пошаговом режиме или опять нажав Ctrl-F9 или F4. В любом случае при достижении очередной точки прерывания выполнение программы будет приостановлено.

При установленных точках прерывания можно вывести на экран список точек прерывания, который позволяет добавлять (удалять) точки прерывания, устанавливать условия или счетчик остановки программы при достижении точки прерывания. При выборе Debug|Breakpoints... на экране появится окно диалога



выбрав в котором кнопку Edit получим еще одно окно диалога, в котором можно модифицировать условия, и счетчик (count):



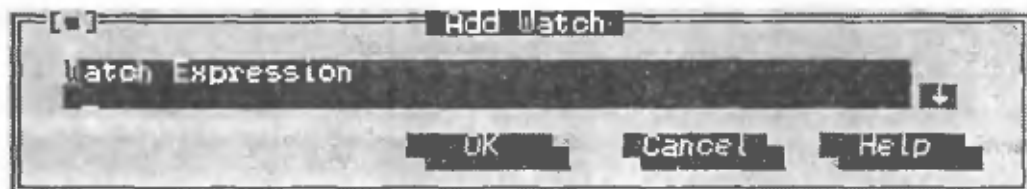
Установка программы на начало (Program Reset). Если выполнение программы в дальнейшем нецелесообразно (найдена ошибка или требуется вернуться в начало программы), надо выбрать пункт меню Run|Program Reset (Ctrl-F2). Сеанс отладки будет прекращен, и программа будет готова к выполнению с самого начала.

### НАБЛЮДЕНИЕ ПЕРЕМЕННЫХ

Одна из самых необходимых возможностей в процессе отладки - возможность посмотреть текущее значение одной или нескольких переменных в процессе выполнения программы. В отсутствие отладчиков эта проблема решалась добавлением в текст программы "отладочной печати". Теперь в этом нет необходимости. Определите переменные, значение которых хотите контролировать, выберите пункт меню Debug | Watches



выберите Add watch. В открывшемся окне диалога



введите имя переменной или выражение. Отладчик откроет окно Watch, в котором будет переменная или выражение и значение переменной или выражения. Можно продолжать добавлять переменные в окно Watch. Можно продолжать выполнение программы. В процессе выполнения программы в пошаговом (пошаговом) режиме значения в окне Watch будут автоматиче-

ски изменяться. Если переменная глобальная, ее значение доступно в любом месте программы. Если же переменная локальная, ее значение доступно лишь в области видимости переменной. Если переменная недоступна, то в окне Watch вместо значения выдается соответствующее предупреждение.

При просмотре выражений в окне Watch есть два ограничения. Во-первых, в выражении запрещен вызов функций. Во-вторых, в выражении не могут применяться макросы, определенные с использованием `#define`.

Отладчик Borland C++ позволяет осуществлять форматный вывод наблюдаемых значений. Для задания формата используется следующая форма:

`expression, format_code.`

Список кодов формата задан в таблице

Код формата	Значение
C	В виде символа
D	Десятичное число
F (n)	Число с плавающей точкой
H или X	Шестнадцатеричное число
M	Показать память (dump)
P	Указатель
R	Структуры: вывести имена и значения членов
S	Вывести управляющие символы

В формате F можно указать число значащих цифр после запятой: `average, F5.`

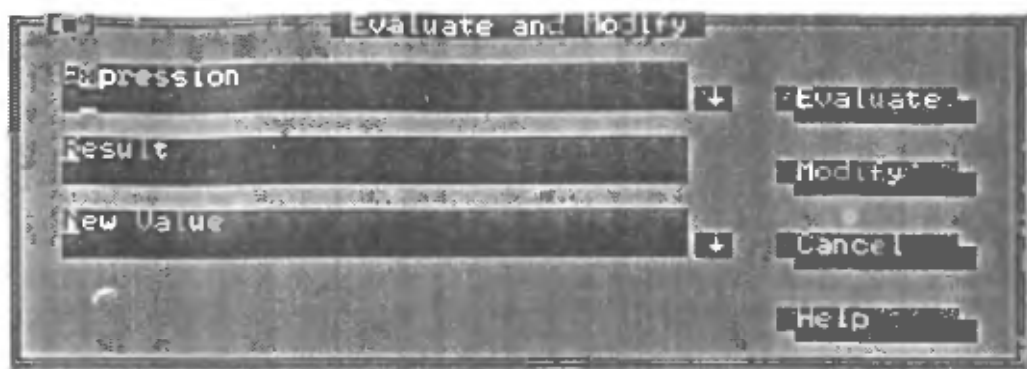
Если формат не указан, отладчик сам подбирает соответствующий тип выбора формата. Если переменная имеет тип `char*`, отладчик выдаст не указатель, а соответствующую этому указателю строку символов. Отладчик позволяет наблюдать и объект языка C++, при этом можно использовать формат R.

### Стек вызовов функций (CALL STACK)

В процессе выполнения программы можно вызвать на экран состояние стека вызовов функций, используя пункт меню `Debug|Call stack`. При этом выдается на экран последовательность вложенных вызовов функций с указанием значения фактических параметров. Локальные переменные и адреса возврата не выдаются. Удобство этой опции можно оценить при отладке рекурсивных функций.

### ВЫЧИСЛЕНИЕ И ИЗМЕНЕНИЕ ЗНАЧЕНИЙ

При выборе пункта меню `Debug|Evaluate/Modify` на экране появится окно диалога



котором можно задать выражение, не содержащее вызовов функций и макросов. Значение этого выражения высветится во второй строке. Если выражение является величиной типа `lvalue` (например, простая переменная), можно в нижней строке ввода задать новое значение, нажать кнопку `Modify` и продолжить выполнение программы с новым значением выражения или переменной.

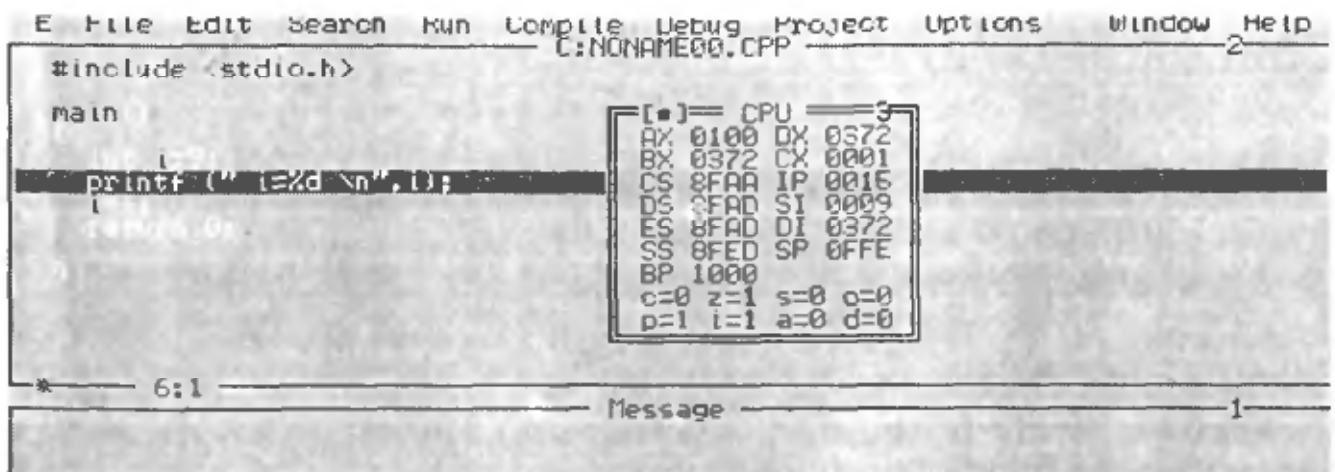
### Окно INSPECT

Хотя информации о переменных при наблюдении в окне `Watch` бывает достаточно, можно получить более подробную информацию, открыв окно `spect`. При выборе пункта меню `Debug|Inspect` откроется окно диалога, в котором можно ввести имя переменной, и на экране появится более подробная информация, включая тип, адрес размещения переменной в памяти и ее значение. Особенно информативно окно `Inspect` при анализе объектов типа `class`. При просмотре объекта класса можно установить курсор внутри окна `Inspect` на член этого класса (переменную или функцию), нажать `Enter` и на экране появится еще одно окно `Inspect` для этого члена класса. При пошаговом выполнении программы данные в окне `Inspect` автоматически изменяются. В отличие от окна `Watch` нельзя открыть окно `Inspect` для переменной вне области действия этой переменной.

Можно открыть несколько окон `Inspect`, разместив их в разных местах экрана.

### РЕГИСТРЫ ПРОЦЕССОРА

В процессе отладки можно просматривать содержимое регистров процессора и установленных флагов. При выборе `Window|Registers` на экране появляется окно с именами и содержимым регистров процессора:



Если вы умеете анализировать имеющуюся информацию, она может оказаться весьма полезной при отладке.

### Окно OUTPUT

При выдаче результатов в процессе отладки отображение происходит на пользовательский экран (User Screen), который обычно закрыт интегрированной средой. Чтобы не переключаться во время отладки для просмотра результатов вывода на экран, целесообразно открыть окно Output, выбрав пункт меню Window|Output. Открывшееся окно можно перемещать по экрану и изменять его размеры. Оно отображает ту часть User Screen, в которой произошло последнее изменение. При пошаговом выполнении программы вывод на экран отображается в этом окне. Правда, есть ограничение: в окне отображается только текстовая информация. Графический режим в окне Output не поддерживается.

Этих сведений хватает для отладки достаточно простых программ. В более сложных функциях возможно использование внешних отладчиков, которые поставляются вместе с компиляторами. Например, отладчик Turbo Debugger фирмы Borland.



## ПИСОК ЛИТЕРАТУРЫ

Керниган Б., Ритчи Д. Язык программирования Си. 2-е изд., М.: Финансы и статистика. 1992

Джехани Н. Программирование на языке Си. М.: Радио и связь, 1988.

Касаткин А. И. Профессиональное программирование на языке Си. Системное программирование. Минск: Высшая школа, 1993.

Касаткин А. И. Профессиональное программирование на языке Си. Управление ресурсами. Минск: Высшая школа. 1993.

Касаткин А. И., Вальвачев А. Н. Профессиональное программирование на языке Си. От Turbo C к Borland C++. Минск: Высшая школа. 1995.

Бочков С. О., Субботин Д. М. Язык программирования Си для персонального компьютера. М.: СП "Диалог", Радио и связь, 1990.

Страуструп Б. Язык программирования C++. 2-е изд.: В 2 т. Киев: ДИАСофт, 1993.

Эллис М., Строуструп Б. Справочное руководство по языку программирования C++ с комментариями. М.: Мир, 1992.

Ирэ, Пол. Объектно-ориентированное программирование с использованием C++. Киев: ДИАСофт, 1995.

Цимбал А. А., Майоров А. Г., Козодаев М. А. Turbo C++: язык и его применение. М.: Джен Ай Лтд, 1993.

Подбельский В. В. Язык Си++. М.: Финансы и статистика, 1995.

Собоцинский В. В. Практический курс Turbo C++. М.: Свет, 1993.

Рассохин Д. От Си к Си++. М.: Эдель, 1993.

Вайнер Р., Пинсон Л. C++ изнутри. Киев: ДИАСофт, 1993.

Березина Н. И., Лопушенко В. В., Посохов И. Н. Symantec C++ Professional.

Жешке Р. Толковый словарь стандарта языка Си. Спб.: Питер. 1994.

Тондо К., Гимпел С. Язык Си. Книга ответов. М.: Финансы и статистика, 1994.

Schildt H. Using Turbo C++. Osborn McGraw-Hill, 1990.

Borland C++ v.3.1. Users Guide.


Borland C++ v.3.1. Programmers Guide.

Borland C++ v.3.1. Library Reference.



# ОГЛАВЛЕНИЕ

---

<b>ВВЕДЕНИЕ.....</b>	<b>3</b>
<b>1. ИНТЕГРИРОВАННАЯ СРЕДА ПРОГРАММИРОВАНИЯ</b>	
<b>СИСТЕМЫ BORLAND C++.....</b>	<b>7</b>
Особенности системы Borland C++ 3.1 .....	7
Работа в интегрированной среде Borland C++ .....	7
Запуск системы Borland C++ .....	8
Выход из системы Borland C++ .....	8
Компоненты интегрированной среды.....	9
Окна системы Borland C++ .....	9
Главное меню.....	11
Блок диалога .....	12
Триггерные и селективные кнопки .....	13
Блоки ввода и блоки списка.....	13
Описание элементов главного меню.....	14
Системное меню -  .....	14
Меню File (Файл) .....	14
Меню Edit (Редактирование) .....	17
Меню Search (Поиск).....	18
Меню Run (Выполнение) .....	22
Меню Compile (Компиляция) .....	23
Меню Debug (Отладка) .....	25
Меню Project (Проект).....	27
Меню Options (Параметры) .....	28
Меню Window (Окно).....	29
Меню Help (Подсказка).....	32
Редактирование файлов в системе Borland C++ .....	33
Поиск и замена .....	36
Поиск парных символов .....	36
Компилятор командной строки.....	38
О других компиляторах языка C++.....	39
<b>2. ВВЕДЕНИЕ В ЯЗЫК С .....</b>	<b>40</b>
Некоторые особенности языка С .....	40
Основные понятия .....	40
Две простые программы .....	42
Немного о функциях языка С .....	46
Два простых оператора: if и for.....	49
Точка с запятой, скобки и комментарии .....	51
Определение некоторых понятий.....	51

Переменные, константы, операции и выражения	52
Базовые типы данных	52
Объявление переменных	55
Константы в языке C	56
Символьные переменные и строки	59
Инициализация переменных	61
Выражения	62
Функции printf() и scanf()	63
Операции языка C	66
Арифметические операции	67
Операции отношения и логические операции	69
Операция присваивания	71
Поразрядные операции (битовые операции)	71
Операции ( ) и [ ]	73
Операция условие ?	73
Операция запятая	74
Операция sizeof	74
Управляющие операторы	74
Условный оператор if	74
Оператор switch	76
Циклы	78
Оператор goto	82
Массивы и указатели	83
Объявление массива в программе	83
Массивы символов. Строки	84
Функции для работы со строками	85
Двумерные массивы	86
Инициализация массивов	87
Указатели	89
Объявление указателей	90
Операции над указателями	90
Связь указателей и массивов	93
Массивы указателей	95
Инициализация указателей	97
Функции в языке C	97
Объявление функции	97
Оператор return	98
Прототипы функций	100
Область действия и область видимости	102
Классы памяти	103
Параметры и аргументы функции	108
Аргументы функции main()	111
Рекурсивные функции	113
Функции с переменным числом параметров	114

Указатель на функцию .....	116
Типы данных, определяемые пользователем .....	118
Динамическое распределение памяти .....	119
Функции malloc() и free() .....	119
Нелокальный переход .....	121
Типы, определяемые пользователем .....	123
Структура .....	124
Доступ к отдельному биту .....	127
Объединения (union) .....	128
Перечислимый тип .....	130
Переименование типов - typedef .....	130
Модели памяти .....	131
Препроцессор языка C .....	132
Директива #define .....	133
Директивы условной компиляции .....	135
Предопределенные макросы .....	137
Стандартные заголовочные файлы .....	139
Библиотеки ввода/вывода и работа с файлами в языке C .....	140
Ввод/вывод на консоль .....	141
Указатель на файловую переменную .....	142
Управление экраном в текстовом режиме в MS DOS .....	146
Основные функции работы в текстовом режиме .....	147
Введение в графику Borland C++ .....	153
<b>3. ЯЗЫК C++ .....</b>	<b>163</b>
C++ - язык объектно-ориентированного программирования .....	163
Что такое объектно-ориентированное программирование .....	163
Особенности языка C++, не связанные напрямую с объектной ориентированностью .....	164
Компиляция программ на языке C++ .....	166
Введение в понятие класса и объекта .....	166
Перегруженные функции .....	171
Перегрузка операций .....	173
Наследование .....	173
Конструкторы и деструкторы .....	175
Новые ключевые слова C++ .....	178
Конструктор с параметрами .....	178
Дружественные функции .....	182
Дружественные классы .....	187
Аргументы функции, задаваемые по умолчанию .....	187
Структуры и классы .....	189
Объединения и классы .....	190
Подставляемые (inline) функции .....	191
Наследование классов .....	193

Конструкторы с параметрами при наследовании .....	196
Множественное наследование .....	199
Передача объектов как аргументов функций .....	202
Массивы объектов .....	203
Указатель на объект .....	204
Перегрузка функций и операций .....	205
Перегрузка конструкторов .....	205
Динамическая инициализация и локальные переменные .....	206
Ключевое слово <code>this</code> .....	208
Перегрузка операций .....	209
Дружественные функции-операции .....	213
Ссылки .....	215
Использование ссылочных переменных для перегрузки унарных операций .....	218
Перегрузка операции индексации [ ] .....	219
Использование виртуальных функций .....	220
Указатели на производные типы .....	220
Виртуальные функции .....	223
Чистые виртуальные функции и абстрактные типы .....	227
Производные классы и их конструкторы и деструкторы .....	229
Порядок вызова конструкторов и деструкторов при множественном наследовании .....	231
Виртуальные базовые классы .....	232
Операции динамического выделения памяти <code>new</code> и <code>delete</code> .....	232
Виртуальные деструкторы .....	236
Шаблоны классов и функций .....	239
Шаблоны функций .....	239
Шаблоны классов .....	241
Статические члены класса .....	243
Локальные классы .....	245
Вложенные классы .....	245
Использование библиотеки ввода/вывода языка C++ .....	246
Потоки языка C++ .....	247
Перегрузка операций ввода/вывода. Инсерторы и экстракторы .....	247
Форматированный ввод/вывод .....	250
Работа с файлами в языке C++ .....	253
<b>ПРИЛОЖЕНИЯ</b> .....	260
1. Примеры использования языка C++ .....	260
2. Использование встроенного отладчика .....	277
Основы использования отладчика .....	277

Березин Начальный ку  
рс C и C++  
НДС 0% Цена 54 00

