М. УЭИТ С. ПРАТА Д. МАРТИН

Язык Си
руководство для начинающих

Перевод с английского Л. Н. Горинович и В. С. Явниловича
под редакцией д-ра техн. наук Э. А. Трахтенгерца

ББК 32.973 У97 УДК 681.3
ISBN 5-03-001309-1 /русск./
ISBN 0-672-22090-3 /англ./

Оглавление

---

1. Вступление

ИСТОРИЯ СИ ДОСТОИНСТВА СИ ЯЗЫКИ КОМПИЛЯЦИИ

Добро пожаловать в мир языка Си. В данной главе мы попробуем подготовить вас к изучению

этого мощного языка, завоевывающего все большую популярность. Что вам для этого нужно? Во-первых, интерес к Си, который, по-видимому, у вас уже есть. Но, чтобы усилить его, мы кратко обрисуем некоторые привлекательные стороны данного языка. Во-вторых, вы нуждаетесь в учебнике по языку Си - и учебником послужит вам эта книга. Кроме того, вам нужен доступ к какой-нибудь вычислительной системе, в которой имеется компилятор с языка Си. Это вы должны обеспечить себе сами. Наконец, вам необходимо научиться выполнять Си-программу на вашей вычислительной системе, и мы в конце главы дадим вам несколько советов по этому поводу.

ПРОИСХОЖДЕНИЕ ЯЗЫКА СИ

Далее Содержание

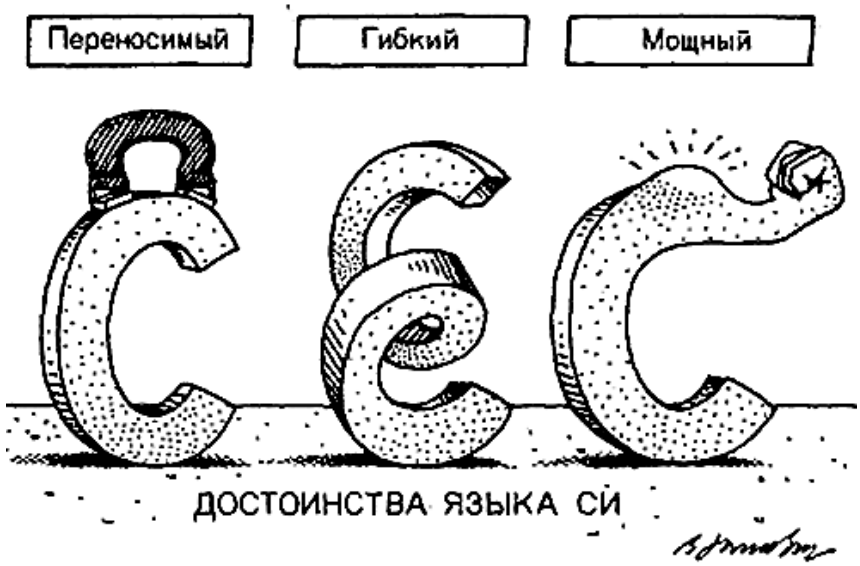
Сотрудник фирмы Bell Labs Деннис Ритчи создал язык Си в 1972 г. во время совместной работы с Кеном Томпсоном над операционной системой UNIX. Ритчи не выдумал Си просто из головы - прообразом послужил язык Би, разработанный Томпсоном, который в свою очередь..., но это уже другая история. Важным моментом для нас является то, что язык Си был разработан как инструмент для программистов-практиков. В соответствии с этим главной целью его автора было создание *удобного* и *полезного* языка.

Мы думаем, что критерий полезности принимался во внимание ПРИ разработке большинства языков программирования, но, кроме того, часто учитывались и другие потребности. Одной из главных Целей при создании языка Паскаль, например, было построение ПРОЧНЫХ основ обучения принципам программирования. Язык Бейсик создавался так, чтобы его синтаксис был близок к синтаксису английского языка; поэтому им легко могли пользоваться студенты, не знакомые с ЭВМ. Все эти цели тоже важны, но они не всегда совместимы с прагматическими, каждодневными требованиями. Предпосылки, послужившие основой создания языка Си как средства программирования, позволили разработать, кроме того, язык, облегчающий труд программиста.

ДОСТОИНСТВА ЯЗЫКА СИ

Далее Содержание

Язык Си быстро становится одним из наиболее важных и популярных языков программирования. Его использование все более расширяется, поскольку часто программисты предпочитают язык Си всем другим языкам после первого знакомства с ним. Когда вы изучите язык Си достаточно хорошо, вы сможете оценить многие из его достоинств. Сейчас мы упомянем лишь некоторые из них.



Си - *современный язык*. Он включает в себя те управляющие конструкции, которые рекомендуются теоретическим и практическим программированием. Его структура побуждает программиста

использовать в своей работе нисходящее проектирование, структурное программирование и пошаговую разработку модулей. Результатом такого подхода является надежная и читаемая программа.

Си - *эффективный язык*. Его структура позволяет наилучшим образом использовать возможности современных ЭВМ. На языке Си программы обычно отличаются компактностью и быстротой исполнения.

Си - *переносимый, или мобильный, язык*. Это означает, что программа, написанная на Си для одной вычислительной системы, может быть перенесена с небольшими изменениями (или вообще без них) на другую. Если модификации все-таки необходимы, то часто они могут быть сделаны путем простого изменения нескольких элементов в "головном" файле, который сопутствует главной программе. Конечно, структура большинства языков программирования подразумевает переносимость, но тот, кто переносил программу, написанную на Бейсике, с персональной ЭВМ IBM PC на машину Apple (они во многом, похожи) или пытался выполнить программу, написанную на Фортране для машины типа IBM, в системе UNIX, знает о многих возникающих при этом мучительных

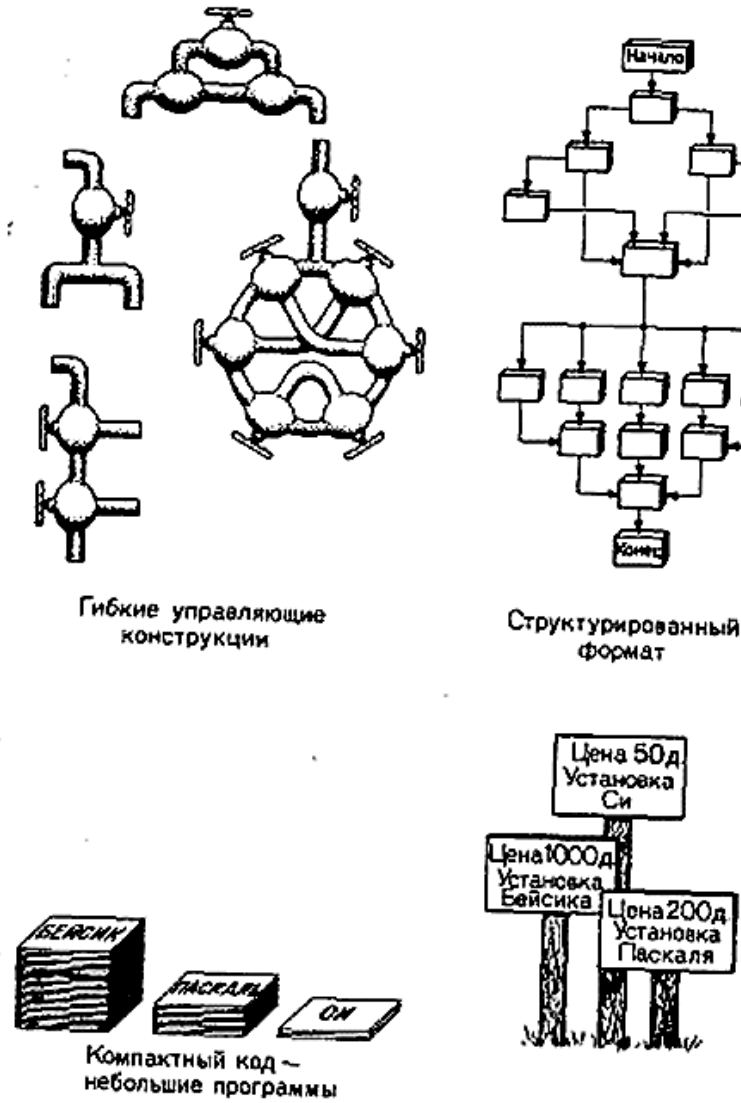


РИС. 1.1. Достоинства языка Си.

проблемах. Язык Си предоставляет исключительные возможности для переноса программ. Компиляторы с данного языка реализованы почти на 40 типах вычислительных систем, начиная от 8-разрядных микропроцессоров и кончая CRAY-1 одним из самых мощных в настоящее время

суперкомпьютеров.

Си - *мощный и гибкий язык* (два излюбленных слова в литературе по вычислительной технике). Например, большая часть мощной и гибкой (!) операционной системы (ОС) UNIX написана на языке Си. Речь идет о компиляторах и интерпретаторах других языков, таких, как Фортран, АПЛ, Паскаль, Лисп, Лого и Бейсик. Поэтому, когда вы используете компилятор с Фортрана в системе UNIX, результирующая объектная программа в конечном счете получается с помощью некоторой программы, написанной на языке Си. Кроме того, программы, написанные на Си, используются для решения физических и технических проблем и даже для производства мультипликационных фильмов.

Си *обладает рядом замечательных конструкций управления*, обычно ассоциируемых с ассемблером. Если вы остановите свой выбор на Си, то сможете реализовывать свои программы с максимальной эффективностью.

Си - *удобный язык*. Он достаточно структурирован, чтобы поддерживать хороший стиль программирования, и вместе с тем не связывать вас смирительной рубашкой ограничений.

Кроме уже упомянутых достоинств язык Си имеет и некоторые другие. У него, безусловно, есть и ряд недостатков, но вместо того чтобы далее углубляться в это, давайте обсудим еще один вопрос.

БУДУЩЕЕ ЯЗЫКА СИ

Далее Содержание

Язык Си уже занимает доминирующее положение в мире мини-компьютеров, работающих под управлением ОС UNIX. Сейчас он распространяется на область персональных ЭВМ. Многие фирмы, производящие программное обеспечение, все чаще обращаются к Си, как к удобному языку для реализации своих проектов: программ обработки текстов, составления крупноформатных таблиц, компиляторов и т. д., поскольку известно, что Си позволяет получить компактные и эффективные программы. Что еще важнее, эти программы могут быть легко модифицированы и адаптированы к новым моделям ЭВМ.

Другой причиной, способствующей проникновению Си в программное обеспечение персональных ЭВМ, является желание пользователей, работающих в системе UNIX, отлаживать свои программы дома. Поскольку уже созданы варианты компилятора с языка Си для некоторых моделей персональных ЭВМ, пользователи имеют возможность это делать.

По нашему мнению, то, что подходит для фирм и программистов с большим опытом работы на Си, хорошо и для остальных пользователей. Все больше и больше программистов останавливают свой выбор на языке Си, чтобы воспользоваться его преимуществами. Поэтому вам совсем не обязательно быть. профессиональным программистом, чтобы следовать их примеру.

Короче говоря, Си суждено стать одним из наиболее важных языков программирования в 80-90-е годы. Он уже применяется на мини-компьютерах и персональных ЭВМ. Он используется фирмами, производящими программное обеспечение, студентами, обучающимися программированию, и различными энтузиастами. И если вы хотите работать в сфере программотехники, то один из первых вопросов, на который вы должны будете отвечать "да", - "Умеете ли вы программировать на Си?".

ИСПОЛЬЗОВАНИЕ ЯЗЫКА СИ

Далее Содержание

Си - язык "компилируемого" типа. Не огорчайтесь, если это звучит для вас пока как непонятный набор слов; вы поймете, что это значит, когда мы опишем этапы процесса создания работающей Си-программы.

Если вы привыкли использовать какой-нибудь язык программирования компилируемого типа, например Паскаль или Фортран, вам будут понятны основные этапы "сборки" программ, написанных на Си. Но если ваш опыт основан на работе с такими языками

РИС. 1.2. Области применения языка Си.

интерпретируемого типа, как Бейсик и Лого, или у вас совсем нет соответствующей подготовки, то процесс сборки может показаться вам поначалу необычным. К счастью, мы можем объяснить вам все детали этого процесса, и вы увидите, что на самом деле он достаточно понятен и прост.
Чтобы дать вам первое представление о процессе создания программы, ниже приводится упрощенная схема того, что необходимо сделать - начиная от написания программы и кончая ее выполнением.

1. Используйте "редактор текстов" для создания программы на языке Си.
2. Попытайтесь осуществить трансляцию вашей программы с помощью удобного для вас компилятора. Он проведет проверку правильности вашей программы и, если обнаружит ошибки, выдаст сообщение об этом. В противном случае компилятор выполнит перевод программы в некоторый внутренний язык ЭВМ и поместит результат в новый файл.
3. Набрав имя этого нового файла на клавиатуре дисплея, вы можете запустить вашу программу.

В некоторых вычислительных системах второй этап может быть разбитым на два или три шага, но его суть от этого не изменится. Давайте рассмотрим теперь каждый этап более подробно.

Использование текстового редактора для подготовки программы

Далее Содержание

В отличие от языка Бейсик у Си нет собственного текстового редактора. В качестве него вы можете использовать любой из редакторов общего типа, имеющихся в вашей вычислительной системе. В операционной системе **UNIX**, например, это чаще всего редакторы **ed**, **ex**, **edit**, **emacs** или **vi**. На персональном компьютере это может быть **ed**, **edlin**, **Wordstar**, **Volkswriter** или любой другой из широкого набора редакторов. При работе с некоторыми из них вам необходимо будет определить конкретную версию редактора (путем задания соответствующих параметров). Например, при использовании редактора **Wordstar** необходимо ввести параметр **N**, указывающий на отсутствие документирования.

При работе с редактором от вас потребуется, во-первых, не ошибаться, набирая текст программы на пульте дисплея, и, во-вторых, выбирать имя для файла, в который будет помещена данная программа. Правила выбора имени выглядят довольно просто: оно должно быть допустимым именем в вашей вычислительной системе и должно оканчиваться символом **.c**. Ниже приведены два правильно построенных имени: **sort.c add.c**

Первая часть имени файла должна напоминать вам, что программа делает. Вторая часть (символ **.c**) указывает на то, что данный файл содержит текст программы, написанной на языке Си. В программировании принято называть часть имени, следующую за точкой, "расширением". Расширения используются для того, чтобы информировать вас и вычислительную систему о типе файла.

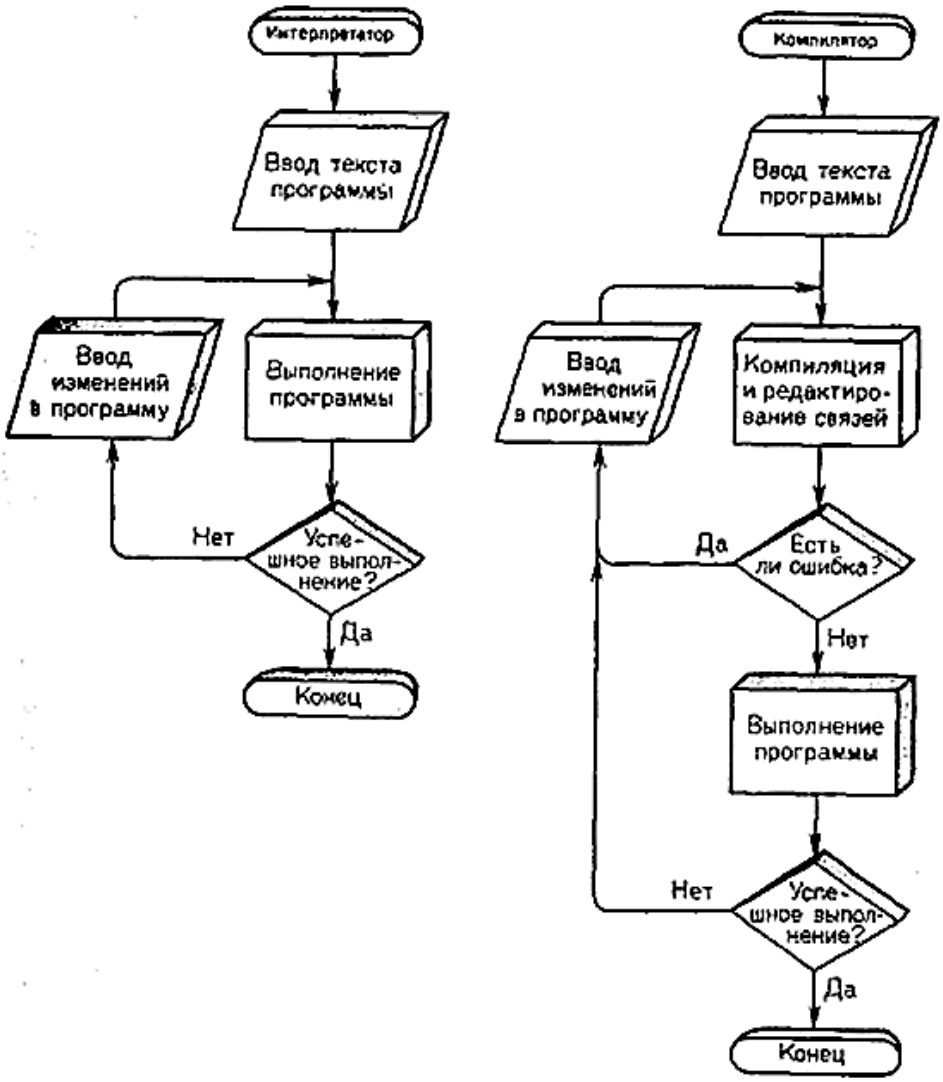


РИС. 1.3. Схема работы интерпретатора и компилятора.

Рассмотрим простой пример. Предположим, что при помощи редактора мы подготовили программу, которая приведена ниже, и поместили ее в файл с именем **inform.c**.

```
#include
main( ) {
  printf (" Символ  .c используется как окончание имени файла с СИ-программой. \n"); [1] }
```

Выше приведенный текст, который мы набрали на клавиатуре дисплея, обычно называется исходным кодом (текстом); он содержится в исходном файле. Важным моментом, который необходимо сразу отметить, является то, что наш исходный файл - это начальный пункт процесса программирования, а не его конец.

Исходные файлы и выполняемые файлы

Далее Содержание

Наша замечательная программа, несмотря на свою лаконичность и простоту, для компьютера является совершенно бессмысленным набором символов, так как он "не понимает" директив типа **#include** или **printf**. Он понимает только специальный язык, называемый машинным кодом, - набор последовательностей двоичных цифр, например, 10010101 и 01101001. Если мы хотим, чтобы

компьютер выполнил программу, мы должны осуществить перевод (трансляцию) *нашего* кода (исходного) в ее код (машинный). В результате этих действий будет получен выполняемый файл[2)] , т. е. файл, содержащий весь необходимый машинный код, требующийся компьютеру для выполнения задания.

Если вышеприведенные рассуждения выглядят скучными и непонятными, не огорчайтесь. Дело в том, что процесс перевода удалось переложить на сам компьютер! "Умные" программы, называемые компиляторами, выполняют весь объем работы, связанный с этим переводом. Детали процесса зависят от особенностей конкретной системы. Ниже кратко описано несколько способов перевода.

Компиляция Си-программы в ОС UNIX

Далее Содержание

Компилятор с языка Си в ОС UNIX называется **cc**. Чтобы осуществить компиляцию нашей программы, на клавиатуре дисплея необходимо набрать только строку:

```
cc inform.c
```

Через несколько секунд интерпретатор команд ОС UNIX выдаст на экран дисплея символ "приглашение", информируя нас, что задание выполнено. (Вообще говоря, мы можем получить предупреждения и сообщения об ошибках в том случае, если программа была написана с ошибками, но давайте предположим, что все было сделано правильно.) Если мы используем директиву ls, осуществляющую вывод на экран списка имен файлов, мы обнаружим новый файл с именем **a.out** - файл с выполняемой программой, содержащий результат трансляции (или "компиляции") нашей исходной программы. Чтобы выполнить ее, мы должны только набрать на клавиатуре символы **a.out** и на экране дисплея появится фраза:

Символ **.c** используется как окончание имени файла с Си-программой.

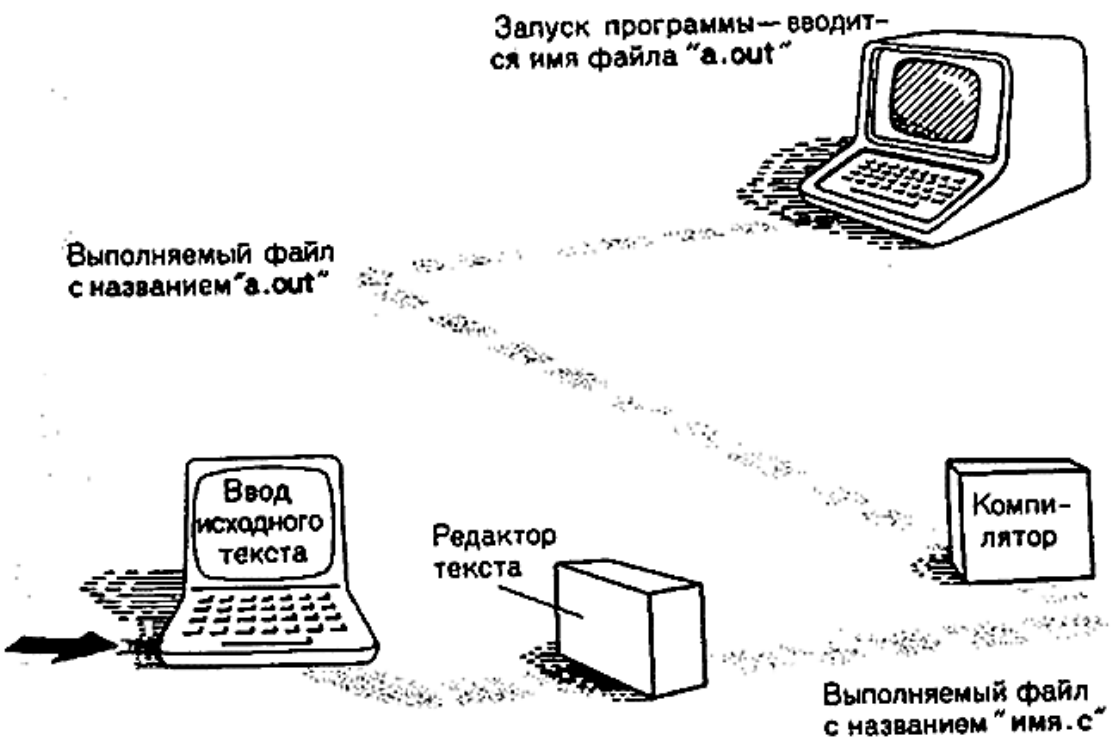


РИС. 1.4. Создание Си-программы в среде ОС UNIX.

Программа-компилятор, называемая **cc**, объединяет несколько последовательных шагов задания в один. Это станет более очевидным, когда мы рассмотрим выполнение аналогичного процесса компиляции на персональном компьютере.

Компиляция Си-программы на IBM PC (компиляторы Microsoft C и Lattice C)

Далее Содержание

Описанное ниже разбиение процесса компиляции программы на последовательные шаги зависит как от операционной системы, так и от самого компилятора. Конкретный пример, который мы здесь рассматриваем, - это функционирование компилятора Microsoft C под управлением операционной системы PC DOS I.I (Компилятор Lattice C, лежащий в основе версии, реализованной фирмой Microsoft, запускается по аналогичным правилам, только вместо команд **mc1** и **mc2** необходимо использовать команды **lc1 lc2**.)
Так же как и прежде, мы считаем, что исходная программа содержится в файле **inform .c**. Наша первая команда выглядит следующим образом:

```
mcl inform
```

(Компилятор интерпретирует строку символов **inform** как **inform.c**.) Если ошибок нет, то в результате будет получен промежуточный файл с именем **inform.q**. Затем мы набираем на клавиатуре следующую команду:

```
mc2 inform
```

в результате выполнения которой будет создан файл с именем **inform.obj**, содержащий так называемый "объектный код" (код на языке машины), соответствующий нашей исходной программе. (Объяснения приведены ниже.) После этого вводится команда

```
link c inform
```

по завершении которой создается файл **inform.exe**. Наша цель достигнута - получен файл, содержащий выполняемую программу. Если мы затем введем команду

```
inform. exe
```

или просто

```
inform
```

то наша программа начнет выполняться.

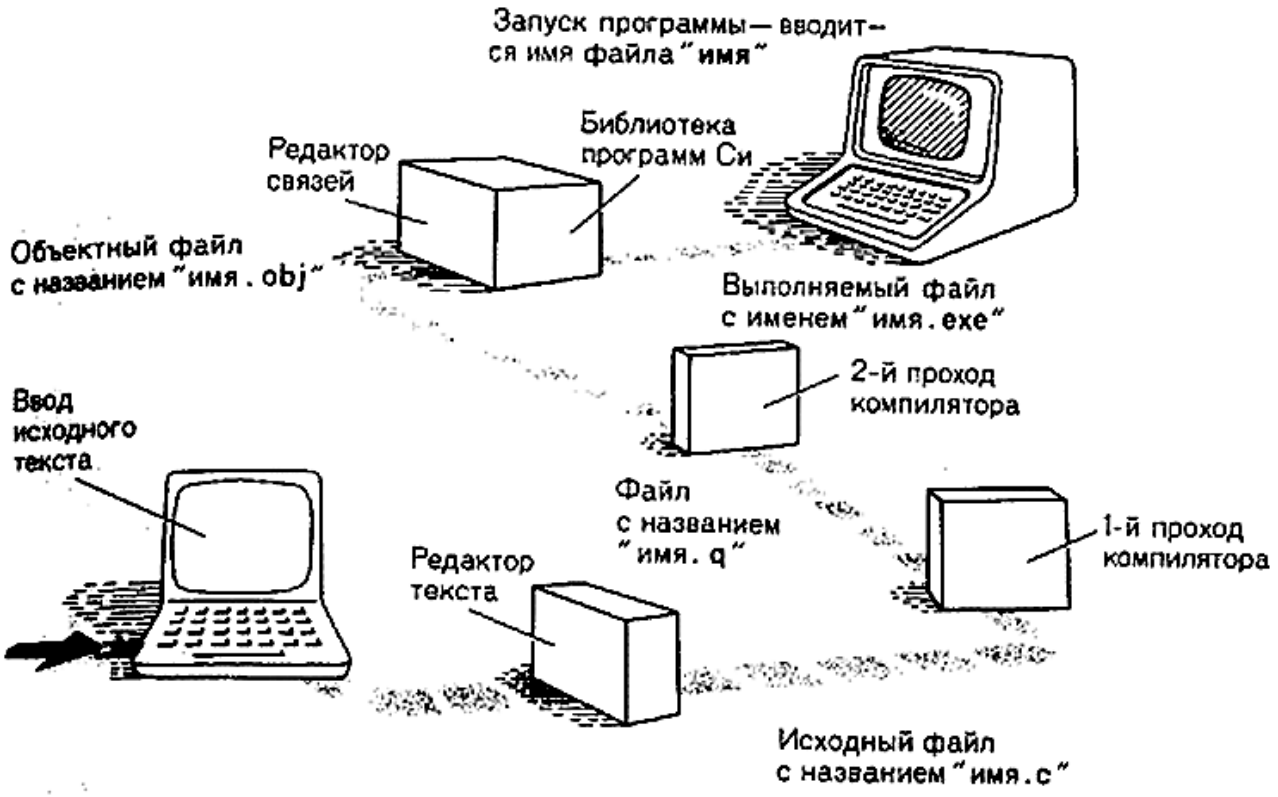


РИС. 1.5. Создание Си-программы при помощи компиляторов Microsoft C и Lattice C.

На самом деле вы можете не знать, что происходит, когда вы пользуетесь вышеописанной процедурой, но, если вам интересно, мы кратко опишем выполняемые при этом действия.

Что здесь нового? Во-первых, новым является то, что вводится файле именем **inform.obj**. Поскольку в нем содержится машинный код, непонятно, почему мы не остановились в этом месте? Ответом может служить то, что полная программа включает в себя части, которые мы не писали. Например, мы использовали команду **printf**, являющуюся программой, помещенной в Си-библиотеку. Вообще говоря, может возникать необходимость использовать в программе стандартные процедуры, помещенные в различные библиотеки. Эта потребность приводит к использованию второго нового понятия - команды **link**.

Программа **link** является частью операционной системы IBM POS. Она связывает наш объектный код (находящийся в файле **inform.obj**) с некоторыми стандартными процедурами, содержащимися в файле **c.obj**, и, кроме того, осуществляет поиск требуемых объектных модулей в той библиотеке, которую мы указываем (программа **link** запрашивает требуемое имя во время выполнения); в данном случае это будет библиотека с именем **lc.lib**. Затем указанная программа объединяет все найденные модули в одну полную программу.

Программа **cc**, работающая под управлением ОС UNIX, во время выполнения проходит аналогичную последовательность шагов; отличие состоит только в том, что она "скрывает" этот факт от нас, уничтожая файл с объектным модулем после его использования для получения полной программы. (Но в случае необходимости в ответ на соответствующий запрос компилятор выдаст нам объектный файл под именем **inform.o**.)

Альтернативный способ трансляции

Далее Содержание

В некоторых компиляторах с языка Си, работающих на персональных ЭВМ, реализованы другие способы трансляции. Метод, который только что обсуждался, можно охарактеризовать тем, что в результате мы получаем файл, содержащий объектный код (имя файла оканчивается символами

.obj), а затем используем системный компоновщик для получения файла с выполняемой программой (его имя оканчивается символами **.exe**). Альтернативный метод состоит в том, что в результате трансляции мы вначале имеем файл, содержащий "ассемблерный код" (имя файла оканчивается символами **.asm**), а затем используем системную программу, называемую ассемблером, для получения файла с выполняемой программой.

Утомленный читатель может воскликнуть: "Как, неужели еще один код?" Поэтому сразу же поясним: ассемблерный код тесно связан с машинным кодом. Фактически это тот же самый код, только представленный в символьном виде. Например, JMP может соответствовать коду 11101001, являющемуся частью машинной команды, в результате выполнения которой осуществляется "перескок" (переход) к другой ячейке. (Вы, вероятно, представляете себе компьютер, снующий по пчелиным сотам, а мы имеем в виду другие ячейки памяти.) Программисты не без основания считают, что ассемблерный код более легок для восприятия, чем чисто машинный код, а задача перевода с одного языка на другой вполне может быть возложена на специальную программу, называемую ассемблером.

Почему компиляция?

Далее Содержание

Читатели, пользовавшиеся языком Бейсик, могут удивиться, зачем столько шагов для того, чтобы выполнить программу. Кажется, что такой способ компиляции требует больше времени (и в некоторых случаях это может быть действительно так). Но, поскольку в результате компиляции программа выполняется гораздо быстрее чем обычная программа, написанная на Бейсике, вам просто приходится испытывать некоторые неудобства при получении гораздо эффективнее работающего конечного продукта.

НЕКОТОРЫЕ СОГЛАШЕНИЯ

Далее Содержание

Теперь мы уже почти готовы начать последовательное описание языка Си. Нам осталось только упомянуть о некоторых соглашениях, которых мы будем придерживаться.

Вид шрифта

Далее Содержание

Для представления текста программ, данных ввода-вывода, имен файлов, программ и переменных мы применяем специальный шрифт, похожий на тот, который вы можете видеть на экране дисплея или на бумаге при выводе на печать. Мы уже использовали его несколько раз, но если вы не обратили на это внимания, то запомните, что он выглядит следующим образом:

```
printf (" Здравствуйте! \n ");
```

Цвет

Далее Содержание

Сообщения компьютера во время диалога с пользователем даются голубым цветом; кроме того, важные слова, отражающие основные идеи или понятия, используемые в данной главе, мы помещаем перед текстом главы и печатаем также голубым цветом.

Устройство ввода-вывода

Далее Содержание

Вообще говоря, существует много способов ведения диалога человека с ЭВМ, но мы будем предполагать, что вы вводите команды при помощи клавиатуры и читаете ответ на экране дисплея.

Обычно вы посылаете команду ЭВМ, нажимая на клавишу с надписью **enter** (ввод), **c/r** (возврат каретки) или **return** (возврат). Названия клавиш иногда обозначаются прописными буквами. Пусть клавиша **[enter] - [ввод]**. Здесь квадратные скобки означают, что вы должны нажать на единственную клавишу, а не набирать все слово по буквам.

Кроме того, мы будем упоминать управляющие символы, назы вая их **[CTRL/d]**. Это обозначение указывает на то, что необходимо нажать клавишу **[d]**, держа одновременно нажатой клавишу **control**.

Некоторые детали реализации языка Си, например объем памяти, требуемый для того, чтобы поместить туда число, зависят от конкретной системы. Когда мы рассматриваем какие-нибудь при. меры и употребляем слова "наша система", мы имеем в виду персональный компьютер IBM PC, работающий под управлением операционной системы DOS I.I, и компилятор с языка Си Lattice C.

В тех случаях, когда мы говорим о программах, работающих в среде ОС UNIX, мы имеем в виду версию BSD 4.1 этой операционной системы, созданную в Калифорнийском университете (Беркли) и функционирующую на ЭВМ VAX 11/750.

СОВЕТ

Вы должны изучать программирование активно, а не просто пассивно читать данную книгу. С этой целью мы включили в на много примеров. Вы должны попытаться решить, хотя бы некоторые из них на вашей вычислительной системе, чтобы получить луч шее представление о том, как они работают. Не бойтесь модифицировать эти программы, чтобы увидеть, к чему это приведет. Пытайтесь отвечать на вопросы и выполнять упражнения, помещенные в конце каждой главы. Если вы будете активны, вы изучи теязык Си быстро и узнаете его достаточно глубоко.
Теперь мы с вами готовы приступить к изучению гл. 2.

[1)] Символы **.\n** - будут объяснены позднее. - *Прим. ред.*

[2)] Этот файл можно называть "файлом с выполняемой программой". - *Прим. ред.*

Предисловие редактора перевода

Созданием языков программирования занимаются в большинстве случаев очень квалифицированные люди, часто группы программистов, а иногда даже международные коллективы. Однако подавляющее большинство языков программирования умирало, едва родившись. Лишь к немногим из них был проявлен интерес, и буквально единицы получили действительно широкое распространение. К таким "счастливым" языкам принадлежит язык Си, разработанный Д. Ритчи. Он появился не на пустом месте. Ему предшествовали и оказали на него серьезное влияние язык BCPL, разработанный М. Ричардсоном, и язык Би (B), созданный К. Томпсоном.

Си - это достаточно выразительный язык программирования, предназначенный для описания широкого круга задач и содержащий современные механизмы управления вычислительным процессом и работы с данными. В то же время язык Си очень прост: в него введены некоторые средства, характерные скорее для ассембле-ров, чем для языков высокого уровня. Простота языка не требует создания слишком сложных компиляторов и позволяет получать достаточно

эффективный объектный код. Эти свойства языка особенно важны при написании операционных систем, но они могут оказаться очень полезными и при разработке прикладных программ.

Наибольшую популярность язык приобрел у системных программистов. Видимо, этому способствовали как сам факт успешного написания на языке Си переносимой операционной системы UNIX, получившей широкое распространение, так и элегантность и лаконичность языка. Чрезвычайно привлекательными для системных программистов оказались возможность использовать память раз-яичных типов в том числе регистровую, введение указателей, являющихся аналогами косвенных адресов, а также возможность работы со сложными структурами данных, применение препроцессора и Удобство работы с символьными строками.

Прикладные программы пишутся обычно на традиционных языках высокого уровня, например на Фортране. Однако в тех случаях, когда некоторые части таких программ оказываются особенно критичными в отношении времени, их можно писать не на ассемблере, как обычно, а на языке Си. Такой подход сократит время разработки прикладных программ, обеспечит их достаточную эффективность, а во многих случаях и переносимость, т. е. использование одной программы на ЭВМ различных типов.

Предлагаемая читателю книга - учебник по языку Си. Книга написана известными американскими специалистами М. Уэйтом, С. Пратой и Д. Мартином с большим педагогическим и методическим мастерством: излагаемый материал проиллюстрирован многочисленными примерами и задачами. Это обеспечивает легкость понимания и усвоения предмета. Поскольку трансляторы с языка Си появились сейчас на отечественных машинах, книга может представлять интерес не только для широкого круга читателей, впервые знакомящихся с языком Си, но и для системных программистов, инженеров и научных работников.

Перевод выполнили В. С. Явнилович (предисловие, гл. 1-9) и Л. Н. Горинович (гл. 10-15 и приложения).

Э. А. Трахтенгерц

Предисловие

Си - простой, изящный язык программирования, на котором останавливает свой выбор все большее число программистов. Эта книга (если вы не посмотрели на титульный лист) называется "Язык Си. Руководство для начинающих"; она представляет собой простой и хороший учебник по языку Си.

Слова "Руководство для начинающих", стоящие в подзаголовке книги, говорят о том, что нашей целью было дать обзор основ языка Си. В программировании опыт - великий учитель; с этой целью в книге приведено много задач учебного и познавательного характера. Мы пытались использовать рисунки всюду, где, как мы надеялись, они помогут внести ясность. Чтобы вы имели возможность проверить себя, в конце каждой главы приводятся вопросы для самоконтроля (и ответы на них). Мы не предполагаем у вас большого опыта работы на каком-нибудь языке программирования, однако иногда будем сравнивать язык Си с другими языками, ориентируясь на тех читателей, которые знают их.

Мы несколько расширили границы обычного учебника: обсудили ряд более сложных тем, таких, как использование структур, приведение типов, работу с файлами; в приложении мы рассмотрели возможности побитовой обработки на языке Си, а также некоторые расширения языка. Мы описали программную среду компилятора с языка Си, функционирующего как с операционной системой UNIX, так и с программным обеспечением микрокомпьютеров: например, обсудили вопрос переключения ввода-вывода и продемонстрировали использование портов в микропроцессорах INTEL 8086/8088. И наконец, мы включили шутливые рисунки как одно из довольно приятных

дополнений.

Мы попытались сделать эту книгу поучительной, понятной и полезной. Чтобы получить максимальную пользу от книги, вы должны работать с ней самым активным образом. Не занимайтесь просто чтением примеров. Вводите их в вашу вычислительную систему и пытайтесь выполнить. Хотя Си и переносимый (или мобильный) язык, вполне возможно, вы найдете различия между тем, как программа работает в вашей системе и у нас. Не бойтесь экспериментировать - измените часть программы, чтобы увидеть, к чему это приведет. Модифицируйте ваши программы, чтобы они слегка отличались от исходных. Попробуйте не обращать внимания на наши иногда встречающиеся предупреждения и посмотрите, что при этом произойдет. Попытайтесь ответить на вопросы и выполнить упражнения. Чем больше вы сделаете самостоятельно, тем большему научитесь.

Мы желаем вам удачи при изучении языка Си. Мы попытались сделать книгу отвечающей вашим потребностям и надеемся, что она поможет вам достичь поставленных целей.

Мы благодарим Роберта Лафора из издательства Waite Group за редакторские советы и Боба Петерсена за техническую помощь. Мы приносим благодарность также компании Lifeboat Associates (в особенности Джошуа Аллену и Тодду Кацу) за возможность использовать компилятор Lattice C. Мы благодарим специалистов компаний C-Systems, Software Toolworks, Telecon Systems и Supersoft за предоставленную нам информацию о своих компиляторах с языка Си. Один из авторов, С. Прата, посвящает свой труд родителям - Вики и Биллу - с любовью.

М Уэйт С. Прата Д. Мартин

---

2. Введение в язык Си

СТРУКТУРА ПРОСТОЙ ПРОГРАММЫ
ОПИСАНИЕ ПЕРЕМЕННЫХ
ИСПОЛЬЗОВАНИЕ КОМЕНТАРИЕВ
ЧИТАЕМОСТЬ ПРОГРАММ
ОПЕРАЦИИ

Как выглядит программа, написанная на языке Си? Возможно, вы уже обратили внимание на пример, приведенный в гл. 1, и нашли, что эта программа выглядит довольно специфически из-за наличия в ней символов типа **(** и **\n"**. Когда вы прочтете данную книгу, вы обнаружите, что появление этих и Других характерных для языка Си символов станет менее странным, более понятным и, возможно, даже предпочтительным! Начало настоящей главы будет посвящено обсуждению довольно простого примера программы и объяснению того, что она делает. При этом мы рассмотрим некоторые из основных средств языка Си. Если какие-то детали остаются для вас неясными и вы захотите получить более подробные, ответы на возникшие вопросы, не огорчайтесь. Мы займемся этим в следующих главах.

ПРИМЕР ПРОСТОЙ ПРОГРАММЫ НА ЯЗЫКЕ СИ

Далее Содержание

Давайте рассмотрим простую программу на языке Си. Следует сразу сказать, что такой пример нужен нам лишь для выявления некоторых основных черт любой программы, написанной на языке Си. Далее мы дадим пояснения к каждой строке, но, перед тем как вы с ними познакомитесь, просто взгляните на программу и попробуйте понять, если сможете, что она будет делать.

```
#include
main( ) /*простая программа*/
{
    int num;
```

```
    num = l;
    printf ("Я простая");
    printf ("вычислительная машина .\n");
    printf ("Мое любимое число %d, потому что оно самое первое .\n", num);
}
```

Если вы считаете, что программа должна вывести нечто на экран дисплея, то вы совершенно правы! Несколько труднее понять, что же появится на экране на самом деле, поэтому давайте выполним программу на ЭВМ и посмотрим к чему это приведет.

Первый шаг заключается в использовании имеющегося у вас текстового редактора для создания файла, содержащего текст программы. Этому файлу необходимо будет присвоить какое-то имя; если вам не приходит в голову ничего оригинального, то назовите его **main.c**. Выполните компиляцию вашей программы. (Для этого вы должны терпеливо ознакомиться с руководством по компилятору, имеющемуся в составе вашей вычислительной системы.) Теперь запустим программу. Если все пойдет хорошо, то результат должен выглядеть следующим образом:

В общем этот результат не кажется особенно неожиданным. Но какую роль в программе выполняют символы **\n** и **%d**? И вообще некоторые строки выглядят немного странно. Здесь необходимы дополнительные пояснения.

ПОЯСНЕНИЯ

Далее Содержание

Мы выполним два просмотра текста программы: во время первого объясним смысл каждой строки, а во время второго - рассмотрим дополнительные вопросы и детали.

Первый просмотр: краткий обзор

#include -включение другого файла.
Эта строка указывает компилятору, что нужно включить информацию, содержащуюся в файле **stdio.h**.

main( ) - имя функции

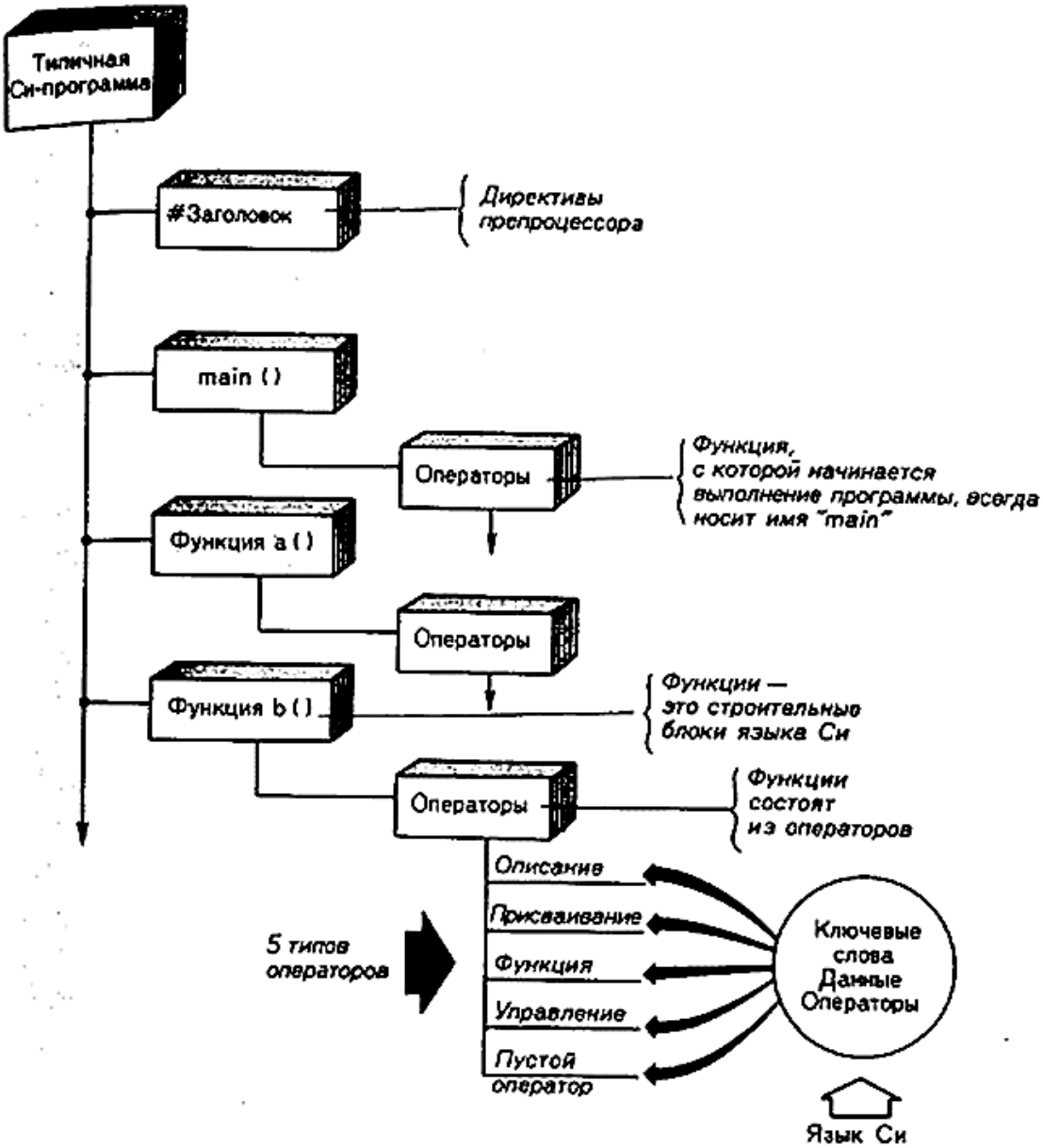


РИС. 2.1. Структура программы, написанной на языке Си.

Любая программа, написанная на языке Си, состоит из одной или более "функций", являющихся основными модулями, из которых она собирается.
Наша программа состоит из одной функции **main**, и круглые скобки указывают именно на то, что **main( )** - имя функции.

/*простая программа*/ - комментарий.
Вы можете использовать пары символов **/*** и ***/** в качестве отбывающей и закрывающей скобок для комментария. Комментарии - это примечания, помогающие понять смысл программы. Они предназначены для читателя и игнорируются компилятором.

- начало тела функции.
Открывающая фигурная скобка отмечает начало последовательности операторов, образующих тело (или определение) функции. Конец определения отмечается закрывающей фигурной скобкой }.

int num; - оператор описания.

С помощью такого оператора мы объявляем, что будем использовать в программе переменную **num**, которая принимает целые **(int)** значения.

num =1; - оператор присваивания.
Этот оператор служит для присваивания переменной **num** значения 1.

printf (" Я простая"); - оператор вывода на печать.
С его помощью выводится на печать фраза, заключенная в кавычки: *Я простая*

printf(" вычислительная машина.\n"); - еще один оператор вывода на печать.
Этот оператор добавляет слова *вычислительная машина.* в конец последней печатаемой фразы. Комбинация символов **\n** указывает компилятору на начало новой строки.

printf ("Мое любимое число %d, потому что оно самое первое. \n", num);
Этот оператор выводит на печать значение переменной **num** (равное 1), содержащееся во фразе в кавычках. Символы **%d** указывают компилятору, где и в какой форме печатать значение этой переменной **num**.

} - конец. Как уже упоминалось, программа завершается закрывающей фигурной скобкой.
Теперь рассмотрим нашу программу более внимательно.

Второй просмотр: детали Далее Содержание

#include < stdio.h> :
Файл с именем **stdio.h** является частью пакета, имеющегося в любом компиляторе языка Си и содержащего информацию о вводе-выводе (например, средства взаимодействия программы с вашим терминалом). В качестве имени файла используется аббревиатура английских слов:

standard input/output header - стандартный заголовок ввода-вывода.

(Программисты называют набор данных, содержащийся в начале файла, заголовком.)В некоторых случаях включение этой строки в начало программы обязательно, а в некоторых - нет. Мы не можем дать однозначную рекомендацию, поскольку ответ зависит как от программы, так и от используемой вами вычислительной системы. При работе на нашей системе вводить указанную строку в эту программу совсем не обязательно, но на системе, имеющейся у вас, она может быть необходимой. В любом случае ее использование не принесет никакого вреда. В дальнейшем мы будем указывать эту строку только тогда, когда действительно необходимо.

Возможно, вас удивляет, почему одно из основных средств языка - процедуры ввода-вывода - не включается компилятором в программу автоматически. Дело в том, что этот пакет используется далеко не всегда, а ведь одна из целей создания языка Си - получение компактного объектного кода. Между прочим, упомянутая строка не является даже оператором языка Си. Символ # указывает, что она должна быть обработана "препроцессором" языка Си. Как вы уже могли предположить из названия, препроцессор осуществляет некоторую предварительную обработку текста программы перед началом компиляции. В дальнейшем мы рассмотрим несколько примеров использования команд препроцессора.

main( )
Выбор имени **main** в качестве названия нашей программы довольно очевиден; более того, назвать ее как-то по-другому и нельзя. Дело в том, что программа, написанная на языке Си, всегда начинает выполняться с функции, называемой **main( )**, поэтому мы имеем возможность выбирать имена для всех используемых нами Функций кроме той, с которой начинается выполнение программы. Зачем здесь скобки? Как уже упоминалось, они указывают на то, что **main( )** - имя функции. Дополнительные вопросы, относящиеся к функциям, будут обсуждаться ниже. Здесь мы только

повторим, что функции - это основные модули программы, написанной иа языке Си.

В круглых скобках в общем случае содержится информация, передаваемая этой функции. В нашем простом примере передача информации отсутствует и, следовательно, в скобках ничего не содержится. Заканчивая обсуждение данного вопроса, дадим вам одщ совет: при написании программы старайтесь не пропускать скобок.

Файл, содержащий программу, может иметь любое имя, правда, с тем ограничением, что оно должно удовлетворять системным соглашениям и оканчиваться символом **.c**. Например, вместо **main.c** мы могли бы выбрать имена **mighty.c** или **silly.c**.

/*простая программа*/:
Использование комментариев облегчает процесс понимания ва шей программы любым программистом (включая вас самих) Большим удобством при написании комментариев является воз можность располагать их на той же строке, что и операции, кото рые они объясняют. Длинный комментарий может помещаться на отдельной строке или даже занимать несколько строк. Все, что на ходится между символом, указывающим на начало комментария **/***, и символом, указывающим на его конец ***/**, игнорируется компилятором, поскольку он не в состоянии интерпретировав язык, отличающийся от Си.

{ и } :
Фигурные скобки **{ }** (и только они) отмечают начало и конец тела функции. Для этой цели не используются ни круглые **( )**, ни квадратные **[ ]** скобки. Фигурные скобки применяются также для того, чтобы объединить несколько операторов программы в сегмент или "блок". Если вы знакомы с такими языками, как Паскаль или Алгол, вы легко сообразите, что такие скобки аналогичны операторам **begin** и **end** в этих языках.

int num;:
"Оператор описания переменной" - одно из важнейших средств языка Си. Как уже упоминалось выше, в нашем простом пример вводятся два понятия. Первое - использование в теле функции "переменной", имеющей имя **num;** второе - с помощью слова **int** объявляется, что переменная **num** принимает целые значения. Точка с запятой в конце строки указывает на то, что в ней содержится оператор языка Си, причем этот символ является здесь частью оператора, а не разделителем операторов, как в Паскале.

Слово **int** служит "ключевым словом", определяющим один из основных типов данных языка Си. Ключевыми словами называют ся специальные зарезервированные слова, используемые для по строения фраз языка; список ключевых слов вы можете найти в приложении в конце книги.

В языке Си все переменные должны быть объявлены. Это означает, что, во-первых, в начале программы вы должны привести список всех используемых переменных, а во-вторых, необходимо указать "тип" каждой из них. Вообще объявление переменных считается "хорошим стилем" программирования.

Здесь вы можете задать три вопроса. Первый: каким образом надо выбирать имена? Второй: что такое типы данных? Третий: зачем вообще требуется объявлять переменные? Ответы на первый и третий вопросы приведены ниже и отмечены вертикальной линейкой голубого цвета.

Второй вопрос мы обсудим в гл. 3, а здесь сделаем краткое замечание. Язык Си имеет дело с некоторыми классами (или "типами") данных: целыми числами, символами и числами с плавающей точкой. Объявление переменной, имеющей целый или символьный тип позволяет компилятору размещать данные в памяти, осуществлять их выборку и интерпретировать нужным образом.

Выбор имени
Мы предполагаем, что вы используете осмысленные обозначения переменных. Имя переменной

может содержать от одного до восьми символов. (Фактически вы можете использовать и большее их число, но компилятор пропустит же символы, начиная с девятого. Поэтому имена **shakespeare** и **shakespencil** считались бы одинаковыми, поскольку первые восемь букв у них совпадают.) Для образования имени переменной разрешается использовать строчные и прописные буквы, цифры и символ подчеркивания, считающийся буквой. Первым символом должна быть обязательно буква.

Правильные имена	*Неправильные имена*
wiggly	$Z^** cat1
1cat Hot_Tub	Hot-Tub _kcaB
	don' t

В библиотечных процедурах часто используются имена, начинающиеся с символа подчеркивания: Это делается в предположении, что пользователи вряд ли выберут имена, начинающиеся с этого символа, поэтому маловероятно, что одно из них будет случайно выбрано для обозначения другого понятия. Старайтесь ив использовать имен, начинающихся с символа подчеркивания, и вам удастся избежать взаимопересечений с множеством библиотечных имен.

Четыре довода в пользу объявления переменных

1. Сведение всех операторов объявления переменных в начало программы облегчает понимание ее смысла. Это особенно справедливо, если вы даете переменным осмысленные имена (например, **taxrate** [налоговый тариф] вместо r) и, кроме того, включаете в программу комментарии для объяснения того, что обозначают переменные. Документирование программы подобным образом является одним из основных признаков хорошего стиля программирования.

2 . Размышление о том, что поместить в секцию объявления переменных, побуждает спланировать программу перед тем, как погрузиться в ее написание. Это эквивалентно получению ответов на вопросы: какая информация необходима программе при запуске? Какую выходную информацию хотелось бы получить?

3. Объявление переменных позволяет избежать одной из наиболее коварных и труднообнаруживаемых ошибок - неправильно написанных имен. Например, предположим, что программируя на некотором языке, вы использовали оператор
B0ZO = 32.4,
а дальше в программе вы ошибочно написали
ANS = 19.7* BOZO - 2.0
случайно заменив цифру **0** буквой **O**. Вследствие этого в программе появится новая переменная с именем **BOZO**, и будет использовано какое-то ее значение (возможно нуль или какой-то "мусор"). В результате переменная **ANS** получит неправильное значение, и вы, возможно, потратите много времени, пытаясь найти причину. Это не может произойти при программировании на языке Си (если только вы не объявили две переменные со столь похожими именами), поскольку компилятор сразу выдаст сообщение об ошибке, как только встретит в программе необъявленную переменную с именем **BOZO**.

4 . Любая программа, написанная на языке Си, не будет выполняться, если не описать все используемые переменные. Мы полагаем, что последний довод окажется решающим в том случае, если первые три вас не убедили.

num = 1; :
"Оператор присваивания" является одним из основных средств языка. Приведенную выше строку программы можно интерпретировать

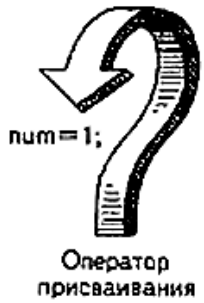


РИС. 2.2. Оператор присваивания - один из основных операторов.

так: "присвоить переменной **num** значение **1**". Дело в том, что, согласно оператору в четвертой строке программы, переменной **num** была выделена ячейка памяти, и только теперь в результате выполнения оператора присваивания переменная получает свое значение. При желании мы могли бы присвоить ей другое значение - вот почему имя **num** обозначает переменную. Отметим, что этот оператор тоже заканчивается точкой с запятой.

```
printf (" Я простая");
printf ("вычислительная машина. \n");
printf ("Мое любимое число %d, потому что оно самое первое.\n", num);
```

Во всех этих строках используется стандартная функция языка Си, называемая **printf( );** скобки указывают на то, что мы, конечно же, имеем дело с функцией. Строка символов, заключенная в скобки, является информацией, передаваемой функции **printf( )** из нашей главной функции **[main( )]**.

Такая информация называется "аргументом"; в первом случае аргументом является строка "Я простая". Возникает вопрос: что функция **printf( )** делает с этим аргументом? Ответ довольно очевиден: она просматривает все символы, содержащиеся между кавычками, и выводит их на экран терминала.

РИС. 2.3. Вид функции **printf ( )** и ее аргумента.

Данная строка дает нам пример того, как мы "вызываем" функцию или "обращаемся" к ней, программируя на языке Си. Для этого требуется только указать имя функции и заключить требуемый аргумент (или аргументы) в скобки. Когда при выполнении ваша программа "достигнет" этой строки, управление будет передано указанной функции [в данном случае **printf( )**]. Когда выполнение функции будет завершено, управление вернется обратно в исходную ("вызывающую") программу.

Что можно сказать по поводу следующей строки программы? В ней имеются символы **\n**, которые не появились на экране. В чем дело? Эти символы служат директивой начать новую строку на устройстве вывода. Комбинация **\n** на самом деле представляет собой один символ, называемый "новая строка". Его смысл кратко формулируется так: начать вывод новой строки с самой левой колонки. Другими словами, с помощью этого символа осуществляются те же функции, что и с помощью клавиши [ввод], имеющейся на обычном терминале. Но вы можете сказать, что комбинация **\n** выглядит, как два символа, а не как один. Вы, конечно же, правы, но просто по смыслу они представляют собой один символ, для которого не существует соответствующей клавиши на клавиатуре. Возникает вопрос: почему для этой цели нельзя использовать клавишу [ввод]? В ответ скажем, что это может быть интерпретировано как некоторая директива вашему текстовому редактору, а не как команда, которая должна быть помещена в память ЭВМ. Другими словами, когда вы нажимаете клавишу [ввод], редактор прекращает заполнение текущей строки, с которой вы в данный момент работаете, и начинает новую строку, оставляя старую неоконченной.

Символ "новая строка" служит одним из примеров того, что называется "управляющей последовательностью". Эта последовательность используется для представления символов, которые трудно или вообще невозможно вводить с обычной клавиатуры. Другими примерами служат **\t** для табуляции и **\b** для возврата на одну позицию. В любом случае управляющая последовательность начинается со знака ****. Мы вернемся к обсуждению этого вопроса в гл. 3.

Теперь, мы думаем, стало понятно, почему три оператора печати вывели на экран только две строки: аргумент первого оператора не содержал символа "новая строка".

Вид второй строки, появившейся на экране, может вызвать недоуменный вопрос: почему отсутствуют символы **%d**, имеющиеся в операторе вывода? Напомним, что напечатанная строка имела следующий вид:

```
Мое любимое число1, потому что оно самое первое.
```

Вы, наверное, уже догадались - при печати вместо символов **%d** было подставлено число **1**, являющееся значением переменной **num**. По-видимому, комбинация символов **%d** служит своего рода указателем места в строке, куда необходимо вставить значение переменной **num** при печати. На языке Бейсик аналогичный оператор печати выглядел бы следующим образом:

```
PRINT "Мое любимое число" ; num; "потому что оно самое первое".
```

На самом деле в языке Си обсуждаемый оператор позволяет сделать несколько больше. Символ **%** сигнализирует программе, что, начиная с этой позиции, необходимо напечатать число, а **d** указывает, что переменную необходимо печатать в десятичном формате. Функция **printf( )** предоставляет возможность выбора соответствующего формата для печати переменных. Буква **f** в имени функции **printf( )** фактически служит напоминанием, что это оператор печати с заданным форматом.

СТРУКТУРА ПРОСТОЙ ПРОГРАММЫ

Далее Содержание

Теперь, после того как мы привели конкретный пример, вы готовы к тому, чтобы познакомиться с несколькими общими правилами, касающимися программ, написанных на языке Си. Программа состоит из одной или более функций, причем какая-то из них обязательно должна называться **main( )**. Описание функции состоит из заголовка и тела. Заголовок в свою очередь состоит из директив препроцессора типа **#include** и т. д. и имени функции.

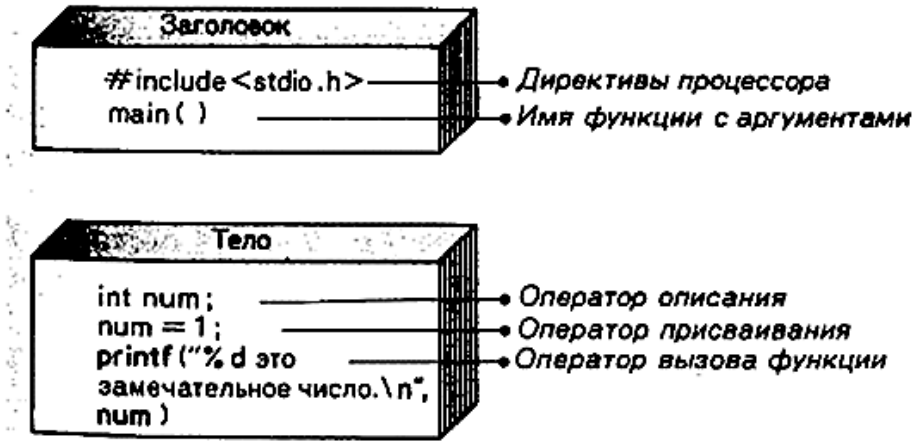


РИС. 2.4. Структура функции в языке Си: заголовок и тело.

Отличительным признаком имени функции служат круглые скобки, причем аргумент, вообще говоря, может отсутствовать. Тело функции заключено в фигурные скобки и представляет собой набор операторов, каждый из которых оканчивается символом "точка с запятой". В нашем примере тело состояло из оператора описания, в котором объявлялись имя и тип используемой переменной, оператора присваивания, с помощью которого переменная получила некоторое значение, и, наконец, трех операторов печати, каждый из которых представляет собой вызов функции **printf( )**.

НЕСКОЛЬКО СОВЕТОВ, КАК СДЕЛАТЬ ПРОГРАММУ ЧИТАЕМОЙ

Далее Содержание

Создание читаемой программы служит признаком хорошего стиля программирования. Это приводит к облегчению понимания смысла программы, поиска ошибок и в случае необходимости ее модификации. Действия, связанные с улучшением читаемости программы, кроме того, помогут более четко понять, что программа делает. На протяжении всего изложения мы будем пытаться указывать полезные приемы, способствующие достижению этой цели.

Мы уже упоминали о двух таких способах: выбор осмысленных обозначений для переменных и использование комментариев. Заметим, что эти два метода дополняют друг друга. Если вы дали переменной имя **width** (ширина), то необходимость в комментарии, сообщающем о том, что данная переменная определяет ширину, отпадает.

Еще один прием состоит в использовании пустых строк для того, чтобы отделить одну часть функции, соответствующую некоторому семантическому понятию, от другой. Например, в нашей простой программе одна пустая строка отделяет описательную часть от выполняемой (присваивание значения и вывод на печать). Синтаксические правила языка Си не требуют наличия пустой строки в данном месте, но поскольку это стало уже традицией, то и мы делаем также.

Четвертый принцип, которому мы следуем, заключается в том, чтобы помещать каждый оператор на отдельной строке. Опять же это только соглашение, которое никак не регламентируется правилами языка, так как Си имеет "свободный формат". Вы можете поместить несколько операторов на одной строке или распространить один оператор на несколько строк. Нижеследующий пример является абсолютно правильной программой:

```
main( )
{
 int four; four = 4;
printf(" %d \n", four);
}
```

Совершенно очевидно, что символ "точка с запятой" указывает компилятору, где кончается один оператор и начинается следующий, но логика программы окажется проще, если вы последуете соглашениям, приведенным выше. Поскольку в нашем примере запутанной логики нет, вид программы в данном случае не влияет на понимание ее смысла, но, по нашему мнению, лучше прививать хорошие привычки с самого начала.

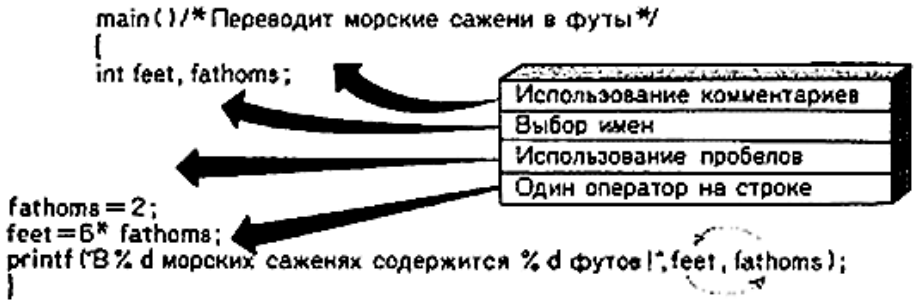


РИС. 2.5. Способы улучшения читаемости программы.

СЛЕДУЮЩИЙ ШАГ

Далее Содержание

Наша первая программа была довольно простой, и следующий пример будет ненамного сложнее. Он выглядит так:

```
main( )/* Переводит 2 морские сажени в футы*/ 1)
{
int feet, fathoms;
fathoms = 2;
feet = 6 *fathoms;
printf (" В %d морских саженях содержится %d футов!" , feet, fathoms);
}
```

Что здесь нового? Во-первых, мы описали две переменные вместо одной. Для этого потребовалось только разделить в операторе описания имена двух переменных запятой.

Во-вторых, мы выполнили вычисления - использовали громадную вычислительную мощность

нашего компьютера для умножения 2 на 6. В Си, так же как и во многих других языках, символ ***** обозначает умножение. Поэтому смысл оператора

feet = 6 *fathoms;

заключается в следующем: взять величину переменной **fathoms**, умножить ее на **6** и присвоить результат переменной **feet**. (Судя по этой парафразе, обычный английский язык менее лаконичен, чем простой язык Си; это одна из причин, лежащих в основе разработки языков программирования.)

И наконец, мы использовали функцию **printf( )** более сложным образом. Если вы выполните эту программу на компьютере, то результат должен выглядеть так:

Можно заметить, что было произведено две подстановки: первое вхождение символов **%d** в строку, заключенную в кавычки, было заменено значением первой переменной (**fathoms**) из списка, следующего за указанной строкой, а второе - значением второй переменной (**feet**) из этого же списка. Обратите внимание, что список печатаемых переменных расположен в конце оператора.

Область применения данной программы несколько ограниченна, но она может послужить прообразом программы перевода морских саженей в футы. Все, что нам потребуется - специальный способ присваивания произвольных значений переменной **feet**; о том, как это делается, вы узнаете несколько позже.

ДОПОЛНИТЕЛЬНЫЙ ПРИМЕР

Далее Содержание

Здесь мы приведем еще один пример. До сих пор в наших программах использовалась только стандартная функция **printf( )**. В данном разделе мы хотим продемонстрировать, как включить в программу[2)] и использовать функцию, которую вы сами написали.

```
main( )/* butler*/
{
     printf("Я вызываю функцию butler.\n");
     butler( );
     printf ("Да. Принесите мне чашку чая и гибкие диски.\n");
}
     butler( )
{
     printf("Вы вызывали, сэр?\n");
}
```

Результаты работы программы выглядят следующим образом:

```
Я вызываю функцию butler. Вы вызывали, сэр? Да. Принесите мне чашку чая и гибкие диски.
```

Функция **butler( )** определяется точно так же, как и функция **main();** ее тело заключено в фигурные скобки. Вызов функции осуществляется путем простого указания ее имени, включая круглые скобки. Мы вернемся к этому важному вопросу только в гл. 9, а здесь хотели продемонстрировать ту легкость, с которой вы можете включать в программу свои собственные функции.

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Ниже приведена краткая сводка строгих правил (но не чрезмерно жестких), которые, мы надеемся, вы усвоили. Мы включили сюда же краткие примеры.

Как назвать файл, содержащий вашу программу: **eye.c**, или **black.c**, или **infan.c** и т. п.
Какое имя можно использовать в качестве названия программы, состоящей из одной функции: **main()**.
Структура простой программы: заголовок, тело, фигурные скобки, операторы. Как описать целую переменную: **int varname;**
Как присвоить значение переменной: **varname = 1024**;
Как напечатать фразу **printf ("Хотите купить утку?");**
Как напечатать значение переменной: **printf ("%d", varname);**
Символ новая строка: **\n**
Как включать комментарии в программу: **/*анализ движения наличных денег*/**

ВОПРОСЫ И ОТВЕТЫ

Ниже приведено несколько вопросов, которые помогут вам проверить и расширить свое понимание материала данной главы.

Вопросы

1. Икабод Боуди Марфут (ИБМ)[3] подготовил программу, приведенную ниже, и принес ее вам для проверки. Пожалуйста, помогите ему найти в ней ошибки.

```
include studio, h main{} /*эта программа печатает число недель в году/*
(
int s

s: = 56;
print (В году s недель.);
```

2. Что будет напечатано в каждом из примеров, приведенных ниже, в предположении, что они являются частями некоторых полных программ?

а. **printf(" Б-э-э Б-э-э, Черная Овца.");**
а. **printf("У тебя есть шерсть?\n");**
б. **printf("Убирайся!\n Мешок сала!");**
в. **printf("Что?\n Нет/n Кларнет?\n");**
г. **int num;**
num = 2;
printf(" %d + %d = %d", num, num, num + num);

Ответы

1.
Строка 1:
данная строка должна начинаться с символа # правильное написание имени файла - **stdio.h;** имя файла должно быть заключено в угловые скобки.

Строка 2:
вместо фигурных скобок **{}** необходимо использовать круглые **( );**
комментарий должен оканчиваться символами ***/**, а не **/***

Строка 3:

вместо круглой скобки **(** должна стоять фигурная **{**.

Строка 4:
оператор должен оканчиваться символом "точка с запятой".

Строка 5:
эту строку (пустую) м-р ИБМ написал Совершенно правильно!

Строка 6:
в операторе присваивания необходимо использовать символ **=**, а не **:=**. (К сожалению, м-р ИБМ имеет представление о языке Паскаль.)
В году 52 недели, а не 56.

Строка 7:
оператор должен выглядеть так **printf ("В году %d недель. \n", s);**

Строка 8:
отсутствует, но она обязательно должна быть и содержать закрывающую фигурную скобку - **}**.

2. а. **Б-э-э Б-э-э. Черная Овца. У тебя есть шерсть?**
(Заметим, что пробел после точки отсутствует. Для того чтобы поместить в это место пробел, необходимо было вместо **"У тебя"** писать **" У тебя"**)
б. **Убирайся!**
Мешок сала!
(Отметим, что курсор теперь находится в конце второй строчки.)
в. **Что?**
Нет /n Кларнет?
Заметим, что символ **(/)** производит не тот же эффект, как символ **(\)**
г. **2 + 2 = 4**
(Отметим, что каждое вхождение комбинации символов **%d** в строку заменяется значением соответствующей переменной из списка. Заметим также, что символ **+** означает сложение и что таким образом вычисления могут быть проведены "внутри" оператора **printf( )**.)

УПРАЖНЕНИЯ

Чтобы изучить язык Си, одного только чтения недостаточно. Вы должны попробовать сами написать одну или две простые программы и посмотреть, пройдет ли все так же гладко, как это может показаться в результате чтения данной главы. Мы хотим предложить вам несколько идей, но, если желаете, вы можете воспользоваться своими собственными соображениями на этот счет.
1. Напишите программу, печатающую ваше имя.
2. Напишите программу, печатающую ваши имя и адрес, используя три или более строк.
3. Напишите программу, которая укажет ваш возраст, данный в годах, в днях. Не усложняйте ее, учитывая високосные и невисокосные годы.

[1)] 1 морская сажень = 1,83 м; 1 фут = 30,5 см. - *Прим. ред.*

[2)] В программе используется слово **butler** (англ.) - дворецкий. - *Прим. перев.*

[3)] Аббревиатура этого шутливого имени совпадает с аббревиатурой названия крупнейшей американской фирмы по выпуску вычислительной техники ИБМ (IBM). - *Прим. ред.*

3. Данные, язык Си и вы

int, short, long, unsigned, char, float, double, sizeof

Программы имеют дело с данными. Мы вводим в компьютер числа, буквы и слова и ожидаем, что он будет проводить над ними какие-то операции. В этой и следующей главах мы сосредоточим наше внимание на данных различных типов и их свойствах. В соответствии с этим мы будем последовательно останавливаться на каждом из типов и смотреть, как их можно использовать. Но, по скольку заниматься одним только обсуждением представляется нам не очень веселым делом, мы рассмотрим также небольшие про граммы обработки данных.

Эта глава в основном посвящена обсуждению двух важнейших классов типов данных целым числам и числам с плавающей точкой. Язык Си предоставляет программисту возможность использовать несколько разновидностей этих типов Мы займемся изучением следующих вопросов что такое типы данных, как их описать, как и когда их использовать. Кроме того, мы обсудим различия между константами и переменными.

Теперь, так же как прежде, пришло время рассмотреть какую нибудь простую программу. Возможно, вы обнаружите в ней какие-то непонятные места Мы постараемся разъяснить их вам при последующем обсуждении в данной главе Общий смысл про граммы должен быть достаточно понятен, поэтому попробуйте осуществить компиляцию и выполнить эту программу[1)] . Для экономии времени можете опустить комментарии при вводе программы в машину. (Замечание: мы включили имя программы в ее со став как комментарий, в дальнейшем, приводя программы, будем придерживаться этого правила).

```
/*Ваш золотой эквивалент*/
/*определение стоимости количества золота, равного вашему весу*/
main( )
{
    float weight, value, /* 2 переменные с плавающей точкой */
    char beep;           /* символьная переменная */
    beep = ' /007' ;     /* присваивание специального символа  переменной beep */
    printf(" Стоите ли вы своего веса в золотом эквиваленте? \n");
    printf(" Укажите, пожалуйста, свой вес в фунтах и узнаете \n");
    scanf("%f ",  &weight;);  /* получение данных */
    value= 400.0 *weight*14,5833; /* предполагаемая цена золота -
                                  400 долл за тройскую унцию */
     /* коэффициент 14,5833 служит для перевода в тройские унции */
    printf(" %c Стоимость вашего веса в золотом эквиваленте
                              $%2,2 f%c.\n",  beep, value, beep);
    printf("Вы несомненно стоите столько' Если цена золота упадет,");
    printf(" ешьте больше, \n чтобы сохранить свою стоимость \n");
}
```

При вводе этой программы в компьютер вы можете захотеть заменить число **400.00** величиной текущей цены золота. Мы надеемся, однако, что вы не будете пускаться на обман, заменяя число **14,5 833**, равное числу унций в фунте (Речь идет об унциях в тройской системе мер, применяемой при взвешивании благородных металлов, и фунтах в обычной системе мер, используемой при взвешивании всего остального). Заметим, что слова "укажите свой вес" означают, что необходимо набрать на клавиатуре число, выражающее вес, и нажать клавишу "ввод" или "возврат" (Не надо только вставать на клавиатуру!) Нажав эту клавишу, вы тем самым сообщаете компьютеру, что вы уже закончили ввод.

Данная программа имеет также некоторую незаметную на первый взгляд особенность. Чтобы обнаружить ее и понять, в чем дело, вы должны сами запустить эту программу; имя одной из переменных служит довольно недвусмысленным намеком.
Что нового содержится в этой программе?

1. По-видимому, вы уже заметили, что мы ввели описание двух новых типов переменных. До этого мы использовали только целые переменные, а теперь добавили переменные с плавающей точкой и символьные переменные, так что теперь мы можем обрабатывать данные более общего вида.

2. Мы использовали в программе несколько новых способов задания констант. Теперь мы умеем вводить в программу числа с десятичной точкой и знакомы с довольно специфическим способом представления значения символьной переменной **beep**.

3. Для вывода на печать этих переменных нового типа в операторе **printf( )** мы использовали спецификации **%f** и **%c**, соответствующие переменной с плавающей точкой и символьной переменной. Модификаторы в спецификации **%f** применяются для улучшения вида результата на экране дисплея.

4. Возможно, самой существенной новой особенностью является то, что эта программа "диалоговая". Компьютер запрашивает У вас информацию, а затем использует число, которое вы ввели. Работать с диалоговой программой гораздо интереснее, чем с программами недиалогового типа, которые мы использовали до этого. Более важным является еще и то, что такой подход позволяет нам создавать более гибкие программы. Например, нашу программу можно использовать при задании любого веса (конечно, в разумных пределах), а не только веса в 175 фунтов. Нам не нужно переписывать программу всякий раз, когда мы захотим обработать вес еще одного человека. Функции **scanf( )** и **printf( )** делают это вполне возможным. Функция **scanf( )** читает данные, набираемые на пульте дисплея, и вводит их в программу. В гл. 2 мы уже видели, что функция **printf( )** читает данные из программы и выводит их на экран. Вместе эти две функции позволяют установить двустороннюю связь с вашей программой, что делает общение с компьютером гораздо более приятным.

В данной главе мы рассмотрим два первых пункта - переменные и константы различных типов данных. Оставшиеся два пункта мы обсудим в следующей главе, но функции **scanf( )** и **printf( )** будем использовать по-прежнему.

ДАННЫЕ: ПЕРЕМЕННЫЕ И КОНСТАНТЫ

Далее Содержание

Компьютер, выполняя программу, может заниматься разнообразной деятельностью. Он может складывать числа, сортировать имена, заниматься распознаванием речи и изображения на экране видеодисплея, вычислять орбиты комет, подготавливать список почтовых адресов абонентов, чертить фигуры, делать логические выводы или что-нибудь еще, что только вы можете себе представить. Чтобы заниматься всем этим, программам необходимо работать с "данными" - числами и символами, т. е. объектами, которые несут в себе информацию, предназначенную для использования. Некоторые данные устанавливаются равными определенным значениям еще до того, как программа начнет выполняться, а после ее запуска сохраняют такие значения неизменными на всем протяжении работы программы. Это "константы". Другие данные могут изменяться, или же им могут быть присвоены значения во время выполнения программы; они называются "переменными". (Мы уже использовали данный термин в предыдущей главе, но формально вы зна-комитесь с ним только здесь.) В нашей простой программе **weight** - это переменная; число **16.0** - константа. Что можно скаэать по поводу числа **400.00**? Совершенно очевидно, что в действительности цена золота не остается неизменной, но в нашей программе мы считаем ее константой.

Различие между переменной и константой довольно очевидно: во время выполнения программы значение переменной может быть изменено (например, с помощью присваивания), а значение константы - нет. Указанное различие приводит к тому, что обработка переменных компьютером оказывается немного сложнее и требует больше времени, чем обработка констант, но, несмотря на это, он вполне справляется с такой деятельностью.

ДАННЫЕ: ТИПЫ ДАННЫХ

Далее Содержание

Помимо различия между переменными и константами существует еще различие между типами данных. Некоторые данные в программе являются числами, некоторые - буквами, или, более обобщенно, символами. Компьютер должен иметь возможность идентифицировать и обрабатывать требуемым образом данные любого из этих типов. В языке Си предусмотрено использование нескольких основных типов данных. Если величина есть константа, то. компилятор обычно может распознать ее тип только по тому виду, в каком она присутствует в программе. Однако в случае переменной необходимо, чтобы ее тип был объявлен в операторе описания.

Дополнительные детали, относящиеся к типам данных, мы будем сообщать вам по мере изложения. Рассмотрим основные типы данных, имеющиеся в языке Си. В стандарте языка Си используется семь ключевых слов, указывающих на различные типы данных. Приведем список этих ключевых слов:

```
int long short unsigned

char

float double
```

Первые четыре ключевых слова используются для представления целых, т. е. целых чисел без десятичной дробной части. Они могут появляться в программе по отдельности или в некоторых сочетаниях, как, например, **unsigned short**. Следующее слово **char** предназначено для указания на буквы и некоторые другие символы, такие, как #, **$**, **%** и **&**. Последние два ключевых слова используются для представления чисел с десятичной точкой. Типы, обозначаемые этими ключевыми словами, можно разделить на два класса по принципу размещения в памяти машины. Первые пять ключевых слов определяют "целые" типы данных, в то время как последние два - типы данных с "плавающей точкой".

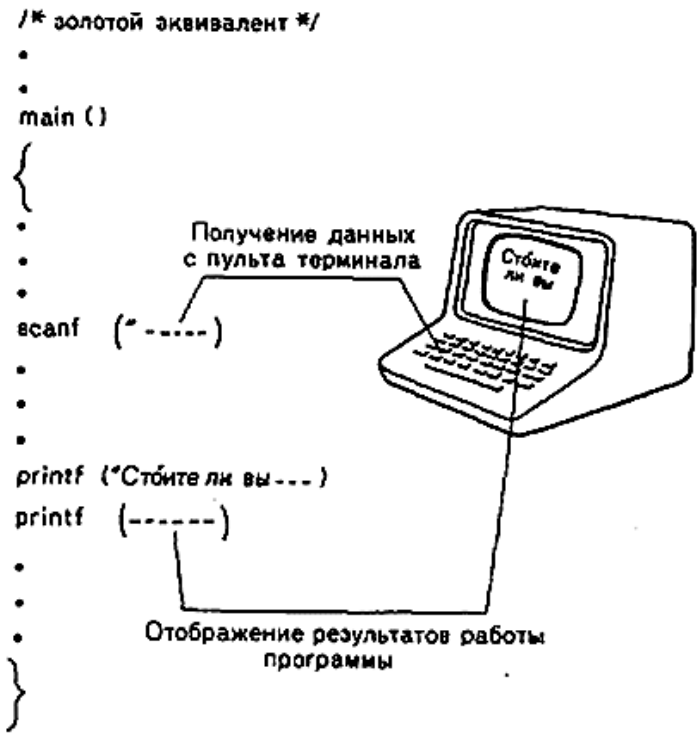


РИС. 3.1. Работа функций **scanf( )** и **printf( )**.

В этом месте у некоторых читателей могут появиться недоумен-ные вопросы: "Целые типы данных? Типы данных с плавающей теневой?" Не пугайтесь. Если эти термины кажутся вас непривычными или непонятными, мы дадим краткое объяснение их смысла. Те, кто не знаком с

терминами "биты", "байты" и "слова", могут, перед тем как двинуться дальше, прочесть приводимое ниже объяснение, отмеченное вертикальной голубой линией. Можно спросить: а нужно ли знать все эти детали? На самом деле необязательно. Пожалуй, не больше, чем вы должны знать о принципах работы двигателя внутреннего сгорания для того, чтобы управлять машиной. Но некоторое представление о том, что происходит в компьютере или двигателе, может иногда выручить вас. Кроме того, это может помочь вашему превращению в прекрасного "собеседника".

Термины "бит", "байт" и "слово" обычно используются для описания как элементов данных, которые обрабатывает компьютер, так и элементов памяти. Здесь мы займемся рассмотрением второго смысла этих терминов.

Наименьшая единица памяти называется бит. Она может принимать одно из ДВУХ значений: 0 или 1. (Иначе говоря, бит может находится в состояниях "включен" или "выключен"; эта фраза совершенно аналогична первому выска зыванию.) В один бит нельзя поместить достаточное количество информации но в машине содержится большое число битов; дело в том, что бит - основной "строительный блок", из которых создается память компьютера.

Байт - более удобный элемент памяти. В большинстве машин байт состоит из 8 бит. Поскольку каждый бит можно установить либо в состояние 0, либо в состояние 1, всего в байтовом формате можно представить 256 (два в восьмой степени) различных комбинаций из нулей и единиц. Такие комбинации можно использовать, например, для представления целых чисел в диапазоне от 0 до 255 или для кодирования набора символов. Это можно получить при помощи "дво-ичного кода", в котором для представления чисел используются только нули и единицы. Обсуждение структуры двоичного кода мы поместили в приложение (вы вполне можете его не читать, если не захотите).

При современном подходе к проектированию компьютеров слово является самым естественным элементом памяти. В 8-разрядных микрокомпьютерах, таких, как ЭВМ фирмы Sinklair иди первые модели машин фирмы Apple, слово занимает как раз 1 байт. Многие более новые персональные вычислительные системы, такие, как IBM PC и Lisa фирмы Apple, являются 16-разрядными. Это означает, что размер слова у них 16 бит, т. е. 2 байта. Большие компьютеры могут иметь 32-, 64-разрядные слова или даже более длинные. Совершенно очевидно, что чем длиннее слово, тем больше информации можно туда поместить. Обычно в компьютерах предусмотрена возможность объединять вместе два или более слов для того, чтобы помещать в память элементы данных большей длины, но этот процесс сильно замедляет работу компьютера.

В наших примерах мы предполагаем, что длина слова равна 16 бит, если мы не оговорили противного.

Для человека различие между целым числом и числом с плавающей точкой выражается в способе записи. Для компьютера различие выражается в способе занесения этих чисел в память. Давайте рассмотрим по очереди каждый из двух классов чисел.

Целые числа

Далее Содержание

У целого числа никогда не бывает дробной части и, согласно правилам языка Си, десятичная точка в его записи всегда отсутствует. В качестве примера можно привести числа **2**, **-23** и **2456**. Числа вида **3.14** и **2/3** не являются целыми. Представив целое число в двоичном виде, его нетрудно разместить в памяти машины.

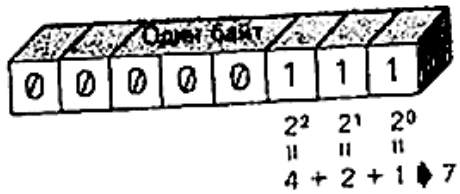


РИС. 3.2. Двоичное представление числа **7** в памяти машины.

Например, число **7** в двоичном виде выглядит как **111**. Поэтому, чтобы поместить это число в 1-байт слово, необходимо первые 5 бит установить в **0**, а последние 3 бит - в **1** (рис. 3.2).

Числа с плавающей точкой

Далее Содержание

Числа с плавающей точкой более или менее соответствуют тому, что математики называют "вещественными числами". Они включают в себя числа, расположенные между целыми. Вот некоторые из них: **2.75**, **3.16E7**, **7.00** и **2e-8**. Очевидно, что любое число с плавающей точкой можно записать несколькими способами. Более полное обсуждение "E-нотации" будет проведено дальше, а мы Только кратко поясним, что запись вида **"3.16E7"** означает число, полученное в результате умножения 3.16 на 1,0 в седьмой степени, т. е. на 1 с семью нулями. Число 7 называется "порядком" (показателем степени при основании 10).

Наиболее существенным моментом здесь является то, что способ кодирования, используемый для помещения в память числа с плавающей точкой, полностью отличается от аналогичной схемы для размещения целого числа. Формирование представления числа с плавающей точкой состоит в его разбиении на дробную часть и порядок; затем обе части раздельно помещаются в память. Поэтому число 7.00 из вышеприведенного списка нельзя поместить в память тем же способом, что и целое число 7, хотя оба имеют одно и то же значение. В десятичной записи (точно так же как и в двоичной) число "7.0" можно было бы записать в виде "0.7E1"; тогда "0.7" будет дробной частью, а "1" - порядком. Для размещения чисел в памяти машины будут, конечно, использоваться двоичные числа и степени двойки вместо степеней десяти. Дополнительную информацию, относящуюся к этому вопросу, вы сможете найти в приложении Ж. Здесь же мы остановимся лишь на различиях, связанных с практическим использованием чисел этих двух типов.

1. Целые числа не имеют дробной части, в то время как числа с плавающей точкой могут представлять как целые, так и дробные числа.

2. Числа с плавающей точкой дают возможность представлять величины из более широкого диапазона, чем целые (см. табл. 3.1).

3. При некоторых арифметических операциях, например при вычитании одного большого числа из другого, использование чисел с плавающей точкой приводит к большей потере точности.

4. Операции над числами с плавающей точкой выполняются, как правило, медленнее, чем операции над целыми числами. Однако сейчас уже появились микропроцессоры, специально ориентированные на обработку чисел с плавающей точкой, и в них эти операции выполняются довольно быстро.

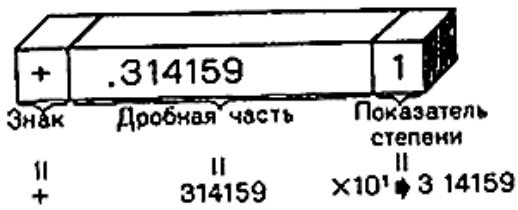


РИС. 3.3. Десятичное представление числа p в формате с плавающей точкой.

Возьмите некоторое число. Добавьте к нему 1, а затем вычтите из полученной суммы исходное число. Что у вас получится? У нас получилась 1. Но вычисления, производимые над числами с плавающей точкой, могут дать и совершенно неожиданный результат:

```
/*ошибка вычислений*/
main( )
{
   float a, b;
   b = 2.0e20 + 1.0;
   a = b - 2.0e20;
   printf(" %f \n", a);
}
```

Результат равен

```
0000000
```

Причина появления такого странного результата состоит в отсутствии доста точного числа разрядов для выполнения операций с требуемой точностью. Число **2.0e20** записывается как двойка с последующими двадцатью нулями, и, до бавляя к нему **1**, мы пытаемся изменить 21-ю цифру Чтобы выполнить эту опе рацию корректно, программа должна иметь возможность поместить в память число, состоящее из **21** цифры. Но число типа **float** (т е. с плавающей точкой) путем изменения порядка можно увеличить или уменьшить лишь на **6** или **7** цифр. Попытка вычисления оказалась неудачной. С другой стороны, если бы мы использовали, скажем, число **2.0e4** вместо **2.0e20**, мы смогли бы получить правильный ответ, поскольку в этом случае мы пытались бы изменить 5-ю цифру, и точность представления чисел типа **float** оказалась бы вполне достаточной для этого.

ТИПЫ ДАННЫХ В ЯЗЫКЕ СИ

Далее Содержание

Давайте теперь рассмотрим некоторые специфические особенности основных типов данных, используемых в языке Си. Для каждого типа мы покажем, как описать переменную, как представить константу и как лучше всего использовать данные этого типа. В некоторых компиляторах с языка Си не реализована обработка всех типов данных; поэтому вам необходимо свериться с руководством по языку Си, имеющимся в комплекте вашей машины, чтобы посмотреть, какие из типов доступны для использования.

Типы int, short и long

Далее Содержание

В языке Си имеется несколько целых типов, поэтому у вас есть возможность вносить изменения в свою программу, чтобы она удовлетворяла требованиям конкретной машины или определенного задания. Если вы не хотите заботиться о таких деталях, то, вообще говоря, вы можете просто остановиться на типе **int** и не думать больше о других возможностях.

Все данные типов **int**, **short** и **long** являются "числами со знаком", т. е. допустимыми значениями переменных этих типов могут быть только целые числа - положительные, отрицательные и нуль. Один бит используется для указания знака числа, поэтому максимальное число со знаком, которое можно представить в слове, меньше, чем максимальное число без знака. Например, в формате 16-битного слова можно представить любые целые числа без знака, из диапазона от 0 до 65535. Точно так же 16-битное слово можно использовать для представления целых чисел со знаком из диапазона от -32768 до +32767.

Заметим, что длины диапазонов в обоих случаях одинаковые.

Язык Си предоставляет пользователям возможность выбора размера элемента памяти (одного из трех) для представления це лых чисел. Типу **int** обычно соответствует стандартная длина слова, принятая на используемой машине. При этом гарантируется, что размер элементов памяти, отводимых под данные типа **short** и **long**, будет соответственно не больше и не меньше длины элемента памяти, выделяемого типу **int**. В некоторых вычислительных системах один или оба этих типа реализованы точно так же, как **int**. Все зависит от того, какое представление лучше соответствует архитектуре конкретной ЭВМ. В табл. 3.1 для каждого компьютера из некоторого множества приведено число битов, используемое для представления данных различных типов, а также диапазоны отображаемых чисел.

Описание данных целого типа

Далее Содержание

При описании данных необходимо ввести только тип, за которым должен следовать список имен переменных. Ниже приведены некоторые возможные примеры описаний:

```
int erns;
short stops;
long johns;
int hogs, cows, goats;
```

В качестве разделителя между именами переменных необходимо использовать запятую; весь список должен оканчиваться символом "точка с запятой". Вы можете собрать в один оператор описания переменных с одним и тем же типом или, наоборот, разбить одно описание на несколько операторов. Например, описание

```
int erns, hogs, cows, goats;
```

будет давать тот же самый эффект, что и два отдельных описания типа **int** в предшествующем примере. При желании вы даже могли бы использовать четыре различных описания данных типа **int** - по одному для каждой переменной. Иногда вам могут встретиться сочетания ключевых слов, как, например, **long int** или **short int**. Эти комбинации являются просто более длинной записью ключевых слов **long** и **short**.

Целые константы

Далее Содержание

Согласно правилам языка Си, число без десятичной точки и без показателя степени рассматривается как целое. Поэтому **22** и **-273** - целые константы. Но число **22.0** нецелое, потому что в его записи имеется десятичная точка, и число **22E3** тоже нецелое, поскольку в записи использован порядок. Кроме того, указывая целое число, нельзя использовать запятые. Нужно записать 23456 вместо 23,456.

Если вы хотите ввести некоторую константу типа **long**, то можете это сделать, указав признак **L** или **l** в конце числа. Использование прописной буквы **L** более предпочтительно, поскольку ее труднее спутать с цифрой **1**. Примером такой константы служит число **212L**. Очевидно, что само по себе число **212** не очень большое, но добавление признака **L** гарантирует, что в памяти для него будет отведено необходимое число байтов. Это может оказаться важным для достижения совместимости, если данное число должно использоваться вместе с другими переменными и константами типа **long**.

Вполне возможно, что вам уже ничего больше не нужно знать про то, как записывают константы, но в языке Си имеются еще и два других способа.

Первый: если целое начинается с цифры **0**, оно интерпретируется как "восьмеричное" число. Восьмеричные числа - это числа, представляемые "по основанию восемь" (т. е. их запись состоит из комбинаций степеней числа восемь). Например, **020** - это удвоенная первая степень основания восемь, т. е. восьмеричный эквивалент числа **16**. При отсутствии в первой позиции нуля это просто обыкновенное (десятичное) число **20**.

Второй: целое, начинающееся с символом **0x** или **0X** интерпретируется как шестнадцатеричное число, т. е. число, записываемое по основанию **16**. Поэтому запись **0x20** представляет собой удвоенную первую степень числа **16**, или **32**.

Восьмеричные и шестнадцатеричные числа чрезвычайно популярны среди программистов. Поскольку **8** и **16** являются степенями числа **2**, а **10** - нет, использование этих систем счисления при работе на машине является вполне естественным. Например, число 65536, которое часто возникает при программировании на 16-разрядных компьютерах, в шестнадцатеричной записи имеет вид 10000. Если вы захотите больше узнать о восьмеричных и шестнадцатеричных числах, вы сможете найти дополнительный материал в приложении Ж.

Инициализация переменных целого типа

Далее Содержание

Константы часто применяются при "инициализации" перемен ных. Это означает присваивание переменной некоторого значения перед началом обработки. Ниже приводятся примеры использова ния инициализации:

```
erns = 1024;
stops = -3;
johns = 12345678;
```

Если захотите, вы можете инициализировать переменную в операторе описания. Например:

```
int hogs = 23;
int cows = 32, goats = 14;
short dogs, cats = 92;
```

Заметим, что в последней строке была инициализирована только переменная **cats**. При невнимательном чтении может создаться впечатление, что переменная **dogs** тоже

инициализирована значением **92**, поэтому лучше избегать смешивания инициализируемых и неинициализируемых переменных в одном операторе описания.

Рекомендации

Далее Содержание

Какие переменные целого типа со знаком лучше всего использо вать? Одной из целей введения в язык трех классов целых чисел, имеющих различные размеры, было предоставить возможность со гласования типа переменной с требованиями задачи. Например, если переменная типа **int** занимает одно слово, а переменная типа **long** - два, то тип **long** позволяет обрабатывать большие числа. Если в вашей задаче такие большие числа не используются, то незачем и вводить в программу переменную типа **long**, так как, если вместо числа, занимающего одно слово памяти, используется число, занимающее два слова, работа машины замедляется. Вообще говоря, необходимость введения данных типа **long** целиком зависит от вашей вычислительной системы, поскольку под данные типа **int** на одной машине может отводиться больше памяти, чем под данные типа **long** на другой. В конце мы еще раз хотим напомнить вам, что обычно вполне достаточно использовать переменную типа **int**.

Что происходит, когда в процессе обработки данных появляется значение, лежашее вне того диапазона чисел, которому соответствует данный целый тип? Давайте присвоим некоторой переменной целого типа наибольшее возможное значение, выполним операцию сложения и посмотрим, что произойдет

```
/* переполнение */
main( )
{
   int i = 32767,
   printf( %d  %d  %d\n , i, i+l, i+2),
}
```

Ниже приведен результат работы этой программы, выполненной на нашей вычислительной системе

```
3
```

Целая переменная **i** ведет себя здесь как одометр[2)] в машине. Когда его показания достигают максимума, данная величина "сбрасывается", и все начинается сначала. Основное отличие состоит в том, что показания одометра растут, начиная с нуля, а значения нашей переменной типа **int** - с величины - 32768.

Заметим, что при этом вам не сообщают, что переменная **i** превысила макси мальное значение Для регистрации подобных событий вы должны использовать свои программные средства.

Описанный подход не вытекает непосредственно из правил языка Си, а является довольно распространенным способом реализации.

Тип данных unsigned

Далее Содержание

Обычно данный тип служит модификатором одного из трех ранее описанных типов. Поэтому мы можем использовать комбинация ключевых слов **unsigned int** или **unsigned long** как обозначения типов. Для указания типа **unsigned int** достаточно привести только ключевое слово **unsigned**. Некоторые вычислительные системы никак не обеспечивают аппаратную реализацию типа **unsigned long;** кроме того, существуют модели микропроцессоров в которых **unsigned** - специальный тип фиксированного размера.

Целые беззнаковые константы записываются точно так же, как и обычные целые константы, с тем лишь исключением, что использование знака - запрещено.

Целые переменные без знака описываются и инициализируются совершенно аналогично тому, как это делается в случае обычных целых переменных. Ниже приведено несколько примеров:

```
unsigned int students;
unsigned players;
unsigned short ribs = 6;
```

Применение данного типа при введении в программу некоторой переменной гарантирует, что она никогда не станет отрицательной. Кроме того, если вы имеете дело только с положительными числами, вы сможете воспользоваться тем, что данные указанного типа могут принимать большие значения, чем данные эквивалентного типа со знаком. Обычно это применяется при адресации памяти и организации счетчиков.

Тип данных char

Далее Содержание

Этот тип определяет целые числа без знака в диапазоне от 0 до 255. Обычно такое целое размещается в одном байте памяти. В машине используется некоторый код для перевода чисел в символы и обратно. В большинстве компьютеров это код ASCII, описанный в приложении в конце книги. Во многих компьютерах фирмы IBM (но не IBM PC) применяется другой код, называемый EBCDIC. На протяжении всей книги мы будем использовать код ASCII и, проведя различные примеры, будем ссылаться на него.

Описание символьных переменных

Далее Содержание

Для описания символьной переменной применяется ключевое слово **char**. Правила, касающиеся описания более чем одной переменной и инициализации переменных, остаются теми же, что и для других основных типов. Поэтому строки, приведенные ниже, являются допустимыми операторами.

```
char response;
char intable, latan;
char isma = ' S ';
```

Символьные константы

Далее Содержание

В языке Си символы заключаются в апострофы. Поэтому, когда мы присваиваем какое-то значение переменной **broiled** типа **char**, мы должны писать

```
broiled = ' T ';  /* ПРАВИЛЬНО */,
```

а не

```
broiled = T;  /* НЕПРАВИЛЬНО */
```

Если апострофы опущены, компилятор "считает", что мы используем переменную с именем **T**, которую забыли описать.

В стандарте языка Си принято правило, согласно которому значениями переменной или константы типа **char** могут быть только одиночные символы. В соответствии с этим последовательность операторов, указанная ниже, является недопустимой, поскольку там делается попытка присвоить переменной **bovine** значение, состоящее из двух символов:

```
ehar bovine;
bovine = ' ox ';  /*НЕПРАВИЛЬНО */
```

Если вы посмотрите на таблицу кода ASCII, то увидите, что некоторые из "символов" в ней не выводятся на печать. Например, при использовании в программе символа номер 7 терминал компьютера издает звуковой сигнал. Но как использовать символ, который невозможно набрать на клавиатуре? В языке Си для этого имеются два способа.

В *первом способе* используется сам код ASCII. Вы должны только указать номер символа вместе с предшествующим знаком "обратная косая черта". Мы уже делали это в нашей программе "золотой эквивалент". Вот эта строка

```
beep = ' \007 ';
```

Здесь имеются два важных момента, которые вы должны отчетливо представлять себе. Первый - это то, что последовательность знаков заключается в апострофы точно так же, как это делается с обычным символом. Второе - то, что номер символа должен быть записан в восьмеричном виде. При записи последовательности знаков мы можем случайно пропустить нули в первых позициях; в этом случае для представления кода "сигнал" мы могли бы использовать **'\07'** или даже **'\7'**. Но ни в коем случае не опускайте в записи последние нули! Последовательность символов **'\020'** можно записать в виде **'\20'**, но не **'\02'**.

При использовании кода ASCII необходимо отметить различие между числами и символами, обозначающими числа. Например, символу *"4"* соответствует код ASCII, равный 52. Это символ *"4"* а не число 4.

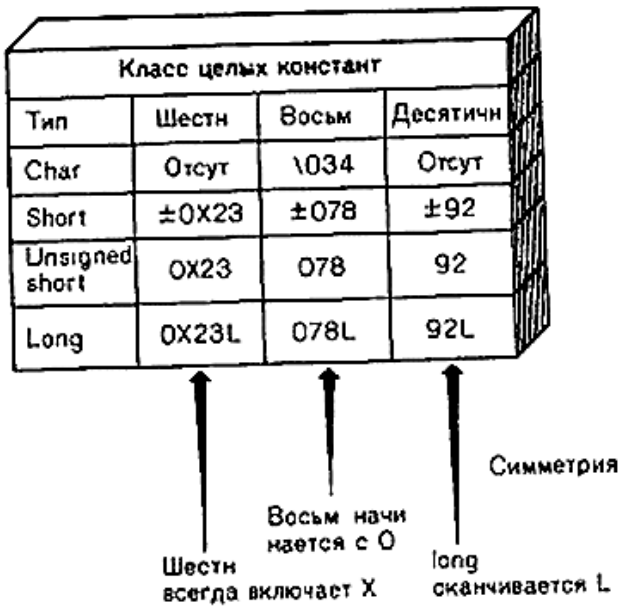


РИС. 3. 4. Формы записи констант целых типов

Во *втором способе* представления "неудобных" знаков используются специальные последовательности символов. Они называются управляющими последовательностями и выглядят следующим образом:

\n новая строка
\t табуляция
\b шаг назад
\r возврат каретки
\f подача бланка
\\ обратная косая черта (\)
\' апостроф (')

\" кавычки (")

При присваивании символьной переменной эти последовательно сти тоже должны быть заключены в апострофы. Например, мы могли бы написать оператор

```
nerf = ' \n ';
```

а затем вывести на печать переменную **nerf;** это приведет к продви жению на одну строку вперед на печатающем устройстве или на экране дисплея.

Первые пять управляющих последовательностей являются общепринятыми символами, предназначенными для управления работой печатающего устройства: символ "новая строка" вызывает пе реход к новой строке; символ "табуляция" сдвигает курсор или печатаюшую головку на некоторое фиксированное число позиций 5 или 8; символ "шаг назад" производит сдвиг назад на одну позицию; символ "возврат каретки" осуществляет возврат к началу строки; символ "подача бланка" вызывает протяжку бумаги на одну страницу. В последних трех управляющих последовательностях символы ****, **'**, **"** можно считать символьными константами [поскольку они служат для *определения* символьных констант и непосредственно используются в операторе **printf( )**, применение их самих в качестве символов могло бы привести к ошибке]. Если вы хотите вывести на печать строку.

```
Заломните, " символ \ называется  обратная косая черта".
```

оператор будет выглядеть так:

```
printf(" Запомните, \" символ \\ называется обратная косая черта. \" \n");
```

Здесь у вас могут возникнуть два вопроса. Во-первых, почему мы не заключили управляющие последовательности в апострофы? Во-вторых, в каких случаях необходимо использовать код ASCII и когда управляющие последовательности, которые мы только что обсуждали? (Мы надеемся, что у вас возникли как раз эти вопросы, ПОТОМУ что мы собираемся отвечать именно на них.)

1. Когда символ является частью строки символов, заключенной в кавычки, он входит туда без апострофов независимо от того, является ли он управляющим или нет. Заметим, что все остальные символы в нашем примере (3,а, п, о, м, н, и т. д.) тоже присутствуют в этой строке без кавычек. Строка символов, заключенная в кавычки, называется символьной строкой или цепочкой. Мы обсудим этот вопрос в следующей главе.

2. Если у вас есть возможность выбора одной из двух форм записи некоторой специальной управляющей последовательности, скажем **'\f'**, или эквивалентного кода из таблицы кодов ASCII - **'\016'**, то рекомендуем использовать **'\f'**. Во-первых, это более наглядно. Во-вторых, лучше согласуется с требованием переносимости программ, поскольку даже в том случае, когда в системе не используется код ASCII, обозначение **'\f'** будет продолжать "работать".

Программа

Далее Содержание

Ниже приводится короткая программа, позволяющая узнавать номер кода символа даже в том случае, если на вашей машине не используется код ASCII.

```
main( )   /* определяет номер кода символа */
{
   char ch;
   printf(" Введите, пожалуйста, символ .\n");
   scanf(" %c", &ch);   /* ввод пользователем символа */
   printf("Код символа %c равен %d.\n", ch, ch);
}
```

При работе с этой программой не забывайте нажимать клавишу [ввод] или [возврат] после ввода символа. Затем функция **scanf( )** прочтет введенный символ; знак амперсанд **(&)** указывает, что символ должен быть присвоен переменной **ch**. Функция **printf( )** выводит на печать величину **ch** дважды: первый раз как символ (в соответствии со спецификацией **%c**), а затем как десятичное целое число (в соответствии со спецификацией **%d**).

Типы данных float и double

Далее Содержание

В большинстве проектов разработки программного обеспечения оказывается вполне достаточным использовать данные целых типов. Однако в программах вычислительного характера часто применяются числа с плавающей точкой. В языке Си такие данные описываются типом **float;** они соответствуют типу **real** в Фортране и Паскале. Указанный подход, как вы могли заметить при вни мательном чтении, позволяет представлять числа из гораздо более широкого диапазона, включая и десятичные дроби. Числа с плавающей точкой совершенно аналогичны числам в обычной алгебраической записи, используемой при работе с очень большими или малыми числами. Давайте рассмотрим ее подробнее.

Алгебраическая запись числа представляет собой произведение некоторого десятичного числа на степень, основание которой равно десяти. Ниже приведено несколько примеров.

Число	*Алгебраическая запись для ввода запись в машину*
1 000000000	= 1.0 ґ 10^{9} = 1.0e9
123000	= 1.23 ґ 10^{5} = 1.23e5
322.56	= 3.2256 ґ 10^{2} = 3.2256e2
0.000056	= 5.6 ґ 10^{-5} = 5.6e-5

В первом столбце числа изображены в обычной записи, во втором приведена соответствующая алгебраическая запись, а в третьем столбце числа показаны в том виде, в котором они обычно представляются при вводе в машину и при выводе из нее - с символом **e**, за которым следует показатель степени по основанию десять (порядок).

Обычно для размещения в памяти числа с плавающей точкой отводится 32 бита - 8 бит для представления порядка и знака и 24 бита - для мантиссы (т. е. коэффициента при степени десяти). Важным фактом, который вам необходимо знать, является то, что такой способ дает возможность представлять числа с точностью до 6-7 десятичных цифр в диапазоне ±(10^{-37} - 10^{38}). Это может оказаться удобным, если вам понадобится обрабатывать числа того же порядка, что масса Солнца (2.0e30 кг) или заряд протона (1.6e-19 Кл). (Многим нравится использовать подобные числа.)

Во многих ЭВМ предусматривается обработка данных типа **double** (вычислений с двойной точностью), когда для представления чисел используется удвоенное число битов, чаще всего 64. В некоторых машинах все 32 добавочных бита используются для хранения мантиссы. Это увеличивает число значащих цифр и уменьшает ошибку округления. В других машинах некоторое число битов из дополнительного набора используется для хранения большего порядка: это расширяет диапазон представления чисел.

Другой способ определения данных типа **double** заключается в использовании ключевых слов **long float**.

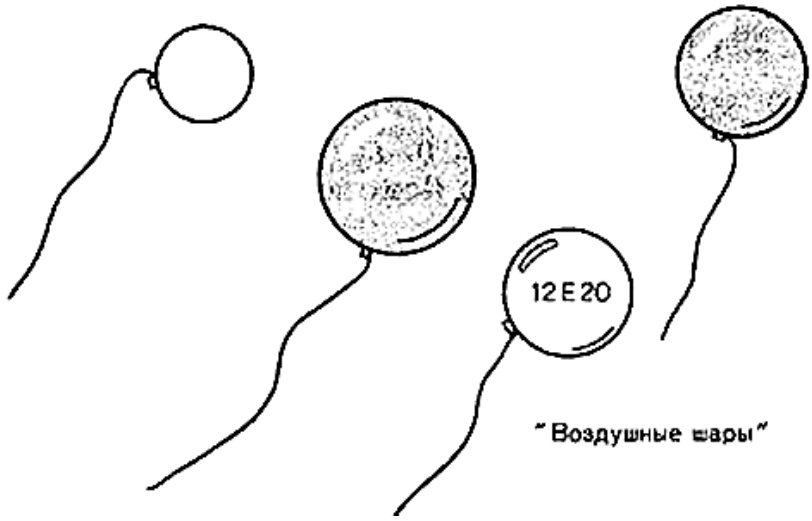


РИС. 3.5. Примеры чисел с плавающей точкой.

Описание переменных с плавающей точкой

Далее Содержание

Переменные с плавающей точкой описываются и инициализируются точно таким же образом, что и переменные целого типа. Ниже приведено несколько примеров:

```
float noah, jonah;
double trouble;
float planck = 6.63e- 34;
```

Константы с плавающей точкой

Далее Содержание

Правила языка Си допускают несколько способов записи констант с плавающей точкой. Наиболее общая форма записи константы - это последовательность десятичных цифр со знаком, включающая в себя десятичную точку, затем символ **e** или **E** и показатель степени по основанию **10** со знаком. Вот два примера:

```
-1.56E+12    2.87e-3
```

Знак **+** можно не писать. Разрешается также опускать либо десятичную точку, либо экспоненциальную часть, но не одновременно. Кроме того, можно не писать дробную или целую часть, но не обе сразу. Ниже приведено еще несколько правильно записанных констант с плавающей точкой:

```
3.14159    .2    4e16    .8E-5    100.
```

Использовать пробелы при записи констант запрещается

```
1.56E+ 12 - НЕПРАВИЛЬНО
```

В процессе обработки константы с плавающей точкой рассматриваются в формате с удвоенной точностью. Предположим, например, что переменная **some** типа **float** получает свое значение в результате выполнения оператора

```
some = 4.0*2.0;
```

В этом случае константы 4.0 и 2.0 размещаются в памяти как данные типа **double**, т. е. для

каждой из них (обычно) отводится 64 бит. Их произведение (равное 8) вычисляется с помощью операции умножения, выполняемой с двойной точностью, и только после этого производится усечение результата до нормального размера, соответствующего типу **float**. Все это обеспечивает максимальную точность ваших вычислений.

Переполнение и потеря значимости
при обработке чисел с плавающей точкой

Что произойдет, если значение переменной типа **float** выйдет за установленные границы? Например, предположим, что вы умножаете 10e38 на 100 (переполнение) или делите 10e - 37 на 1000 (потеря значимости). Результат целиком зависит от реакции вашей вычислительной системы. В нашей системе при возникновении состояния "переполнение" результат операции заменяется максимально допустимым числом, а при потере значимости - нулем. В других системах в подобной ситуации могут выдаваться предупреждающие сообщения, выполиение задачи можно приостановить, или вам будет предоставлена возможность предпринять что-нибудь самому. Если этот вопрос окажется для вас сушественным, вам необходимо будет свериться с правилами, действующими для вашей ЭВМ. В случае если вы не сможете найти никакой информации, не бойтесь пробовать другие возможности.

Резюме: основные типы данных

Ключевые слова

Данные основных типов вводятся в программу при помощи следующих семи ключевых слов: **int**, **long**, **short**, **unsigned**, **char**, **float**, **double**.

ЦЕЛЫЕ ЧИСЛА СО ЗНАКОМ

Данные этих типов могут принимать положительные и отрицательные значения.

int: основной целый тип, используемый в вычислительной системе;

long или **long int**: может содержать целое значение, не меньшее максимальной величины, допускаемой типом **int**, или даже большее;

short или **short int**: максимальное целое число типа **short** не больше, чем максимальное целое число типа **int**, а может быть, и меньше. Обычно числа типа **long** бывают больше чисел типа **short**, а тип **int** реализуется как один из двух указанных типов. Например, компилятор Lattice C на IBM PC под данные типов **short** и **int** отводит 16 бит, а под данные типа **long** - 32 бита. Все зависит от конкретной системы.

ЦЕЛЫЕ ЧИСЛА БЕЗ ЗНАКА

Данные этих типов принимают только положительные значения или нуль. Это расширяет диапазон возможных положительных значений. При указании типа используйте ключевое слово **unsigned**: **unsigned int**, **unsigned long**, **unsigned short**. Просто **unsigned** соответствует написанию **unsigned int**.

СИМВОЛЫ

Эти знаки соответствуют типографским символам, таким, как A, &, + и т. п. Обычно под каждый символ отводится 1 байт памяти.

Char: ключевое слово, используемое для указания данных этого типа.

Данные этих типов могут принимать положительные и отрицательные значения.

float: основной тип данных с плавающей точкой в системе;

double или **long float** при размещении в памяти чисел с плавающей точкой такого типа отводится (возможно) элемент памяти большего размера; при этом может допускаться либо большее число значащих цифр, либо большее значение порядка.

Резюме:

как описывать простые переменные

1. Выбрать требуемый тип данных.
2. Выбрать имя для переменной.
3. Для оператора описания использовать нижеследующий формат:
   спецификация-типа имя-переменной;
   Спецификация-типа формируется из одного или более ключевых слов.
   Вот несколько примеров:
   int erest;
   unsigned short cash;
4. Вы можете описать в одном операторе несколько переменных одного типа, разделяя их имена запятыми:
   char ch, **unit**, **ans**;
5. В операторе описания вы имеете возможность инициализировать переменную:
   float mass = 6.0E24;

Другие типы Далее Содержание

Этот раздел завершает рассмотрение основных типов данных. Некоторым читателям их число может показаться слишком большим. Остальные могут полагать, что описанных типов недостаточно; например, им захочется иметь булев тип или строковый тип данных. В языке Си они отсутствуют, но, несмотря на это, он вполне подходит для написания программ, связанных с обработкой логических данных или строк. Самые простые возможности работы со строками мы рассмотрим в следующей главе.

В языке Си имеются и другие типы данных, построенные с использованием основных типов. Они включают в себя массивы, указатели, структуры и объединения. Хотя эти типы являются пред метом рассмотрения последующих глав, мы, не подозревая об этом, уже применили указатели в примерах, приведенных в данной главе. [Указатели используются функцией **scanf( );** признаком этого в данном случае служит префикс **&**.]

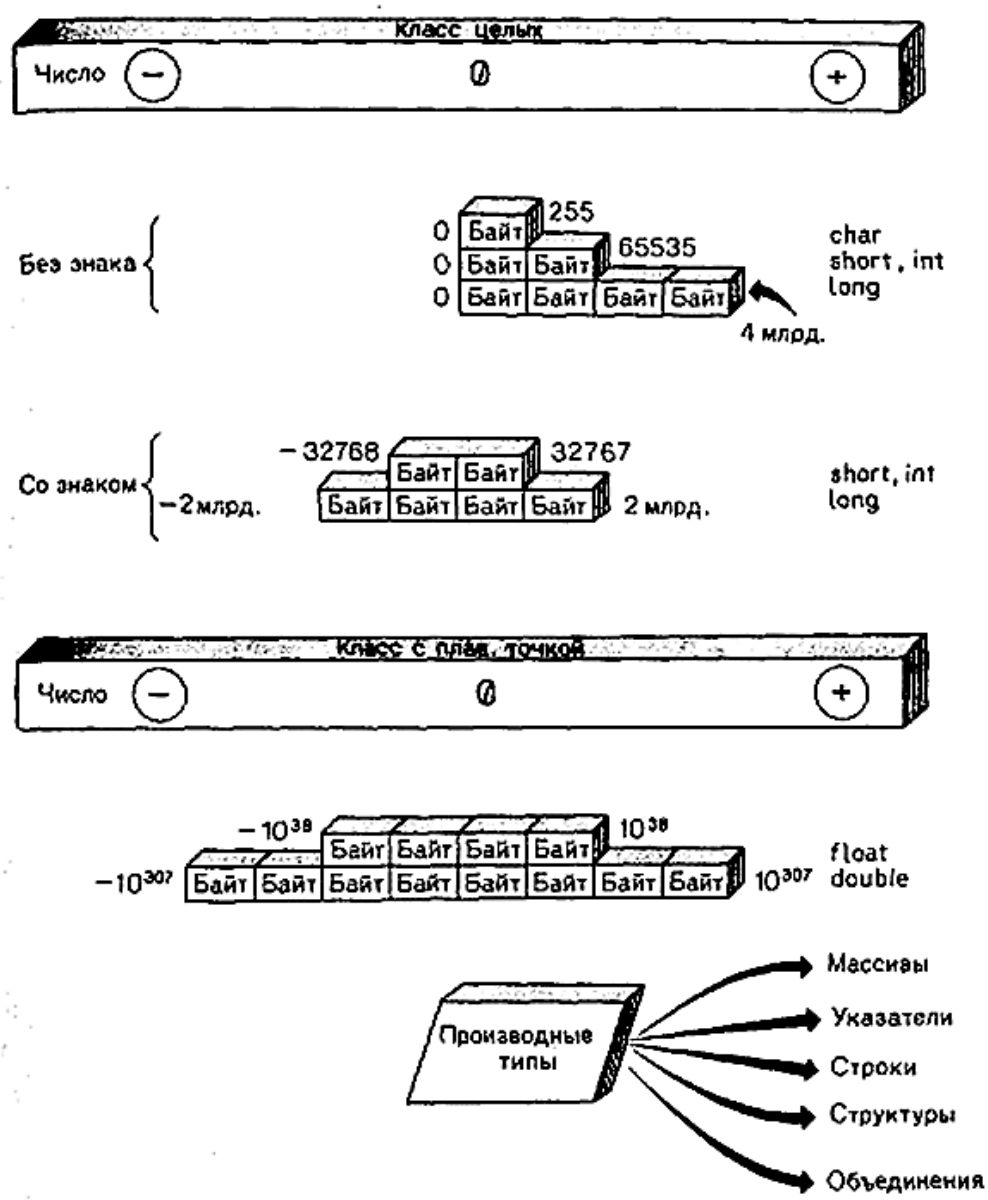


РИС. 3.6. Реализация типов данных языка Си в обычной вычислительной системе.

Размеры данных

Далее Содержание

Приведем таблицу размеров данных для некоторых распространенных вычислительных систем.

Таблица 3.1.

Представление типов данных в некоторых известных вычислительных системах

Размер слова	DEC PDP-11 16 бит	DEC VAX 32 бита	Interdata 8/3 32 бита	IBM PC (Lattice C) 16 бит
char	8	8	8	8
int	16	32	32	16
short	16	16	16	16
long	32	32	32	32
float	32	32	32	32
double	64	64	64	64
Диапазон порядка	±38	±38	±76	-307 ё 308 **(double)**

Как обстоит дело на вашей машине? Попробуйте выполнить нижеследующую программу:

```
main( )
{
    printf(" Данные типа int занимают %d байта. \n", sizeof (int));
    printf(" Данные типа char занимают %d байт.\n", sizeof (char));
    printf(" Данные типа long занимают %d байта.\n", sizeof (long));
    printf(" Данные типа double занимают %d байт.\n", sizeof (double));
}
```

В языке Си имеется встроенная операция sizeof, которая позволяет определить размер объектов в байтах.

Мы определили размеры данных только четырех типов, но вы легко можете модифицировать эту программу и найти размер объекта любого другого интересующего вас типа.

ИСПОЛЬЗОВАНИЕ ТИПОВ ДАННЫХ

Далее Содержание

Во время разработки программы вам необходимо составить список требуемых переменных и указать при этом, какого они должны быть типа. Скорее всего вы будете использовать тип **int** или, возможно, **float** для определения чисел и тип **char** для символов. Описывайте эти данные в самом начале тела функции, в которой они используются. Имена переменных выбирайте таким образом, чтобы они указывали на их смысл. При инициализации переменной следите за тем, чтобы тип константы соответствовал типу переменной.

```
int apples = 3; /* ПРАВИЛЬНО */
int oranges = 3.0; /* НЕПРАВИЛЬНО */
```

Язык Си "рассматривает" такие несоответствия менее жестко, чем, скажем, Паскаль, но в любом случае лучше учиться избегать дурных привычек.

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

В данной главе мы рассмотрели довольно большой материал. Суммируя его, мы обратим основное внимание на практическую сторону тех вопросов, которые здесь обсудили. Так же как и в предыдущей главе, мы дадим краткие примеры. Ниже приводится сводка тех фактов, которые вы должны были узнать из этой главы.

Что такое основные типы данных языка Си: **int**, **short**, **long**, **unsigned**, **char**, **float**, **double**.
Как описать переменную любого типа: **int beancount**, **float root-beer**; и т. д.
Как зависать константу типа **int**: **256**, **023**, **OXF5** и т. д.
Как записать константу типа **char**: **'r'**, **'U'**, **'\007'**, **'?'** и т. д.
Как записать константу типа **float**: **14,92**, **1.67e-27** и т. д.
Что такое слова байты и биты.
В каких случаях используются различные типы данных.

ВОПРОСЫ И ОТВЕТЫ

Содержание

Рассмотрение приводимых ниже вопросов должно помочь вам глубже УСВОИТЬ материал данной главы.

Вопросы

1. Какими типами вы будете пользоваться при обработке данных следующего вида
   а. Население Рио Фрито
   б. Средний вес картины Рембрандта
   в. Наиболее часто встречающаяся буква в тексте данной главы
   г. Сколько раз указанная буква встречается в тексте

2. Определите тип и смысл (если он есть) каждой из следующих констант
   а. '\b'
   б. 1066
   в. 99 44
   г. OXAA
   д. 20e30

3. Вирджила Анн Ксенопод (ВАКС)[3] написала программу с множеством ошибок Помогите ей обнаружить их:

```
#include < stdio h>
main
(
        float g, h,
        float tax, rate,
        g = e21, tax = rate*g,
)
```

Ответы

1. а. **int**, возможно **short**, население выражается целым числом
   б. **float**, маловероятно, что среднее окажется целым числом
   в. **char**
   г. **int**, возможно **unsigned**

2. а. **char**, символ "шаг назад"
   б. **int**, историческая дата[4]
   в. **float**, степень чистоты после мытья
   г. Шестнадцатеричное число типа **int**, десятичное значение **170**
   д. **float**, масса Солнца в кг

3. Строка 1: правильная
   Строка 2: должна содержать пару круглых скобок вслед за именем **main**, т. е. **main( )**
   Строка 3: нужно использовать **{**, а не **(**
   Строка 4: **g** и **h** должны разделяться запятой, а не точкой с запятой
   Строка 5: правильная
   Строка 6: (пустая) правильная
   Строка 7: перед **e** должна стоять по крайней мере одна цифра: например, **1e21** или **1 0e21**
   Строка 8: правильная
   Строка 9: нужно использовать **}**, а не **)**

Недостающие строки:
   первая - переменной **rate** нигде не присваивается значеиие.
   вторая - переменная **h** нигде не используется.
Кроме того, программа ни как не информирует нас о результатах вычислений. Ни одна из этих ошибок не будет служить препятствием для выполнения программы (хотя при компиляции может

быть выдано предупреждение о неиспользуемой переменной), но все ошибки существенно снижают эффективность программы и без того уже ограниченную.

[1)] В программе используются следующие единицы 1 фунт - 454 г, 1 тройская унция 31 г - *Прим ред*

[2)] Одометр - прибор для определения пройденного расстояния - *Прим перев.*

[3)] Аббревиатура этого имени совпадает с обозначением широко используемой сейчас вычислительной машины VAX (ВАКС) фирмы DEC - *Прим ред.*

[4)] Год битвы при Гастингсе - *Прим перев.*

---

4. Символьные строки, директива #define, функции printf( ) и scanf( )

В этой главе мы продолжим нашу "игру" с данными покопаемся в вопросах, выходящих за пределы тех, которые были связаны с типами данных, и рассмотрим символьную строку Сначала опи шем важное средство языка - препроцессор Си - и узнаем, как задавать и использовать символические константы. Затем вновь об судим способы ввода и вывода данных, при этом более полно ис следуем возможности функций **printf( )** и **scanf( )**. Ну, а теперь вы вероятно, ожидаете примера программы, который должен быть помещен в начале главы; мы не будем вас разочаровывать и приве дем его

```
/* непринужденный разговор */
# define DENSITY 62 4  /* плотность тела человека в фунтах на кубический фут */
main( )  /* любопытствующая программа*/
{
  float weight, volume;
  int size, letters;
  char name [40];   /* или попробуйте "static char name [40], */
  printf(" Привет! Как вас зовут?\n" );
  scanf(" %s" , name);
  printf("%s,  Каков ваш вес в фунтах?\n", name);
  scani("%f", &weight);
  size = sizeof name;
  letters = strlen (name);
  volume = weight/DENSITY;
  printf(" Прекрасно, %s, ваш объем %2 2f кубических фута.\n", name, volume);
  printf(" Кроме того, ваше имя состоит из %d букв,\n", letters);
  printf(" и для его размещения в памяти у нас есть %d байт.\n", size);
}
```

Результат работы программы "непринужденный разговор" может, например, выглядеть следующим образом:

```
Привет ! Как вас зовут?
Анжелика
Анжелика  Каков ваш вес в фунтах?
102,5
Прекрасно, АНЖЕЛИКА ваш объем 1,64 кубических фута
Кроме того, ваше имя состоит из 8 букв
и для его размещения в памяти у нас есть 40 байт
```

Перечислим основные новые черты этой программы:

1. Мы использовали "массив" для хранения "символьной строки" - в данном случае для некоторого имени.

2. При вводе и выводе строки была использована "спецификация преобразования" **%s**.

3. Для определения символической константы **DENSITY** был использован препроцессор языка Си.

4. Для нахождения длины строки была использована функция **strlen( )**.

Способ ввода-вывода, реализованный в языке Си, может показаться вначале несколько более сложным по сравнению с вводом-выводом, предусмотренным, например, в Бейсике. Однако эта сложность окупается улучшенными возможностями управления вводом-выводом и большей эффективностью получаемых программ. Указанная трудность - не единственная, с которой вы столкнетесь в дальнейшем. Давайте исследуем все новые моменты более детально.

СИМВОЛЬНЫЕ СТРОКИ - ВВЕДЕНИЕ

Далее Содержание

"Символьная строка" - это последовательность, состоящая из одного или более символов В качестве примера рассмотрим следующую строку:

```
"Строки изливались прямо из сердца!"
```

Кавычки не являются частью строки. Они вводятся только для того, чтобы отметить ее начало и конец, т.е. играют ту же роль, что и апострофы в случае одиночного символа.

В языке Си нет специального типа, который можно было бы использовать для описания строк Вместо этого строки представля ются в виде "массива" элементов типа **char**. Это означает, что символы в строке можно представить себе расположенными в со седних ячейках памяти - по одному символу в ячейке (рис. 41).

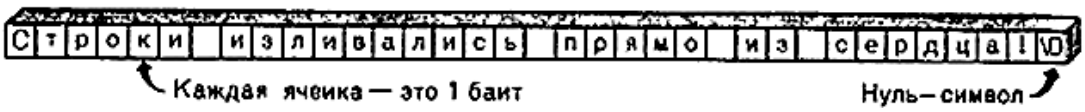


РИС. 4. 1. Строка как массив ячеек

Необходимо отметить, что на рисунке последним элементом массива является символ **\0**. Это "нуль-символ", и в языке Си он используется для того, чтобы отмечать конец строки Нуль-символ - не цифра **0**; он не выводится на печать и в таблице кода ASCII[1] имеет номер **0**. Наличие нуль-символа означает, что количество ячеек массива должно быть по крайней мере на одну больше, чем число символов, которые необходимо размещать в памяти.

Ну, а теперь спросим, что такое массив? Массив можно пред ставить себе как совокупность нескольких ячеек памяти, объединен ных в одну строку Если вы предпочитаете более формальные и строгие определения, то массив - это упорядоченная последова тельность элементов данных одного типа В нашем примере мы со здали массив из 40 ячеек памяти, в каждую из которых можно по местить один элемент типа **char**. Мы осуществили это с помощью оператора описания

```
char name [40];
```

Квадратные скобки указывают, что переменная **name** - массив, 40 - число его элементов, а **char** задает тип каждого элемента. (В комментариях к программе было отмечено, что при желании вы можете воспользоваться более сложным оператором описания).

```
static char name [40],
```

Ввиду некоторой специфики, связанной с реализацией функции **scanf( )** в нашей системе, мы

вынуждены использовать эту вторую

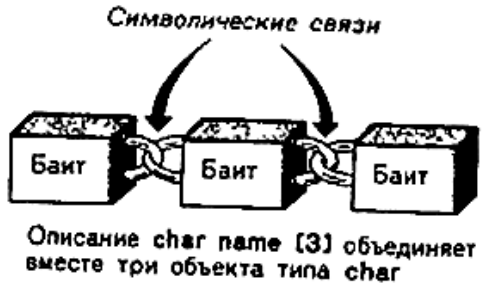


РИС.4.2. Описание имени массива типа **char**

форму, но весьма вероятно, что вы сможете выбрать любую из них. Если обнаружится, что при работе с первой формой оператора описания у вас возникнут трудности при решении наших примеров, попробуйте воспользоваться второй В действительности вторая форма должна работать в любой системе, но мы не хотим применить тип **static** до тех пор, пока не рассмотрим в гл 10 понятие классов памяти).

На первый взгляд все это выглядит довольно сложным: вы должны создать массив, расположить символы в виде строки и не забыть добавить в конце **\0**. К счастью, о большинстве деталей компилятор может "позаботиться" сам.

Попробуйте выполнить приведенную ниже программу, чтобы посмотреть, как просто все происходит на практике:

```
/* похвала 1*/
#define PRAISE " Вот эта да, какое великолепное имя"
main( )
{
char name [50];
printf(" Как вас зовут? \n" );
scanf(" %s", name);
printf(" Привет, %s %s\n" , name, PRAISE);
}
```

Символ **%s** служит указанием функции **printf( )** напечатать строку. Результат выполнения программы **похвала 1** может выглядеть, например, так

```
Как вас зовут ?
Элмо Бланк Привет, Элмо,  Вот эта да, какое великолепное имя !
```

Как видите, нам не пришлось самим помещать нуль символ в конец массива. Эта задача была выполнена за нас функцией **scanf( )** при чтении вводимой строки. **PRAISE** - "символическая строковая константа". Ниже мы рассмотрим директиву **#define** более подробно, а пока вы должны знать, что кавычки, в которые за ключена фраза, следующая за строковой константой **PRAISE**, иден тифицируют эту фразу как строку, и поэтому в ее конец будет помещен нуль-символ.

Заметим (и это очень важно), что функция **scanf( )** при вводе строки "Элмо Бланк" читает только имя Элмо. Дело в том, что, встретив какой-нибудь разделитель (пробел, символ табуляции или перевода строки), функция **scanf( )** прекращает ввод символов, т е в данном случае она прекращает опрос переменной **name** в тот момент, когда доходит до пробела между "Элмо" и "Бланк". Вообще говоря, функция **scanf( )** вводит только одиночные слова, а не целую фразу в качестве строки. Для чтения входной информации в языке Си имеются другие функции, например функция **gets( )**, предназначенная для обработки строк общего вида. Более полно работу со строками мы рассмотрим в последующих главах.

Необходимо заметить также, что строка **"x"** не то же самое, что символ **'x'**. Первое различие: **'x'** - объект одного из основных типов **(Char)**, в то время как **"x"** - объект производного типа (массива элементов типа **char**). Второе различие: **"x"** на самом де ле состоит из двух символов - символа **'x'** и нуль-символа.

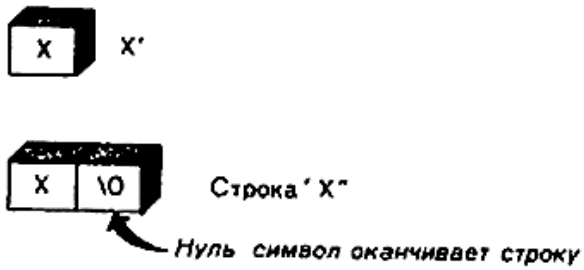


РИС.4.3. Символ 'x' и строка "x"

Длина строки - функция strlen( )

Далее Содержание

В предыдущей главе мы практически без объяснений использовали операцию **sizeof**, которая дает нам размер объектов в байтах Функция **strlen( )** позволяет определять длину строки числом символов. Поскольку для размещения одного символа в памяти отводится 1 байт, можно было бы предположить, что в результате применения любой из этих двух операций к одной строке будет получен одинаковый результат. Оказывается, это не так. Давайте немного изменим нашу предыдущую программу (добавим к ней несколько строк), и тогда мы поймем, в чем дело.

```
/*похвала 2*/
#define PRAISE " Вот это да, какое великолепное имя!"
main( )
{
char name [50];
printf(" Как вас зовут?\n");
scanf(" %s", name);
printf(" Привет, %s. %s\n" , name, PRAISE);
printf(" Ваше имя состоит из %d букв и занимает %d ячеек памяти. \n",
strlen (name), sizeof name);
printf(" Хвалебная фраза состоит из %d букв", strlen (PRAISE));
printf(" и занимает %d ячеек памяти. \n", sizeof PRAISE);
}
```

Заметим, что случайно мы воспользовались двумя методами для обработки длинных операторов **printf()**. В первом случае мы записав один оператор печати в двух строках программы[2)] . Мы сделали это, поскольку разрешается разбивать строку между аргументами, но не посередине строки. В другом случае использовались два оператора **printf()** для печати одной строки; мы указали символ "новая строка" (**\n**) только во втором из них. Представленный ниже результат работы данной программы поможет понять подобную ситуацию:

```
Как вас зовут ?
Перки
Привет, Перки. Вот это да, какое великолепное имя!
Ваше имя состоит из 5 букв и занимает 50 ячеек памяти.
Хвалебная фраза состоит из 35 букв и занимает 36 ячеек памяти.
```

Давайте посмотрим, в чем дело. Массив **name** занимает 50 ячеек памяти, и именно об этом сообщает операция **sizeof**. Но для хранения имени **Перки** требуются только первые пять ячеек, и как раз об этом нас информирует функция **strlen( )**. В шестой ячейке массива **name** содержится

нуль-символ, и его появление служит сигналом для функции **strlen( )** прекратить подсчет символов

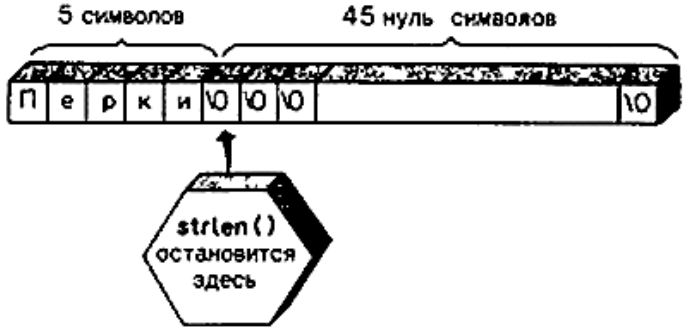


РИС.4.4. Распознавание функцией **strlen( )** конца строки

При переходе к обработке константы **PRAISE** обнаруживается, что функция **strlen( )** опять дает нам точное число символов (включая пробелы и знаки пунктуации) в строке. Результат операции **sizeof** оказывается на единицу большим, поскольку при этом учи тывается и "невидимый" нуль-символ, помещенный в конец строки. Мы не указываем компилятору, какой объем памяти он должен отвести для размещения всей фразы, он сам подсчитывает число символов между кавычками.

Еще одно замечание в предыдущей главе была использована операция **sizeof** со скобками, а в этой - без них. Решение, исполь зовать ли скобки или нет, зависит от того, что вы хотите знать объем памяти, отводимый под элементы конкретного типа, или объем памяти, занимаемый определенным объектом В первом слу чае вы писали бы **sizeof(char)** или **sizeof(float)**, а во втором - **sizeof name** или **sizeof 6.28**.

В данном разделе операции **strlen( )** и **sizeof** использовались только для удовлетворения нашего любопытства, в действительности они представляют собой важные программные средства Функция **strlen( )**, например, полезна в любого сорта программах обра ботки строк или символов, в чем вы сможете убедиться, ознакомившись с гл 13. Приступим теперь к рассмотрению директивы **#define**.

КОНСТАНТЫ И ПРЕПРОЦЕССОР ЯЗЫКА Си

Далее Содержание

Иногда возникает необходимость использовать в программах константы. Например, оператор, позволяющий определять длину окружности, можно было бы записать в следующем виде

```
circ = 3.14 * diameter,
```

Приведенная здесь константа **3. 14** - известное число р. Чтобы ввести ту или иную константу в программу, нужно указать ее фак тическое значение, как было сделано выше. Однако существуют веские причины использовать вместо этого "символические кон станты", например, мы могли бы применять оператор

```
circ = pi * diameter,
```

а позже компилятор подставил бы в него фактическое значение константы.
В чем достоинства такого метода? Во-первых, имя говорит нам больше, чем число. Сравним два оператора

```
owed = 0 015 * housevl, owed = taxrate * housevl,
```

Если мы изучаем большую программу, то второй вариант будет нам более понятен.
Во-вторых, предположим, что некоторая константа использова лась в нескольких местах программы

и впоследствии возникла не обходимость изменить ее значение - ведь в конце концов и налого вые тарифы (taxrate) меняются, и, к примеру, некое законодатель мое собрание приняло однажды закон впредь считать число p равным $3^1/_7$. (Весьма вероятно, что окружности пришлось при этом скрываться от правосудия). В таком случае требуется только изменить определение символической константы, а не отыскивать каждый случай ее появления в программе.

Теперь осталось выяснить, как можно создать такую символическую константу? Первый способ заключается в том, чтобы описать некоторую переменную и положить ее равной требуемой константе. Мы могли бы сделать это следующим образом

```
float taxrate, taxrate = 0 015,
```

Такой способ подходит для небольшой программы, в других же случаях он несколько неэкономичен, поскольку каждый раз при использовании переменной **taxrate** компьютер должен будет обращаться к той ячейке памяти, которая отведена данной переменной Это служит примером подстановки "во время выполнения", так как она производится именно при выполнении программы. К счастъю, в языке Си имеется и другой, лучший способ.

Этот способ реализуется с помощью препроцессора языка Си В гл. 2 мы уже видели, как препроцессор использует директиву **#include** для включения информации из другого файла в программу. Кроме того, препроцессор дает нам возможность задавать константы Для этого в начало файла, содержащего вашу программу, необходимо добавить только одну строку, аналогичную следующей

```
#define TAXRATE  0.015
```

При компиляции программы каждый раз, когда появится переменная **TAXRATE**, она будет заменяться величиной 0.015. Это называ ется подстановкой "во время компиляции". К тому моменту, когда вы начнете выполнение своей программы, все подстановки будут уже сделаны.

Несколько замечаний по поводу формата. Сначала идет ключевое слово **#define**. Оно должно начинаться с самой левой позиции. Потом следует символическое имя константы, а затем ее величина. Символ "точка с запятой" не используется, поскольку это не оператор языка Си. Почему имя **TAXRATE** пишется прописными буква ми? В процессе использования языка Си выработалась традиция писать константы прописными буквами. Если при просмотре программы вам встретится имя, написанное прописными буквами, вы сразу поймете, что имеете дело с константой, а не с переменной. Это еще один способ улучшить читаемость программы. Ваша программа будет работать даже и тогда, когда вы будете писать кон станты строчными буквами, но при этом вы должны чувствовать свою вину, поскольку нарушаете традицию.
Приведем простой пример[3)]

```
/* пицца */
#define PI  3,14159
main( ) /* изучение вашей пиццы */
{
float area, circum, radius;
printf("Чему равен радиус вашей пиццы? \n");
scanf("%f", &radius);
area = PI * radius * radius;
printf(" Основные параметры вашей пиццы следующие \n");
printf(" длина окружности = %1.2f, площадь =%1.2f \n circum, area);
}
```

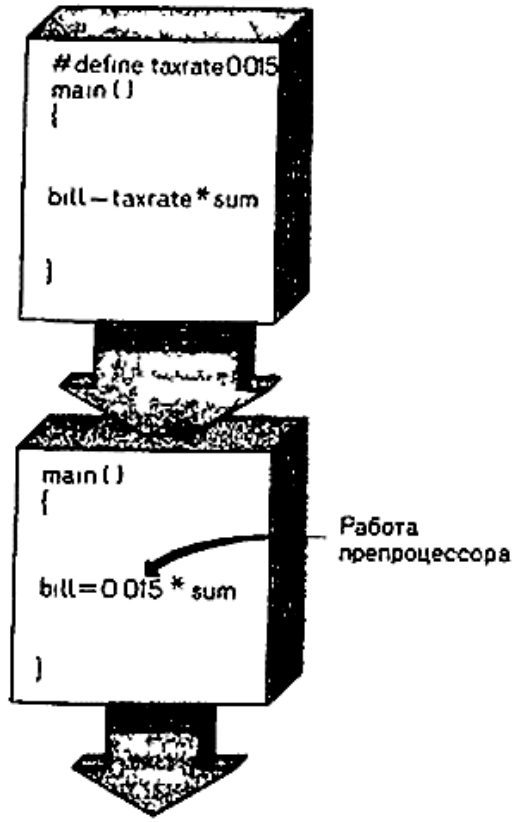


РИС.4.5. Обработка текста программы препроцессором

Использование спецификации **%1.2f** в операторе **printf( )** приведет к тому, что при печати результаты будут округлены до двух десятичных цифр. Мы понимаем, конечно, что написанная выше программа может и не отражать ваши собственные вкусы, касаю щиеся пиццы, но во множестве программ, посвященных этому вопросу, она займет свое скромное место. Вот один из примеров ее выполнения.

```
Чему равен радиус вашей пиццы ? 6.0
Основные параметры вашей пиццы следующие: длина окружности = 37.70,
площадь окружности = 113.40.
```

Директиву **#define** можно также использовать для определения символьных и строковых констант. Необходимо использовать знак "апостроф" в первом случае и кавычки - во втором. Примеры, приведенные ниже, вполне правомерны

```
#define BEEP  '\007'
#define ESS 'S'
#deline NULL '\0'
#define OOPS  "Ну вот, вы и сделали это!"
```

А теперь мы хотим обрадовать лентяев. Предположим, вы раз рабатываете целый пакет программ, использующих один и тот же набор констант. Вы можете произвести следующие действия:

1. Соберите все ваши директивы **#define** в один файл и назовите его, например, **const.h**.
2. В начало каждого файла[4] , содержащего программу, включите директиву **#include "const.h."**

Тогда, если вы будете выполнять программу, препроцессор прочтет файл с именем **const.h** и использует все директивы **#define** вашей программы. Получилось так, что символ **.h** в конце имени файла напомнит вам, что этот файл является "заголовком", т.е. в нем содержится вся информация, которая должна попасть в начало вашей программы. Самому препроцессору безразлично,

используете ли вы символ **.h** в имени файла или нет.

Язык Си - искусный фокусник: создание псевдоимен

Далее Содержание

Возможности директивы **#define** не исчерпываются только символическим представлением констант. Рассмотрим, например, следующую программу:

```
#include "alias. h"
program begin
whole yours, mine then
spitout(" Введите, пожалуйста, целое число.\n" )
then takem(" %d", & yours)
then mine = yours times TWO then
spitout(" %d в два раза больше вашего числа! \n" , mine) then end
```

Странно, текст что-то смутно напоминает, язык немного похож на Паскаль, но программа не похожа на Си-программу. Секрет лежит, конечно, в файле с именем **alias.h.** Давайте посмотрим, что в нем содержится?

```
alias. h #define program main( )
#define degin { #define enf  } #define then;
#define takein   scanf
#define spilout   printf
#define TWO  2
#define times   *
#define whole int
```

Этот пример иллюстрирует, как работает препроцессор. Он просматривает вашу программу и проводит поиск элементов, определяемых директивами **#define**. Обнаружив такие элементы, он полностью заменяет их. В нашем примере во время компиляции все слова **then** заменяются символами "точка с запятой", **end** - **}** и т.д. Результирующая программа будет полностью идентична той, которую мы могли бы получить, если бы с самого начала писали ее в обычных терминах языка Си.

Эту мощную возможность языка можно использовать для задания макрокоманд, являющихся одним из вспомогательных средств программирования. Мы вернемся к обсуждению этой темы в гл. 11.

Теперь необходимо упомянуть о некоторых ограничениях. На-пример, части программы, заключенные в кавычки, закрыты для подстановок. Операторы, приводимые ниже, служат иллюстрацией такого положения:

```
#define MN "минимифидианизм"
printf(" Он глубоко верил в MN.\n");
```

Распечатка будет выглядеть так:

```
Он глубоко верил в MN.
```

Однако после выполнения оператора

```
printf(" Он глубоко верил в %s.\n" , MN);
```

мы получим следующий результат:

```
Он глубоко верил в минимифидианизм.
```

В последнем случае константа с именем **MN** находилась вне кавычек и поэтому была заменена соответствующим значением.
Препроцессор языка Си является полезным вспомогательным средством, поэтому при написании

программ старайтесь пользоваться преимуществами, которые он предоставляет. По мере изложения мы покажем вам дополнительные возможности применения препроцессора.

ИЗУЧЕНИЕ И ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ printf( ) И scanf( )

Далее Содержание

Функции **printf( )** и **scanf( )** дают нам возможность взаимодействовать с программой. Мы называем их функциями ввода-вывода. Это не единственные функции, которыми мы можем вос пользоваться для ввода и вывода данных с помощью программ на языке Си, но они наиболее универсальны. Указанные функции не входят в описание языка Си. И действительно, при работе с язы ком Си реализация функций ввода-вывода возлагается на созда телей компилятора; это дает возможность более эффективно орга низовать ввод вывод на конкретных машинах. Однако в интересах обеспечения совместимости различные системы имеют дело с неко торыми вариантами функций **scanf( )** и **printf( )**. Все, о чем мы здесь говорим, должно быть в основном справедливо для большинства систем, но не удивляйтесь, если обнаружите некоторые отличия в имеющейся у вас версии.

Обычно функции **printf()** и **scanf()** "работают" во многом одинаково - каждая использует "управляющую строку" и список "аргументов". Сначала мы рассмотрим работу функции **printf()**, затем **scanf( )**.

Инструкции, передаваемые функции **printf( )**, когда мы "просим" ее напечатать некоторую переменную, зависят от того, какого типа эта переменная. Например, при выводе на печать целого числа применяется формат **%d**, а при выводе символа - **%c**. Ниже перечислены все форматы, указываемые при обращениях к функции **printf()**, а затем показано, как они используются. Каждому форма ту соответствует тип выводимой (с их помощью) информации, причем первые пять покрывают большинство возникающих по требностей, а остальные четыре применяются достаточно редко.

Формат	*Тип выводимой информации*
%d	Десятичное целое число
%c	Один символ
%s	Строка символов
%e	Число с плавающей точкой, экспоненциальная запись
%f	Число с плавающей точкой, десятичная запись
%g	Используется вместо записей
%f	или %e, если он короче
%u	Десятичное целое число без знака
%o	Восьмеричное целое число без знака
%x	Шестнадцатеричное целое число без знака

Посмотрим теперь, как эти форматы применяются.

Использование функции printf( )

Далее Содержание

Приведем программу, иллюстрирующую обсуждаемые вопросы

```
/* печать чепухи*/
#define PI  3.14159
main( )
{
      number = 5;
      float ouzo =13,5;
      int cost = 31000;
```

```
      printf("%d женщин выпили %f стаканов ликера. \n",
               number, ouzo);
      printf(" Значение числа pi равно %f \n", PI);
      printf(" Прощай! Твое искусство слишком дорого для меня \n");
     printf(" %c%d\n", '$', cost);
}
```

Результат выглядит так:

```
5 женщин выпили 13,50000 стаканов ликера.
Значение числа pi равно 3,14159.
Прощай! Твое искусство слишком дорого для меня.
$31000
```

Формат, указываемый при обращении к функции **printf( )**, выглядит следующим образом:

```
printf(Управляющая строка, аргумент1,  аргумент2, ...,);
```

Аргумент1, Аргумент2 и т. д. - это печатаемые параметры которые могут быть переменными, константами или даже выражениями, вычисляемыми вначале, перед выводом на печать.
Управляющая строка - строка символов, показывающая, как должны быть напечатаны параметры. Например, в операторе

```
printf(" %d женщин выпили %f стаканов ликера. \n" , number, ouzo);
```

управляющей строкой служит фраза в кавычках (учитывая предыдущие замечания, это - строка символов), а **number** и **ouzo** - аргументы или в данном случае значения двух переменных.

РИС. 4.6. Аргументы функции printf( )

Приведем еще пример.

```
printf(" Значение числа pi равно %f.\n", PI);
```

На этот раз список аргументов содержит только один элемент - символическую константу **PI**.
Мы видим, что в **управляющей строке** содержится информация двух различных видов:
1. Символы, печатаемые текстуально.
2. Идентификаторы данных, называемые также "спецификациями преобразования".

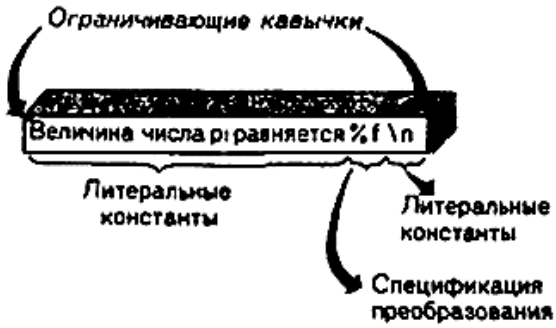


РИС. 4.7. Структура управляющей строки

Каждому аргументу из списка, следующего за управляющей строкой, должна соответствовать одна спецификация преобразования. Горе вам, если вы забудете это основное требование. Никогда не пишите, например, так:

```
printf(" Количество слизняков %d, червяков %d.\n", scorel);
```

Здесь отсутствует аргумент для второй спецификации преобразования **%d**. Способ проявления этой ошибки целиком зависит от вашей вычислительной системы, но в лучшем случае вы получите бессмыслицу.

Если вам нужно напечатать какую-нибудь фразу, то нет необходимости использовать спецификацию преобразования; если же требуется только вывести данные на печать, то можно обойтись и без использования комментария. Поэтому каждый из операторов, приведенных ниже, вполне приемлем.

```
printf(" Прощай! Твое искусство слишком дорого для меня.\n");
printf(" %c%d\n" , '$', cost);
```

Заметим, что во втором примере первый аргумент из печатаемого списка является символьной константой, а не переменной.

Поскольку символ **%** используется в функции **printf( )** для идентифицирования спецификаций преобразования, возникает небольшая проблема в том случае, если вам нужно напечатать сам символ **%**, Если просто написать один знак **%**, то компилятор примет его за ошибочную спецификацию преобразования. Выходом из создавшейся ситуации служит довольно простое решение - писать два символа **%** подряд:

```
pc = 2*6;
printf("Только %d%% стряпни Салли было съедобно.\n", pc);
```

Результат работы программы будет выглядеть следующим образом:

```
Только 12%  стряпни Салли было съедобно.
```

Модификаторы спецификации преобразования, используемые в функции printf( )

Далее Содержание

Мы можем несколько расширить основное определение спецификации преобразования, поместив модификаторы между знаком **%** и символом, определяющим тип преобразования. В приводимой ниже таблице дан список тех символов, которые вы имеете право туда поместить. При использовании одновременно нескольких модификаторов они должны быть указаны в том порядке, в котором пере числены в таблице. Заметим, что при этом допускаются не все комбинации.

Модификатор	*Значение*
-	Аргумент будет печататься с левой позиции поля заданной ширины (как объяснено ниже). Обычно печать аргумента оканчивается в самой правой позиции поля. *Пример:* **%-10d**
строка цифр	Задает минимальную ширину поля. Большее поле будет использоваться, если печатаемое число или строка не помещаются в исходном поле. *Пример:* **%4d**
строка цифр	Определяет точность: для типов данных с плавающей точкой - число печатаемых цифр справа от десятичной точки; для

	символьных строк - максимальное число печатаемых символов *Пример:* **%4.2f** (две десятичные цифры для поля шириной в четыре символа)
l	Соответствующий элемент данных имеет тип **long**, а не **int**. *Пример:* **%ld**

Примеры

Далее Содержание

Посмотрим, как эти модификаторы работают. Начнем с того, что продемонстрируем влияние модификатора ширины поля на печать целого числа. Рассмотрим следующую программу:

```
main( )
{
  printf("/%d/\n", 336);
  printf("/%2d/\n", 336);
  printf("/%10d/\n", 336);
  printf("/%-10d/\n", 366);
}
```

Эта программа печатает одно и то же значение четыре раза, но используются при этом четыре различные спецификации преобразования. Мы вводим также символы **/**, чтобы вы могли видеть, где начинается и кончается каждое поле. Результат выполнения программы выглядит следующим образом:

```
/336/ /336/
/      336 /336       /
```

Первая спецификация преобразования **%d** не содержит модификаторов. Мы видим, что поле печати здесь имеет ширину, равную количеству цифр данного целого числа. Это так называемый выбор "по умолчанию", т. е. результат действия компилятора в случае, если вы не дали ему никаких дополнительных инструкций. Вторая спецификация преобразования - **%2d**. Она указывает, что ширина поля должна равняться **2**, но, поскольку число состоит из трех цифр, доле автоматически расширяется до необходимого размера. Следующая спецификация **%10d** показывает, что ширина поля равна **10**. И действительно, между символами **/** имеется семь пробелов и три цифры, причем число сдвинуто к правому краю поля. Последняя спецификация **%-10d** также указывает ширину поля, равную **10**, а знак - приводит к сдвигу всего числа к левому краю, как показано в приведенном выше примере. Когда вы привыкнете к этой системе обозначений, она покажется вам простой и вы сумеете по вашему усмотрению менять вид выходной информации.

Рассмотрим теперь некоторые форматы, соответствующие данным с плавающей точкой. Допустим, у нас имеется следующая программа:

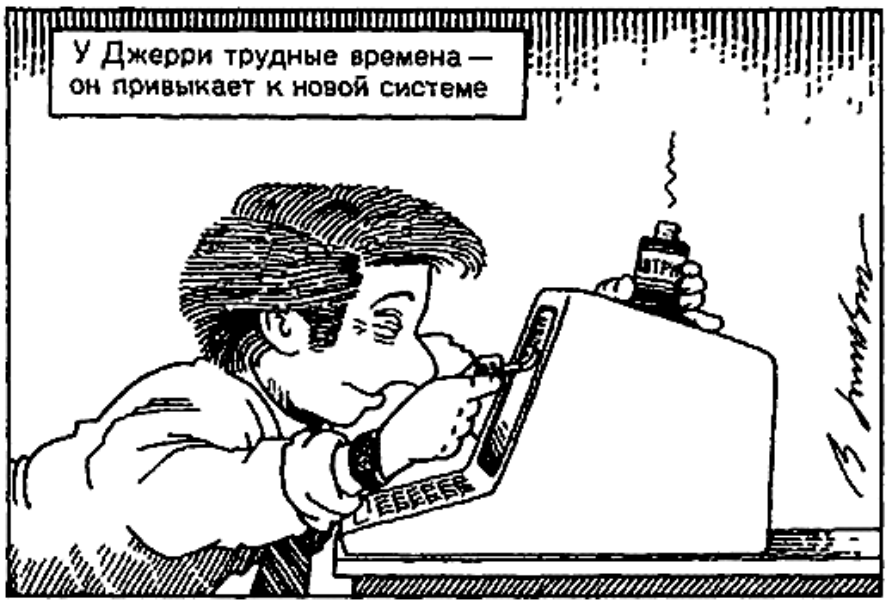
```
main( )
{
        printf(" /%f/\n" , 1234.56);
        printf(" /%e/\n" , 1234.56);
        printf(" /%4.2f/\n" , 1234.56);
        printf(" /%3.1f/\n", 1234.56);
        printf(" /%10.3f/\n" , 1234.56);
        printf(" /%10.3e/\n" , 1234.56);
}
```

На этот раз результат работы программы будет выглядеть так

```
        /1234.560059/
        /1.234560E+03/
        /1234.56/
        /1234.6/
```

```
/   1234.560/
/ 1.234E+03/
```

Мы снова начинаем с варианта, выбранного по умолчанию, т. е. со спецификации **%f**. В этом случае имеется две величины, значене которых используются по умолчанию: ширина поля и число цифр справа от десятичной точки. Вторая величина задает шесть цифр, ширина поля берется такой, чтобы в нем могло поместиться число. Заметим, что печатаемое число несколько отличается от исходного Это происходит потому, что на печать выводится 10 цифр, в то время как числа с плавающей точкой в нашей системе изображаются приблизительно с точностью до **6** или **7** цифр.



Рассмотрим вариант по умолчанию для спецификации **%e**. Как мы видим, при ее использовании печатается одна цифра слева от десятичной точки и шесть справа. В результате получается слишком много цифр! Чтобы избежать этого, необходимо задать число цифр справа от десятичной точки, и последние четыре опера тора программы реализуют как раз указанную возможность Обратите внимание на то, как четвертый и шестой операторы производят округление выводимых данных.
Теперь исследуем некоторые варианты строк. Рассмотрим пример:

```
#define BLURB "Выдающееся исполнение"
main( )
{
printf(" /%2s/\n" , BLURB);
printf(" /'%25.s/\n" , BLURB);
printf(" /'%25.5s/\n" , BLURB);
printf(" /% - 25.5s/\n" , BLURB);
}
```

Вот результат работы программы:

```
/Выдающееся исполнение!/
/   Выдающееся исполнение!/
/                                Выдаю/ /Выдаю                                /
```

Обратите внимание на то, как поле расширяется для того, чтобы поместились все указанные символы. Заметим также, как спецификация точности ограничивает число символов, выводимых на ПЕЧАТЬ. Символы **.5** в спецификации формата указывают функции **printf( )** на необходимость напечатать только пять символов.

Теперь вы ознакомились с некоторым количеством примеров. А знаете ли вы, как подготовить

оператор печати, чтобы напечатав нечто вроде следующей фразы:

```
Семья NAME, возможно, лишь на XXX.XX долларов богаче!
```

Здесь **NAME** и **XXX.XX** представляют значения соответствующих переменных в программе, скажем **name[40]** и **cash**. Вот одно из решений:

```
printf(" Семья %s, возможно, лишь на %.2f долларов
богаче! \n", name, cash);
```

До сих пор мы без тени сомнения применяли спецификации преобразования для переменных разных типов, например, **%f** для типа **float** и т. д. Но, как мы уже видели в нашей программе поиска кода ASCII, для некоторого символа функцию **printf()** можно исполь зовать также для преобразования данных из одного типа в другой. Мы не намерены, однако, терять чувство реальности и по прежнему будем работать с целыми типами.

Использование функции printf( ) для преобразования данных

Далее Содержание

Здесь мы снова займемся выводом на печать целых чисел. По. скольку мы уже осведомлены о полях, то не будем заботиться об использовании символа **/**, чтобы отмечать их начало и конец.

```
main( )
{
printf(" %d\n", 336);
printf(" %o\n", 336);
printf(" %x\n", 336);
printf(" %d\n",  -336);
printf(" %u\n",  -336);
}
```

В нашей системе результат будет выглядеть следующим образом

```
336
520
150
-336
-65200
```

Как вы, по-видимому, и ожидали, при использовании спецификации **%d** будет получено число **336** точно так же, как в примере, обсуждавшемся чуть выше. Но давайте посмотрим, что произойдет, когда вы "попросите" программу напечатать это десятичное целое число в восьмеричном коде. Она напечатает число **520**, являющееся восьмеричным эквивалентом **336 (5x64+2x8+0x 1= 336)**. Аналогично при печати этого числа в шестнадцатеричном коде мы получим **150**.

Таким образом, мы можем использовать спецификации, приме няемые для функции **printf( )** с целью преобразования десятичных чисел в восьмеричные или шестнадцатеричные и наоборот. Или же если вы захотите напечатать данные в желаемом для вас виде, то необходимо указать спецификацию **%d** для получения десятичных чисел, **%o** - для восьмеричных, а **%x**- для шестнадцатеричных. При этом не имеет ни малейшего значения, в какой форме число первоначально появилось в программе.

Сделаем еще несколько замечаний относительно вывода на печать. Печать числа **-336** при использовании спецификации **%d** не вызывает никакого затруднения. При применении же спецификации **%u** (unsigned - беззнаковая) получаем число **65200**, а не **336**, как можно было бы ожидать. Причина получения такого результата лежит в способе представления отрицательных чисел в нашей системе. Здесь используется так называемый "дополнительный код". Числа от **0** до **32767** отображаются обычным образом, а от **32768** до **65535** представляют отрицательные числа, причем **65535** кодирует число **-1**, **65534** - число **-2** и т. д. Поэтому числу **-336** соответствует **65536**,

-336 = **65200**. Этот метод применяется не во всех системах. Тем не менее отсюда следует вывод: не ожидайте, что спецификация преобразования **%u** приводит просто к отбрасыванию знака числа.

Сейчас мы переходим к обсуждению интересного примера, которого мы уже касались ранее, а именно к использованию функции **printf()** для нахождения кода ASCII некоторого символа. Например оператор

```
printf(" %c%d\n" , ' A', ' A');
```

выдаст следующий результат:

```
A 65
```

A - это буква, а **65** - десятичный код ASCII символа **A**. Мы могли бы использовать спецификацию **%o**, если бы хотели получить восьмеричный код ASCII символа **A**.

Все вышесказанное дает хороший способ нахождения кодов ASCII для различных символов и наоборот. Вполне возможно, конечно, что вы предпочтете ему поиск кодов в приложении Ж. Что произойдет, если вы попробуете преобразовать число, больше **255**, в символ? Следующая строка и результат ее выполнения дадут ответ на этот вопрос:

```
printf(" %d %c\n" , 336, 336);
336 P
```

Десятичный код ASCII символа **P** равен **80**, а **336** - это **256 + 80**. Данное число, очевидно, интерпретируется по модулю **256**. (Это математический термин, обозначающий остаток от деления числа на **256**.) Другими словами, всякий раз при получении чис ла, кратного 256, отсчет начинается сначала, и 256 рассматривается как 0, 257 - как 1, 511 - как 255, 512 - как 0, 513 - как 1 и т. д.

И наконец, попытаемся напечатать число (65616), превышающее максимальное значение, которое могут принимать данные типа **int** в нашей системе (32767):

```
printf(" %1d %d\n" , 65616, 65616);
```

Результат будет выглядеть так:

```
65616   80
```

Мы снова видим, что действия выполняются по "модулю" На этот раз счет ведется группами по 65536. Числа между 32767 и 65536 будут выводиться на печать как отрицательные из-за способа их представления в памяти машины. Системы с разными размера ми ячеек памяти, отводимых под данные целого типа, ведут себя в общем одинаково, но при этом дают разные числовые значения.

Мы не исчерпали всех возможных комбинаций данных и спецификаций преобразования, поэтому вы можете пытаться экспериментировать сами. Но будет лучше, конечно, если вы сможете за ранее предсказать результат, который будет получен при печати данных, когда используется какая-нибудь спецификация преобразования, выбранная вами.

Применение функции scanf( )

Далее Содержание

Поскольку в дальнейшем мы будем пользоваться функция **scanf( )** лишь эпизодически, мы рассмотрим здесь только основные особенности ее применения.

Так же как для функции **printf( )**, для функции **scanf( )** указыва ются управляющая строка и следующий за ней список аргументов. Основное различие двух этих функций заключается в особенности данного списка. Функция **printf( )** использует имена переменных константы и выражения, в то время как функция **scanf( )** - только указатели на переменные. К счастью, при применении этой функ ции мы ничего не должны знать о таких указателях. Необходимо помнить

только два правила:

1. Если вам нужно ввести некоторое значение и присвоить его переменной одного из основных типов, то перед именем переменной требуется писать символ **&**.

2. Если вы хотите ввести значение строковой переменной, использовать символ **&** не нужно.

Приведем правильную программу

```
main( )
{
int age;
float assets;
char pet [30];
printf(" Укажите ваш возраст, состояние и любимое животное.\n" );
scanf(" %d  %f" , &age, &assets);
scanf(" %s" , pet);  /* & отсутствует при указании массива
символов */ printf("%d  $%.0f  %s\n", age, assets, pet);
}
```

Вот пример диалога:

```
Укажите ВАШ ВОЗРАСТ, состояние и любимое животное.
82
8345245.19  носорог
82  $8345245  носорог
```

Функция **scanf( )** использует некоторые специальные знаки (про белы, символы табуляции и "новая строка") для разбиения входного потока символов на отдельные поля. Она согласует последова тельность спецификаций преобразования с последовательностью полей, опуская упомянутые специальные знаки между ними. Обратите внимание, что наша входная информация располагается на двух строках. Точно так же мы могли бы использовать одну или пять строк при условии, что вводимые величины разделяются по крайней мере одним знаком типа "новой строки", пробела или символа табуляции. Единственным исключением из этого является спецификация **%c**, обеспечивающая чтение каждого следующего символа даже в том случае, если это "пустой символ".

Функция **scanf( )** использует практически тот же набор символов спецификации преобразования, что и функция **printf( )**. Основные отличия в случае функции **scanf( )** следующие:

1. Отсутствует спецификация **%g**.

2. Спецификации **%f** и **%e** эквивалентны. Обе спецификации до пускают наличие (или отсутствие) знака, строки цифр с десятичной точкой или без нее и поля показателя степени.

3. Для чтения целых чисел типа **short** применяется спецификация **%h**.

Функция **scanf( )** не является одной из наиболее часто используемых функций языка Си. Мы обсуждаем ее здесь главным образом из-за ее универсальности (она позволяет читать данные всех имею щихся типов); однако в Си имеется еще несколько других функций, осуществляющих ввод, например **getchar( )** и **gets( )**, которые более удобны для выполнения конкретных задач - чтения одиночных символов или строк, содержащих пробелы. Мы рассмотрим неко торые из этих функций в гл. 6, 13 и 15.

СОВЕТЫ ПО ПРИМЕНЕНИЮ

Далее Содержание

Задание фиксированной ширины полей оказывается полезным при печати данных столбцами. Поскольку шириной поля по умолчанию является "ширина" числа, при повторном использовании оператора

```
printf(" %d %d %d\n" , val1, val2, val3);
```

будут получены неровные столбцы чисел, если эти числа состоят из разного количества цифр.

Например, результат мог бы выглядеть следующим образом:

```
12  234  1222
4  5  23
22334  2322  10001
```

(Здесь предполагается, что между обращениями к оператору печати значения переменных изменялись.)

Эти же данные можно представить в улучшенном виде, если за дать достаточно большую фиксированную ширину поля. При ис пользовании оператора

```
printf( %9d %9d %9d\n" , val1, val2, val3);
```

результат будет выглядеть так:

```
      12     234    1222
        4         5        23
22334   2322  10001
```

Наличие пробелов между спецификациями преобразования гарантирует, что даже в том случае, если все поле будет заполнено, символы, соответствующие данному числу, не перейдут в следующее поле.

Это вызвано тем обстоятельством, что обычные символы, имеющиеся в управляющей строке, включая пробелы, всегда печатаются.

С другой стороны, если печатаемое число включено в некоторую фразу, то часто при его выводе оказывается удобным задать поля равной или меньше требуемой. Это дает возможность включить число в фразу без добавления лишних пробелов. Например результатом работы оператора

```
printf(" Скороход Беппо пробежал %.2f мили за 3 ч.\n", distance);
```

могла бы быть следующая фраза:

```
Скороход Беппо пробежал 10.22  мили за 3 ч.
```

Изменяя спецификацию преобразования на **%10.2f**, получим

```
Скороход Беппо пробежал                  10.22 мили за 3 ч.
```

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Что такое строка символов: несколько символов, расположенных в ряд.
Как записывать строку символов: " несколько символов, расположенных в ряд".
Как строка хранится в памяти: " несколько символов, расположенных в ряд\0".
Где разместить строку: **char phrase[25]** или **static char phrase[25]**.
Как определить длину строки: использовать функцию **strlen(строка)**.
Как распечатать строку: **printf(" %s", phrase)**.
Как прочитать строку, состоящую из одного слова: **scanf(" %s " ,&name)**.
Как задать числовую константу: **#define TWO 2**.
Как задать символьную константу: **#define WOW '!'**.
Как задать строковую константу: **#define WARN "Не делай этого!"**.
Спецификации преобразования при вводе-выводе: **%d %f %e %g %c %s %u % o %x**.
Как улучшить вид входной информации: **%-10d %3.2f**.
Как выполнять преобразования: **printf(" %d %o %c\h", WOW, WOW, WOW);**

Вопросы

1. Выполните снова программу, приведенную в начале данной главы, но на этот раз в ответ на вопрос о вашем имени введите имя и фамилию. Что произойдет? Почему?

2. Что выведет на печать каждый из нижеприведенных программных фрагментов в предположении, что они являются частью некоторой полной программы?

а. printf("Он продал картину за $%2 2f \n", 2 345e2),
б. printf("%c%c%c\n", 'H', 105, '\41'),
в. #define Q "Его Гамлет был смешным, но не вульгарным "
printf("%s\n имеет %d символов \n", Q, strlen(Q)),
г. printf("%2 2e то же самое, что и %2 2f?\n", 1201 0, 1201 0),

3. Какие изменения необходимо внести в программу п. 2в, чтобы строка **Q** была вы ведена на печать заключенной в апострофы?

4. Очередная задача по обнаружению ошибок в программе

```
define в а-яй яй
define X 10
main( )
{ I
int age, char name,
printf(" Укажите, пожалуйста, свое имя ");
scanf(" % s", name);
printf(" Прекрасно, %с, сколько вам лет?\n", name);
scanf(" %f , age), xp = age + X;
printf(" %s  Вам должно быть по крайней мере %d \n", в, xp),
}
```

Ответы

1. "Взрывоопасная" программа Первый оператор **scanf( )** читает ваше имя, остав ляя фамилию непрочитанной; при этом она все таки попадает во входной "буфер" (Этот буфер выполняет функции области памяти, используемой для вре менного хранения поступающих данных). Следующий оператор **scanf( )** должен ввести в программу величину вашего веса, он начинает вводить символы как раз с того места, где завершился предыдущий ввод, и поэтому читает вашу фамилию, принимая ее за вес. В результате в программу попадает "мусор" С другой сторо ны, если вы в ответ на вопрос об имени введете строку типа **"Саша 144"**, то вели чина **144** будет рассматриваться как ваш вес, несмотря на то что вы ввели ее до того, как программа запросила величину веса.

2. а. Он продал картину за 234 50 долл
б. Hi!*Примечание:* первый символ - это символическая константа, второй - деся тичное целое число, преобразованное в символ, а третий - представлен символической константы в коде ASCII.
в. Его Гамлет был смешным, но не вульгарным имеет 41 символ.
г. **1.20E+03** то же самое, что и **1201,00**?

3. Вспомните, что в гл 3 говорилось по поводу управляющих последовательностей, и попробуйте записать оператор в таком виде **printf(" \" %s \" \n имеет %d символов \n", Q, strlen(Q))**.

4. Строка 1: символ # опущен; вместо **a-яй-яй** должно стоять **"a-яй-яй"**
Строка 2: символ # опущен
Строка 6: переменная **name** должна быть массивом, например **char name[25]**
Строка 8: в управляющей строке должен стоять символ **\n**
Строка 10: вместо **%c** должно быть **%s**
Строка 11 поскольку переменная **age** целого типа, необходимо использовать **%d**, а не **%f**, кроме того, вместо **age** должно стоять **&age**
Строка 12: имя **xp** нигде не было описано
Строка 13: правильная, но при выводе на печать результат будет испорчен из-за ошибки, допущенной при определении **B**
Кроме того, программа служит примером плохого стиля программирования.

[1)] ASCII (American Standard Code for Information Interchange) - Американским стандартный код для обмена информацией - *Прим ред*.

[2)] Строкой программы считается строка до запятой - *Прим перев*.

[3)] Пицца - национальный итальянский пирог с несколькими слоями различной начинки - *Прим перев*.

[4)] Каждый файл содержит один из компилируемых модулей программы - *Прим ред*.

---

5. Операции, выражения и операторы

ОПЕРАЦИИ И ОПРАТОРЫ
ВЫПОЛНЕНИЕ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ
ИСПОЛЬЗОВАНИЕ ОПЕРАТОРА while
ВЫРАЖЕНИЯ
ПРОСТЫЕ СОСТАВНЫЕ ОПЕРАТОРЫ
ПРЕОБРАЗОВАНИЯ ТИПОВ

КЛЮЧЕВЫЕ СЛОВА
while

ОПЕРАЦИИ
+ - * / % ++ -- (тип)

ВВЕДЕНИЕ

В гл 3 и 4 мы говорили о типах данных, используемых в языке Си. Здесь же мы рассмотрим способы обработки данных - для этого язык Си имеет широкий набор возможностей. Начнем с основных арифметических операций сложения, вычитания, умноже ния и деления. Чтобы сделать наши программы более интересными и поучительными, мы впервые в этой главе коснемся циклов. А пока, чтобы ввести вас в курс дела, приведем простую программу, выполняющую несложные арифметические действия

```
/* размер обуви1 */
#define OFFSET  7 64
#define SCALE  0 325
main( )
```

```
{
/* пересчет размера обуви в размер ноги в дюймах */
float shoe, foot;
shoe =90;
foot = SCALE*shoe + OFFSET,
printf(" Размер обуви (мужской) размер ноги\n"),
printf(" %10 lf %13 2f дюйм\n" , shoe, foot),
}
```

Здорово в нашей программе выполняется умножение и сложе ние, т.е. берется ваш размер обуви (если вы носите размер 9), а вам сообщается длина стопы в дюймах. Вы скажете, что могли бы решитъ данную задачу в уме за меньшее время, чем потребовалось для вода самой программы в машину. Это, конечно, правильно. Создание программы, способной оперировать только одним размером обуви, выглядит как ненужная трата времени и усилий. Мы бы придать программе большую эффективность, сделав ее диалоговой, но и это окажется непроизводительным использовани ем возможностей машины.

Нам нужно лишь каким то образом заставить компьютер выполнить повторяющиеся вычисления. Вообще говоря, именно эта по является одной из главных причин использования машин. Для выполнения арифметических вычислений. Язык Си предлагает несколько способов реализации повторяющихся вычислений, сейчас обсудим один из них. Данный способ, называемый **"while"**, дает возможность использовать операторы языка более интересным образом. Ниже приводится модификация нашей программы, занимающейся пересчетом размеров обуви.

```
/* размер обуви2 */
#define OFFSET 7 64
#define SCALE 0 325
main()
{
/* пересчет размера обуви в размер ноги в дюймах */
float shoe, foot;
printf("Размер обуви (мужской) размер ноги\n");
 shoe = 3.0;
while(shoe < 18.5)
{
   foot = SCALE *shoe + OFFSET;
   printf(" %10 lf  %13 2f  %l6 2f дюйма\n" , shoe, foot);
   shoe = shoe + 1.0;
}
printf"'Если эта обувь годится вам, носите ее \n");
}
```

Вот результат работы программы, **размер обуви2**, приведенный в сокращенном виде.

```
Размер обуви (мужской)                Размер ноги
3.01                                  8.61 дюйма
4.0                                   8.94 дюйма
...                                   ...
...                                   ...
17.0                                  13.16 дюйма
18.0                                  13.46 дюйма

Если эта обувь годиться вам, носите ее.
```

(Значения констант для данной программы пересчета были по лучены во время нашего визита инкогнито в обувной магазин. В единственном обнаруженном там классификаторе размеров приво дились данные только относительно мужской обуви. Лица, интере сующиеся размерами женской обуви, должны посетить обувной магазин сами).

Цикл **while** работает следующим образом. Когда программа в процессе выполнения впервые достигает оператора **while**, осушествляется проверка истинности условия, заключенного в круглые скобки. В этом случае соответствующее выражение имеет вид:

```
shoe < 18.5
```

где символ **<** означает "меньше". Вначале переменная **shoe** была инициализирована значением **3.0**, которое, как видно, меньше **18.5**. Поэтому данное условие истинно, и осуществляется переход к следующему оператору, который переводит размер обуви в дюймы. После этого результаты выводятся на печать. Следующий оператор

```
shoe = shoe + 1.0;
```

увеличивает значение переменной **shoe** на **1.0**, делая его равным **4.0**. В этом месте программы происходит возврат к началу фрагмента **while**, где вышеупомянутое условие проверяется вновь. По чему именно здесь? Это происходит потому, что следующей строкой программы является закрывающая фигурная скобка **}** - тело цикла **while** заключено в фигурные скобки. Операторы, располо женные между ними, являются той частью программы, которая может выполняться повторно. Теперь давайте вернемся к нашей программе: **4** меньше **18.5** ? Безусловно Поэтому весь набор операторов, заключенный в фигурные скобки и следующий за ключевым словом **while**, выполнится опять. (Специалисты по вычислительной технике в этом случае говорят, что программа выполняет эти операторы "в цикле"). Это продолжается, пока значение переменной **shoe** не достигнет величины **19.0**. Когда условие

```
shoe < 18.5
```

станет ложным, поскольку **19.0** не меньше **18.5**. При этом произойдет передача управления оператору, следующему сразу за телом цикла **while**. В нашем случае им является завершающий оператор **printf()**.

Вы можете легко модифицировать эту программу, чтобы она осуществляла другие преобразования. Например, замените значение константы **SCALE** на **1.8**, а константы **OFFSET** - на **32.0**, и вы изучите программу, которая переводит температуру по Цельсию в температуру по Фаренгейту. Если заменить значение **SCALE** на **0.6214**, а **OFFSET** на **0**, то программа будет переводить мили в километры. Производя эти изменения, вы, по видимому, во избежание путаницы должны будете поменять также и печатаемые сообщения.

Цикл **while** служит удобным и гибким средством управления выполнения программы. Вернемся теперь к обсуждению набора основных операций, которые мы можем использовать в программах.

ОСНОВНЫЕ ОПЕРАЦИИ

Далее Содержание

"Операции" в языке Си применяются для представления арифметических действий. Например, выполнение операции **+** приводит к сложению двух величин, стоящих слева и справа от этого знака. Если слово "операция" кажется вам странным, подумайте тогда, как назвать эти понятия. Нам слово "операция" представляется лучшим термином, чем, скажем, "арифметические транзакторы" мы рассмотрим операции **=**, **+**, **-**, ***** и **/**. (В языке Си нет операции возведения в степень. В одной из следующих глав будет представлена программа, реализующая данную функцию).

Операция присваивания: =

Далее Содержание

В языке Си знак равенства не означает "равно". Он означает операцию присваивания некоторого значения. С помощью оператора

```
bmw = 2002,
```

переменной с именем **bmw** присваивается значение **2002**, т.е. элемент слева от знака **=** - это *имя* переменной, а элемент справа - ее *значение*. Мы называем символ **=** "операцией присваивания". Еще раз хотим обратить ваше внимание на то, что смысл указанной строки не выражается словами "**bmw** равно 2002". Вместо этого нужно говорить так "присвоить переменной **bmw** значение 2002". В

этой операции действие выполняется справа налево.

Возможно, различие между именем переменной и ее значением покажется вам незначительным. В таком случае давайте рассмотрим следующий сравнительно часто используемый при программировании оператор

```
i =i + 1;
```

С математической точкизрения это бессмыслица. Если вы прибавляете единицу к конечному числу, результат не может быть равен исходному числу. Но как оператор присваивания данная строка имеет вполне определенный смысл, который можно выразить, например, такой длинной фразой "Взять значение переменной с именем **i**, к нему прибавить 1, а затем присвоить новое значение переменной с именем **i**".

i = i + 1,

РИС. 5.1.

Оператор вида

```
2002 = bmw,
```

на языке Си не имеет смысла, поскольку **2002** - число. Вы не можете присвоить константе какое-то значение; ее значением является она сама. Поэтому, сидя за клавиатурой, помните, что элемент, стоящий слева от знака **=**, всегда должен быть именем переменной.

Тем из вас, кто предпочитает знать правильные названия понятий, скажем, что вместо использованного ранее термина "элемент" обычно употребляют слово "операнд". Операнды - это то, над чем выполняются операции. Например, вы можете описать процесс "поедания" гамбургера как применение операции "поедание" к операнду "гамбургер"[1)].

Операция присваивания в языке Си представляется несколько более интересной, чем в большинстве других языков. Попробуйте выполнить приведенную ниже короткую программу.

```
/* таблица результатов турнира по гольфу */
main( ) {
int Jane, tarzan, cheeta, cheeta = tarzan = jane = 68;
printf("cheeta tarzan jane\n");
printf("Счет первой партии %4d %8d  %8d \n", cheeta, tarzan, jane);
}
```

В то время как многие языки запрещают применять такое троиное присваивание, присутствующее в данной программе, для Си это обычная практика. Присваивания выполняются справа налево сначала переменная **jane** получает значение **68**, затем переменная **tarzan** и наконец переменная **cheeta**. Результат выглядит так:

```
cheeta  tarzan  jane
Счет первой партии      68      68      68
```

В языке Си имеется несколько других операции присваивания, которые отличаются от операции, описанной в данном разделе, и мы обещаем рассказать о них в следующей главе.

Операция сложения: + Далее Содержание

Выполнение операции **+** приводит к сложению двух величин, стоящих слева и справа от этого знака. Например, в результате работы оператора

```
printf(" %d", 4 + 20);
```

на печать будет выведено число 24, а не выражение

```
4 + 20
```

Операнды могут быть как переменными, так и константами. Поэтому при выполнении оператора

```
income = salary + bribes,
```

компьютер возьмет значения двух переменных, стоящих в правой части, сложит их и присвоит затем полученную сумму переменной **income**.

Операция **+** называется "бинарной", или "диадической". Эти названия отражают тот факт, что она имеет дело с *двумя* операндами.

Операция вычитания: -

Далее Содержание

Выполнение операции вычитания приводит к вычитанию числа, расположеного справа от знака **-**, из числа, стоящего слева от этого знака. Оператор

```
takehome = 224.00 - 24.00;
```

присваивает переменной **takehome** значение 200.

Операция изменения знака: -

Далее Содержание

Знак минус используется также для указания или изменения алгебраического знака некоторой величины. Например, в результате выполнения последовательности операторов

```
rocky = -12;
smokey = -rocky;
```

переменной **smokey** будет присвоено значение 12.

Когда знак минус используется подобным образом, данная операция называется "унарной". Такое название указывает на то, что она имеет дело только с *одним* операндом.

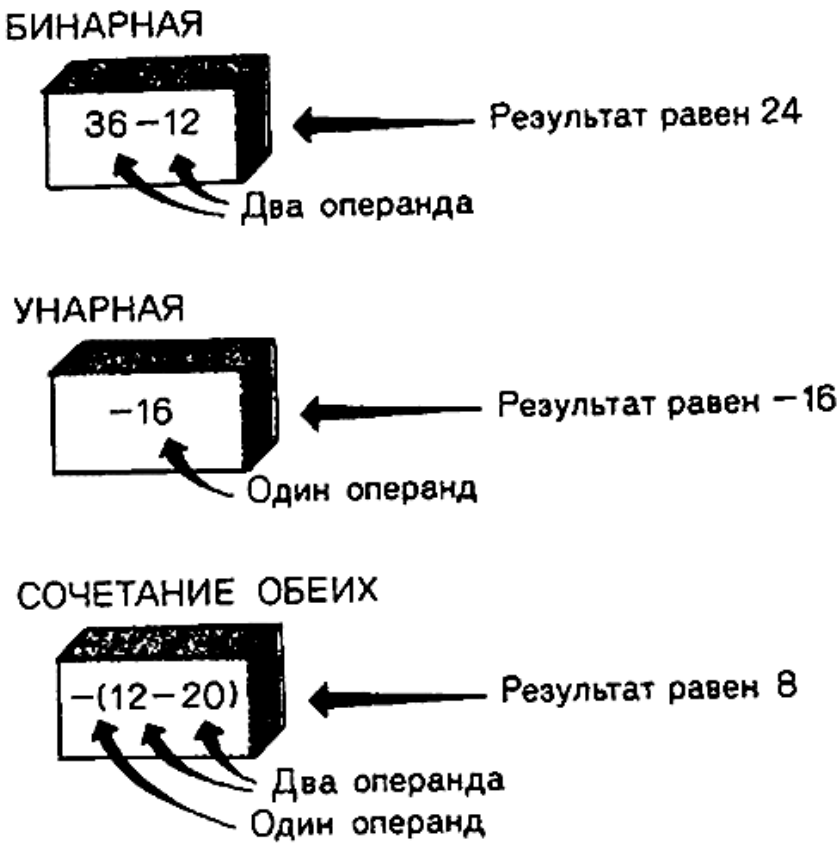


РИС. 5.2. Унарные и бинарные операции

Операция умножения: *

Далее Содержание

Операция умножения обозначается знаком ***. При выполнении оператора

```
cm = 2.54 * in;
```

значение переменной **in** умножается на **2.54**, и результат присваивается переменной **cm**.

Вы хотите иметь таблицу квадратов натуральных чисел? В языке Си нет функции, осуществляющей возведение в квадрат; вместо этого мы можем использовать умножение.

```
/* квадраты чисел */
main( )  /* получение таблицы квадратов */
{
    int num = 1;
    while(num < 21) {
    printf("%10d %10d", n, n*n);
    n = n + 1;
}
```

Эта программа выводит на печать первые 20 целых чисел и их квадраты, в чем вы сами легко можете убедиться. Теперь давайте рассмотрим более интересный пример. Вы, наверное, слышали историю о том, как один могущественный правитель обещал наградить ученого, оказавшего ему большую услугу. Ученый, когда его спросили, что бы он хотел получить в награду, указал на шахматную доску и промолвил: "Положите одно пшеничное зерно на первую клетку, два - на вторую, четыре на третью, восемь на следующую и т. д.". Правитель, которому явно недоставало математической эрудиции, был поражен, услышав такую скромную просьбу,- ведь он уже приготовил богатые дары. Программа, приведенная ниже, показывает, в какое смешное положение

попал правитель. В ней вычисляется количество зерен, которое надо положить на каждую клетку, а также промежуточные результаты (суммы зерен по числу клеток). Поскольку вы, возможно, не знакомы с урожаями пшеницы, мы, кроме того, сравниваем промежуточные суммы зерен с приблизительными цифрами годового урожая пшеницы в США.

```
/* пшеница */
#define SQUARES 64 /* число клеток на шахматной доске */
#define CROP 7E14 /* урожай пшеницы в США, выраженный в числе зерен */
main( ) {
double current, total, int count = 1;

printf(" клетка число зерен сумма зерен доля\n");
printf("от урожая в США\n"),
total = current = 1.0; /* начинаем с одного зерна */
printf("%4d %15.2e %13.2e %12.2e\n", count, current, total, total/CROP);
while (count < SQUARES){
count = count + 1;
current =2.0 * current; /* у двоение числа зерен на следующей клетке*/
total = total + current; /* коррекция суммы */
printf(" %4d %15.2e %13.2e %12 2e\n" , count, current, total, total/CROP); }
}
```

Вначале результаты работы программы выглядят довольно безобидно.

На первых 10 клетках оказалось чуть более тысячи зерен пшеницы. Но давайте посмотрим, сколько зерен на пятидесяти клетках.

Добыча ученого превысила весь годовой урожай пшеницы в США. Если вы захотите узнать, что окажется на 64 клетках, выполните программу сами.

Этот пример иллюстрирует феномен экспоненциального роста возрастание населения земного шара и использование нами энергетических ресурсов подчиняются тому же закону.

Операция деления: /

Далее Содержание

В языке Си символ **/** указывает на операцию деления. Величина, стоящая слева от этого знака, делится на величину, расположенную справа от него. Например, в результате выполнения оператора:

```
four = 12.0/3.0;
```

переемменной **four** будет присвоено значение 4.0. Заметим, что над данными целого типа операция деления производится не так, как над данными с плавающей точкой в первом случае результат будет целым числом, а во втором - числом с плавающей точкой. У целого числа нет дробной части, что делает деление **5** на **3** затруднительным, поскольку результат не является целым. В языке Си принято правило, согласно которому дробная часть у результата деления целых чисел отбрасывается. Это действие называется "усечением".

Попробуйте выполнить приведенную ниже программу, чтобы посмотреть, как осуществляется усечение результата и чем деление чисел отличается от деления чисел с плавающей точкой.

```
/*Примеры деления */
main()
{
printf(" деление целых:  5/4 это %d \n" , 5/4);
printf(" деление целых 6/3 это %d \n" , 6/3);
printf(" деление целых 7/4 это %d \n" , 7/4);
printf(" деление чисел с плавающей точкой 7 /4 это  %2.2f \n", 7 /4 );
printf(" смешанное деление 7 /4 это %2.2f \n" , 7 /4);
}
```

Мы включили в нашу программу также случай "смешанных" типов, осуществляя детение вещественного числа на целое. Язык Си менее строго "подходит" к подобным вопросам, чем некоторые другие языки, и позволяет выполнять такие операции, но, вообще говоря, смещения типов следует избегать. Вот результаты выполнения указанной программы. Обратите внимание на то, что результат деления целых чисел округляется не до ближайшего целого, а всегда до меньшего целого числа. Когда мы смешиваем целые числа и числа с плавающей точкой, результат будет таким же, как если бы оба операнда были числами с плавающей точкой, поскольку в этом случае перед деле нием целое преобразуется в число с плавающей точкой.

Указанные свойства операции деления целых чисел оказываются довольно удобными при решении некоторых задач. Очень скоро мы приведем соответствующий пример. Нам осталось рассмотреть еще один важный вопрос что происходит в тех случаях, когда в одном операторе используется несколько операций? Это и послу жило нам темой обсуждения, приведенного ниже.

Порядок выполнения операций Далее Содержание

Рассмотрим следующую строку:

```
butter = 25.0 + 60.0 * n / SCALE;
```

В этом операторе имеются операции сложения, умножения и деления. Какая операция будет выполнена первой? Будет ли **25.0** складываться с **60.0**, затем результат **85.0** умножаться на **n**, а произведение делиться на значение константы **SCALE**? Или **60.0** умножается на **n**, результат складывается с **25.0**, а сумма затем делится на величину **SCALE**? Или же существует какой-то другой порядок вы полнения операций? Пусть переменная **n** равна **6.0**, а константа **SCALE** - **2.0**. Если вы выполните данные операции, используя эти значения, вы найдете, что при первом способе вычисления результат равен **255**, а при втором - **192.5** При выполнении данной Си программы на машине реализуется, по-видимому, какой-то другой порядок вычислений, поскольку на деле переменная **butter** получит значение **205.0**.

Совершенно очевидно, что изменение порядка выполнения действий может приводить к различным результатам, поэтому язык Си нуждается в наборе непротиворечивых правил, указывающих, какое действие осуществлять первым. Язык Си делает это, за давая приоритет той или иной операции. Каждой операции назна чается уровень старшинства. Умножение и деление имеют более высокий уровень, чем сложение и вычитание, поэтому они выполняются первыми. Если же две операции имеют один и тот же уровень старшинства, они выполняются в том порядке, в котором присутствуют в операторе. Для большинства операций обычный порядок - слева направо. (Операция **=** является исключением из этого правила.) Поэтому в операторе

```
butter = 25.0 + 60.0 * n / SCALE;
```

порядок операций следующий:
60.0 * n - первое умножение (или, возможно, деление) (если **n** = 6, то **60.0 * n = 360.0**).
360.0/SCALE - второе умножение (или, возможно, деление) и наконец (поскольку **SCALE = 2.0**):
25.0 + 180.0 - первое сложение (или, возможно, вычитание) дает **205.0**.

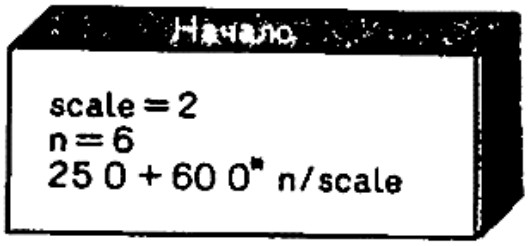
Многие программисты предпочитают представлять порядок вычислений с помощью диаграммы специального вида, называемой "правом выражения". Ниже приводится пример такой диаграммы. Диаграмма показывает, как исходное выражение сводится к одному значению.

Если вы захотите, скажем, чтобы сложение выполнялось перед делением, тогда вы должны делать то же, что и мы в приведенной ниже строке:

```
hour = (25.0 + 60.0 * n) / SCALE;
```

В первую очередь выполняется все, что заключено в скобки; внутри действуют обычные правила. В

данном примере сначала вы умножение, а затем сложение. С помощью этих действий вычисляется выражение в скобках, и только потом результат делится на значение константы **SCALE**.



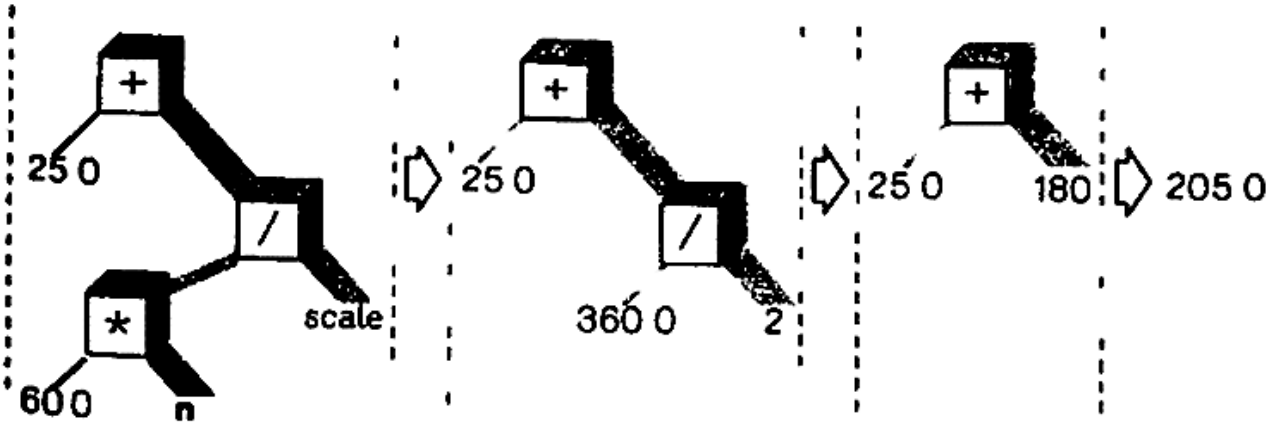


Рис. 5.3. Деревья выражении, построенные на основе операции и операндов, и порядок вычислении.

Мы можем составить таблицу правил, касающихся уже использованных нами операции. (В приложении В в конце книги приведена таблица, где содержатся правила, относящиеся ко всем операциям языка Си.)

Таблица 5.1.

Операции в порядке уменьшения уровня старшинства

ОПЕРАЦИИ	ПОРЯДОК ВЫЧИСЛЕНИЯ
( )	слева направо
-(унарный)	слева направо
* /	слева направо
+ -(вычитание)	слева направо
=	справа налево

Заметим, что два различных по смыслу употребления знака минус имеют разные приоритеты (уровни старшинства). Столбец "порядок вычисления" указывает, как операция связана со своими операндами. Например, унарный знак минус связан с величиной, стоящей справа от него, а при делении левый операнд делится на правый.
Попытаемся применить эти правила на более сложном примере

```
/* применение правил старшинства */
main( ){
```

```
int top, score;
top = score = -(2 + 5)*6 + (4 + 3*(2 + 3));
printi("top = %d \n", top);
}
```

Какое значение будет выведено на печать в результате работы данной программы? Вначале вычислите его сами, а затем выполните программу или прочитайте нижеследующее объяснение, чтобы проверить свой ответ. (Надеемся, что вы получите правильный ре зультат.)

Итак, выражения, стоящие в скобках, имеют наивысший приоритет. Двигаясь слева направо, встречаем первое выражение скобках (2+5). Вычисляя его, получаем

```
top = score = -7*6 + (4 + 3*(2 + 3))
```

Следущее выражение в скобках - это (4 + 3*(2 + 3)). Отбрасываем скобки, получаем 4 + 3*(2 + 3). Вот как! Еще одни скобки! Тогда первым шагом является нахождение суммы 2+3. Выражение примет вид:

```
top = score = -7*6 + (4 + 3*5)
```

Мы должны еще завершить вычисление выражения в скобках. По-скольку умножение ***** имеет приоритет более высокий, чем сложение, выражение теперь выглядит так

```
top = score = -7*6 + (4 + 15)
```

имеем

```
top = score = -7*6 + 19.
```

Что же дальше? Если вы предполагаете, что нужно найти произведение 7*6, то вы ошибаетесь. Заметим, что унарный минус (измениние знака) имеет более высокий приоритет, чем умножение *****. Поэтому сначала число 7 заменяется на -7, а затем -7 умножается на 6. Строка примет вид

```
top = score = -42 + 19
```

после этого в результате сложения получим

```
top  = score = -23
```

Затем переменной **score** присваивается значение -23, и, наконец, переменная **top** получает то же значение -23. Напомним, что операция **=** выполняется справа налево.

НЕКОТОРЫЕ ДОПОЛНИТЕЛЬНЫЕ ОПЕРАЦИИ

Далее Содержание

В языке Си имеется около 40 операций, но некоторые из них используются гораздо чаще, чем другие. Те операции, которые мы только что рассмотрели, являются наиболее общеупотребительными. Кроме того, нам хотелось бы привести еще три полезные операции.

Операция деления по модулю: %

Далее Содержание

Операция деления по модулю используется в целочисленной арифметике. Ее результатом является остаток от деления целого числа, стоящего слева от знака операции, на число, расположенное справа от него. Например, **13 % 5** (читается как "13 по модулю 5") имеет значение 3, поскольку справедливо равенство 13 = 2*5 + 3.

Не пытайтесь производить данную операцию над числами с плавающей точкой она просто не будет выполняться.

На первый взгляд эта операция может показаться некоторым экзотическим средством, используемым лишь математиками, но на самом деле она применяется на практике и довольно

удобна при программировании ряда задач. Одно широко распространенное применение - содействие пользователю в управлении ходом программы. Предположим, например, что вы пишите программу обработки счетов, которая должна предусматривать дополнительную плату раз в три месяца. Для этого нужно только вычислить оста ток от деления номера месяца на 3 (т.е. **month % 3**), проверить, равен ли результат **0**, и, если равен, добавить к счету величину до полнительной платы. После того как вы познакомились с "оператором **if**", вы сможете лучше представить себе, как все это работает.

Приведем пример программы, использующей операцию **%**

```
/* секунды в минуты */
/* переводит секунды в минуты и секунды */
#define SM 60 /* число секунд в минуте */
main( )
{
int sec, mm, left;
printf(" Перевод секунд в минуты и секунды ! \n");
printf(" Укажите число секунд, которое вы хотели бы перевести в минуты \n" );
scanf(" %d", &sec); /* ввод числа секунд */
mm = sec %  SM; /* число минут */
left = sec % SM; /* оставшееся число секунд */
printf(" %d секунды это %d минуты, %d секунды \n", sec, mm, left);
}
```

Вот результат ее работы

```
Перевод секунд в минуты и секунды!
Укажите число секунд, которое вы хотели бы перевсти в минуты.
234
234 секунды это 3 минуты 54 секунды.
```

Недостатком этой диалоговой программы является то, что она обрабатывает только одну входную величину. Сумеете ли вы сами изменить программу так, чтобы она предлагала вам вводить новые значения? Мы вернемся к этой задаче в разделе вопросов в конце главы, но, если вы найдете свое собственное решение, мы будем очень рады.

Операции увеличения и уменьшения: ++ и --

Далее Содержание

Операция увеличения осуществляет следующее простое действие: она увеличивает значение своего операнда на единицу. Существуют две возможности использования данной операции, первая:

когда символы **++** находятся слева от переменной (операнда), - "префиксная" форма,

и вторая:

когда символы **++** стоят справа от переменной, - "постфиксная" форма.

Эти две формы указанной операции различаются между собой только тем, в какой момент осуществляется увеличение операнда. Сначала мы обсудим сходство указанных двух форм, а затем вернемся к различиям. Короткий пример, приведенный ниже, показывает, как выполняется данная операция.

```
/*выполнение сложения */
main( ) /*увеличение префиксная и постфиксная формы */
{
int ultra = 0, super = 0;

while (super < 6) {
    super++;
    ++ultra;
    printf(" super = %d, ultra = %d\n", super, ultra); }
}
```

Результаты работы программы "выполнение сложения" выглядят следующим образом

```
super = 1, ultra = 1
super = 2, ultra = 2
super = 3, ultra = 3
super = 4, ultra = 4
super = 5, ultra = 5
```

Вот это да! Мы досчитали до 5! Дважды! Одновременно! (Если вы захотите считать дальше, вам необходимо будет только изменить параметр, определяющий верхний предел счета в операторе **while**).

Признаемся, что мы могли бы получить тот же результат, заменив два оператора увеличения следующими операторами присваивания

```
super = super + 1, ultra = ultra + 1,
```

Данные операторы выглядят достаточно простыми. В связи с этим возникает вопрос зачем нужен еще один дополнительный оператор, не говоря уже о двух, да еще в сокращенной форме?

Во первых, компактная форма делает ваши программы более изящными и легкими для понимания. Эти операции придают им блеск, что приятно само по себе.
Например, мы можем переписать часть программы **"размер обуви2"** следующим образом.

```
size = 3.0;
while(size < 18.5) {
foot = SCALE*size + OFFSET;
printf("%10.1f %20.2f дюймов\n", size, foot);
++size;
}
```

При этом способе мы еще не воспользовались всеми преимуществами операции увеличения. Мы можем сократить данный фрагмент так

```
size = 2.0;
while(++size < 18.5) {
foot = SCALE *size + OFFSET;
printf( %10.1f  %20.2f дюйма\n" , size, foot); }
```

Здесь мы объединили в одном выражении операцию увеличения переменной на 1 и проверку истинности условия в операции **while**. Подобного типа конструкция настолько часто встречается в языке Си, что заслуживает более подробного рассмотрения. Во-первых, как она работает. Очень просто значение переменнной **size** увеличивается на единицу, а затем сравнивается с 18.5. Если оно меньше, то выполняются операторы, заключенные в фигурные скобки. После этого переменная **size** увеличивается на единицу один раз и т. д. Данный цикл повторяется до тех пор, пока значение переменной **size** не станет слишком большим. Мы изменили значение переменной **size** с 3.0 на 2.0, чтобы скомпенсировать увеличение переменной **size** перед ее первоначальным использованием для вычисления переменной **foot**.

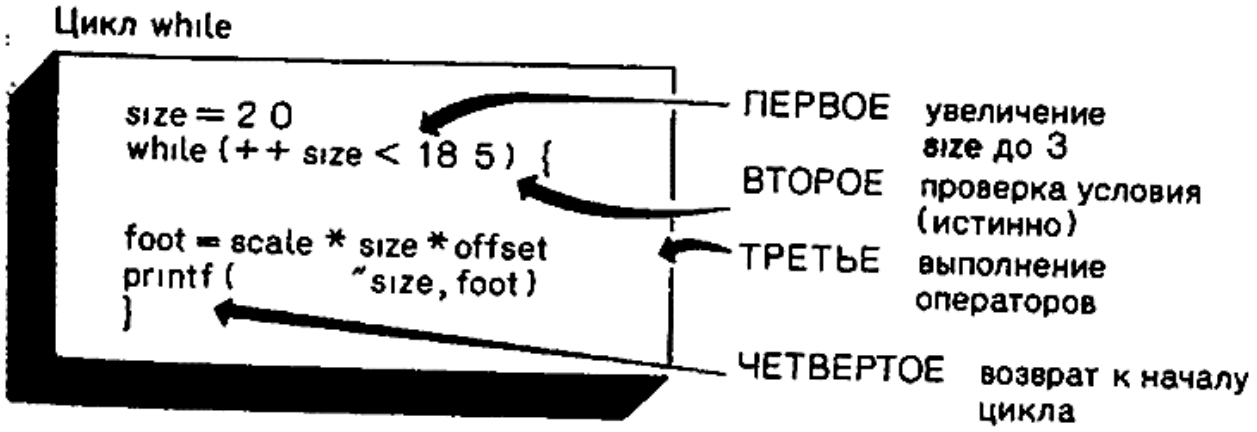


РИС. 5.4. Выполнение одного шага цикла

Во-первых, чем нас привлекает этот способ? Своей компактностью. Кроме того (что еще более важно), с его помощью можно объединить в одном выражении два процесса, управляющие циклом. Первый процесс - это проверка, можно продолжать или нет? В данном случае проверяется, меньше ли размер обуви 18.5. Второй процесс заключается в изменении переменной, значение которой проверяется, - в данном случае размер обуви увеличивается на 1. Предположим, мы забыли изменить ее значение. Тогда переменная **size** *всегда* будет меньше 18.5, и выполнение цикла никогда не закончится. При выполнении программы компьютер, "пойманный в бесконечный цикл", будет выводить на печать одну за другой идентичные строки. В конце концов вы можете потерять интерес, ожидая результатов, и должны будете каким-то образом пре кратить выполнение программы. Наличие проверки и изменения параметра цикла в одном выражении помогает программистам не забывать вводить в программу коррекцию параметра цикла.

Дополнительное преимущество использования операции увеличения заключается в том, что обычно в результате компиляции по лучается несколько более эффективный объектный код, поскольку она идентична соответствующей машинной команде.

И наконец, эти операции имеют еще одну особенность, которую можно использовать в ряде затруднительных ситуаций. Чтобы узнать, что это за особенность, попробуйте выполнить следующую программу:

```
main( )
{
 int a = 1, b = 1;
 int aplus, plusb;
 aplus = a++;  /* постфиксная форма */
 plusb = ++b; /* префиксная форма */
```

```
 printf(" a aplus b plusb \n");
 printf("%5d %5d %5d %5d\n", a, aplus, b, plusb);
}
```

Если вы все сделали правильно, то, насколько мы помним, в качестве результата вы должны получить следующие строки

```
a   aplus   b   plusb
2   1       2   2
```

Как и предполагалось, значения обеих переменных, **a** и **b**, увеличились на 1. Однако переменной **aplus** значение **a** было присвоено перед изменением **a**, в то время как переменной **plusb** значение **b** было присвоено после изменения **b**. В этом и заключается разница между префиксной и постфиксной формами.

aplus = a++ - постфикс: переменная **a** изменяется после того как ее значение используется
plusb = ++b - префикс: переменная **b** изменяется перед тем как ее значение используется

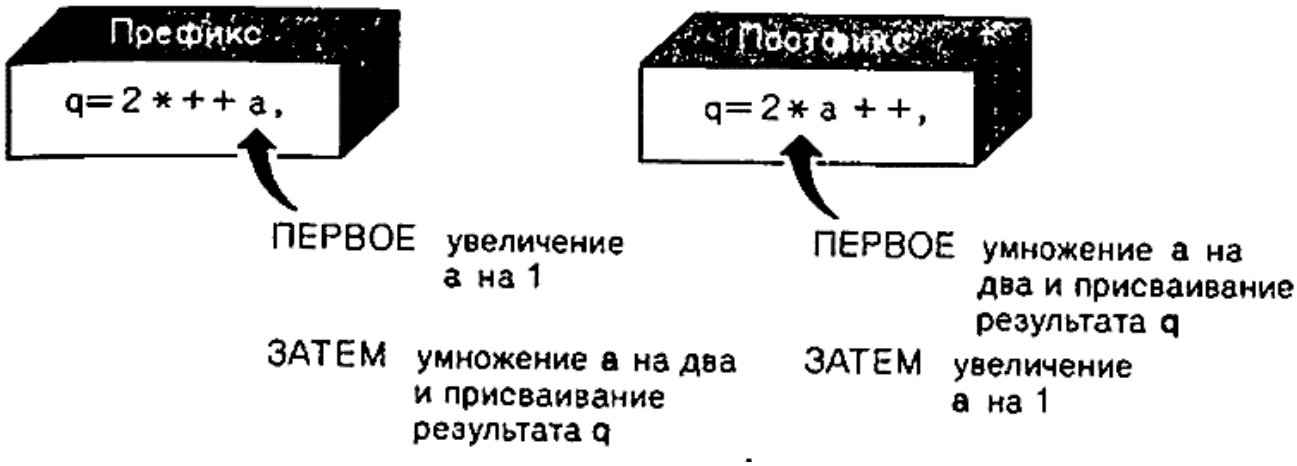


РИС. 5.5. Префиксная и постфиксная формы.

В тех случаях, когда одна из этих операций увеличения используется сама по себе, как, например, в операторе **ego++**, не имеет значения какой формой вы пользуетесь. Выбор приобретает смысл, когда операция и ее операнд являются частью некоторого "высшего" выражения, как, например, в операторах присваивания, которые мы только что рассматривали. В подобной ситуации необходимо иметь представление о результате, который вам хотелось бы получить. Напомним, например, следующий оператор

```
while(++size < 18.5)
```

При его использовании мы получили таблицу перевода вплоть до размера 18. Но, если бы мы вместо этого записали операцию увеличения в виде **size++**, в таблицу попал бы и размер 19, поскольку значение переменной **size** увеличивалось бы после сравнения, а не до этого.

Конечно, вы могли бы использовать менее красивый способ - оператор присваивания

```
size = size +1;
```

Тогда никто не поверит вам, что вы умеете по-настоящему программировать на языке Си.

Думаем, что при чтении книги вы уже обратили внимание на приведенные примеры использования операций увеличения. Как вы думаете, могли ли мы всегда пользоваться любой из них или внешние условия диктовали нам конкретный выбор? Говоря о примерах, нам необходимо привести еще один.

Спят ли когда-нибудь компьютеры? Конечно, спят, но они обычно не рассказывают нам об этом. Программа, приведенная ниже показывает, что происходит в действительности.

```
/* овцы */
#define MAX 40
main( ) {
int count = 0,
printf(" Я считаю овец, чтобы уснуть \n");
while(++ count < MAX)
 printf(" %d миллионов овец, а я еще не уснул \n", count);
printf(" %d миллионов овец, а я хр-р-р р   \n" , count);
}
```

Попробуйте выполнить ее и посмотрите, работает ли она так, как должна по вашему мнению. Конечно, значение константы **MAX** для вашего компьютера можно взять другим. (Кстати, что произойдет призамене префиксной формы операции увеличения постфиксной формой?)

Операция уменьшения: --

Далее Содержание

Каждой операции увеличения соответствует некоторая операция уменьшения, при этом вместо символов **++** мы используем **--**

```
-- count, /* префиксная форма операции уменьшения */
count --, /* постфиксная форма операции уменьшения */
```

Ниже приводится пример, иллюстрирующий, как машины могут быть законченными лириками

```
/* бутылки */
#define MAX 100
main( ) {
int count = MAX + 1;
while(--  count > 0) {
    printf(" %d бутылок пива на полке, %d бутылок пива!\n", count, count);
    printf(" Сними одну и пусти ее по кругу, \n");
    printf("%d бутылок пива! \n \n", count-1); }
}
```

Начальные результаты выглядят так:

```
100 бутылок пива на полке, 100 бутылок пива!
Сними одну ипусти ее по кругу,
99 бутылок пива!
99 бутылок пива на полке, 99 бутылок пива!
Сними одну ипусти ее по кругу,
98 бутылок пива!
```

Постепенно количество бутылок сходит на нет, и программа завершит свою работу следующим образом

```
1 бутылок пива на полке, 1 бутылок пива!
Сними одну ипусти ее по кругу,
0 бутылок пива!
```

По-видимому, у нашего законченного лирика имеются трудности со склонением существительных с количественными числительными, но это можно устранить, используя условные операторы, показываемые в гл. 7. Необходимо заметить, что смысл операции **>** словами выражается как "больше". Так же как и операция **<**, она является "операцией отношения". Подробнее операции отношения мы рассмотрим внизу.

Старшинство операций

Далее Содержание

В соответствии с принятым в языке Си порядком вычислений операции увеличения и уменьшения имеют очень высокий уровень старшинства; только круглые скобки обладают более высоким приоритетом. Поэтому выражение **x*y++** означает **(x)*(y++)**, а не **(x*y)++**, что очень удобно, поскольку последнее выражение смысла не имеет. (Операции увеличения и уменьшения применяются к переменной, в то время как произведение **x*y** само по себе не является переменной в отличие от сомножителей).

Не путайте только старшинство этих операций с порядком вычислений. Предположим, у нас есть последовательность операторов:

```
y  = 2;
n  = 3;
nextnum = (y + n ++ )*6;
```

Какое значение примет переменная nextnum? Подставляя в выражение соответствующие значения, получаем

```
nextnum = (2 + 3)*6= 5*6 = 30
```

Только после того как выражение вычислено, значение переменной **n** увеличивается до **4**. Старшинство операций говорит, что операция **++** имеет отношение только к **n**; кроме того, оно указывает, когда значение переменной **n** используется при вычислении выражения, но момент изменения значения **n** определяется семантикой данной операции.

Не будьте слишком умными

Далее Содержание

Вы можете попасть в глупое положение, если попытаетесь использовать операцию увеличения в неподходящих случаях. Например, вы могли бы захотеть улучшить нашу программу вывода на печать целых чисел и их квадратов, заменив имеющийся там цикл **while** следующей конструкцией :

```
while (num < 21)
{
 printf("%10d  %10d\n", num*num++);
}
```

Эта модификация выглядит разумной. Мы печатаем число **num**, умножаем его само на себя, чтобы получить его квадрат, а затем увеличиваем значение **num** на единицу. На некоторых машинах эта программа даже может работать. Но не на всех. Проблема состоит в том, что при выполнении функции **printf()**, когда определяются печатаемые значения, вычисление последнего аргумента может выполниться сначала, и приращение переменной **n** произойдет до того, как будет определен первый аргумент. Поэтому, вместо, скажем, такой строки

```
5
```

будет напечатано

```
6
```

Правила языка Си предоставляют компилятору возможность выбрать, какой аргумент функции вычислять первым, это повышает эффективность работы компилятора, но может приводить и к некоторым проблемам, если операция увеличения выполняется над одним из аргументов функции.

Другим возможным источником неприятностей служит оператор вида

```
ans = num/2 + 5*(1 + num++);
```

Опять проблема заключается в том, что компилятор может выполнять действия не в том порядке, который вы ожидали. Вы можете считать, например, что сначала он определит значение **num/2**, а затем перейдет к другой части выражения. Но компилятор может вычислить сначала

последний член, увеличить переменною **num**, а затем использовать новое значение при нахождении **num/2**. Никакой гарантии в этом случае не существует.

Избежать эти трудности достаточно просто:

1. Не применяйте операции увеличения или уменьшения к переменной присутствующей в более чем одном аргументе функции.
2. Не применяйте операции увеличения или уменьшения к переменной, которая входит в выражение более одного раза.

ВЫРАЖЕНИЯ И ОПЕРАТОРЫ

Далее Содержание

Мы использовали термины "выражение" и "оператор" на протяжении всех первых глав; теперь настало время изучить их более подробно. Операторы служат основными элементами, из которых строится программа на языке Си; большинство же операторов сосостоит из выражений. Исходя из этого, вначале разумно рассмотреть выражения, что мы и сделаем.

Выражения

Далее Содержание

Выражение представляет собой объединение операций и операндов. (Напомним, что операндом называется то, над чем выполняется операция.) Простейшее выражение состоит из одного операнда отталкиваясь от него, вы можете строить более сложные конструкции. Приведем несколько выражений.

```
4
-64+21
a*(b + c/d)/20
q = 5*2
x = ++q % 3 q > 3
```

Нетрудно заметить, что операнды могут быть константами, переменными или их сочетаниями. Некоторые выражения состоят из меньших выражении, которые мы можем назвать подвыражениями. Например, **c/d** - это подвыражение в нашем четвертом примере.

Важным свойством языка Си является то, что каждое выражение в Си имеет значение. Чтобы определить это значение, мы выполняем операции в порядке, определяемом уровнями старшинства. Значения первых нескольких выражений очевидны, но что можно сказать относительно выражений со знаком **=** ? Они просто имеют те же значения, что и переменная, стоящая слева от знака **=**. Эта переменная получает его в результате вычисления выражения, стоящего справа от знака. А выражение **q > 0**? Подобное выражение, связанное с операцией отношения, имеет значение 1, если оно истинно, и 0, если оно ложно. Приведем несколько выражении и их значения

```
Выражение           Значение
-4+6                2
c = 3 + 8           11
5 > 3               1
6 + (c = 3 + 8)     17
```

Последний пример выглядит довольно странно. Но он полностью соответствует правилам языка Си, поскольку данное выражение представляет собой сумму двух подвыражении, каждое из которых имеет значение.

Операторы

Далее Содержание

Операторы служат основными строительными блоками программы. Программа состоит из последовательности операторов с добавлением небольшого количества знаков пунктуации. Оператор является законченной инструкцией для компьютера. В языке Си указанием на наличие

оператора служит символ "точка с запятой", стоящий в конце него. Поэтому

```
legs = 4
```

это всего лишь выражение (которое может быть частью большего выражения), но

```
legs = 4;
```

является оператором. Что делает инструкцию законченной? Она должна выполнять некоторое действие полностью. Выражение

```
2 + 2
```

не является законченной инструкцией, а служит указанием компьютеру сложить **2** и **2**, но не говорит, что делать с результатом.

```
kids = 2 + 2;
```

служит указанием компилятору (а затем компьютеру) поместить результат (4) в ячейку памяти, помеченную именем **kids**. После записи в память числа 4 компьютер может приступить к выполнению следующих действий.

До сих пор мы познакомились с четырьмя типами операторов. Далее приводится краткий пример, в котором используются все четыре типа.

```
/ * сумма */
main( ) /* нахождение суммы первых 20 целых чисел */
{
int count, sum;   /* оператор описания */
count = 0;   /* оператор присваивания */
sum = 0;    /* то же самое */
while(count++ < 20)   /* while */
    sum = sum + count;   /* оператор */
printf (" sum = %d\n" , sum);   /* вызов функции */
}
```

Давайте обсудим этот пример. К данному моменту оператор описания должен быть вам уже довольно хорошо знаком. Тем не менее мы напомним, что с его помощью определяются имена и типы переменных и им отводятся ячейки памяти.

Оператор присваивания - это основная рабочая сила большинства программ с его помощью переменной присваивается некоторое значение. Он состоит из имени переменной, за которым следует знак операции присваивания **(=)**, а затем выражение, оканчивающееся символом "точка с запятой". Отметим, что оператор **while** включает в себя оператор присваивания. Оператор вызова функции приводит к выполнению последовательности операторов, образующих тело функции. В нашем примере функция **printf( )** вызывается для того, чтобы вывести на печать результаты работы программы.

Оператор **while** состоит из трех различных частей: это ключевое слово **while**, затем проверяемое условие, заключенное в круглые скобки, и, наконец, оператор, который выполняется в том случае если условие истинно. Тело цикла состоит из одного оператора. Он может быть простым, как в данном примере (причем в этом случае не требуется заключать его в фигурные скобки), или составным как в некоторых предыдущих примерах (тогда фигурные скобки абсолютно необходимы). Вы сможете прочесть о составных операторах чуть позже.

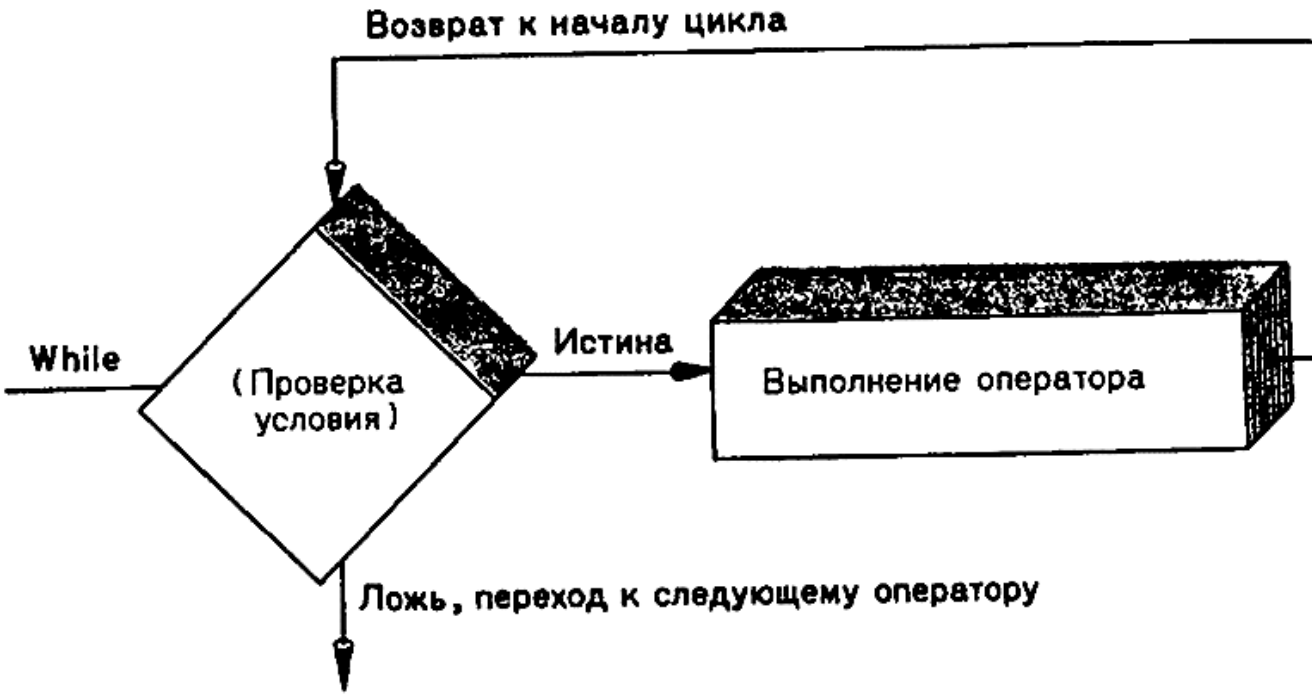


РИС. 5.6. Структура простого цикла while

Оператор **while** принадлежит к классу операторов, иногда называемых "структурированными операторами", поскольку они обладают структурой более сложной, чем структура простого оператора присваивания. В следующих главах мы познакомимся с другими типами структурированных операторов.

Составные операторы (блоки)

Далее Содержание

"Составной оператор" представляет собой два или более операторов, объединенных с помощью фигурных скобок; он называется также "блоком". В нашей программе **размер обуви2** мы использовали такой оператор, чтобы иметь возможность включить в оператор **while** несколько простых операторов. Сравните между собой фрагменты программы:

```
/* фрагмент1 */
index = 0;
while (index ++  < 10) sam = 10 * index +  2;
printf(" sam = %d\n", sam);

/* фрагмент2*/
index = 0;
while(index ++  < 10) {
sam = 10*index + 2;
printf(" sam = %d\n", sam);
}
```

В фрагменте 1 в цикл **while** включен только оператор присваивания. (При отсутствии фигурных скобок область действия оператора **while** распространяется от ключевого слова **while** до следующего символа "точка с запятой".) Печать данных будет произведена только один раз - после завершения цикла.

В фрагменте 2 наличие фигурных скобок гарантирует, что оба оператора являются частью цикла **while**, и печать результатов будет изводиться на каждом шаге работы цикла. Весь составной оператор рассматривается как один оператор, являющийся составной частью оператора **while**.

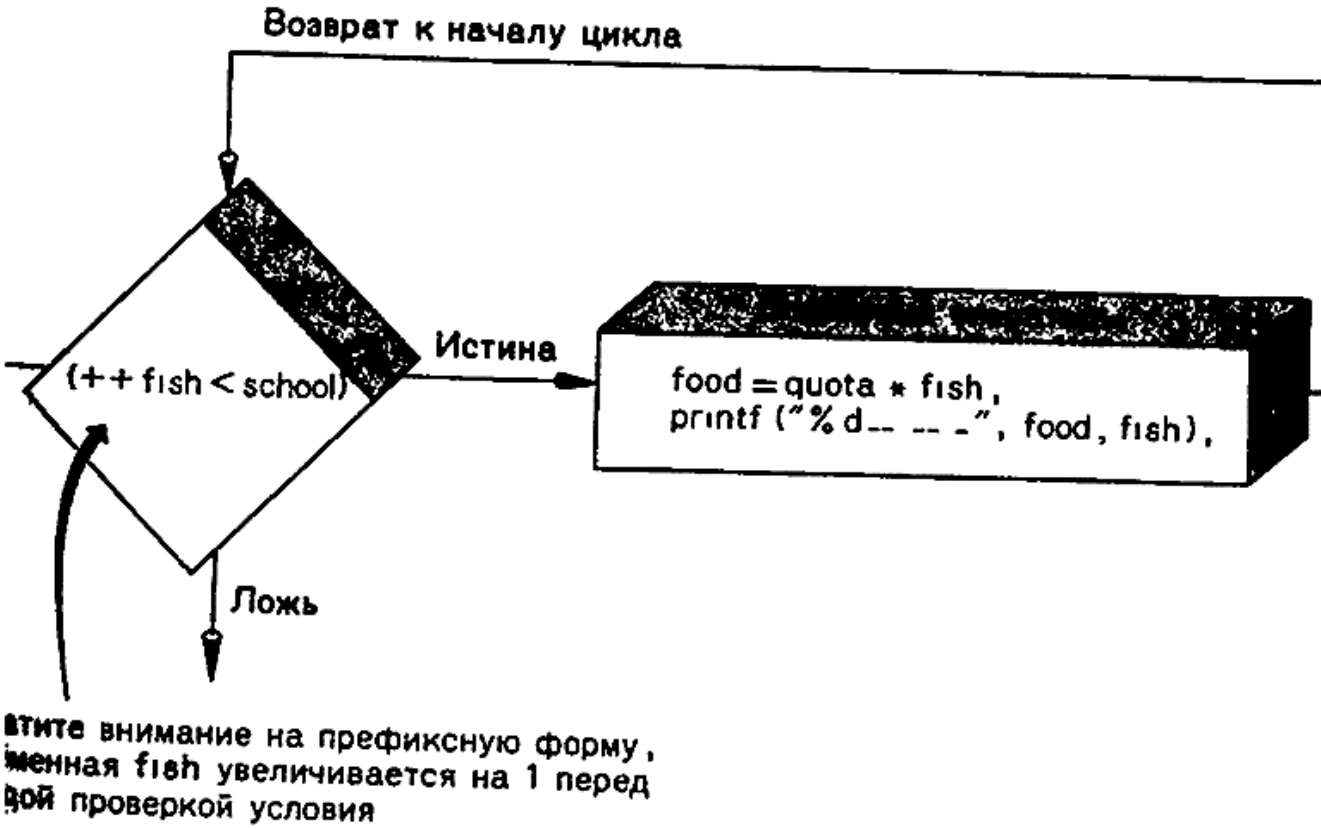


РИС. 5.7. Цикл while с составным оператором

Давайте опять посмотрим на фрагменты, содержащие цикл **while**, и обратим внимание на то, как мы использовали отступы от поля в строках для выделения тела циклов **while**. Для компилятора отступы в строке не имеют никакого значения; решения вопроса о том, как интерпретировать наши инструкции, он использует фигурные скобки и свое "знание" правил формирования структуры цикла **while**. Отступы в строках предназначены для нас, чтобы с первого взгляда можно было понять, как устроена программа. Ранее мы показали вам один популярный способ расстановки фигурных скобок, служащий для указания блока или составного оператора. Другой, тоже довольно распространенный способ выглядит следующим образом

```
while(index++ < 10) {
sam = 10 * index + 2;
printf(" sam = %d\n", sam);
}
```

Этот способ акцентирует внимание на том, что данные операторы образуют блок, в то время как способ, обсуждавшийся выше, указывает на принадлак ность блока оператору **while**. Заметим снова, что поскольку дело касается компилятора, обе формы являются абсолютно идентичными. Подводя итоги, скажем применяйте отступы от поля в строках, чтобы сделать структуру программы наглядной.

Резюме: выражения и операторы

Выражение состоит из операций и операндов. Примерами простейших выражений являются константы или переменные (операция отсутствует), такие, как **22** или **beebop**. Более сложные выражения - это **55 + 22** и **vap = 2*(vip + (mgx = 4))**.

Оператор служит камандой компьютеру. Операторы бывают простыми и составными. *Простые* операторы оканчиваются символом "точка с запятой". Примеры:

```
1. Операторы описания                int toes;
2. Операторы присваивания            toes = 12;
3. Операторы вызова функции          printf(" %d\n", toes);
4. Управляющие операторы             while (toes < 20) toes = toes + 2;
5. Пустой оператор                           ;
```

Составные операторы, или *блоки*, состоят из одного или более операторов (ко торые в свою очередь сами могут быть составными), заключенных в фигурные скобки. Оператор **while**, приведенный ниже, содержит, например, составной оператор

```
while(years < 100) {
wisdom = wisdom + 1;
printf(" %d  %d\n" , years, wisdom);
}
```

ПРЕОБРАЗОВАНИЕ ТИПОВ

Далее Содержание

В операторах и выражениях, вообще говоря, должны использоваться переменные и константы только одного типа. Если все же вы смешаете типы в одном выражении, то компилятор с языка Си не считает программу неправильной, как это произошло бы при программировании на Паскале. Вместо этого компилятор использует набор правил для автоматического преобразования типов. Это очень удобно, но может оказаться и опасным, особенно если вы допустили смешение типов нечаянно. (Например, программа **lint**, работающая в операционной системе UNIX, проверяет несоответствие типов.) Нам представляется разумным привести несколько основных правил, касающихся преобразования типов:

1. Если операция выполняется над данными двух различных типов, обе величины приводятся к "высшему" из двух типов. Этот процесс называется "повышением" типа.

2. Последовательность имен типов, упорядоченных от "высшего" к "низшему", выглядит так: **double**, **float**, **long**, **int**, **short** и **char**. Применение ключевого слова **unsigned** повышает ранг соответствующего типа данных со знаком.

3. В операторе присваивания конечный результат вычисления выражения в правой части приводится к типу переменной, которой должно быть присвоено это значение. Данный процесс может привести к "повышению" типа, как описано выше, или к "понижению, при котором величина приводится к типу данных, имеющему более низкий приоритет.

Повышение" типа обычно происходит гладко, в то время как понижение" может привести к затруднениям. Причина этого проста: все число целиком может не поместиться в элементе данных низшего типа. Переменная типа **char** может иметь целое значение 101, но не 22334. Пример, приведенный ниже, иллюстрирует применение этих правил.

```
/* Преобразования */
main()
{
 char ch;
 int i;
 float fl;
 fl = i  = ch = 'A'; /* строка 8 */
 printf(" ch = %c, i = %d, fl = %2.2f\n", ch, i, fl);
 ch = ch + 1;   /* строка 10 */
 i = fl + 2*ch; /* строка 11 */
 fl = 2.0*ch + 1;       /* строка 12*/
 printf(" ch = %c,      i =  %d, fl = %2.2f\n", ch, i, fl);
 ch = 2.0e30;   /* строка 14 */
 printf(" Теперь ch     =  %c \n" , ch);
}
```

Выполнив программу "преобразования", получим следующие результаты:

```
ch =A, i = 65, fl = 65.00
```

```
ch =B, i = 197, fl = 329.00
Теперь  ch =
```

Вот что происходит в программе.
Строки 8 и 9: Величина **' A'** присваивается символьной переменной **ch**. Переменная **i** получает целое значение, являющееся преобразованием символа **' A'** в целое число, т. е **' 65'**. И наконец, перемен ная **fl** получает значение **65.00**, являющееся преобразованием числа **65** в число с плавающей точкой.
Строки 10 и 13: Значение символьной переменной **'A'** преобразуется в целое число **65**, к которому затем добавляется **1**. После этого получившееся в результате число **66** преобразуется в код символа **B** и помещается в переменную **ch**.
Строки 11 и 13. При умножении на **2** значение переменной **ch** преобразуется в целое число **(66)**. При сложении с величиной переменной **fl** получившееся в результате число **(132)** преобразуется в число с плавающей точкой. Результат **(197.00)** преобразуется в число целого типа и присваивается переменной **i**.
Строки 12 и 13. Перед умножением на **2.0** значение переменной **ch(' B')** преобразуется в число с плавающей точкой. Перед выполнением сложения величина переменной **i(197)** преобразуется в число с плавающей точкой, а результат операции **(329.00)** присваивается переменной **fl**.
Строки 14 и 15: Здесь производится попытка осуществить преобразование типов в порядке убывания старшинства - переменная **ch** полагается равной сравнительно большому числу. Результаты оказываются неутешительными. Независимо от переполнения и усечения, которые имеют место, в итоге на нашей системе мы получили код, соответствующий какому-то непечатаемому знаку.

На самом деле существует еще один вид преобразования типов. Для, сохранения точности вычислений при арифметических операциях все величины типа **float** преобразуются в данные типа **double**. Это существенно уменьшает ошибку округления. Конечный результат, естественно, преобразуется обратно в число типа **float**, если это диктуется соответствующим оператором описания. Вам нет не обходимости заботиться о выполнении подобных преобразований, но должно быть приятно сознавать, что компилятор стоит на страже ваших интересов.

Операция приведения

Далее Содержание

Самое лучшее - это вообще избегать преобразования типов особенно в порядке убывания ранга. Но иногда оказывается удобным применять такие преобразования при условии, что вы ясно представляете себе смысл выполняемых действий. Преобразования типов, которые мы обсуждали до сих пор, выполнялись автоматически. Кроме того, существует возможность точно указывать тип данных, к которому необходимо привести некоторую величину.

Этот способ называется "приведением" типов и используется следующим образом: перед данной величиной в круглых скобках записывается имя требуемого типа. Скобки и имя типа вместе образуют операцию приведения". В общем виде она записывается так

```
(тип)
```

где фактическое имя требуемого типа подставляется вместо слова *"тип"*.
Рассмотрим две нижеследующие строки, где **mice** - это переменная типа **int**. Вторая строка содержит две операции приведения

```
mice = 1.6 + 1.7;
mice = (int)1.6  + (int)l.7;
```

В первом примере используется атоматическое преобразование типов. Сначала числа **1.6** и **1.7** складываются - результат равен **3.3**. Затем путем отбрасывания дробной части полученное число преобразуется в **3** для согласования с типом **int** переменной **mice**. Во втором примере **1.6** и **1.7** преобразуются в целые числа **1**, так что переменной **mice** присваивается значение, равное **1+1**, или

2.

Вообще говоря, вы не должны смешивать типы; вот почему в некоторых языках это запрещено. Но бывают ситуации, когда это оказывается полезным. Философия языка Си заключается в том, чтобы не устанавливать барьеров на вашем пути, но при этом возложить на вас всю ответственность за злоупотребление предоставленной свободой.

Резюме: операции в языке Си

Ниже перечислены операции, которые мы уже обсудили.

```
=           Присваивает величину справа от знака переменной слева от него
+           Прибавляет величину справа от знака к величине слева от него
-           Вычитает величину справа от знака из величины слева от него
-           Унарная операция, изменяет знак величины справа от знака
*           Умножает величину справа от знака на величину слева от него
/           Делит величину слева от знака на величину справа от него.
            Результат усекается, если оба операнда целые числа
%           Дает остаток при делении величины слева от знака на величину
            справа от него (только для целых чисел)
++          Прибавляет 1 к значению переменной слева от знака (префиксная форма)
            или к значению переменной справа от знака (постфиксная форма)
--          Аналогичная операции ++, но вычитает 1
sizeof      Дает размер операнда, стоящего справа, в байтах.
            Операнд может быть спецификацией типа, заключенного в круглые скобки,
            как, например, sizeof (float), или именем конкретной переменной,
            массива и т. п., например sizeof foo
(тип)       Операция приведения: приводит величину, стоящую справа, к типу,
            определяемому ключевым словом (или словами) в скобках. Например,
```

(float)9 преобразует целое число **9** в число с плавающей точкой 9.0.

ПРИМЕР ПРОГРАММЫ

Далее Содержание

На рис. 5.8 приведена программа, которая может оказаться полезной тем, кто занимается бегом, и которая иллюстрирует некоторые положения данной главы. Она выглядит довольно длинной, но все вычисления в ней выполняются шестью операторами, помещенными в конце. Большей частью программа занимается организацией диалога между машиной и пользователем. Мы ввели в программу достаточно большое число комментариев, чтобы сделать ее почти самодокументируемой. Просмотрите ее, а затем мы объясним некоторые ее моменты.

```
/* бег */
# define SM     60 /* число секунд в минуте */
# define SH   3600 /* число секунд в часе */
# define MK  0.62137 /* число миль в километре */
main()
{
float distk, distm;    /* дистанция в км и милях */
float rate;   /* средняя скорость в милях в час */
int min, sec;   /* время бега в минутах и секундах */
int time;  /* время бега в секундах */
float mtime;    /* время пробега одной мили в секундах */
int mmin, msec;    /* время пробега одной мили в минутах и секундах */

printf(" Эта программа пересчитывает ваше время пробега дистанции, выраженной в км, \n");
printf(" во время, требуемое для пробега одной мили, и вашу среднюю \n");
printf(" скорость в милях в час,\n");
printf(" Укажите, пожалуйста, дистанцию в километрах.\n");
scanf(" %f ", &distk);
printf(" Введите затем время в минутах и секундах. \n ");
printf(" Начните с ввода минут. \n");
scanf(" %d", &min);
printf(" Теперь вводите секунды. \n");
scanf(" %d", &sec);
time = SM * mm + sec; /* переводит время в секунды */
distm = MK * distk; /* переводит километры в мили */
rate = distm / time*SH; /* число миль в сек * число
                    сек в час = число миль в час */
mtime = (float)time / distm;  /* время/дистанция = время на милю */
mmin = (int)mtime / SM; /* находит целое число минут */
msec = (int)mtime % SM; /* находит остаток в секундах */
printf("Вы пробежали %1.2f KM (%1.2f мили) за %d мин %d с \n",
distk, distm, mm, sec);
printf(" Эта скорость соответствует пробегу мили за %d : мин", mmin);
printf("%d с.\n Ваша средняя скорость %l.2f миль/ч \n", msec, rate);
}
```

РИС. 5. 8. Программа, полезная для тех, кто занимается бегом

Здесь мы применили тот же подход, который использовали в программе секунды в минуты для перевода времени, выраженного в секундах, в минуты и секунды. Кроме того, нам пришлось воспользоваться преобразованиями типов. Почему? Потому что для той части программы, которая занимается переводом секунд в ми нуты, нам требуются аргументы целого типа, а при преобразовании из метрической системы в мили используются числа с плавающей точкой. Мы применили операцию приведения для того, чтобы сделать эти преобразования явными.

Честно говоря, нашу программу можно было бы написать, используя только автоматическое преобразование типов. Мы так и делали, применяя операцию приведения переменной **mtime** к типу **int**, чтобы при вычислении времени все операнды были целого типа. Однако такая версия

компилятора работает всего на одной из двух доступных нам систем. Использование операции приведения не только проясняет ваш замысел человеку, знакомящемуся с вашей программой, но и упрощает ее компиляцию.

Вот результат работы данной программы.

```
Эта программа пересчитывает ваше время пробега дистанции, выраженной в км,
во время, требуемое для пробега одной мили, и вашу среднюю скорость, в милях в час.
Укажите, пожалуйста дистанцию в километрах.
10,0.
Введите затемвремя в минутах и секундах.
Начните с ввода минут.
36.
Теперь введите секунды.
23
Вы пробежали 10,00 км (6,21 мили) за 36 мин. 23 с.
Эта скорость соответствует пробегу мили за 5 мин 51 с.
Ваша средняя скорость 10.25 миль/ч
```

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Как использовать операции: +,-,*,/,%,++, --, (type).
Что такое операнд: это - величина, над которой выполняется операция.
Что такое выражение: совокупность операций и операндов.
Как вычислять значение выражения: в соответствии с порядком старшинства.
Как распознать оператор: по символу.
Виды операторов: операторы описания, присваивания, **while**, составной.
Как сформировать составной оператор: заключить последовательность операторов в фигурные скобки **{}**.
Как сформируется оператор **while**: **while** (проверка условия) оператор.
Как вычисляются выражения со смешанными типами данных: с помощью автоматического преобразования типов.

ВОПРОСЫ И ОТВЕТЫ

Далее Содержание

1. Предположим, все переменные имеют тип **int**. Определите значение каждой из последующих переменных:
   а. x = (2+3)*6,
   б. x = (12+6)/2*3,
   в. y = x = (2+3)/4,
   г. y = 3 + 2*(x = 7/2),
   д. x = (int)3.8 + 3.3,

2. Мы подозреваем, что в программе, приведенной ниже, имеется несколько ошибок. Сумеете ли вы помочь нам их обнаружить?

```
main( )
{
int i = 1,
float n;

printf(" Внимание! Сейчас появится несколько дробей. \n");
while (i < 30)
    n = 1/ i;
    printf(" %f", n);
printf(" Вот и все! Конец! \n"),
}
```

3. Ниже приведена первая попытка сделать программу "секунды в минуты" диалоговой. Программа нас не удовлетворяет. Почему? Как ее улучшить?

```
#define SM 60
main( )
{
int sec, mm, left,
printf(  Эта программа переводит секунды в минуты и секунды \n );
printf( 'Укажите число секунд \n ),
printf( Для окончания работы программы необходимо ввести 0      \n);
while (sec < 0){
scanf( %d", &sec),
mm = sec/SM,
left = sec % SM,
printf("%d с это % d  мин %d с \n", sec, mm, left),
printf(" Введите следующее значение \n"),
}
printf( До свидания!\n ),
}
```

Ответы

1. а. 30
   б. 27(а не 3). Результат 3 можно получить в случае (12 + 6)/(2*3)
   в. **x** = 1, **y** = 1 (деление целых чисел)
   г. **x** = 3 (деление целых чисел) и **y** = 9
   д. **x** = 6, так как (**int**)3.8=3.3 + 3.3 = 6.3, это число будет преобразовано в число 6, поскольку **x** имеет тип **int**

2. Строка 3: должна оканчиваться точкой с запятой, а не запятой.
   Строка 7: оператор **while** представляет собой бесконечный цикл, потому что величина переменной **i** остается равной 1 и всегда будет меньше 30. По всей видимости, мы собирались написать **while(i+ + < 30)**.
   Строки 7-9: отступы в строках подразумевают, по видимому, что из операторов, расположеных в строках 8 и 9, мы намеревались сформировать блок, но отсутствие фигурных скобок означает, что цикл **while** включает в себя только оператор, расположенный на строке 8; поэтому фигурные скобки должны быть обязательно добавлены.
   Строка 8: поскольку 1 и **i** - оба целого типа, результат деления будет равен 1 при **i**, равном 1, и 0 - для всех больших значений. Необходимо писать так **n = 1.0/i;** перед делением значение переменной **i** будет приведено к типу данных с плавающей точкой и будет получен ненулевой результат.
   Строка 9: мы опустили символ "новая строка" в управляющей строке; это приведет к тому, что числа будут печататься на одной строке, если так допускается устройством вывода.

3. Основная трудность лежит в согласовании между оператором, выполняющим проверку (величина переменной **sec** больше 0 или нет?), и оператором **scanf( )**, осуществляющим ввод значения переменной **sec**. В частности, когда проверка выполняется первый раз, переменная **sec** в программе еще не получает своего значения, и поэтому сравнение будет производиться с некоторой случайной величиной (мусором"), которая может оказаться в соответствующей ячейке памяти. Одно решение, хотя и некрасивое, заключается в инициализации переменной **sec**, скажем величиной 1, в результате чего первый раз сравнение выполнится. Но здесь обнаруживается вторая проблема. Когда при окончании работы мы набираем величину 0, чтобы остановить программу, оказывается, что значение переменной **sec** проверяется только после завершения шага цикла и происходит вывод на печать результатов для 0 секунд. На самом деле нам хотелось бы, чтобы оператор **scanf( )** выполнялся перед тем, как осуществляется проверка в операторе **while**.

Этого можно достичь путем следующей модификации средней части программы

```
scanf(" %d," &sec);
while(sec > 0){
mm = sec / SM;
left = sec % SM;
printf(" %d с это %d мин %d с \n", sec, mm, left);
printf(" Введите следующее значение \n");
scanf(" %d  ", &sec);
}
```

В первый раз ввод указанной величины в программу осуществляется функцией **scanf( )**, помещенной перед циклом, а ввод каждой последующей величины будет выполняться функцией **scanf** в конце цикла (и, следовательно, как раз перед тем, как начнется выполнение очередного шага цикла). Этот подход является общим способом решения проблем подобного сорта.

УПРАЖНЕНИЯ

Ниже приводятся задачи, решения которых мы не даем. Чтобы узнать, работает ли ваша программа, необходимо выполнить ее на вашей машине.
1. Измените нашу программу "сумма" так, чтобы она определяла сумму первых 20 чисел. (Если хотите, можете считать, что эта программа вычисляет, сколько денег вы получите за 20 дней, если в первый день вы получите 1 долл , во второй - 2, в третий - 3 и т.д.). Модифицируйте потом свою программу таким образом, чтобы вы могли в диалоговом режиме указать ей, до какого дня следует вести расчет, т. е. замените константу 20 переменной, значение которой присваивается в результате операции ввода.

2. А теперь модифицируйте свою программу так, чтобы она вычисляла сумму квадратов целых чисел (Или, если вам так больше нравится, сколько вы всего получите денег, если в первый день вам заплатят 1 долл , во второй - 4, в третий - 9 и т. д. Это гораздо более прибыльное дело!) Учтите, что в языке Си нет функции возведения в квадрат, но вы можете использовать тот факт, что квадрат числа n - это просто n*n.

3. Измените свою программу так, чтобы после завершения вычислений она запрашивала у вас новое значение переменной и выполняла вычисления повторно. Окончание работы программы должно происходить при вводе 0. (Указание, используйте такую конструкцию, как цикл в цикле См также вопрос 3 и решение к нему).

[1)] Гамбургер - булочка с котлетой - *Прим перев.*

---

6. Функции и переключение ввода-вывода

ввод - вывод
ФУНКЦИИ *getchar( )* и *putchar( )*
КОНЕЦ ФАЙЛА
ПЕРЕКЛЮЧЕНИЕ < И >
СИСТЕМНО ЗАВИСИМЫЙ ВВОД-ВЫВОД
ЦИКЛЫ РЕАЛИЗУЮЩИЕ ЗАДЕРЖКУ ПО ВРЕМЕНИ

В вычислительной технике слова "ввод" и "вывод" применяются в нескольких разных смыслах. Мы можем говорить об устройствах ввода и вывода, таких, как терминалы, накопители на магнитных

дисках, точечно-матричные принтеры и т. п., или о данных, используемых при вводе и выводе, или же, наконец, о функциях, реализующих ввод и вывод. Основной целью данной главы является обсуждение функций, применяемых при вводе и выводе, но, кроме этого, мы коснемся и двух других аспектов этого понятия.

Под функциями ввода-вывода подразумеваются функции, которые выполняют транспортировку данных в программу и из нее. Мы уже использовали две такие функции: **printf( )** и **scanf( )**. Теперь же рассмотрим несколько других возможностей, предоставляемых языком Си.

Функции ввода-вывода не входят в определение языка Си; их разработка возложена на программистов, реализующих компилятор с языка Си. Если вы являетесь проектировщиком такого компилятора, то можете реализовать любые функции ввода-вывода. Если вычислительная система, для которой вы его создаете, обладает той или иной особенностью, например тем, что каналы ввода-вывода построены на основе портов микропроцессора INTEL 8086, вы можете встроить в нее специальные функции ввода-вывода, ориентированные на эту особенность. Мы рассмотрим пример применения такого подхода в конце данной главы. С другой стороны, выгода использования стандартного набора функций ввода-вывода на всех системах очевидна. Это дает возможность писать "переносимые" программы, которые легко можно применять на разных машинах. В языке Си имеется много функций ввода-вывода такого типа, например **printf( )** и **scanf( )**. Ниже мы рассмотрим функции **getchar( )** и **putchar( )**.

Эти две функции осуществляют ввод и вывод одного символа при каждом обращении к ним. На первый взгляд, выполнение операций подобным образом может показаться довольно странным так как, учитывая все сказанное выше, мы уже можем с легкостью осуществить ввод нескольких символов подряд. Но этот способ ввода данных лучше соответствует возможностям машины. Более того, такой подход служит основой построения большинства про грамм обработки текстов, являющихся последовательностями обычных слов. Мы увидим, как можно применять эти функции в программах, занимающихся подсчетом символов, чтением и копированием файлов. Попутно мы узнаем про буферы, эхо-печать и переключение ввода-вывода.

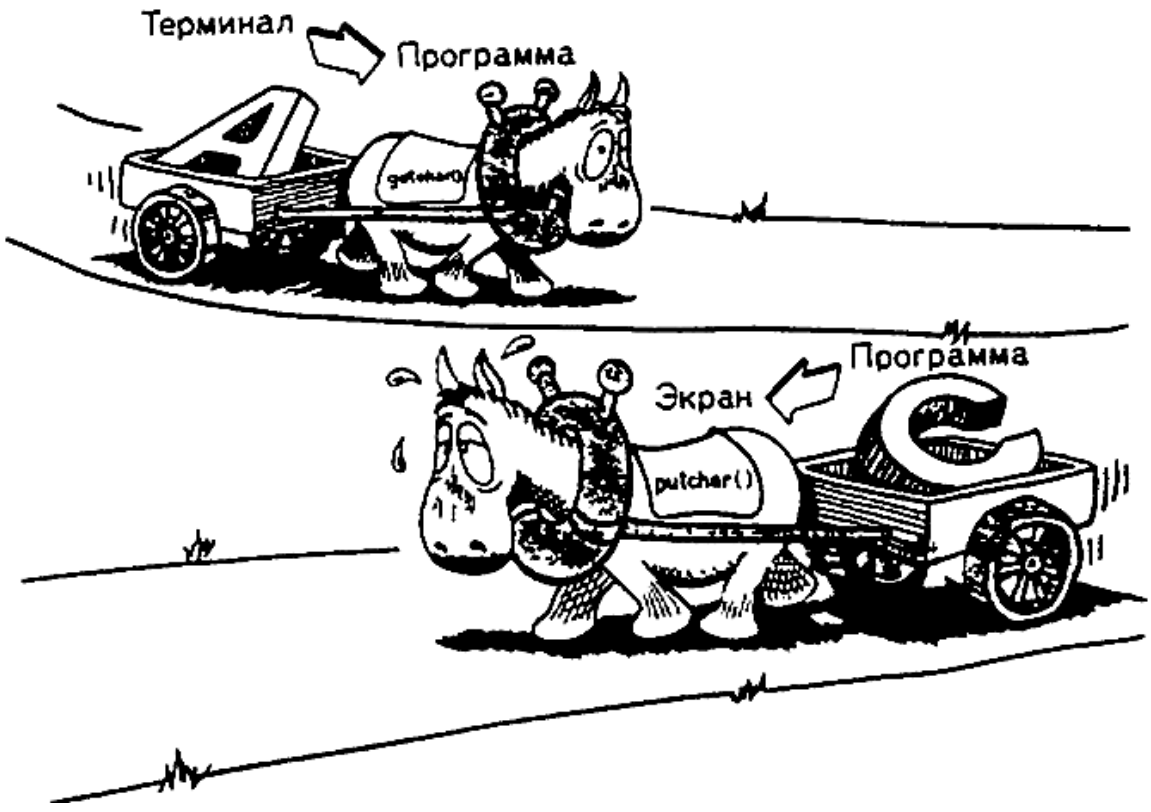


РИС. 6.1. Функции **getchar( )** и **putchar( )** рабочие лошади программы обработки текстов

ВВОД И ВЫВОД ОДНОГО СИМВОЛА: ФУНКЦИИ getchar( ) И putchar( )

Далее Содержание

Функция **getchar()** получает один символ, поступающий с пульта терминала (и поэтому имеющий название), и передает его выполняюшейся в данный момент программе. Функция **putchar( )** получает один символ, поступающий из программы, и пересылает его для вывода на экран. Ниже приводится пример очень простой программы. Единственное, что она делает, это принимает один символ с клавиатуры и выводит его на экран. Мы будем постепенно модифицировать данную программу до тех пор, пока она не приобретет ряд полезных возможностей. Из дальнейшего вы узнаете, что представляют из себя эти возможности, но сначала давайте посмотрим на наш скромный первый вариант

```
/* ввод-вывод1 */
#include
main( )
{
char ch;
ch = getchar( ); /* строка 1 */
putchar (ch);     /* строка 2 */ }
```

Для большинства систем спецификации функций **getchar** и **putchar** содержатся в системном файле **stdio.h**, и только по этой причине мы указали данный файл в программе. Использование такой программы приводит к следующему:

```
g [ввод] g
```

или, возможно, к

```
gg
```

Обозначение [ввод] служит указанием, что вы должны нажать клавишу [ввод]. В любом случае, первый символ **g** вы набираете на клавиатуре сами, а второй выводится компьютером.

Результат зависит от того, есть в вашей системе "буферизованный" ввод или нет. Если перед тем как получить на экране ответ, вы должны нажать клавишу [ввод], то буферизация в вашей системе имеется. Давайте закончим рассмотрение функций **getchar( )** и **putchar( )** перед тем, как приступить к обсуждению понятия буферов.

Функция **getchar( )** аргументов не имеет (т. е. при ее вызове в круглые скобки не помещается никакая величина). Она просто получает очередной поступающий символ и сама возвращает его значение выполняемой программе. Например, если указанная функция получает букву **Q**, ее значением в данный момент будет эта буква. Оператор, приведенный в строке 1, присваивает значение функции **getchar( )** переменной **ch**.

Функция **putchar( )** имеет один аргумент. При ее вызове необходимо в скобках указать символ, который требуется вывести на печать. Аргументом может быть одиночный символ (включая знаки представляемые управляющими последовательностями, описаными в гл. 3), переменная или функция, значением которой является одиночный символ. Правильным обращением к функции **putchar( )** является указание любого из этих аргументов при ее вызове.

```
putchar ('S');      /* напомним, что символьные */
putchar ('\n');     /* константы заключаются в апострофы */
putchar ('\007');
putchar (ch);       /* ch - переменная типа char */
putchar (getchar ( ));
```

Форму записи, приведенную в последнем примере, мы можем использовать для того, чтобы представить нашу программу в следующем виде:

```
#include
main( )
{
        putchar (getchar( ));
}
```

Такая запись очень компактна и не требует введения вспомогательных переменных. Кроме того, в результате компиляции такая программа оказывается более эффективной, но, пожалуй, менее понятной.

После того как мы ознакомились с работой этих двух функций, можно перейти к обсуждению понятия буферов.

БУФЕРЫ

Далее Содержание

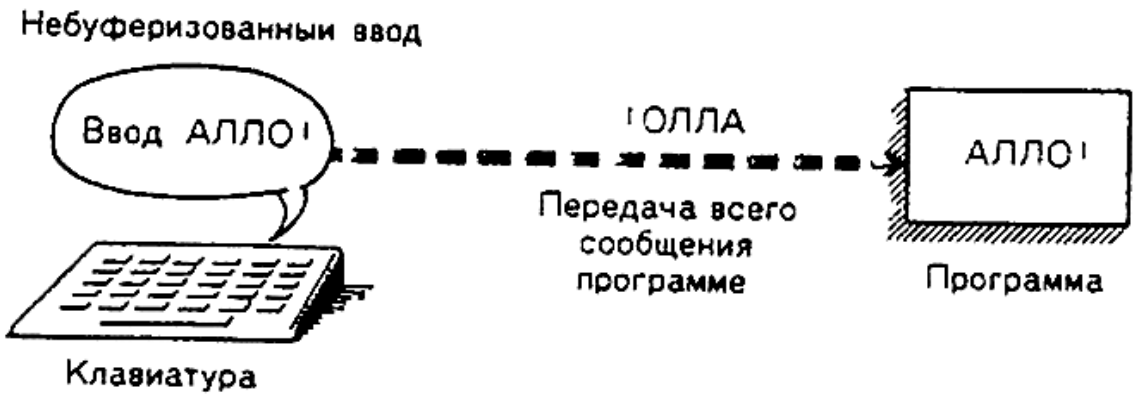
При выполнении данной программы (любой из двух ее версий) вводимый символ в одних вычислительных системах немедленно появляется на экране ("эхо-печать"), в других же ничего не происходит до тех пор, пока вы не нажмете клавишу [ввод]. Первый случай относится к так называемому "небуферизованному" ("прямому") вводу, означающему, что вводимый символ оказывается немедленно доступным ожидающей программе. Второй случай служит примером "буферизованного" ввода, когда вводимые символы собираются и помешаются в некоторую область временной памяти, называемую "буфером". Нажатие клавиши [ввод] приводит к тому, что блок символов (или один символ) становится доступным программе. В нашей программе применяется только первый символ, поскольку функция **getchar( )** вызывается в ней один раз. Например, работа нашей программы в системе, использующей буферизованный ввод, будет выглядеть следующим образом:

```
Вот длинная входная строка. [ввод] в
```

В системе с небуферизованным вводом отображение на экране символа **В** произойдет сразу, как только вы нажмете соответствующую клавишу. Результат ввода-вывода при этом может выглядеть, например, так:

```
Ввот длинная входная строка
```

Символ **В**, появившийся на второй позиции данной строки, - это непосредственный результат работы программы. В каждом случае, программой обрабатывается только один символ, поскопьку функция **getchar( )** вызывается лишь один раз.



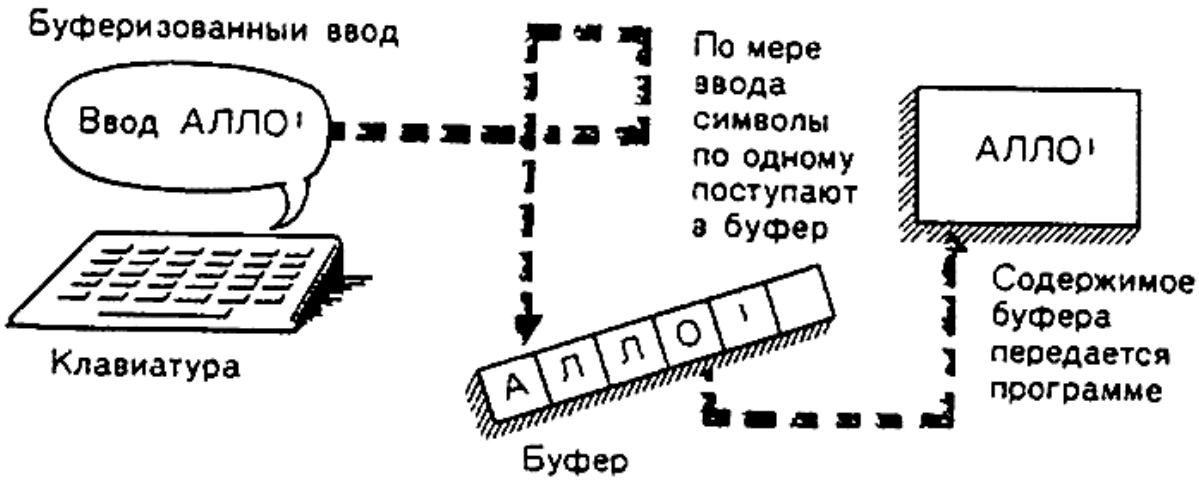


РИС. 6.2. Схема буфернзованного и небуферизованного ввода

Зачем нужны буферы ? Во-первых, оказывается, что передачу нескольких символов в виде одного блока можно осуществить гораздо быстрее, чем передавать их последовательно по одному. Во-вторых, если при вводе символов допущена ошибка, вы можете воспользоваться коректирующими средствами терминала, чтобы ее исправить. И когда в конце концов вы нажмете клавишу [ввод], будет произведена передача откорректированной строки.

Однако для некоторых диалоговых программ небуферизованный ввод может оказаться приемлемым. Например, в программах обработки текстов было бы желательно, чтобы каждая комманда вводилась, как только вы нажимаете соответствующую клавишу. Поэтому как буферизованный, так и небуферизированный ввод имеет свои достоинства.

Вы можете захотеть узнать, какой способ ввода реализован в вашей системе. Для этого необходимо выполнить нашу программу и посмотреть, как будет выглядеть результат ее работы. Некоторые компиляторы с языка Си предоставляют возможность выбора требуемого способа. В нашей микрокомпьютерной системе, например, функция **getchar ( )** реализует буферизированный ввод, между тем как функция **getch( )** - прямой.

СЛЕДУЮЩИЙ ШАГ

Далее Содержание

Теперь возьмемся за что-нибудь несколько более сложное чем чтение и вывод на печать одного символа - например за вывод на печать групп символов. Желательно также, чтобы в любой момент можно было остановить работу программы; для этого спроектируем ее так, чтобы она прекращала работу при получении какого-нибудь специального символа, скажем *****. Поставленную задачу можно решить, используя цикл **while**:

```
/*ввод-вывод2 */
/*ввод и печать символов до поступления завершающего символа*/
#include
#define STOP *  /*дает символу * символическое имя STOP*/
main()
{
char ch;
ch = getchar;  /* строка 9 */
while(ch!= STOP){  /* строка 10 /
putchar (ch);        / * строка 11  */
ch=getchar ();      / * строка 12  */ }
}
```

В данном примере была использована структура программы, обсуждавшаяся нами в конце гл. 5 (вопрос 3). При первом прохождении тела цикла функция **putchar()** получает значение своего аргумента в результате выполнения оператора, расположенного в строке 9; в дальнейшем, вплоть до завершения работы цикла, значением этого аргумента является символ, передаваемый программе функцией **getchar( )**, расположенной в строке 12. Мы ввели новую операцию отношения **!=**, смысл которой выражается словами "не равно". В результате всего этого цикл **while** будет осуществлять чтение и печать символов до тех пор, пока не поступит признак **STOP**. Мы могли бы опустить в программе директиву **#define** и использовать лишь символ ***** в операторе **while**, но наш способ делает смысл данного знака более очевидным.

Перед тем как приступить к выполнению этой замечательной программы на своей машине, взгляните на ее следующий вариант. Программа, приведенная ниже, делает то же самое, но стиль ее написания лучше отвечает духу языка Си:

```
/* ввод-вывод3 */
#include
#define STOP *
main( ) {
char ch;

while ((ch=getchar( )) != STOP) /* строка 8 */
putchar (ch);
}
```

Одна строка 8 этой программы заменяет строки 9, 10 и 12 программы **ввод-вывод2**. Как же работает этот оператор? Начнем с того, что рассмотрим содержимое внутренних скобок:

```
ch = getchar( )
```

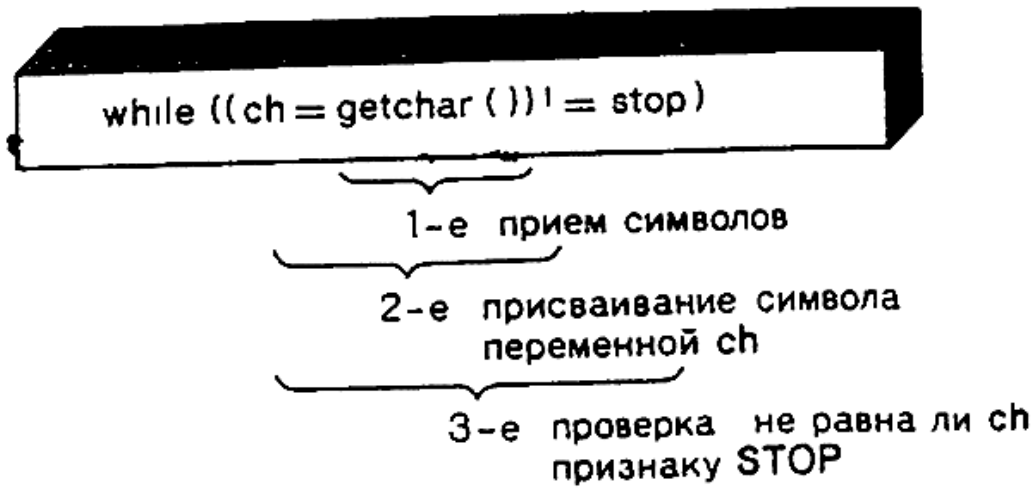
Это - выражение. Его смысл заключается в вызове функции **getchar( )** и присваивании полученного значения переменной **ch**. Одним таким действием мы выполним то, чему в программе **ввод-вывод2** были посвящены строки 9 и 12. Далее напомним, что любое выражение имеет значение и что значение выражения, включающего в себя операцию присваивания, совпадает со значением переменной, расположенной слева от знака **=** . Следовательно, значение выражения **(ch = getchar( ))** - это величина переменной **ch**, так что

```
(ch = getchar( )) ! = STOP
```

имеет то же действие, что и

```
ch != STOP
```

Тем самым выполняется проверка, которую в программе **ввод-вывод2** осуществлял оператор, расположенный в строке 10. Конструкции подобного сорта (объединение в одном выражении операций присваивания и сравнения) довольно часто используются при программировании на языке Си:



Аналогично нашему предыдущему примеру, в котором применя лась конструкция **while (++ size < 18.5)**, данная форма записи обладает тем преимуществом, что позволяет объединять в одном выражении проверку условия окончания цикла и действие по изме-нению одного из операндов операции сравнения. Подобная структура очень напоминает нам рассуждения, которыми мог бы сопровождаться данный процесс: "Я читаю символ, анализирую его и решаю, что делать дальше".

Теперь вернемся к нашей программе и попробуем ее выполнить. Если в вашей системе реализован небуферизованный ввод, результат может выглядеть, например, следующим образом:

```
Иинттеерреесснноо   ррааббооттааеетт   ллии   ооннаа . Думаю что да.
```

При вводе все символы вплоть до признака **STOP** (звездочка), медленно отображаются на экране (эхо-печать). Дублируются даже пробелы. Однако, как только вы ввели признак **STOP**, работа программы прекращается и все, что вы набираете на пульте после этого, появляется на экране без эхо-дублирования.

Теперь посмотрим, что будет происходить в системе, обладающей буферизованным вводом. В этом случае программа не начнет работать до тех пор, пока вы не нажмете на клавишу [ввод]. Вот пример возможного диалога

```
Интересно, работает ли она. Гм , не знаю [ввод].
Интересно, работает ли она.
```

Первая строка была целиком передана программе. Программа последовательно читает эту строку по одному символу и также по одному символу выводит на печать до тех пор, пока не встретит символ ***.

Теперь напишем несколько более полезную программу. Мы заставим ее подсчитывать символы, которые она читает. Нам требуется для этого ввести в предыдущую программу лишь некоторые изменения

```
 /* подсчет символов! */
#define STOP *
main(  )
{
        char ch;
        int count =0;  /* инициализация счетчика символов 0 */
        while ((ch = getchar( ))!= STOP) {
           putchar (ch);
           count++;       /* прибавить 1 к счетчику */
}
printf (" \n Всего было прочитано %d символов \n ' , count);
 }
```

Если мы хотим просто подсчитывать число введенных символов без отображения их на экране, функцию **putchar( )** можно опустить. Эта маленькая программа подсчитывает символы, и нам осталось сделать всего лишь несколько шагов для получения программы, которая будет подсчитывать строки и слова. В следующей главе будут описаны необходимые для этого средства.

Чтение одной строки

Далее Содержание

Давайте подумаем, какие дополнительные усовершенствования можно ввести в программу, используя только те средства, которыми мы владеем. Первое, что легко можно сделать - это заменить признак окончания ввода данных. Но можно ли предложить что-то лучшее, чем символ ***?** Одной из возможностей является использование символа "нова строка" (**\n**). Для этого нужно лишь переопределить признак **STOP**.

```
#define STOP   ' \n '
```

Какой это даст эффект? Очень большой ведь символ "новая строка" пересылается при нажатии клавиши [ввод], следовательно, в результате наша программа будет обрабатывать одну вводимую строку. Предположим, например, что мы внесли указанное изменение в программу **подсчет символов1**, а затем при ее выполнении ввели следующую строку:

```
O! Быть сейчас во Фресно, когда здесь лето, [ввод]
```

В ответ на экране появятся следующие строки

```
O! Быть сейчас во Фресно,  когда здесь лето,  Всего бьпо прсчитано 43 символа
```

(Если бы мы не включили в управляющую строку оператора **printf( )** в качестве первого символа признак **\n**, второе сообщение появилось бы справа от запятой, после слова **лето**. Мы предпочли избежать такого склеивания строк).

Признак, появляющийся в результате нажатия клавиши [ввод] не входит в число символов (**43**), подсчитанных программой, поскольку подсчет осуществляется внутри цикла.

Теперь у нас есть программа, которая может прочесть одну строку. В зависимости от того, какой оператор помещен в тело цикла **while**, программа может осуществлять эхо-печать, подсчет числа символов в строке или и то и другое. Эти средства представляются нам в принципе полезными, скажем как часть некоторой большей программы. Но было бы хорошо иметь возможность читать большие порции данных, например файл данных. Это может быть осуществлено путем надлежащего выбора признака **STOP**.

Чтение одиночного файла

Далее Содержание

Каким может быть идеальный признак **STOP**? Это должен быть такой символ, который обычно не используется в тексте и следовательно, не приводит к ситуации, когда он случайно встретится при вводе, и работа программы будет остановлена раньше чем мы хотели бы.

Проблема подобного сорта не нова, и, к счастью для нас, она уже была успешно решена проектировщиками вычислительных систем. На самом деле задача, которую они рассматривали, была не сколько отличной от нашей, но мы вполне можем воспользоваться их решением. Занимавшая их проблема касалась "файлов". Файлом можно назвать участок памяти, в который помещена некоторая информация. Обычно файл хранится в некоторой долговременной памяти, например на гибких или жестких дисках или на магнитной ленте. Чтобы отмечать, где кончается один файл и начинается другой, полезно иметь специальный символ, указывающий на конец файла. Это должен быть символ, который не может появиться где-нибудь в середине файла, точно так же как выше нам требовался символ, обычно не встречающийся во вводимом тексте. Решением

указанной проблемы служит введение специального признака, называемого "**End-of-File**" (конец файла), или **EOF**, для краткости. Выбор конкретного признака **EOF** зависит от типа системы он может состоять даже из нескольких символов. Но такой признак всегда существует, и компилятор с языка Си, которым вы пользуетесь, конечно же "знает", как такой признак действует в вашей системе.

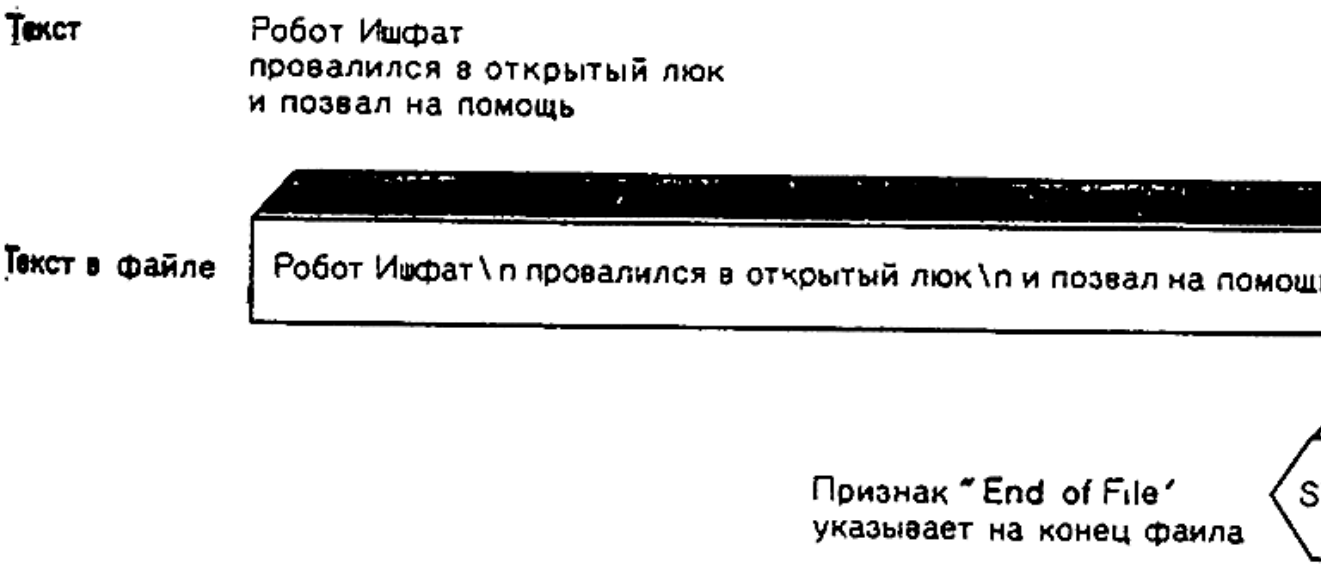


РИС. 6.4. Структура текстового файла с признаком **EOF**

Каким образом можно воспользоваться символом **EOF**? Обычно его определение содержится в файле . Общеупотребительным является определение

```
#define EOF (-1)
```

Это дает возможность использовать в программах выражения, подобные, например, такому

```
while ((ch=getchar( ))!= EOF)
```

Поэтому мы можем переписать нашу предыдущую программу, осуществляющую ввод и эхо-печать символов, так:

```
/* ввод-вывод4 */
#include < stdio.h>
main( )
{
int ch;
while ((ch = getchar( ))! = EOF)
putchar (ch);
}
```

Отметим следующие моменты:

1. Нам не нужно самим определять признак **EOF**, поскольку заботу об этом берет на себя файл **stdio.h.**

2. Мы можем не интересоваться фактическим значением символа **EOF**, поскольку директива **#define**, имеющаяся в файле **stdio.h**, позволяет нам использовать его символическое представление.

3. Мы изменили тип переменной **ch** с **char** на **int**. Мы поступили так потому, что значениями переменных типа **char** являются целые числа без знака в диапазоне от 0 до 255, а признак **EOF** может иметь числовое значение **-1**. Эта величина недопустима для переменной типа **char**, но вполне подходит для переменной типа **int**. К счастью, функция **getchar()** фактически возвращает

значение типа **int**, поэтому она в состоянии прочесть символ **EOF**.

4. Переменная **ch** целого типа никак не может повлиять на работу функции **putchar( )**. Она просто выводит на печать символьный эквивалент значения аргумента.

5. При работе с данной программой, когда символы вводятся с клавиатуры, необходимо уметь вводить признак **EOF**. Не думайте, что вы можете просто указать буквы **E-O-F** или число **-1**. (Число **-1** служит эквивалентом кода ASCII данного символа, а не самим этим символом. Вместо этого вам необходимо узнать, какое представление используется в вашей системе. В большинстве реализаций операционной системы UNIX, например, ввод знака [CTRL/d] (нажать на клавишу [d], держа нажатой клавишу [CTRL]) интерпретируется как признак **EOF**. Во многих микрокомпьютерах для той же цели используется знак [CTRL/z].

Приведем результат[1)] работы программы **ввод-вывод4** в системе, обладающей буферизованным вводом:

```
0на идет во всей красе -
Она идет во всей красе -
Светла, как ночь ее страны.
Светла, как ночь ее страны.
                      Лорд Байрон
                      Лорд Байрон
[CTRL/z]
```

Каждый раз при нажатии клавиши [ввод] производится обработка символов, попавших в буфер, и копия строки выводится на печать. Это продолжается до тех пор, пока мы не введем признак **EOF**.

Давайте остановимся и подумаем о возможностях программы **ввод-вывод4**. Она осуществляет вывод на экран символов независимо от того, откуда они поступают. Предположим, мы сумели сделать так, что программа вводит символы из некоторого файла. В этом случае она будет осуществлять вывод содержимого файла на экран и остановится только тогда, когда достигнет конца файла, поскольку обнаружит признак **EOF**. Или предположим, что у нас есть способ организовать вывод результатов работы программы в некоторый файл. Тогда можно набрать какой-нибудь текст на клавиатуре и при помощи программы **ввод-вывод4** поместить его во внешнюю память. Или мы могли бы выполнить оба действия одновременно: например, осуществить ввод данных из одного файла и переслать их в другой. В этом случае программа **ввод-вывод4** использовалась бы для копирования файлов. Следовательно, наша маленькая программа могла бы просматривать содержимое файлов, создавать новые файлы и получать копии файлов. Неплохо для такой короткой программы! Ключ к решению этих проблем - в управлении вводом и выводом. Это послужит темой представленного ниже обсуждения.

ПЕРЕКЛЮЧЕНИЕ И РАБОТА С ФАЙЛАМИ

Далее Содержание

Понятие ввода-вывода включает в себя функции, данные и устройства. Рассмотрим, например, нашу программу **ввод-вывод4**. В ней используется функция **getchar( )**, осуществляющая ввод, причем устройство ввода - клавиатура (в соответствии с нашим предположением), а входные данные - отдельные символы. Нам бы хотелось сохранить функции ввода и тип данных, но изменить источник их поступления в программу. Зададимся вопросом: откуда программа узнает, где искать входные данные?

По умолчанию Си-программа рассматривает "стандартный ввод" как источник поступления данных. "Стандартным вводом называется устройство, принятое в качестве обычного средства ввода данных в машину. Это может быть устройство чтения данных с магнитной ленты или перфокарт, телетайп или (как мы продолжаем считать) терминал. Современная машина - это послушный инструмент, и мы можем воздействовать на нее так, чтобы она вводила данные из любого источника. В частности, мы можем сообщить программе, что источник входных данных - файл, а не клавиатура.

Существуют два способа написания программ, работающих с файлами. Первый способ заключается в явном использовании специальных функций, которые открывают и закрывают файлы, организуют чтение и запись данных и т. п.; мы не хотим пока касаться этого вопроса. Второй способ состоит в том, чтобы использовать программу, спроектированную первоначально в предположении что данные в нее вводятся с клавиатуры и выводятся на экран, переключить ввод и вывод на другие информационные каналы: например, из файла в файл. Этот способ в некоторых отношениях обладает меньшими возможностями, чем первый, но зато гораздо проще в использовании. Мы изучим понятие переключения в данном разделе.

Операция переключения - это средство ОС UNIX, а не самого языка Си. Но она оказалась настолько полезной, что при переносе компилятора с языка Си на другие вычислительные системы часто вместе с ним переносится в какой-то форме и эта операция. Более того, многие из вновь созданных операционных систем, таких, как MS-DOS 2, включают в себя данное средство. Поэтому, даже если вы не работаете в среде ОС UNIX существует большая вероятность того, что вы в той или иной форме сможете воспользоваться операцией переключения. Мы обсудим сначала возможности этой операции в ОС UNIX, а затем и в других системах.

Предположим, вы осуществили компиляцию программы **ввод-вывод4** и поместили выполняемый объектный код в файл с именем **getput4**. Затем, чтобы запустить данную программу, вы вводите с терминала только имя файла

```
getput4
```

и программа выполняется так, как было описано выше, т. е. получает в качестве входных данных символы, вводимые с клавиатуры. Теперь предположим, что вы хотите посмотреть, как наша программа работает с "текстовым файлом" с именем **words**. (Текстовый файл - это файл, содержащий некоторый текст, т е. данные в виде символов. Это может быть, например, очерк или программа на языке Си. Файл, содержащий команды на машинном языке, например файл, полученный в результате компиляции данной программы, *не* является текстовым. Поскольку наша программа занимается обработкой символов, она должна использоваться вместе с текстовыми файлами.) Все, что для этого требуется - ввести вместо команды, указанной выше, следующую:

```
getput4  < words
```

Символ **<** служит обозначением операции переключения, используемой в ОС UNIX. Выполнение указанной операции приводит к тому, что содержимое файла **words** будет направлено в файл с именем **getput4**. Сама программа **ввод-вывод4** не знает (и не должна знать), что входные данные поступают из некоторого файла, а не с терминала; на ее вход просто поступает поток символов, она читает их и последовательно по одному выводит на печать до тех пор, пока не встретит признак **EOF**. В операционной системе UNIX файлы и устройства ввода-вывода в логическом смысле представляют собой одно и то же, поэтому теперь файл для данной программы является "устройством" ввода-вывода. Если вы попробуете ввести команду

```
getput4  < words
```

то в результате на экране могут появиться, например, следующие строки[2)]. :

```
В одном мгновеньн видеть вечность,
Огромный мир - в зерне песка,
В единой горсти - бесконечность,
И небо - в чашечке цветка.
```

Но мы, конечно, не можем гарантировать, что в файле, который выберете вы, тоже окажется четверостишие Вильяма Блейка.

Теперь предположим (если вы еще не устали и в состоянии что-нибудь предположить), вам хочется, чтобы слова, вводимые с клавиатуры, попадали в файл с именем **mywords**. Для этого вы должны ввести команду

```
getput4 > mywords
```

и начать ввод символов. Символ **>** служит обозначением еще одной операции переключения, используемой в ОС UNIX. Ее выполнение приводит к тому, что создается новый файл с именем **mywords**, а затем результат работы программы **ввод-вывод4**, представляющий собой копию вводимых символов, направляется в данный файл. Если файл с именем **mywords** уже существует, он обычно уничтожается, и вместо него создается новый. (В некоторых реализациях ОС UNIX, однако, вам предоставляется возможность защитить существующие файлы.) На экране в данном случае появятся лишь вводимые вами символы; их же копии будут направлены в указанный файл. Чтобы закончить работу программы, введите признак **EOF**; в системе UNIX это обычно символ [CTRL/d]. Попробуйте воспользоваться описанной здесь операцией. Если вам ничего другого не придет в голову, просто воспроизведите на своей машине пример, приведенный ниже. Знак приглашения, выводимый на экран интерпретатором команд **SHELL**, обозначается здесь символом **%**. Не забывайте оканчивать каждую введенную строку символом [возврат], чтобы содержимое буфера пересылалось в программу.

```
% getput4 > mywords
```

у вас не должно быть трудностей с запоминанием того, какая операция переключения для чего предназначена. Необходимо помнить только, что знак каждой операции указывает на направление информационного потока. Вы можете по ассоциации представлять себе этот знак в виде воронки. [CTRL/d]

После того как введен символ [CTRL/d], программа заканчивает свою работу и возвращает управление операционной системе UNIX, на что указывает повторное появление знака приглашения. Как убедиться в том, что наша программа вообще работала? В ОС UNIX существует команда **ls**, которая выводит на экран имена файлов; обращение к ней должно продемонстрировать вам, что файл с именем **mywords** теперь существует. Чтобы проверить его содержимое, вы можете воспользоваться командой **cat** или запустить заново программу **ввод-вывод4**, направляя в нее на этот раз содержимое входного файла.

```
% getput4 < mywords
```

У вас не должно быть трудностей с запоминанием того, какая операция переключения для чего предназначена. Необходимо помнить только, что знак каждой операции указывает на направление информационного потока. Вы можете по ассоциации представлять себе этот знак в виде воронки.

Комбинированное переключение

Далее Содержание

Предположим теперь, что вы хотите создать копию файла **mywords** и назвать ее **savewords**. Введите для этого команду

```
getput4 < mywords > savewords
```

и требуемое задание будет выполнено. Команда

```
getput4  > savewords < mywords
```

приведет к такому же результату, поскольку порядок указания операций переключения не имеет значения. Нельзя использовать в одной команде один и тот же файл и для ввода и для вывода одновременно.

```
getput4 mywords              НЕПРАВИЛЬНО
```

Причина этого заключается в том, что указание операци, **> mywords** приводит к стиранию исходного файла перед его использованием в качестве входного.

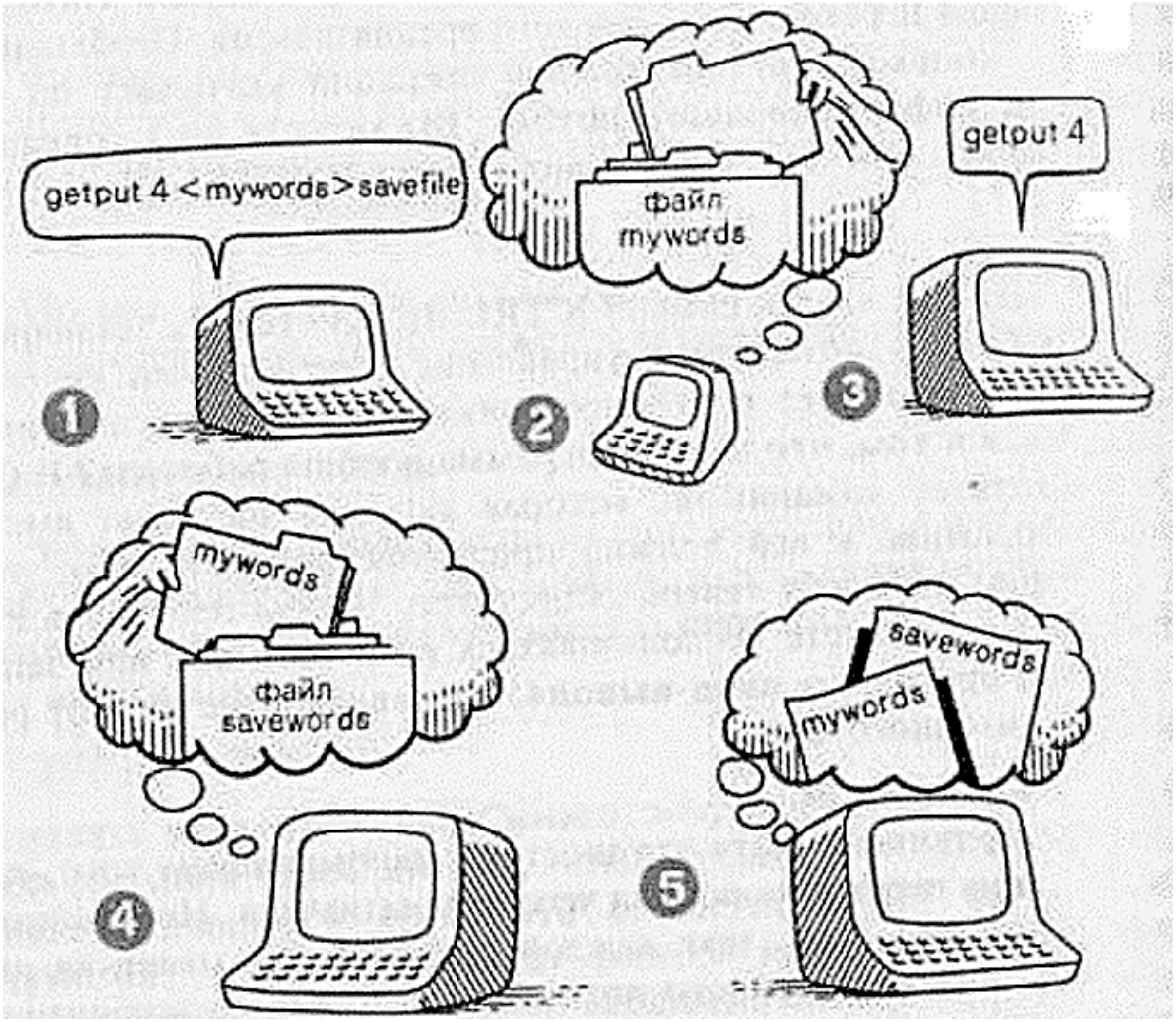


РИС. 6.5. Комбинированное переключение.

Теперь, мы думаем, настало время суммировать правила, касающиеся использования двух операций переключения **<** и **>**.

1. Операция переключения связывает *выполняемую* программу (в том числе и стандартные команды ОС UNIX) с некоторым файлом. Она не может использоваться для связи одного файла с другим или одной программы с другой.
2. Имя выполняемой программы должно стоять слева от знака операции, а имя файла - справа от него.
3. При использовании этих операций ввод не может осуществляться более чем из одного файла, а вывод - более чем в один файл.
4. Обычно между именем и операцией пробелы не обязательны кроме тех редких случаев, когда используются некоторые символы специального назначения в интерпретаторе команд **UNIX**. Мы могли бы писать, например, так: **getput4 < words**, или, что более предпочтительно, **getput4 < words**.

Мы уже привели выше несколько примеров правильного использования операций переключения. Ниже дается несколько ошибочных примеров (**addup** и **count** - выполняемые программы, а **fish** и **stars** - текстовые файлы).

```
fish > stars            Нарушение правила 1
addup < count           Нарушение правила 1
stars > count           Нарушение правила 2
addup < fish < stars    Нарушение правила 3
```

```
count > stars fish      Нарушение правила 3
```

В ОС UNIX применяются также операция **>>**, позволяющая добавлять данные в конец существующего файла, и операция "канал" (**|**), связывающая файл вывода одной программы с вводом другой. Для получения более детальной информации обо всех этих операциях вам необходимо обратиться к руководству по ОС UNIX (по аналогии с этим нам приходит в голову название "ОС UNIX: руководство для начинающих").

Рассмотрим еще один пример: напишем очень простую программу, шифрующую сообщения; с этой целью мы немного изменим программу **ввод-вывод4** и получим

```
/* простой шифр */
/* заменяет каждый символ текста */
/* следующим по порядку из кода ASCII */
#include
main( )
{ int ch;
  while ((ch = getchar ( )) ! = EOF)
  putchar (ch + 1);
}
```

Функция **putchar( )** переводит целое "**ch + 1**" в соответствующий символ. Выполните теперь компиляцию программы и поместите выполняемый объектный код в файл с именем **simplecode**. Затем занеси те приведенные ниже строки[3)] в файл с именем **original**. (Для этого можно воспользоваться системным текстовым редактором или, как было показано ранее, программой **ввод-вывод4**) .

```
Good spelling is an aid
to clear writing.
```

Теперь введите команду

```
simplecode < original
```

Результат должен выглядеть приблизительно так:

```
!!!!!Hppe!tqfmmjoh!jt!bo!bje>Kup!dmfbs!xsjujohl> k
```

Буква **G** заменится на **H**, **o** на **p** и т.д. Вас может удивить следующее: во-первых, что пробелы превратились в восклицательные знаки. Это служит напоминанием, что пробел - такой же символ, как и все остальные. Во-вторых, две строки слились в одну. Почему?

Потому что в тексте, содержащемся в файле **original**, в конце первой строки находиться символ "новая строка", служащий указанием компьютеру начать вывод следующего слова с новой строки. Но этот символ также был изменен. В нашей системе он был заменен символом **^К**, являющимся аналогом специального символа [CTRL/k], и поэтому последующий вывод на печать был продолжен на прежней строке. Если мы хотим иметь программу шифровки сообщений, сохраняющую первоначальную структуру текста по строкам), нам необходимо средство, позволяющее изменять все символы, кроме символа "новая строка". В следующей главе мы узнаем, как это сделать.

Операционные системы, отличные от ОС UNIX

Далее Содержание

Здесь мы главным образом рассмотрим, чем отличаются другие операционные системы от ОС UNIX; поэтому если вы пропустили предыдущий раздел, вернитесь назад и прочтите его.

Все отличия можно разделить на две группы:

1. В других операционных системах реализована операция переключения.
2. Компиляторы с языка Си предоставляют возможность использовать операцию переключения.

Мы не можем рассмотреть все возможные операционные системы, поэтому приведем пример только одной из них, но весьма широко распространенной. Это система MS-DOS 2; она вначале была просто "отпрыском" ОС CP/M, а сейчас самостоятельно развивается в сторону операционной системы XENIX, подобной ОС UNIX. В версию MS-DOS были введены операции переключения **<** и **>**; они работают в ней точно так же, как было описано в предыдущем разделе.

У нас нет возможности рассмотреть все компиляторы с языка Си. Однако в пяти из шести версий компилятора, предназначенных для микрокомпьютеров, с которыми мы имели дело, для указания операции переключения используются символы **<** и **>**. Операция переключения, реализуемая компилятором с языка Си, отличается от аналогичной операции, выполняемой ОС UNIX, в двух аспектах:

1. Указанная операция выполняется при работе программ, написанных только на Си, в то время как в ОС UNIX она может использоватъся при работе любой программы.
2. Между именем программы и знаком операции должен быть один пробел, а между знаком операции и именем файла пробел должен отсутствовать. Ниже приведен пример правильной команды:

```
input4  <words
```

Коментарий

Далее Содержание

Операция переключения - это простое, но мощное средство. С ее помощью мы можем превратить нашу крошечную программу **ввод-вывод4** в инструмент для создания, чтения и копирования файлов. Данный способ служит иллюстрацией подхода, принятого в языке Си (и ОС UNIX) и заключающегося в конструировании простых средств, которые можно комбинировать различным образом для выполнения конкретных задач.

Резюме: как переключать ввод и вывод

Далее Содержание

На большинстве машин, в которых реализован компилятор с языка Си, операцию переключения можно использовать либо для всех программ, благодаря поддержке операционной системы, либо только для программ, написанных на Си, благодаря наличию компилятора с этого языка. Ниже **prog** будет именем выполняемой программы, а **file1** и **file2** - именами файлов.

Переключение вывода в файл: >

```
prog  >file1
```

Переключение ввода в файл: <

```
prog  <file2
```

Комбинирванное переключение:

```
prog  <file2  >filel или prog  >filel  <file2
```

В обеих формах записи файл с именем **file2** используется для ввода данных, а файл с именем **filel** - для вывода.

Расположение пробелов

Некоторые системы (в особенности компиляторы с языка Си) требуют наличия пробела слева от знака операции переключения и его отсутствия справа от этого знака. Другие системы (ОС UNIX например) допускают любое число пробелов (в том числе и ни одного) слева и справа от знака данной операции.

Графический пример

Мы можем воспользоваться функциями **getchar( )** и **putchar( )** для изображения геометрических фигур при помощи символов. Ниже приведена программа которая это делает. Она читает символ, а затем печатает его некоторое число раз зависящее от кода ASCII этого символа. Кроме того она печатает на каждой строке требуемое число пробелов чтобы текст оказывался в центре строки.

```
/* фигуры */
/* изображает симметричную фигуру из символов */
#include
main( )
{
int ch; /* переменная для ввода символа */
int index;
int chnum;

while ((ch=getchar( )) != '\n' ){
chnum - ch %26; /* получение числа от 0 до 25 */
index = 0;
while (index++ <(30 - chnum))
putchar( ); /* печать пробелов сдвига к центру */
index = 0;
while (index++ <(2* chnum + 1))
putchar (ch); /* повторная печать символа */
putchar( \n ); }
}
```

Единственный новый технический прием здесь - это использование подвыражений таких, как (**30-chnum**) при записи условии в циклах **while**. Один цикл **while** управляет печатью необходимого числа начальных пробелов в каждой строке, а второй - выводом символов на печать. Результат работы программы зависит от данных которые вводятся. Если например, вы введете[4].

```
What is up?
```

то на экране появится следующее

```
wwwwwwwwwwwwwwwwww
h
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
ttttttttttttttttttttttt iiiiiiiiiiiiiiiiiiiiiiii
sssssssssssssssssssss

uuuuuuuuuuuuuuuuuuuuuuuuuu ppppppppppppppppp
??????????????????????????
```

Что вы можете делать с помощью этой программы? Можете просто игнори ровать ее, или же (переписав ее по другому) изменять вид фигур которые она выводнт на печать, либо наконец искать такие комбинации входных символов, что в результате на экране будут появляться привлекательные фигуры например при вводе такой последовательности:

```
h i j k l m n o p q r s t u i i i
```

Результат работы программы будет выглядеть так

```
h i i i j j j j j
k k k k k k k l l l l  l l l l l
m m m m m m m m m m  n n n n n n n n n n n n
o o o o o o o o o o o o o o o o
p p p p p p  p p p p p p p p p p p
```

СИСТЕМНО-ЗАВИСИМЫЕ СРЕДСТВА: ПОРТЫ ВВОДА-ВЫВОДА МИКРОПРОЦЕССОРОВ INTEL 8086/8088

Далее Содержание

Рассмотрим различные устройства ввода-вывода, поскольку теперь мы хотим обсудить вопрос о том, как приспособить реализацию компилятора с языка Си к требованиям конкретной вычислительной системы. Многие из современных микрокомпьютеров спроектированы на основе микропроцессорных интегральных схем (ИС) INTEL 8086 и INTEL 8088. Наиболее известным является персональный компьютер IBM PC, в котором применяются ИС второго типа. Конкретный пример, который приведен ниже, относится к упомянутому компьютеру, но обсуждаемые принципы применимы и при рассмотрении других мини-машин, построенных на базе семейства микропроцессоров 8086/8088.

В компьютере типа IBM PC кроме ИС 8088 имеются и другие устройства, например клавиатура, громкоговоритель, возможно, накопитель на мини-кассете или магнитном диске, монитор, встроенная память, таймеры, а также микропроцессоры для управления потоком данных. Центральный процессор (встроенный в кристалл 8088) должен иметь возможность взаимодействовать с остальными частями компьютера. Некоторые из таких взаимодействий осуществляются при помощи адресов памяти, другие - при помощи "портов" ввода-вывода. У микропроцессора 8088 имеется 65536 портов, которые могут использоваться при различных взаимодействиях. Для связи с этим микропроцессором каждому устройству назначается свой определенный порт или порты. (Заметим, что используются не все 65536 портов!) Например, порты 992, 993, 1000-1004 используются для связи с адаптером цветной графики. Работа громкоговорителя управляется портом с номером 97. Это выглядит несколько проще, чем управление адаптером цветной графики, поэтому мы используем его для иллюстрации работы портов ввода-вывода.

Порт 97 не управляет непосредственной работой громкоговорителя. Устройство, осуществляющее эти функции, называется "Программируемый параллельный интерфейсный контроллер 8255". Этот микропроцессор имеет три "регистра" (небольших, легко до ступных элемента памяти), в каждом из которых содержится некоторое число. Числа в регистрах используются для управления работой данного устройства. Каждый регистр связан с ИС 8088 через порт, и регистру, управляющему громкоговорителем, выделен для связи порт 97. С его помощью управление данным устройством осуществляется путем изменения числа в регистре. При посылке правильного" числа громкоговоритель издает звуковой сигнал; посылка же "неправильного" числа может вызвать ряд проблем. Поэтому нам необходимо знать, какие числа требуется посылать и как их нужно посылать. В частности, нам хотелось бы знать, как использовать язык Си для подобного рода операций.

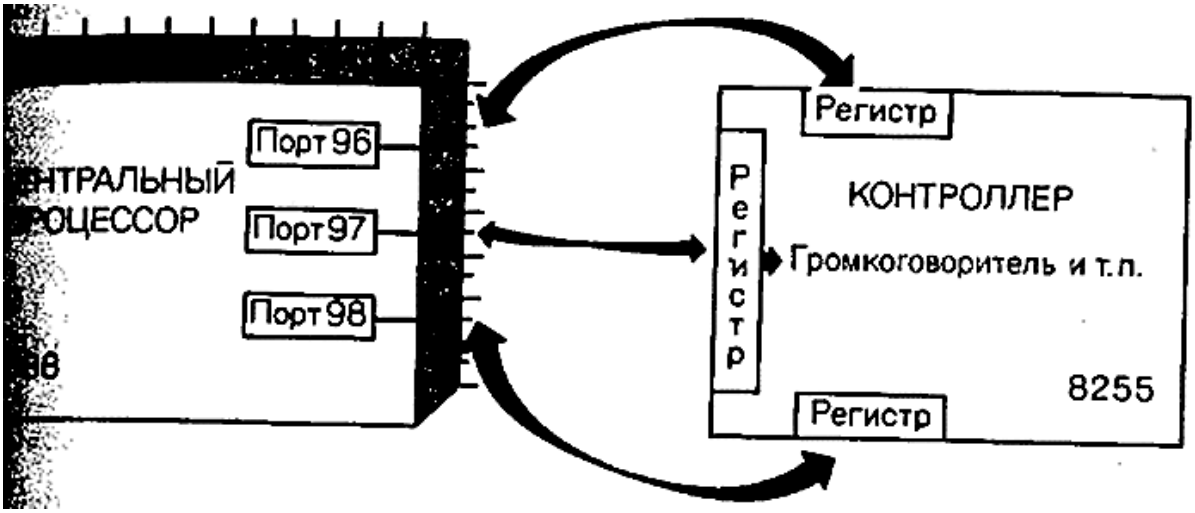


РИС. 6.6. Связь контроллера 8255 с микропроцессором INTEL 8088.

Давайте сначала посмотрим, какие нужно посылать числа. Первым необходимо знать - регистр контроллера 8255 может принять 8- разрядное число, которое помещается туда в двоичном коде, например, 01011011. Каждый из восьми разрядов памяти рассматривается как переключатель "включено-выключено" для соответствующего устройства или воздействия. Наличие 0 или 1 в соответствующей позиции определяет, включено или нет соответствующее устройство. Например, разряд 3 (разряды нумеруются от 0 до 7 справа налево) определяет, включен или нет электродвигатель нателя на мини-кассете, а разряд 7 разрешает или запрещает работу с клавиатурой терминала. При передаче числа в регистр необходимо соблюдать осторожность. Если при включении громкоговорителя мы не обратим внимания на остальные разряды, то случайно можем выключить клавиатуру! Поэтому давайте посмотрим с помощью рис. 6.7, чему соответствует каждый разряд. (Используемая информация взята из технического справочного руководства фирмы IBM, и мы вовсе не должны знать, что большинство из этих разрядов означает.)

```
разряд 0 + включение громкоговорителя через таймер 2

разряд 1 + наличие данных для работы громкоговорителя

разряд 2 + (чтение ключа размера оперативной памяти) или (чтение резервного ключа)

разряд 3 + выключение двигателя накопителя на мини-кассете

разряд 4 - разблокировка оперативной памяти

разряд 5 - разблокировка контроля ввода-вывода

раздяр 6 - поддержание низкой тактовой частоты задающего генератора клавиатуры

разряд 7 - (разблокировка клавиатуры) или + (сброс клавиатуры &
разрешение опроса программно-опрашиваемых переключателей)
```

РИС. 6.7. Порт 97 назначение управляющих разрядов

Обратите внимание на знаки **+** и **-** на рис. 6.7. Знак **+** указывает, что в соответствующем разряде выполнение условия обозначается через 1, а знак **-** указывает, что выполнение условия в разряде обозначается через 0. Поэтому 1 в 3-м разряде показывает, что двигатель накопителя на мини-кассете выключен, в то время как 0 в 4-м разряде указывает на возможность доступа к памяти.

Каким образом можно включить громкоговоритель? Оказывается, для этого необходимо в 0-й разряд (включение громкоговорителя через таймер 2) и в 1-й разряд (наличие данных для работы

громкоговорителя) заслать 1. Это означает, что для включения громкоговорителя через порт 97 необходимо послать в регистр двоичное число 11 (или десятичное число 3). Но, перед тем как приступить к этому, учтите, что данная операция имеет такие побочные эффекты, как, например, установка разряда 4 в 0, что может оказаться вовсе нежелательным. Одна из причин, по которой мы не рассказали, как использовать порты, заключается в том, чтобы предотвратить неприятные последствия вашей поспешности.

Для надежности мы должны проверить сначала, что содержится в регистре. К счастью, это совсем не трудно (мы продемонстрируем это чуть позже). Ответ выглядит так: в регистре обычно содержатся числа "76" или "77 ". Давайте переведем их в двоичную систему. (Здесь вам, возможно, захочется заглянуть в таблицу преобразования в двоичный код, которая приводится в конце книги в приложении.) Результаты преобразования некоторых чисел приве-дены в табл. 6.1.

Не вдаваясь в подробности по поводу значения слов "поддержание низкой тактовой частоты задающего генератора клавиатуры

Таблица 6.1.

Двоичное преобразование некоторых десятичных чисел

Десятичное число	Номер разряда	7	6	5	4	3	2	1	0
76		0	1	0	0	1	1	0	0
77		0	1	0	0	1	1	0	1
78		0	1	0	0	1	1	1	0
79		0	1	0	0	1	1	1	1

Можно сказать, что надежный способ выполнения указанной операции заключается в том, чтобы оставить без изменения значения всех разрядов, кроме нулевого и первого. Это достигается путем передачи в регистр двоичного числа 0100111 (или десятичного 79). В качестве дополнительных мер предосторожности мы должны запомнить исходное значение, содержащееся в регистре, а затем после звукового сигнала громкоговорителя восстановить содержимое указанного регистра. (Битовые операции, рассматриваемые в приложении в конце данной книги, предоставляют другую возможность для занесения некоторого значения в регистр.) Теперь мы готовы к тому, чтобы заставить громкоговоритель подать звуковой сигнал.

Использование порта

Далее Содержание

Существуют две операции, которые могут выполняться с помощью порта: микропроцессор 8088 может послать информацию в подсоединенное устройство или прочитать данные из него. В языке Асемблера эти операции выполняются при помощи команд **OUT** и **IN**, а в языке Си использование указанных средств зависит от компилятора. Некоторые из них предоставляют возможность вызова специальных функций (в соответствии с тем, как это обычно делается в языке Си). В компиляторах Lattice C и Supersoft C, например с этой целью применяются функции **outp( )** и **inp( )**, в других же аналогичные функции могут носить другие имена. Если вы работаете с компилятором, в котором такие возможности отсутствуют для задания указанных функций можно либо воспользоваться ассемблером, либо просто включить в свою программу соответствующий ассемблерный код (что очень просто). В любом случае вам нообходимо ознакомиться с документацией по вашему компилятору. Пока же будем предполагать, что у вас имеется возможность вызова функций **outp( )**

и **inp( )**.

Приведем пример программы, представляющей собой первую попытку извлечь звуковой сигнал из громкоговорителя:

```
/* сигнал1 */
/* заставляет громкоговоритель подавать сигнал */
main( )
{
int store;
store = inp (97);               /* запоминание начального значения с помощью порта 97 */
printf("порт  97 = %d \n", store); /* проверка результатов*/
outp(97, 79);                    /* посылает 79 в порт 97; включение громкоговорителя */
outp(97, store);                  /* восстановление начального значения */
}
```

Несмотря на то что, по-видимому, вы и сами можете догадаться, что выполняют функции **inp( )** и **outp( )**, ниже приведем их формальное описание:

inp(номер порта) — Эта функция возвращает (т. е. формирует) 8-разрядное целое значение (которое преобразуется в 16-разрядное число типа **int** путем добавления нулей слева), полученное из **порта ввода** с указанным **номером**. Обращение к ней не зависит от номера подключенного порта.

oup(номер порта, значение) — Эта функция передает 8-разрядное целое значение в **порт** вывода с указанным **номером**.

Заметим, что один и тот же порт может быть как портом ввода, так и портом вывода в зависимости от того, как он используется.

Давайте теперь выполним программу. В итоге вы можете быть не совсем удовлетворены, поскольку компьютер выключает громкоговоритель довольно быстро после включения. Было бы лучше если бы мы смогли заставить компьютер подождать немного, прежде чем выключить громкоговоритель. Как это можно сделать? Довольно просто! Нужно только дать компьютеру какую-нибудь работу" на это время. Приведенная ниже программа показывает, как этого достичь.

```
/* сигнал2 */
/* более длинный сигнал */
#define LIMIT 10000
int store;
int  count = 0; /* счетчик для организации задержки */
store= inp (97);
outp (97, 79);
while (count++ < LIMIT)
; /* задержка на время работы пустого цикла */
outp (97, store);
```

Заметим, что вся работа оператора **while** состоит в увеличении на каждом шаге цикла) значения переменной **count** до тех пор, пока оно не станет равным величине константы **LIMIT**. Символ "точка с запятой", следующий за оператором **while**, - это "пустой" оператор, который не выполняет никаких действий. Поэтому программа **сигнал2** включает громкоговоритель, считает до 10000, а затем выключает его. Вы можете изменять значение константы **LIMIT** чтобы регулировать продолжительность звучания, или можете заменить константу **LIMIT** переменной и использовать функцию **scanf( )** для ввода соответствующего значения, определяющего продолжительность сигнала.

Было бы прекрасно иметь возможность регулировать и высоту тона. Это и в самом деле осуществимо. После того как мы изучим функции более полно, в приложении в конце книги вы сможете познакомиться с программой, которая превращает клавиатуру терминала в клавиатуру музыкального инструмента.

Мы опять имели дело с устройствами, данными и функциями ввода- вывода. В качестве устройств рассматривались контроллер 8255 и громкоговоритель, в качестве данных - числа, пересылаемые в один из регистров контроллера (а также из него), в качестве функции - функции **inp( )** и **oupt( )**. Использование этих функций или их эквивалентов на ассемблере необходимо в том случае, если мы хотим пользоваться портами ввода вывода ИС INTEL 8086/8088, и компиляторы с языка Си предоставляют нам одну или обе эти возможности.

ИСПОЛЬЗОВАНИЕ СКРЫТОЙ МОЩНОСТИ (В ЛОШАДИНЫХ СИЛАХ) ВАШЕГО КОМПЬЮТЕРА

Далее Содержание

Хотите узнать чудовищный потенциал машины для "перемалы вания чисел"? Как раз для этого мы написали замечательную программу (приведенную на рис 6.8). Чтобы оценить ее полностью вам необходимо выполнить ее на вашем компьютере. *Предупреждение* для получения желаемого эффекта вы должны выбрать подходящую для вашей системы величину константы **LIMIT**. Дополнительные подробности будут обсуждены ниже, а сначала рассмотрим саму программу

```
/* Ганс */
#include
#define LIMIT 8000L
main( )
{
int num1, num2;
long delay =0;
int count = 0;
printf("Лошадь-компьютер Ганс сложит для вас два очень");
printf(" маленьких целых числа \n" );
printf("Укажите, пожалуйста, первое маленькое число \n");
scanf("%d", &num1);
printf("Спасибо А теперь введите второе число \n");
scanf("%d", &num2);
printf("Итак, Ганс, сколько у тебя получится? \n"),
while(delay++ < LIMIT);
while(count++ < num1 + num2 - 1))
{
putchar ('\007' );
delay = 0;
while (delay++  < LIMIT);
putchar ('\n');
}
printf ("Да? И это все? \n");
delay = 0;
while (delay++  < 3*LIMIT);
putchar ('\007');
printf(" Прекрасно, Ганс!\n");
}
```

РИС. 6.8. Программа для "перемалывания чисел"

Технические замечания операторы **while**, в которых содержится переменная **delay**, не делают ничего другого, кроме организации ЗАДЕРЖКИ по времени Символ "точка с запятой" в конце строки показывает на конец тела цикла **while**, т.е. последующие операторы в него не входят. Цикл **while**, использованный внутри другого цикла **whi1e**, называется "вложенным". Мы полагаем, что на IBM PC подходящим значением для константы **LIMIT** является число 8000 AX 11/750 мы предпочитаем число порядка 50000, но на выбор может влиять также уровень загрузки системы, работающей в режиме разделения времени. Мы полагаем **LIMIT** равной значению константы типа **long** (как раз на это и указывает символ **L**, стоящий в конце) для того, чтобы избежать трудностей, связанных с превышением максимального значения величины типа **int** (Для 8000 подобные меры предосторожности на самом деле обязательны, но, например, его замена числом 12000 на IBM PC делает это необходимым, поскольку тогда выражение **3*LIMIT** будет равно 36000, что превышает

максимальное значение величины типа **int** в этой системе).

Если в вашей вычислительной системе отсутствует громкоговоритель или звонок, вы могли бы заменить оператор **putchar('\007')** на **printf ("Стук копыт")**. Эта программа произведет впечатление на ваших друзей и, возможно, успокоит тех, кто боится компьютеров. Мы думаем, такая программа может составить ядро какого-нибудь "Си-вычислителя", но оставляем развитие этой идеи нашим читателям.

ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Что делает функция **getchar()** вводит в программу символ, поступающий с клавиатуры терминала.
Что делает функция **putchar(ch)** отображает символ, содержащийся в переменной **ch**, на экран.
Что символы **!=** означают: не равно.
Что такое **EOF**: специальный символ, указывающий на конец файла.
Как переключить стандартный ввод на ввод из файла:

```
program  < file
```

Как переключить стандартный вывод на вывод в файл:

```
program  > file
```

Что такое порты: средства доступа к подсоединенным устройствам.
Как использовать порты: путем вызова функций **inp( )** и **outp( )**.

ВОПРОСЫ И ОТВЕТЫ

Далее Содержание

Вопросы

1. Выражение **putchar(getchar( ))** является правильным. Будет ли правильным вы ражение **getchar(putchar( ))**?

2. Что произойдет в результате выполнения каждого из следующих операторов?
   а. **putchar('H')**;
   б. **putchar(' \007')**;
   в. **putchar('\n')**;
   г. **putchar(' \b')**

3. Допустим, у вас есть программа **count**, подсчитывающая число символов в файле. Напишите команду, в результате выполнения которой будет произведен подсчет числа символов в файле **essay**, а результат будет помещен в файл **essayct**.

4. Даны программа и файлы, описанные в вопросе 3. Какие из приведенных ниже команд правильны?
   а. **essayct <essay**
   б. **count essay**
   в. **count < essayct**
   г. **essay > count**

5. Что делает оператор **outp(212, 23)**?

Ответы

1. Нет. У функции **getchar( )** аргумент должен отсутствовать, а у функции **putchar( )** аргумент обязательно должен быть.

2.
   а. печать буквы **H**
   б. вывод символа **'\007'**, в результате чего сработает громкоговоритель
   в. переход на новую строку на устройстве вывода
   г. шаг назад на одну позицию.

3. **count** < **essay** > **essayct** или иначе **count** > **essayct** < **essay**

4.
   а. неправильно, поскольку **essayct** не является выполняемой программой
   б. неправильно, поскольку опушен знак операции переключения. (Позже вы научитесь писать программы, для которых не нужно будет использовать операцию переключения)
   в. правильно, при выполнении этой команды число символов, полученное в результате работы программы **count** из вопроса 3, появится в виде сообщения на экране.
   г. неправильно, имя выполняемой программы должно стоять первым

5. Он посылает число **23** через **порт 212**.

УПРАЖНЕНИЯ

1. Напишите программу, описанную в п. 3, т. е. программу, подсчитывающую число символов в файле.

2. Модифицируйте программу **count** так, чтобы при учете каждого символа раздается звуковой сигнал. Введите в программу короткий цикл, реализующий временную задержку, для того чтобы отделить один сигнал от другого.

3. Модифицируйте программу **сигнал2** так, чтобы во время ее выполнения можно было вводить величину, определяющую число повторении тела цикла.

[1)] Используется отрывок из произведения Д. Байрона "Еврейские мелодии" перевод С.Я. Маршака - *Прим. перев.*

[2)] Приводится отрывок из произведения В. Блейка "Прорицания невинности Перевод С.Я. Маршака. - *Прим. перев.*

[3)] Перевод: знание орфографии - залог четкости письма - *Прим. перев.*

[4)] Перевод: что случилось? - *Прим перев.*

---

7. Выбор вариантов

if, else, switch, break, case, default
> >= <= < == != && || / :?

Хотите научиться создавать мощные, "интеллектуальные", универсальные и полезные программы? Тогда вам потребуется язык, обеспечивающий три основные формы управления процессом выполнения программ. Согласно теории вычислительных систем (которая является наукой о вычислительных машинах, а не наукой, создаваемой такими машинами), хороший язык должен обеспечивать реализацию следующих трех форм управления процессом выполнения программ:

1. Выполнение последовательности операторов.
2. Выполнение определенной последовательности операторов до тех пор, пока некоторое условие истинно.
3. Использование проверки истинности условия для выбора между различными возможными способами действия.

Первая форма хорошо известна; все наши предыдущие программы представляли собой некоторую последовательность операторов. Цикл **while** служит одним из примеров использования второй формы; другие способы будут рассмотрены в гл. 8. Последняя форма - выбор между различными возможными способами действия - делает программы гораздо более "интеллектуальными" и чрез вычайно увеличивает эффективность работы компьютера. В данной главе мы и займемся этим вопросом.

ОПЕРАТОР if

Далее Содержание

Начнем с очень простого примера. Мы уже видели, как нужно написать программу, подсчитывающую число символов в файле. Предположим, вместо символов мы хотим подсчитать строки. Это можно сделать путем счета числа символов "новая строка" в файле. Ниже приводится соответствующая программа:

```
/* подсчет строк */
#include < stdio.h>
main( )
{
int ch;
int linecount = 0;
while((ch = getchar( )) != EOF)
if(ch =='\n')
linecount ++;
printf(" Я насчитала %d строк \n", linecount);
}
```

Сердцевиной" программы является оператор

```
if(ch == '\n') linecount ++;
```

Этот "оператор **if**" служит указанием компьютеру увеличить значение переменной **linecount** на 1, если только что прочитанный символ (содержимое переменной **ch**) представляет собой символ "новая строка". Знак **==** не является опечаткой; его смысл выражается словами "равняется". Не путайте эту операцию с операцией присваивания (**=**).

Что происходит в случае, когда значение переменной **ch** не является символом "новая строка"? Тогда в цикле **while** производится чтение следующего символа.

Оператор **if**, который мы только что применили в программе, считается одиночным оператором, начинающимся от ключевого слова **if** и завершающимся символом "точка с запятой". Вот почему мы не использовали фигурные скобки, чтобы отметить начало и конец тела цикла **while**.

Совсем несложно усовершенствовать программу так, чтобы она подсчитывала символы и строки одновременно; давайте попробуем это сделать.

```
/* 1cc - подсчет числа строк и  символов */
#include
main( )
{
int ch;
int linecount = 0;
int charcount = 0;
while((ch = getchar( )) != EOF)
{
    charcount ++ ;
    if(ch == '\n' ) linecount++ ;
}
printf(" Я насчитала %d символов и %d строк.\n", charcount, linecount);
}
```

Теперь в цикл **while** входят два оператора; поэтому мы использовали фигурные скобки, чтобы отметить начало и конец тела цикла.

Мы вызываем скомпилированную программу **lcc** и используем операцию переключения, чтобы подсчитать число символов и строк в файле с именем **chow**.

```
Ice  <chow

Я насчитала 8539 символов и 233 строки
```

Следующий шаг в "развитии" этой программы - придание ей способности подсчитывать число слов. Это лишь немного труднее того, что мы делали до сих пор, но сначала нам нужно изучить ряд дополнительных вопросов, связанных с использованием операторов **if**.

РАСШИРЕНИЕ ОПЕРАТОРА if С ПОМОЩЬЮ else

Далее Содержание

Простейшей формой оператора **if** является та, которой мы только что воспользовались:

```
if(выражение)
оператор
```

Обычно под выражением здесь понимают условное выражение, с его помощью сравниваются значения двух величин (например **x > y** или **c == 6**). Если такое выражение истинно (**x** больше **y** или **c** равно **6**), то оператор выполняется. В противном случае он пропускается. Вообще говоря, в качестве условия может быть использовано любое выражение, и если его значение равно **0**, то оно считается **ложным** - дальнейшие детали мы обсудим чуть позже. Оператор может быть простым, как в нашем примере[1)] , или составным (блоком), начало и конец которого отмечены фигурными скобками:

```
if (score > big)  printf(" Полная победа!\n"); /* простой оператор */

if (joe > ron)                                  /* составной оператор */
{
  joecash + + ;
  printf(" Ты проиграл, Рон.\n");
}
```

Простая форма оператора **if** позволяет выбрать оператор (возможно, составной) или пропустить его. Язык Си предоставляет также возможность выбрать любой из двух операторов путем использования конструкции **if-else**.

Выбор: конструкция if-else

Далее Содержание

В предыдущей главе мы привели очень простую программу шифровки сообщений, которая заменяла каждый символ следующим по порядку в таблице ASCII. К сожалению, она заменила даже символ "новая строка", что привело к объединению нескольких строк в одну. Можно исключить эту проблему, написав программу, реализующую простой выбор: если символ - "новая строка", то оставить его без изменений, в противном случае преобразовать его. Вот как это можно запрограммировать на языке Си:

```
/* код1 */
#include
main( )
{
 char ch;
 while((ch = getchar( )) != EOF) {
    if(ch =='\n' ) /* оставить символ */
        putchar(ch); /* "новая строка" неизменным */
    else
        putchar(ch + 1); /* заменить остальные символы */
 }
}
```

В прошлый раз был использован файл, содержащий следующий текст[2)] :

```
Good spelling is an aid to clear writing.
```

Его использование в качестве теста для нашей новой программы приводит к результату:

```
Ура! Она работает!
```

Между прочим, совсем несложно написать и программу дешифровки. Скопируйте для этого программу **код1**, но только замените выражение **(ch + 1)** на **(ch - 1)**.

Вы обратили внимание на общий вид оператора **if-else**. Он выглядит следующим образом:

```
if (выражение)
оператор else
оператор
```

Если выражение истинно, то выполняется первый оператор, а если ложно, то выполняется оператор, следующий за ключевым словом **else**. Операторы могут быть простыми или составными. Правила языка Си не требуют отступа в строке, но это стало обычной практикой. Отступ позволяет с первого взгляда заметить те операторы, выполнение которых зависит от проверяемого условия.

Простая конструкция **if** позволяет нам выбирать: выполнить или нет некоторое действие; конструкция же **if-else** дает возможность выбрать одно из двух действий. Что делать, если нам требуется осуществить выбор из большого числа вариантов?

Множественный выбор: конструкция else-if

Далее Содержание

Часто нам приходится осуществлять выбор более, чем из двух вариантов. Чтобы учесть это, мы можем расширить структуру **if-else** конструкцией **else-if**. Рассмотрим конкретный пример. Расценки, назначаемые за коммунальные услуги некоторыми компаниями зависят от количества потребляемой энергии. Приведем расценки, установленные за пользование электроэнергией:

```
Первые          240 кВт/ч:    0.05418 долл. за кВт/ч
Следующие       300 кВт/ч:    0.07047 долл. за кВт/ч
Свыше           540 кВт/ч:    0.09164 долл. за кВт/ч
```

Если вас занимает этот вопрос, мы могли бы подготовить программу, вычисляющую стоимость потребляемой энергии. Приведем пример программы, являющейся первым шагом, сделанным в

этом направлении:

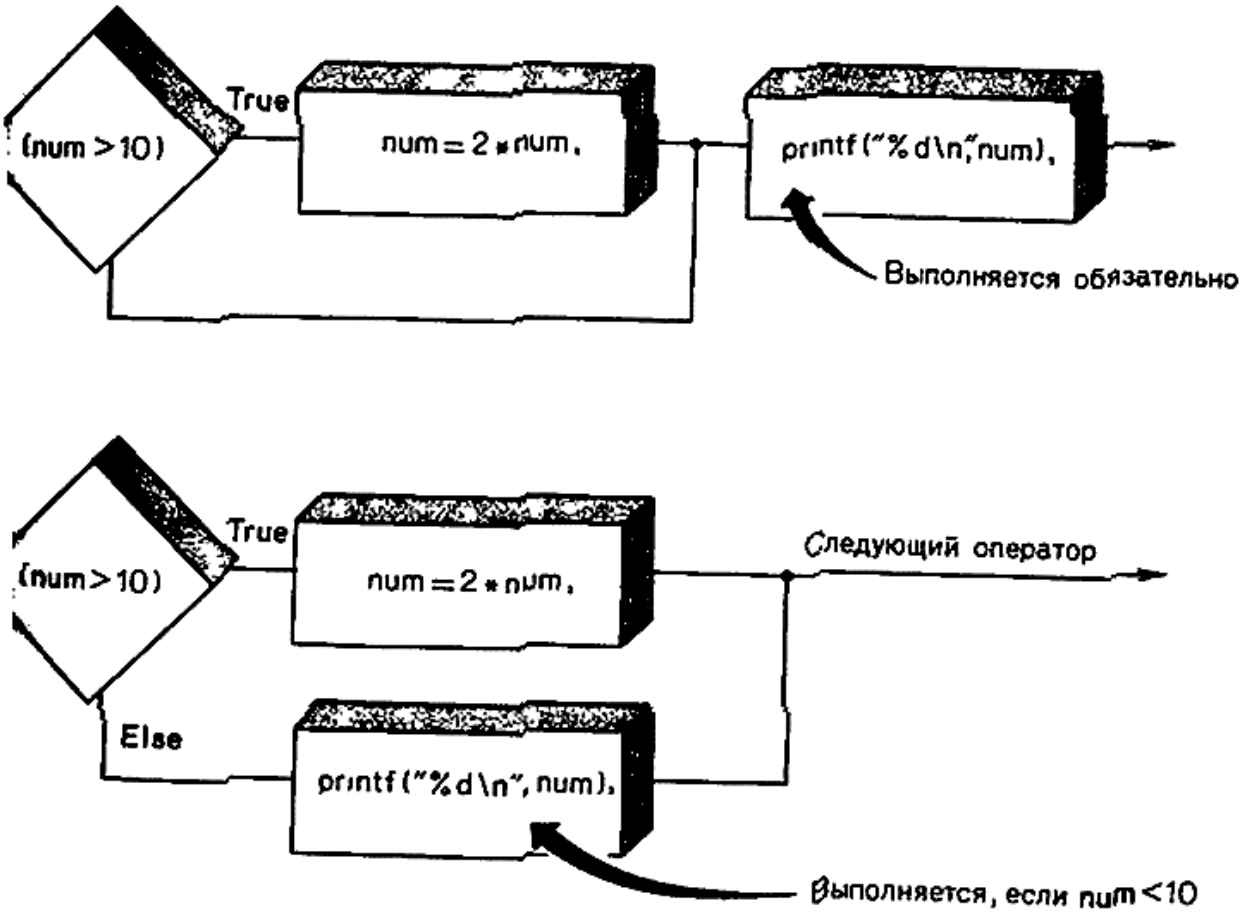


РИС. 7.1. Операторы **if** и **if else**

```
/* счет за электроэнергию */
/* вычисляет плату за электроэнергию */
#define RATE1 0.05418 /* тариф за первые 240 кВт/ч */
#define RATE2 0.07047 /* тариф за следующие 300 кВт/ч */
#define RATE3 0.09164 /* тариф за потребление свыше 540 кВт/ч */
#define BASE1 13.00   /* плата за первые 240 кВт/ч */
#define BASE2 34.14   /* плата за первые 540 кВт/ч */
#define BREAK1 240.0  /* величина, определяющая первое условие ветвления */
#define BREAK2 540.0  /* величина, определяющая второе условие ветвления */
main( )
{
float kwh;  /* количество использованных кВт/ч */
float bill; /* плата */
printf(" Укажите, пожалуйста, количество использованных кВт/ч.\n");
scanf(" %f, &kwh);
if (kwh < BREAK1)
    bill = RATEl * kwh;
else if(kwh < BREAK2) /* количество кВт/ч между 240 и 540 */
     bill = BASE1 + RATE2*kwh;
else                  /* количество кВт/ч свыше 540 */
     bill = BASE2 + RATE3*kwh;
printf(" Плата за %.1f  кВт/ч составляет $%1.2f. \n", kwh, bill);
}
```

Для обозначения тарифов были использованы символические константы, которые поэтому оказались собранными в одном месте. Если электрическая компания изменит свои расценки (а это может случиться), то такое расположение констант не позволит нам забыть скорректировать какую-нибудь из них. Мы задали в символическом виде и константы, соответствующие граничным значениям потребляемой мощности Они также подвержены изменениям. Управляющая логика программы реализуется путем простого выбора одной из трех расчетных формул в зависимости от значения переменной **kwh**, что иллюстрируется на рис. 7.2. Мы хотим подчеркнуть, что программа в процессе выполнения может достичь первого употребления **else** только в том случае, если величина переменной **kwh** больше или равна 240. Поэтому строка **else if(kwh < BREAK2)** эквивалентна требованию, чтобы значение **kwh** было заключено между 240 и 540, это мы и указали в коментарии к программе. Совершенно аналогично последнее употребление **else** может быть достигнуто, только если значение **kwh** больше или равно 540. И наконец, отметим, что константы **BASE1** и **BASE2** представляют собой величину платы за первые 240 или 540 кВт/ч электроэнергии соответственно. Поэтому требуется только прибавить дополнительную плату за количество потребляемой электроэнергии, превышающее эти величины.

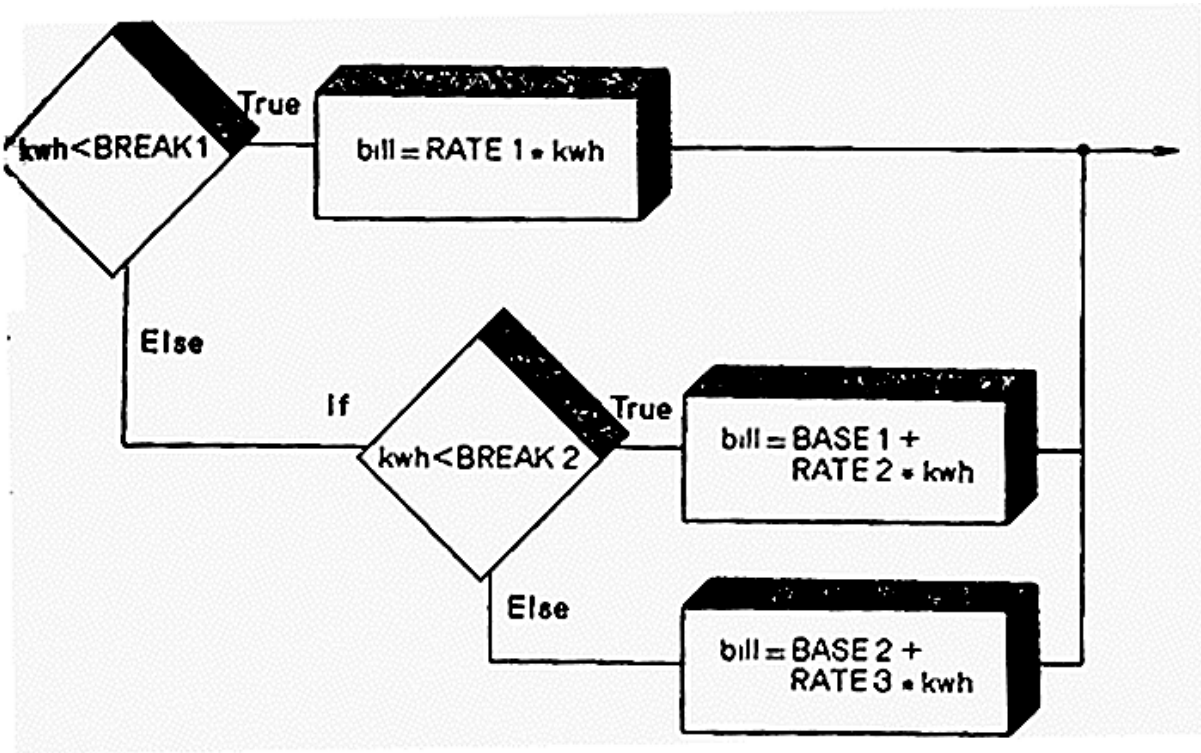


РИС. 7.2. Блок схема программы расчета платы за электроэнергию

Фактически конструкция **else-if** является видоизмененным способом задания условного оператора, с которым мы познакомились раньше. Ядро нашей программы представляет собой другую форму записи следующей последовательности операторов.

```
if(kwh < BREAK1) bill = RATEl*kwh;
else if(kwh < BREAK2)
        bill = BASE1 + RATE2*kwh;
else
    bill = BASE2 + RATE3*kwh;
```

Отсюда видно, что программа состоит из оператора **if-else**, для которого часть **else** представляет собой другой оператор **if-else**.

Про второй оператор **if-else** говорят, что он "вложен" в первым. (Между прочим, вся структура

if-else считается одним оператором. Вот почему мы не должны заключать вложенную конструкцию **if-else** в фигурные скобки).

Эти две формы записи являются абсолютно эквивалентными. Единственное отличие - дополнительные пробелы и символы "новая строка", но они игнорируются компилятором. Тем не менее первая форма записи предпочтительнее, поскольку она более четко показывает, что мы осуществляем выбор из трех возможностей. Кроме того, она облегчает просмотр программы и понимание семантики каждого варианта. Применяйте форму записи, использующую вложение операторов там, где это необходимо - например когда требуется проверить значения двух разных величин или (в на шем случае) если бы была установлена 10%-ная дополнительная плата за потребление энергии свыше 540 кВт/ч только в летние месяцы.

В одном операторе можно использовать столько конструкций **else if**, сколько нужно, что иллюстрируется приведенным ниже фрагментом:

```
if (score < 1000)
bonus = 0; else if (score < 1500)
bonus = 1; else if (score < 2000)
bonus = 2; else if (score < 2500)
bonus = 4; else bonus = 6;
```

(Этот фрагмент мог быть взят из игровой программы, где пере менная bonus представляет собой количество дополнительных "фотонных бомб" или "питательных гранул", получаемых игроком для следующей партии).

Объединение операторов if и else

Далее Содержание

Когда у вас в программе имеется несколько конструкции **if** и **else**, каким образом компилятор решает, какому оператору **if** соответствует какой оператор **else**? Рассмотрим, например, фрагмент программы:

```
if(number > 6)
 if(number < 12)
    printf ("Вы закончили!\n");
else
 printf("Простите, вы потеряли ход.\n");
```

В каком случае фраза "Простите, вы потеряли ход!" будет напечатана? Когда значение переменной **number** меньше или равно 6, или когда оно больше 12? Другими словами, чему соответствует **else**: первому **if** или второму?

Ответ выглядит так: **else** соответствует второму **if**, т.е. при выполнении программы результаты будут такими:

```
Число:        Результат:
5             Нет
10            Вы закончили!
15            Простите, вы потеряли ход!
```

Существует правило, которое гласит, что **else** соответствует ближайшему **if**, кроме тех случаев, когда имеются фигурные скобки. Мы сознательно записали этот фрагмент так, как будто **else** соответствует первому **if**, но вспомните, что компилятор не обратит внимания на отступы в строках. Если мы действительно хотим чтобы **else** соответствовал первому **if**, необходимо данный фрагмент оформить следующим образом:

```
if(number > 6)
{
    if(number < 12) printf (" Вы закончили '\n");
```

```
}
else
    printf(" Простите, вы потеряли ход!\n");
```

Теперь результат может выглядеть так

```
Число:      Результат:
5           Простите, вы потеряли ход!
10          Вы закончили!
15          Нет
```

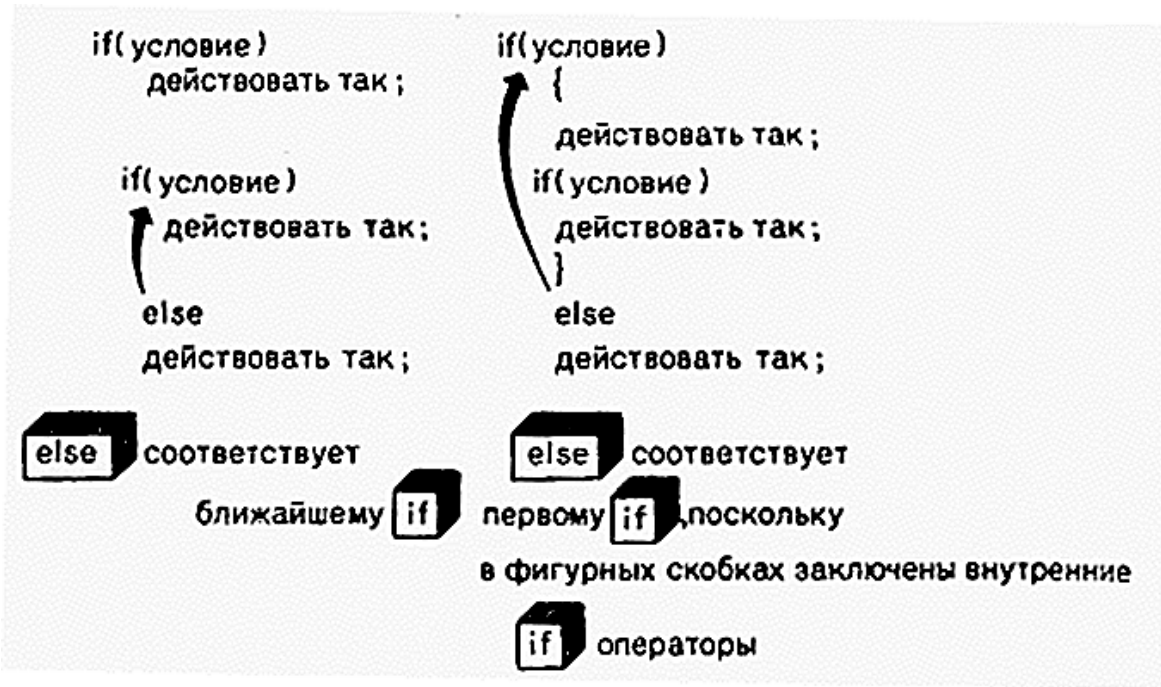


РИС. 7.3. Структура соответствия **if** и **else**.

Резюме: использование операторов if для организации выбора из нескольких вариантов

КЛЮЧЕВЫЕ СЛОВА: if, else

ОБЩИЕ ЗАМЕЧАНИЯ:

В каждой из последующих форм *оператором* может быть либо простои оператор, либо составной. Выражение "истинно" в обобщенном смысле, если его значение - ненулевая величина.

ФОРМА ЗАПИСИ 1:
if(*выражение*) *оператор*

Оператор выполняется, если *выражение* истинно.

ФОРМА ЗАПИСИ 2:
if(*выражение*)
оператор1 else
оператор2

Если *выражение* истинно, выполняется *оператор1*, в противном случае - *оператор2*.

ФОРМА ЗАПИСИ 3:
if(*выражение1*) *оператор1*
else if(*выражение2*) *оператор2*
else *оператор3*

Если *выражение1* истинно, выполняется *оператор1*. Если *выражение1* ложно, но *выражение2* истинно, выполняется *оператор2*. В случае когда оба выражения ложны, выполняется *оператор3*.

ПРИМЕР:[3)]

```
if(legs == 4)
    printf("Это, возможно, лошадь. \n");
else if (legs > 4)
    printf(" Это не лошадь. \n");
else   /* случай, когда legs < 4 */
{ legs++;
  printf(" Теперь животное имеет еще одну ногу.\n");
}
```

ЧТО ВАЖНЕЕ: ОПЕРАЦИИ ОТНОШЕНИЯ ИЛИ ВЫРАЖЕНИЯ

Далее Содержание

Операции отношения используются для сравнений. Мы уже использовали ранее некоторые из них, а сейчас приведем полный список операций отношения, применяемых при программировании на языке Си.

Операция:	*Смысл:*
<	меньше
<=	меньше или равно
= =	равно
>=	больше или равно
>	больше
!=	не равно

Этот список довольно хорошо соответствует возможным числовым соотношениям. (Вообще говоря, числа, даже комплексные, менее сложны, чем люди[4)]). Главное предостережение, которое мы хотим сделать, состоит в том, чтобы не использовать знак **=** вместо **= =**. В некоторых языках программирования (например. Бейсике) один и тот же знак используется и для операции присваивания, и для операции отношения "равенство", хотя они совершенно различны. С помощью операции присваивания некоторое значение *присваивается* переменной слева от знака равенства. В то же время с помощью операции отношения "равенство" проверяется равно ли выражение, стоящее слева от знака, выражению справа от него. Эта операция не изменяет значения переменной в левой части, если она там присутствует.

canoes = 3 присваивает значение 3 переменной **canoes**
canoes = = 5 проверяет, равняется ли значение переменной **canoes** 5

При программировании требуется аккуратность, потому что в ряде случаев компилятор не сможет обнаружить ошибки, связанной с неправильным использованием знаков этих операций, что приведет к результатам, отличным от тех, которые вы должны были получить. Ниже мы кратко остановимся на одном примере.

Таблица 7.1.

Операции присваивания и отношения "равенство" в некоторых языках программирования

Язык	Операция присваивания	Операция отношения "равенство"
Бейсик	=	=
Фортран	=	EQ
Си	=	==
Паскаль	=	=
ПЛ/1	=	=
Лого	make	-

Операции отношения применяются при формировании условных выражений, используемых в операторах *if* и *while*. Указанные операторы проверяют, истинно или ложно данное выражение. Ниже приводятся четыре не связанные между собой оператора, содержащие условные выражения; их смысл, мы надеемся, понятен.

```
if(number < 6)
printf(" Ваше число слишком мало.\n" );



whi1e(ch!= '$')
count ++;



if(total == 100) printf(" Вы набрали максимум очков!\n");



if (ch > 'м')
printf(" Отправьте этого человека по другому маршруту \n");
```

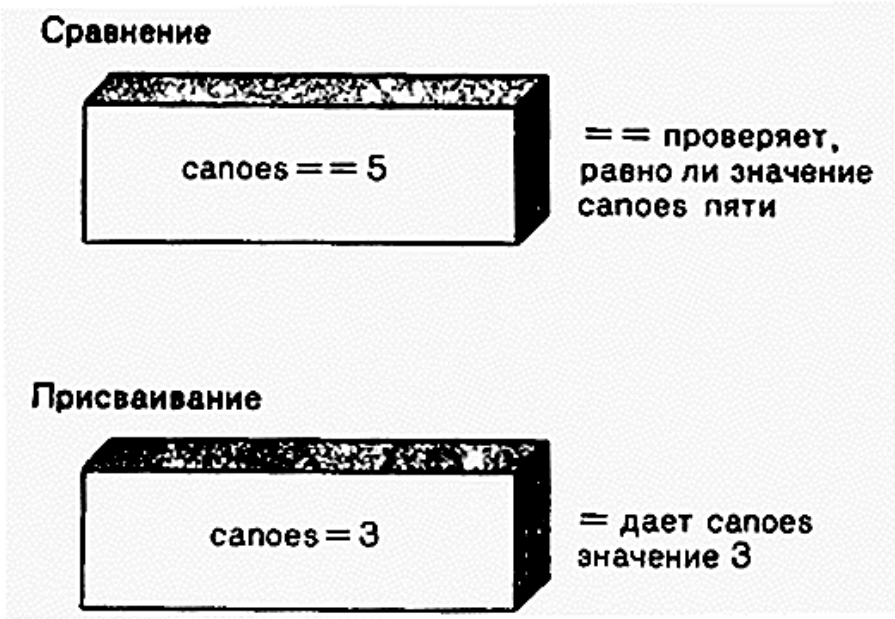


РИС. 7.4. Операции = и = =.

Обратите внимание, что в условных выражениях разрешается использовать также и символы, причем при сравнении их берется машинный код (который, как предполагалось вначале, является ко дом ASCII). Однако использовать операции отношения для сравнения строк не разрешается в гл 13. мы представим средства работы со строками. Операции отношения можно использовать также при работе с числами с плавающей точкой. Однако при сравнениях этих чисел мы рекомендуем вам ограничиться лишь операциями **<** и **>**, так как ошибки округления могут привести к тому, что числа окажутся неравными, хотя по логике они должны быть равны. Рассмотрим подобную ситуацию на примере десятичных чисел. Очевидно, произведение 3 и 1/3 равно 1.0, но если мы представим 1/3 в виде 6 разрядной десятичной дроби, то произведение будет равно 0,999999, что не равняется в точности 1.

Что такое истина? Далее Содержание

Каждое условное выражение проверяется "истинно" ли оно или ложно. При этом возникает интересный вопрос: Что такое истина?

Мы можем ответить на этот вечный вопрос покрайней мере так, как он решен в языке Си. Напомним, во-первых, что выражение в Си всегда имеет значение. Это утверждение остается верным даже для условных выражений, как показывает пример, приведенный ниже. В нем определяются значения двух условных выражений, одно из которых оказывается истинным, а второе - ложным.

```
/* истина и ложь */
main( )
{
    int true, false;
    true = (10 > 2); /* отношение истинно */
    false = (10 = =2); /* отношение ложно */
    printf("true = %d; false = %d\n" , true, false);
}
```

В данном примере значения двух условных выражений присваиваются двум переменным. Чтобы не запутать читателя, мы присвоили переменной **true** значение выражения, которое оказывается истинным, а переменной **false** - значение выражения, которое оказывается ложным. При выполнении программы получим следующий простой результат:

```
true = 1; false = 0
```

Вот как! Оказывается, в языке Си значение "истина" - это 1, а "ложь" - 0. Мы можем это легко проверить, выполнив программу, приведенную ниже.

```
/* проверка истинности */
main( ) {
if(1)
    printf(" 1 соответствует истине.\n" );
else
    printf(" 1 не соответствует истине. \n");
if(0)
    printf(" 0 не означает ложь. \n");
else
    printf(" 0 означает ложь. \n");
}
```

Мы скажем, что 1 должна рассматриваться как истинное утверждение, а 0 - как ложное. Если наше мнение верно, то в первом операторе **if** должна выполниться первая ветвь (ветвь **if**, а во втором операторе **if** - вторая (ветвь **else**). Попробуйте запустить программу, чтобы узнать, правы ли мы.

Итак чему же все-таки соответствует истина? Далее Содержание

Мы можем использовать 1 и 0 в качестве проверочных значений оператора **if**. Спрашивается,

можем ли мы использовать другие числа. Если да, то что при этом происходит? Давайте проведем эксперимент.

```
/* if - тест */
main( )
{
if (200)
    printf("200 - это истина. \n");
if(-33)
    printf(" -33 - это истина \n");
}
```

Результаты выглядят так

```
200 - это истина
-33- это истина
```

Очевидно, в языке Си числа 200 и -33 тоже рассматриваются как "истина". И действительно, все ненулевые величины принимаются в качестве "истины" и только 0 - как "ложь". В языке Си понятие истины оказывается чрезвычайно растяжимым.

Многие программисты обычно пользуются этим определением истинности. Например, строку

```
if(goats !=0)
```

можно заменить такой

```
if(goats)
```

поскольку выражение (**goats != 0**) и выражение (**goats**) оба примут значение 0, или "ложь", только в том случае, если значение переменной **goats** равно 0. Мы думаем, что смысл второй формы записи менее очевиден, чем первой. Однако в результате компиляции она оказывается более эффективной, так как при реализации про граммы требует меньшего числа машинных операций.

Осложнения с понятием "истина"

Далее Содержание

Растяжимость понятия истина в языке Си может привести и к неприятностям. Рассмотрим следующую программу:

```
/* занятость */
main( )
{
int age = 20;
while(age++  <= 65)
{ if((age % 20) = =0) /* делится ли возраст на 20 ? */
    printf(" Вам %d. Поздравляем с повышением!\n", age);
  if (age = 65)
    printf(" Вам %d. Преподносим Вам золотые часы \n", age);
} }
```

С первого взгляда вам может показаться, что результат работы программы будет выглядеть, например, так:

```
Вам 40. Поздравляем с повышением
Вам 60. Поздравляем с повышением
Вам 65. Преподносим Вам золотые часы
```

На самом деле, однако, выход будет таким:

```
Вам 65. Преподносим Вам золотые часы
Вам 65. Преподносим Вам золотые часы
Вам 65. Преподносим Вам золотые часы
```

```
Вам 65. Преподносим Вам золотые часы
Вам 65. Преподносим Вам золотые часы
```

и т. д.- до бесконечности.
В чем дело? Это произошло не только потому, что мы плохо спроектировали программу, но и потому, что мы забыли свои собственные предостережения и использовали выражение:

```
if (age = 65)
```

вместо

```
if (age == 65)
```

Последствия ошибки оказались катастрофическими. Когда в процессе выполнения программа достигает указанного оператора, она проверит выражение (**age = 65**). Это выражение, включающее в себя опреацию присваивания, имеет значение, которое совпадает со значением переменной, расположенной слева от знака, т.е. с 65 (в любом случае). Поскольку 65 не равно нулю, выражение считается истинным" и выполняется команда вывода на печать. Затем, когда в программе происходит передача управления на команду проверки условия в цикле **while**, значение переменной **age** по-прежнему равно 65, что меньше или равно 65. Условие оказывается истинным и величина **age** увеличивается до 66 (ввиду наличия операции увеличения **++** в постфиксной форме), и тело цикла выполняется еще раз. Прекратится ли его выполнение на следующем шаге? Должно было бы, поскольку величина **age** теперь больше, чем 65. Когда программа опять попадает на наш ошибочный оператор **if** переменная **age** снова получит значение 65. В результате сообщение будет напечатано еще раз, затем тело цикла выполнится еще раз, и т.д. - до бесконечности (Конечно, если вы в конце концов не захотите остановить программу).

Подводя итоги, можно сказать, что операции отношения используется для образования условных выражений. Условное выражение имеет значение "1", когда оно истинно, и "0", если оно ложно. В операторах (таких как **while** и **if**), где обычно используются условные выражения для задания проверяемых условий, могут применяться любые выражения, причем ненулевое значение является истиной", а нуль - "ложью".

Приоритеты операций отношения

Далее Содержание

Приоритет операций **отношения** считается меньшим, чем у операций **+** и **-**, и больше, чем у операции **присваивания**. Например, значение выражения:

```
x > y + 2
```

то же, что и выражения

```
x > (y + 2)
```

Это означает также, что выражение

```
ch = getchar( ) != EOF
```

эквивалентно

```
ch = (getchar( ) != EOF)
```

поскольку наличие у оператора **!=**, более высокого приоритета говорит о том, что она выполняется перед присваиванием. Поэтому значение переменной **ch** может стать либо 1, либо 0 ввиду того, что **(getchar( ) != EOF)** - условное выражение, значение которого присваивается переменной **ch**. Поскольку в примерах программ рассмотренных до сих пор, предполагалось, что переменная **ch** получает свое значение от функции **getchar( )**, мы использовали скобки, чтобы организовать выполнение операций в нужном порядке.

```
(ch = getchar( )) != EOF
```

Сами операции отношения можно разбить на две группы в соответствии с назначенными приоритетами:

группа операций более высокого приоритета: < <= => >
группа операций более низкого приоритета: = = !=

Подобно большинству остальных операций операции отношения выполняются слева направо. Поэтому под записью:

```
ex != wye == zee
```

подразумевается

```
(ex != wye) == zee
```

Следовательно, в соответствии с правилами языка Си сначала проверяется, равны ли значения переменных **ex** и **wye**. Результирующая величина, равная 1 или 0 (истина или ложь), затем сравнивается со значением **zee**. Мы не видим реальной необходимости использовать подобного сорта конструкцию, но считаем своим долгом указать на возможные следствия принятого порядка выполнения операций.

Читателю, озабоченному сохранением своего высокого приоритета, хотим напомнить, что полный список всех операций вместе с их приоритетами приведен в приложении В в конце книги.

Резюме: операции отношения и выражения

1. Операции отношения

С помощью каждой из приведенных ниже операции величина слева от знака сравнивается с величиной справа от него:

2. Больше
3. Больше или равно
4. Равно
5. Меньше или равно
6. Меньше
7. Не равно

УСЛОВНЫЕ ВЫРАЖЕНИЯ

Понятие условное выражение состоит из знака операции отношения и операндов, расположенных слева и справа от него. Если отношение истинно, значение условного выражения равно 1, если отношение ложно, значение условного выражения равно 0.

Примеры:

Отношение **5 > 2**: *истинно* и имеет значение 1.
Отношение **(2 + a) = = a**: *ложно* и имеет значение 0.

ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Далее Содержание

Иногда бывает полезным объединить два или более условных выражения. Например, предположим, нам требуется программа, которая подсчитывает только "непустые" символы, т. е. мы хотим знать число символов, не являющихся пробелами, символами "новая строка" и табуляции. Для этого мы можем использовать "логические" операции. Ниже приводится короткая

программа люстрирующая этот способ подсчета:

```
/* число символов */
/* подсчитывает непустые символы */
main( )
{
int ch;
int charcount = 0;
while ((ch = getchar( )) != EOF)
if(ch !='  ' && ch \='\n' && ch != '\t') charcount++;
    printf(" Всего %d непустых символов. \n", charcount);
}
```

Так же как это обычно происходило в наших предыдущих программах, данная программа начинает свое выполнение с чтением символа и проверки, является ли он признаком конца файла. Дальше появляется нечто новое - оператор, использующий логическую операцию **"и"**, обозначаемую **&&**. Смысл действий, осуществляемых оператором **if** в данном случае, можно пояснить следующим образом:

Если прочитанный символ не пробел, **И** не символ "новая строка", **И** не символ табуляции, то происходит увеличение значения переменной **charcount** на единицу.

Все выражение будет истинным, если указанные три условия истинны. Логические операции имеют меньший приоритет, чем операции отношения, поэтому не было необходимости использовать *дополнительные скобки* для выделения подвыражений.

В языке Си имеются три логические операции:

Операция	*Смысл*
&&	И
||	ИЛИ
!	НЕ

Предположим, что **expl** и **exp2** - два простых условных выражения типа **cat > rat** или **debt = = 1000**. Тогда:

1. **expl && exp2**: истинно в том и только в том случае, когда оба выражения **expl** и **exp2** истинны.
2. **expl || exp2**: истинно в том случае, если какое-нибудь из выражений **expl** или **exp2** истинно или оба истинны.
3. **!expl**: истинно, если выражение **expl** ложно, и наоборот.

Ниже приведено несколько конкретных примеров:

5 > 2 && 4 > 7: ложно, поскольку истинно только одно подвыражение.
5 > 2 || 4 > 7: истинно, поскольку по крайней мере одно из подвыражений истинно.
!(4 > 7): истинно, потому что 4 не больше 7.

Последнее выражение к тому же эквивалентно следующему:

```
4 < = 7.
```

Если вы совсем не знакомы с логическими операциями или испытываете трудности при работе с ними, помните, что **практика && время = = совершенство**.

Приоритеты

Далее Содержание

Операция **!** имеет очень высокий приоритет, он выше, чем у умножения, такой же, как у операций увеличения, и только круглые скобки имеют более высокий приоритет. Приоритет операции **&&** больше чем операции **||**, а обе они имеют более низкий приоритет, чем операции отношения, но более высокий, чем операция присваивания. Поэтому выражение:

```
a > b  &&  b > c  || b > d
```

интерпретировано так:

```
((a > b) && (b > c)) || (b > d)
```

т. е. **b** содержится между **c** и **a** или **b** больше **d**.

Порядок вычислений

Далее Содержание

Обычно в языке Си не определяется, какие части сложного выражения будут вычисляться вначале. Например, в операторе:

```
apples = (5 + 3)*(9 + 6);
```

выражение **5 + 3** может быть вычислено до вычисления выражения **9 + 6**, или наоборот (Приоритеты, присвоенные операциям гарантатируют, что оба выражения будут вычислены перед выполнением операции умножения.) Эта неопределенность была оставлена в языке, чтобы создатели компилятора имели возможность в конкретной системе осуществлять наиболее эффективный выбор. Исключением из этого правила (или его нарушением) является выполнение логических операций. Язык Си гарантирует, что логические выражения вычисляются слева направо. Более того, гарантируется также, что, как только обнаруживается элемент, значение которого устанавливает ложность всего выражения как целого, вычисление данного выражения прекращается. Это дает возможность использовать конструкции типа:

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

В результате вычисления первого подвыражения переменная **c** получает свое значение, которое затем можно использовать во втором подвыражении. Если бы такой порядок вычислений не гарантировался, при выполнении программы компьютер, возможно, проверял бы истинность второго выражения перед нахождением значения переменной **c**.

Приведем еще один пример:

```
if (number !=0  && 12/number ==2) printf(" Число равно 5 или 6.\n" );
```

Если значение переменной **number** равно 0, то все выражение ложно, и поэтому дальнейшее вычисление данного условного выражения прекращается. Это избавляет компьютер от последствий деления на нуль. Многие языки не обеспечивают выполнения подобного требования, и, выяснив, что **number** равно 0, компьютер переходит к проверке следующего условия.

Резюме: логические операции и выражения

1. ЛОГИЧЕСКИЕ ОПЕРАЦИИ

Операндами логических операций обычно являются условные выражения. У операции **!=** имеется только один операнд. Остальные имеют по два - один слева от знака и другой справа от него.

```
&&  И
||  ИЛИ
!   НЕ
```

II. ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ

выражение1 && выражение2: истинно тогда и только тогда, когда оба выражения истинны
выражение1 || *выражение2*: истинно, если какое-нибудь одно или оба выражения истинны
!*выражение*: истинно, если выражение ложно, и на оборот

ПОРЯДОК ВЫЧИСЛЕНИИ

Логические выражения вычисляются слева направо; вычисления прекращаются, как только устанавливается истинность или ложность всего выражения.

ПРИМЕРЫ

Выражение: **6 > 2 && 3 = = 3**: истинно
Выражение: **!(6 > 2 && 3 = = 3)**: ложно
Выражение: **x != 0 && 20/x < 5**: второе выражение вычисляется только при условии, что **x** не равен нулю.

Применим наши знания для написания двух программ. Наш первый пример довольно удобен для этого.

ПРОГРАММА ПОДСЧЕТА СЛОВ

Далее Содержание

Теперь у нас есть возможности для написания программы подсчета числа слов в тексте. (Она может также подсчитывать символы строки.) Решающим моментом является разработка способа, с помощью которого программа будет распознавать слова. Мы будем придерживаться сравнительно простого подхода и определим слово как последовательность символов, которая не содержит "пустых символов". Поэтому **"glymxck"** и **"r2d2"** - это слова. Переменная **word** будет использоваться для хранения указания о том, является ли введенный символ частью данного слова или началом следующего. Появление "пустого символа" (которым может быть пробел, табуляция или "новая строка") служит признаком конца слова. Тогда следующий "непустой" символ будет означать начало нового слова, и мы сможем увеличить значение счетчика слов на 1.
Вот эта программа:

```
#include
#define YES 1
#define NO 0
main( )
{
 int ch;   /* введенный символ */
 long nc = 0L;   /* число символов */
 int n1 = 0;   /* число строк */
 int nw = 0;   /* число слов */
 int word = NO;   /* = = YES, если содержимое   ch - часть слова */
 while((ch = getchar( )) != EOF)
{ nc++ ;   / * подсчет символов * /
  if (ch = = '\n' ) nl++;   /* подсчет строк */
  if (ch != '  ' && ch != '\n'  &&  ch !='\t'  && word ==NO)
    {
        word = YES;   /* начало нового слова    */
        nw++;   /* подсчет слов */
    }
 if ((ch = = '  ' ||  ch = = '\n'  ||  ch = = '\t' ) && word == YES)
    word = NO; /* достигнут конец слова */
}
```

```
 printf(" символов  = %1d, слов = %d, строк = %d\n", nc, nw, nl);
}
```

Поскольку существуют три различных "пустых символа", мы должны использовать логические операции для проверки всех трех возможностей. Рассмотрим, например, следующую строку:

```
if(ch != ' ' && ch != '\n'  && ch != '\t ' && word == NO).
```

В ней говорится: "если содержимое **ch** - *не* пробел, и *не* новая строка, и *не* табуляция, и *не* первый символ слова". (Первые три условия эквивалентны проверке, не является ли содержимое **ch** пустым символом). Выполнение всех четырех условий служит признаком начала нового слова, и значение переменной **nw** увеличивается. Если мы в середине слова, то первые три условия оказываются вы полненными, но значением переменной **word** окажется признак **YES**, и значение переменной **nw** не увеличивается. Когда в процессе ввода встретится очередной "пустой" символ, переменной **word** будет вновь присвоен признак **NO**.

Просмотрите программу и проверьте, правильно ли она интерпретирует случаи, когда между словами находится несколько "пустых" символов подряд.

Если вы захотите применить эту программу для работы с файлами, используйте операции переключения.

ПРОГРАММА, "РИСУЮЩАЯ" СИМВОЛАМИ

Далее Содержание

Давайте теперь займемся чем нибудь менее утилитарным и более декоративным. Нашей целью является создание программы, с помощью которой вы сможете рисовать на экране геометрические фигуры, заполненные символами. Каждая выводимая строка представляет собой сплошной рад одинаковых символов.

Нам предоставляется возмжность выбора символа, длины строки, а текже начальной позиции, с которой она выводиться на печать. Программа читает указываемые параметры до тех пор, пока не встретит признак **EOF**. Текст программы представлен на рис. 7.5.

Положим, мы вызываем программу **sketcher**. Чтобы ее выполнить, мы набираем на клавиатуре ее имя, затем вводим символ и два числа. На экране появляется отклик, после чего мы вводим его набор параметров, отклик появляется вновь, и так до тех пор пока мы не введем признак **EOF**.

В среде ОС UNIX диалог будет выглядеть следующим образом:

```
%   sketcher
B 10 20
      BBBBBBBBBB
Y 12 18
         YYYYYYY
[CTRL/-d]
%
```

```
/* художник-график */
/* РИСУЕТ  сплошные фигуры */
#include
#define MAXLENGTH  80
main( )
{
int ch; /*печатаемый символ*/
int start, stop;  /* начальная и конечные позици */
int count;   /* счетчик позиций */
while((ch = getchar( )) != EOF)   /* ввод символа */
{
  if(ch != '\n' )  /*пропуск символа "новая строка"*/
```

```
  { scanf(" %d  %d", &start, &stop);  /* ввод граничных значний*/
     if (start > stop ||  start < 1 ||  stop > MAXLENGTH)
        printf(" Введены неправильные граничные значе ния \n");
     else
        { count = 0;
          while(++count < start)
            putchar(' '); /* печать пробелов вплоть до начальной позиции */
          while(count++  <= stop)
            putchar(ch);   /*печать символа до конечной позиции */
          putchar(' \n'); /* закончить печать строки и начать новую */
        }   /* конец части else */
  }   /* конец проверки содержимого ch */
}   /* конец цикла while */
}   /* конец программы */
```

РИС. 7. 5. Программа, рисующая символами

Программа вывела на экран символ **B** в позициях с 10 по 20, а символ **Y** - с 12 по 18. К сожалению, при диалоговой работе с программой на экране наши команды перемежаются выводимым строками. Гораздо более удобным способом использования программы является создание файла, содержащего подходящий набор данных, а затем применение операции переключения для ввода (из него) параметров в программу. Предположим, например, что в файле с именем **fig** содержатся следующие данные:

```
- 30 50
| 30 50
| 30 50
| 30 50
| 30 50
| 30 50
= 20 60
: 31 49
: 31 49
: 29 49
: 27 49
: 25 49
: 30 49
: 30 49
/ 30 49
: 35 48
: 35 48
```

После ввода команды **sketcher < fig** результат работы программы будет выглядеть так, как показано на рис. 7.6.

Внимание: на устройствах печати и экранах дисплеев отношение высоты символа к его ширине может быть различным, в результате символы фигуры при печати выглядят более сжатыми по вертикали, чем изображенные на экране).

Анализ программы

Далее Содержание

Это короткая программа, но она оказалась сложнее тех примеров, которые мы обсуждали до сих пор. Рассмотрим некоторые ее элементы.

Длина строки

Далее Содержание

Мы ввели в программу ограничение на длину печатаемой строки (она может быть больше 80 позиций), поскольку 80 символов - это стандартный формат (по ширине) экрана дисплеев различных типов, а также число символов при нормальной ширине листа бумаги на устройстве

печати. Вы можете, однако, переопределить величину константы **MAXLENGTH**, если при работе с программой захотите воспользоваться устройством, имеющим другую ширину строки.

Структура программы

Далее Содержание

В нашей программе имеются три цикла **while**, один оператор **if** и один оператор **if-else**. Посмотрим, что каждый из них делает:

```
while((ch = getchar()) != EOF)
```

Задачей первого цикла **while** является ввод нескольких наборов данных. (Каждый набор данных состоит из символа и двух целых чисел, указывающих границы его вывода). Производя вначале чтение символа, мы смогли объединить в одном выражении его ввод и проверку того, не является ли он признаком **EOF**. Если прочитан символ **EOF**, то программа останавливается, не делая попытки ввести величины, соответствующие переменным **start** и **stop**. В противном случае при помощи функци **scanf( )** указанным переменным присваиваются введенные значения, затем программа переходит к их обработке. Этим завершается выполнение тела цикла, после чего вводится новый символ, и весь процесс повторяется снова.

Обратите внимание, что для чтения данных мы использовали два оператора, а не один. Почему нельзя было воспользоваться одним оператором?

```
scanf(" %c %D %D", &ch, &start, &stop)
```

Предположим, мы это сделали. Рассмотрим, что происходит, когда программа заканчивает чтение последнего набора данных из файла. Перед началом выполнения очередного тела цикла единственным оставшимся непрочитанным элементом файла будет признак **EOF**. Функция **scanf( )** читает указанный символ и присваивает его переменной **ch**; затем она пытается ввести значение для переменной **start**, но в файле не осталось данных, которые не были бы уже прочитаны! Компьютер выскажет свое недовольство, и ваша программа прекратит работу. Отделяя чтение символа от ввода остальных данных, мы позволяем компьютеру обнаружить появление признака **EOF** *перед* очередной попыткой прочесть оставшиеся данные.

```
if (ch != '\n')
```

Цель введения в программу первого оператора **if** состоит в том, чтобы упростить чтение данных. Мы объясним, как он работает в следующем разделе.

```
if(start > stop || start < 1 ||  stop > MAXLENGTH)
    printf(" Введены неправильные граничные значения \n");
else
```

Цель применения оператора **if-else** состоит в том, чтобы избежать использования в программе таких значении переменных **start** и **stop**, которые могут привести к нежелательным последствиям. Этот вопрос мы также обсудим ниже. Обратите, однако, внимание на то, как мы использовали логические операции и операции отношения, чтобы обнаружить появление любого из трех "опасных" значений.

Основная часть программы представляет собой составной оператор, который следует за ключевым словом **else**.

```
count = 0;
```

Вначале счетчик **count** устанавливается на нуль.

```
while(++count < start)  putchar(' ');
```

Затем в цикле **while** начинается вывод на печать пробелов вплоть до позиции, определяемой значением переменной **start**. Если значение **start**, скажем, равно 10, то печатается девять пробелов. Поэтому вывод символов на печать начнется с 10-й позиции. Обратите внимание, как использование префиксной формы операции увеличения вместе с операцией < позволяет добиться указанного эффекта. Если бы вместо этого мы использовали выражение **count++ < start**, то сравнение проводилось бы перед увеличением значения **count**, и в результате мог быть напечатан один дополнительный пробел.

```
while(count++ <= stop) putchar(ch);
```

Второй цикл **while** в вышеупомянутом блоке осуществляет задачу вывода на печать символа, начиная с позиции, задаваемой переменной **start**, и кончая позицией, задаваемой переменной **stop**. На этот раз мы воспользовались постфиксной формой операции увеличения и операцией <=. Такая комбинация обеспечивает желаемый результат при выводе на печать символа - верхняя граничная позиция входит в поле печати. Для проверки этого факта вы можете воспользоваться логикой или методом проб и ошибок.

```
putchar(' \n');
```

Оператор **putchar('\n')** используется для завершения печати данной строки и перехода на новую.

Форма данных

Далее Содержание

При написании программы необходимо понимать, как она будет взаимодействовать с входными данными. Сейчас мы обратимся к этому вопросу.

Вводимые данные должны быть представлены в форме, совместимой с требованиями, которые налагаются функциями ввода, используемыми в программе. Поэтому вся тяжесть приведения

данных к правильной форме ложится на пользователя. В более сложных программах основной объем работы по такому преобразованию переносится на саму программу. Наилучшей формой представления вводимых данных является следующая:

```
H 10 40
I 9 41
```

где за символом следуют номера начальной и конечной позиции. Но наша программа допускает также и такую форму данных:

```
H
10
40
I
9
41
```

и такую

```
H10 40I 9 41
```

но не

```
H 10 40 I 9 41
```

Почему наличие одних пробелов является необязательным, а других - обязательным? Почему символ "новая строка" может быть помещен между последним целым числом из одного набора данных и первым символом из следующего набора, а пробел нет?

Эти вопросы поднимают проблемы, касающиеся не только данной программы. Рассмотрим работу функций **getchar()** и **putchar( )** и найдем ответы на них.

Функция **getchar( )** читает первый встретившийся символ независимо от того, является ли он алфавитным символом, пробелом, символом "новая строка" или еще чем-нибудь. Функция **scanf()** делает то же самое, если чтение производится в формате **%c** (символ). Но, когда **scanf( )** осуществляет ввод данных в формате **%d** (целые), пробелы и символы "новая строка" пропускаются. Поэтому символы "новая строка" или любые пробелы между символом, считываемым функцией **getchar( )**, и следующим целым числом, считываемым функцией **scanf( )**, игнорируются. Функция **scanf( )** читает цифры до тех пор, пока не встретит нецифровой символ - пробел, символ "новая строка" или букву.

Следовательно, между первым и вторым целыми числами необходимо помещать пробел или символ "новая строка", чтобы функция **scanf( )** могла распознать где кончается одно и начинается другое. Этим объясняется, почему между символом и следующим целым числом может стоять пробел или символ "новая строка", и почему между двумя целыми числами обязательно должен быть разделитель такого вида. Но почему между целым числом, стоящим в конце набора данных, и следующим символом не может быть пробел? Потому что в следующий раз на очередном шаге выполнения цикла **while** функция **getchar( )** осуществляет ввод символа из той позиции, где "остановилась" функция **scanf( )**. Поэтому она прочтет любой следующий символ, стоящий после целого числа, - пробел, символ "новая строка" и т. п. Если бы мы следовали требованиям функции **getchar( )**, структуру данных необходимо было бы организовать так:

```
w10 50a20 60y10 30
```

где между последним целым числом, стоящим в конце группы, и следующим символом разделитель отсутствует. Но такая структура выглядит неуклюже, поскольку число 50 при этом выглядит так, как будто оно помещено в одну группу с **a**, а не с **w**. Поэтому введен оператор

```
if(ch != '\n')
```

чтобы иметь возможность обнаружить, когда значение **ch** равно символу "новая строка". Вместо этого можно использовать данные, вводимые в виде

```
w 10 50 a20 60 y10 30
```

где между числом 50 и **a** помещен символ "новая строка". Программа читает этот символ, игнорирует его и затем переходит к чтению следующего символа.

Контроль ошибок

Далее Содержание

Существует широко распространенная проблема, связанная с вводом в машину данных, которые должны использоваться определенным образом. Один из методов ее решения состоит в "контроле ошибок". Это означает, что, перед тем как приступить к обработке данных, программа должна проверить их правильность. В нашей программе мы сделали первую попытку осуществить такой контроль ошибок с помощью операторов:

```
if(start > stop || start < 1 || stop > MAXLENGTH)
     printf(" Введены неправильные граничные значения. \n");
```

Они входят в структуру **if-else**, которая определяет, что основная часть программы будет выполняться только в том случае, если ни один из трех **if**-тестов не окажется истинным.

С какой целью мы принимаем все эти меры предосторожности? Во-первых, совершенно неправильно размещать начальную позицию после конечной, поскольку обычно на терминал данные выводятся слева направо, а не наоборот. Поэтому с помощью выражения **start > stop** проверяется наличие такой потенциальной ошибки. Во-вторых, при выводе на экран первый столбец имеет номер 1; мы не можем выводить данные левее левого края. Выражение **start < 1** служит средством обнаружения такой ошибки. И наконец с помощью выражения **stop > MAXLENGTH** проверяется, не пытаемся ли мы вывести на печать данные правее правого края.

Существуют ли еще какие-нибудь ошибочные значения, которые мы можем присвоить переменным **start** и **stop**? Можно было бы, конечно, попробовать присвоить переменной **start** значение больше чем **MAXLENGTH**. Может ли этот вариант успешно пройти тест? Нет, хотя наличие подобной ошибки мы и не проверяем непосредственно.

Предположим, что величина **start** больше константы **MAXLENGTH**. Тогда либо значение **stop** тоже превышает величину **MAXLENGTH**, что обязательно приведет к обнаружению ошибки, либо **stop** окажется меньшей или равной **MAXLENGTH**. Тогда ее значение должно быть меньше величины **start**, что приведет к обнаружению этой ошибки первым тестом. Другая вероятно ошибка может состоять в том, что значение **stop** окажется левее 1. Мы оставляем читателям в качестве самостоятельного упражнения проверку того, что данная ошибка также не останется не замеченной.

В нашей программе контроль ошибок выглядит весьма простым. Если вы проектируете программу для серьезных целей, вы должны обратить на этот больше внимания, чем мы. Например, выводимые сообщения об ошибках могли бы указывать какие величины неверны и почему; кроме того, вы могли бы прибавить в сообщения что-то от себя и придать им большую эмоциональную окраску.

Приведем несколько примеров:

```
Указаное вами значение stop - 897654  превышает ширину экрана.
Вот это да! У вас START больше, чем STOP.
Попробуйте, пожалуйста, еще раз.

ВЕЛИЧИНА START ДОЛЖНА БЫТЬ БОЛЬШЕ 0, ИНДЮК.
```

Произносимые личностные моменты относятся и к вам.

ОПЕРАЦИЯ УСЛОВИЯ: ?:

Далее Содержание

В языке Си имеется короткий способ записи одного из видов оператора **if-else**. Он называется "условным выражением" и использует операцию условия - **?:**. Эта операция состоит из двух частей и содержит три операнда. Ниже приводится пример оператора с помощью которого находится абсолютное значение числа:

```
x = (y < 0 )? -y : y;
```

Все, что находится между знаком **=** и символом "точка с занятой" представляет собой **условное выражение**. Смысл этого оператора заключается в следующем: если **y** меньше 0, то **x = - y**; в противном случае **x = y**. В терминах оператора **if-else** данный оператор мог выглядеть так:

```
if(y < 0)          x = (y < 0 )? -y : y;
    x = -y;
else
    x = y;
```

В общем виде условное выражение можно записать следующим образом:
выражение1 ? *выражение2* : *выражение3*

Если *выражение1* истинно (больше нуля), то значением всего условного выражения является величина *выражения2*;
если *выражение1* ложно (равно 0), то значение всего условного выражения - величина *выражения3*.

Условное выражение удобно использовать в тех случаях, когда имеется некоторая переменная, которой можно присвоить одно из двух возможных значений. Типичным примером является присваивание переменной значения большей из двух величин:

```
max = (a > b)? a : b;
```

Вообще говоря, использование условных выражений не является обязательным, поскольку тех же результатов можно достичь при помощи операторов **if-else**. Однако условные выражения более компактны, и их применение обычно приводит к получению более компактного машинного кода.

Резюме: операция условия

I. Операция условия: ?:
В этой операции имеются три операнда, каждый из которых является выражением, причем вся запись выглядит следующим образом:

выражение1 ? *выражение2* : *выражение3*.

Значение всего выражения равно величине *выражения2*, если *выражение1* истинно, и величине *выражения3* в противном случае.

II. ПРИМЕРЫ:
Выражение: (5 > 3) ? 1 : 2 имеет значение 1
Выражение: (3 > 5) ? 1 : 2 имеет значение 2
Выражение: (a > b) ? a : b имеет значение большей из величин **a** и **b**.

МНОЖЕСТВЕННЫЙ ВЫБОР: ОПЕРАТОРЫ switch И break

Далее Содержание

Операция условия и конструкция **if-else** облегчают написание программ, в которых осуществляется выбор между *двумя* вариантами. Однако иногда в программе необходимо произвести выбор одного из *нескольких* вариантов. Мы можем сделать это используя конструкцию **if-else if - ... - else**, но во многих случаях оказывается более удобным использовать оператор **switch**. Ниже приводится пример, иллюстрирующий его работу. Программа читает с терминала букву, затем выводит на печать название животного начинеющееся с этой буквы.

```
/*животные*/
main( )
{
char ch;
printf("Введите букву алфавита, а я укажу");
printf("название животного, \n начинающееся с нее.\n");
printf("Введите, пожалуйста,  букву;  для завершения работы введите #. \n");
while((ch = getchar())!= '#')
{
  if(ch != '\n')  /* пропуск символа "новая строка" */
{
    if(ch >= 'a' && ch <= 'я')  /*разрешены только строчные буквы */
    switch (ch)
    { case 'а' : printf(" аргали, дикий горный азиатский баран\n");
                 break;
      case 'б' : printf(" бабирусса,  дикая малайская свинья \n");
                 break;
      case 'в' : printf(" выхухоль, водоплавающий крот \n");
                 break;
      case 'г' : printf(" гиббон, длиннорукая обезьяна \n");
                 break;
      case 'д' : printf(" даман древесный \n");
                 break;
        default: printf(" Это трудная задача!\n");
                 break;
    } else
        printf(" Я распознаю только строчные буквы. \n");
  printf(" Введите, пожалуйста, следующую букву или #.\n");
} /* конец if, пропускающего символ "новая строка" */
} /* конец цикла while */
}
```

РИС. 7.7. Программа, печатающая названия животных.

Нам стало скучно продолжать, и мы остановились на букве **д**. Давайте теперь рассмотрим один пример выполнения программы перед тем, как обсудить использованные в ней новые средства языка.

```
Введите букву алфавита, а я укажу название животного, начинающееся с нее.

Введите, пожалуйста, букву; для завершения работы введите #.
а [возврат]
аргали, дикий горный азиатский баран Введите, пожалуйста, следующую букву или #. г [возврат]
гиббон, длиннорукая обезьяна Введите, пожалуйста, следующую букву или #. р [возврат]
Это трудная задача!
Введите, пожалуйста, следующую букву или #. т [возврат]
Я распознаю только строчные буквы. Введите, пожалуйста, следующую букву или #. # [возврат]
```

Этот пример служит иллюстрацией работы оператора **swith**. Вначале вычисляется выражение в скобках, расположенное за ключевым словом **switch**. В данном случае значением этого выражения будет символ, присвоенный переменной **ch**, который мы ввели перед этим. Затем программа просматривает список "меток" (в этом примере **case 'а':, case'б'**: и т. д.) до тех пор, пока не находит "метку", которая соответствует данному значению. Далее программа переходит к выполнению оператора, расположенного в этой строке. Что произойдет в случае, когда такой подходящей строки не найдется? Если существует строка с "меткой" **case default:**, то будет выполняться оператор,

помеченный этой меткой. В противном случае произойдет переход к оператору, расположенному за оператором **switch**.

Что можно сказать по поводу оператора **break**? Его выполнение приводит к тому, что в программе происходит выход из оператора **switch** и осуществляется переход к следующему за ним оператору (см. рис. 7.8). При отсутствии оператора **break** будут выполнены все операторы, начиная с помеченного данной меткой и завершая оператором **switch**. Если удалить все операторы **break** из нашей программы, то, указав, например, букву **г**, получим следующий диалог:

```
Введите букву алфавита, а я укажу название животного,  начинающееся с нее.
Введите, пожалуйста, букву; для завершения работы введите #.
г [возврат]
гиббон, длиннорукая обезьяна
даман древесный
это трудная задача
Введите, пожалуйста, следующую букву или #.
# [возврат]
```

Мы видим, что выполнились все операторы, начиная от метки **case 'r'** и кончая оператором **switch**. Если вы знакомы с языком Паскаль, то можете заметить, что оператор **switch** в Си похож на оператор **case** в Паскале. Важнейшее отличие состоит в том, что если вы хотите, чтобы в каждом конкретном случае выполнялся только помеченный оператор, то в операторе **switch** необходимо использовать операторы **break**.

Метки, имеющиеся в операторе **switch**, должны быть константами или константными выражениями (выражения, операнды которого константы) целого типа (включая тип **char**). Запрещается использовать в качестве метки переменную. Значением выражения в скобках должна быть величина целого типа (опять же, включая тип **char**). Ниже приводится общая структура оператора **switch**:

```
switch(целое выражение)
{ case константа1 : операторы; (необязательные)
  case константа2 : операторы; (необязательные)
  case default (необязательные) : операторы; (необязательные)
}


switch (number)
{
  case 1: оператор 1;
          break;
  case 2: оператор 2;
          break;
  case 3: оператор 3;
          break;
  default:оператор 4;
}
оператор5;


switch (number) {
 case 1:  оператор 1;
 case 2:  оператор 2;
 case 3:  оператор 3;
defauit:  оператор 4;
}
оператор 5;
```

В обоих случаях значение **number** равно 2.

РИС. 7.8. Ход выполнения программы, использующей оператор **switch** при наличии или в

отсутствии операторов **break**

Когда мы хотим получить одинаковый результат при переходе к разным меткам, мы можем использовать метки без операторов. Например, фрагмент программы

```
case 'E':
case 'e':
         printf(" ехидна, муравьед колючий \n" );
         break;
```

свидетельствует о том, что указание букв **E** или **e** приводит к печати названия **"ехидна"**. Если будет введена буква **E**, то произойдет переход к соответствующей метке, но, поскольку там операторы отсутствуют, будут вы полняться операторы, расположенные ниже, пока не встретится оператор **break**.

Наша программа имеет две небольшие особенности, о которых мы хотели бы упомянуть.

Первая поскольку мы собираемся использовать программу в диалоговом режиме, мы решили воспользоваться символом # вместо **EOF** в качестве признака прекращения ее работы. В работе компьютера могли бы возникнуть сложности, если бы он предложил нам ввести признак **EOF** или даже какой-нибудь управляющий символ, между тем как символ # вполне подходит для этой цели. Поскольку теперь отсутствует необходимость чтения символа **EOF**, мы не должны описывать в программе переменную **ch** типа **int**.

Вторая мы использовали оператор **if**, который позволяет игнорировать символы "новая строка" при вводе символов в программу. Это тоже некоторая плата за возможность диалоговой работы. Без этого оператора **if** каждый раз при нажатии клавиши **[возврат]** программе пришлось бы рассматривать данный признак как прочитанный символ.

Когда требуется использовать оператор **switch**, а когда конструкцию **else-if**? Часто у нас нет возможности выбора. Вы не можете применить оператор **switch**, когда выбор вариантов основывается на вычислении значения переменной или выражения типа **int**. Удобного способа воспользоваться оператором **switch** в случае когда возможные значения переменной попадают в некоторый диапаон, также не существует. Проще написать, например, так:

```
if(integer < 1000 && integer > 2)
```

В то время как замена этой строки оператором **switch** приведет к необходимости ввести в программу метки для всех целых чисел от 2 до 999. Тем не менее, если у вас есть возможность применить оператор **switch**, ваша программа будет выполняться более эффективно.

Резюме: множественный выбор вариантов с помощью оператора switch

Управление в программе передается оператору, у которого в качестве метки используется значение некоторого *выражения*. Затем в процессе прохождения программы будут выполняться оставшиеся операторы, пока не произойдет новый переход.

Как *выражения*, так и метки должны иметь значения целого типа (включая тип **char**), метки должны быть константами или константными выражениями. Если не которому значению выражения не соответствует никакая метка, управление передается оператору с меткой **default** (если такой имеется). Впротивном случае управление передается оператору, следующему за оператором **switch**.

III. Форма:

```
switch (выражение)
{ case метка1: оператор1
```

```
  case метка2: оператор2
      default: оператор3
}
```

В операторе может присутствовать более чем 2 помеченных оператора, а наличие метки **default** является необязательным.

IV. Пример

```
switch (letter)
{ case 'a':
  case 'e': printf(" %c - это гласная \n ", letter);
  case 'c':
  case 'n': printf(" Символ %c в наборе букв \ cane\ n ", letter);
   default: printf(" Добрый день. \n" );
}
```

Если переменная **letter** имеет значение '**a**' или '**e**', будут выведены на печать все три сообщения, если же '**c**' или '**n**', то последние два. В случае остальных значений будет напечатано только последнее сообщение.

Изложенный здесь материал позволит вам писать гораздо более мощные и обладающие большими возможностями программы, чем раньше. В справедливости этого утверждения вы сможете убедиться, если сравните некоторые из примеров, приведенных в данной главе, с программами, рассмотренными в предыдущих главах. Но вы изучили еще далеко не все. Вот почему вам придется одолеть еще немало страниц этой книги.

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Как осуществить выбор из двух возможностей выполнить оператор или пропустить его: с помощью оператора **if**.
Как осуществить выбор одного из двух вариантов: с помощью оператора **if-else**.
Как осуществить выбор одного из нескольких вариантов: с помощью операторов **else-if**, **switch**
Операции отношения: > >= = = <= < !=
Логические операции: && || !
Операция условия: : ?

ВОПРОСЫ И ОТВЕТЫ

Содержание

Вопросы

1. Определите, какие выражения истинны, а какие ложны.
   а. 100 > 3
   б. 'a' > 'c'
   в. 100 > 3 && 'a' > 'c'
   г. 100 > 3 || 'a' > 'c'
   д. !(100 > 3)

2. Запишите выражения, соответствующие следующим условиям
   а. Значение **number** равно или больше 1, но меньше 9
   б. Значение **ch** не равно **q** или **k**
   в. Значение **number** лежит между 1 и 9, но не равно 5
   г. Значение **number** не лежит между 1 и 9

3. В программе, приведенной ниже, наряду с неоправданно сложными условными выражениями имеются и прямые ошибки. Уточните эту программу и исправьте в ней ошибки.

```
main( )            /* 1 */
{       /* 2 */
int  weight, height; /* вес в фунтах, рост в дюймах */
                                /* 4 */
scanf('' %d, weight, height);   /* 5 */
if(weight < 100)        /* 6 */
   if (height >= 72)    /* 7 */
printf(" Для такого веса у вас слишком большой рост \n");
else if (height < 72 && > 64)   /* 9 */
printf(" У вас большой рост для вашего веса. \n" );
else if (weight > 300 && !(weight  < = 300))    /* 11*/
if( !(height >= 48)     /* 12 */
printf(" Для такого веса у вас слишком маленький рост.\n" );
else    /* 14 */
printf(" У вас идеальный вес. \n");     /* 15*/
        /* 16 */
}
```

Ответы

1. Выражения истинны в вопросах а и г

2. а. **number > = 1** && **number < 9**

б. **ch != 'q'** && **ch != k**

Замечание: выражение **ch != q || ch!= k** всегда будет иметь значение "истина", потому что если переменная **ch** равна **q**, то она не может равняться **k**, и второе условие оказывается выполненным врезультате все выражения "ИЛИ" будет истинным.

в. **number > 1** && **number < 9** && **number != 5**

г. **!(number > 1 && number < 9)** или **number < = 1 || number > = 9**

Замечание: сказать, что число **НЕ** лежит между 1 и 9 это то же самое, что сказать: число равно или меньше 1 **ИЛИ** равно или больше 9. Вторая форма несколько неуклюже звучит на словах, но проще записывается в виде выражения.

3. Строка 5: должна выглядеть так **scanf(" %d %d", &weight, &height)**. Не забывайте указывать символы в качестве префиксов имен переменных в функции **scanf( )**. Кроме того, данной строке должна предшествовать строка, предлагающая ввести данные.

Строка 9: подразумеваемое выражение должно выглядеть так: **(height < 72 && height > 64)**. Однако первая часть этого выражения необязательна, поскольку величина **height**, если поток управления достигнет записи **else-if**, будет обязательно меньше 72. Поэтому более простое условие (**height** > 64) в данном случае служит той же цели.

Строка 11: избыточное условие; второе подвыражение (отрицание условия "величина **weight** меньше или равна **300**") означает то же, что и первое. В действительности данное условие записывается так: (**weight** > 300). Но неприятности на этом не кончаются. Строка 11 относится к ошибочному оператору **if**. Очевидно, что эта часть **else** ассоциируется с оператором **if**, расположенным в строке 6, но, согласно правилу, связывающему ее с ближайшим отрицанием условия, содержащегося в **if**, она будет ассоциироваться с оператором **if** на строке 9. Поэтому условие, помещенное на строке 11, будет проверяться в том случае, когда величина **weight** меньше 100, а величина **height** меньше или равна 64. Это делает невозможным превышение переменной **weight** значения 300 при выполнении данного оператора.

Строки 7-9 должны быть заключены в фигурные скобки. Тогда строка 11 станет альтернативой оператору, расположенному на строке 6, а не на строке 9.

Строка 12: данное выражение необходимо упростить так: (**height** < 48)

Строка 14: это ключевое слово **else** относится к последнему оператору **if**, расположенному на

строке 12. Операторы, помещенные на строках 12 и 13, необходимо заключить в фигурные скобки, тогда **else** будет относиться к оператору **if** на строке 11. Обратите внимание, что последнее сообщение будет напечатано для тех, чей вес заключен между 100 и 300 фунтами.

[1)] В программу входит слово **score** - счет (в игре) - *Прим. перев.*

[2)] См. перевод на с. 154 - *Прим. ред.*

[3)] В нем используется слово **legs** - ноги - *Прим. перев.*

[4)] В оригинале - игра слов *complex* (англ.) - сложный и комплексный - *Прим. перев.*

---

8. Циклы и другие управляющие средства

ЦИКЛЫ
ВЛОЖЕННЫЕ ЦИКЛЫ
ОСУЩЕСТВЛЕНИЕ ПЕРЕХОДОВ В ПРОГРАММЕ
ИСПОЛЬЗОВАНИЕ ЦИКЛОВ ПРИ РАБОТЕ С МАССИВАМИ

КЛЮЧЕВЫЕ СЛОВА
while, do, for, break, continue, goto

ОПЕРАЦИИ
+= -= *= /= %=

При усложнении решаемых задач ход выполнения программ становится более запутанным. Чтобы иметь возможность управлять процессом выполнения программ и его организацией, вам попадаютя структуры и некоторые специальные операторы. Язык предоставляет эффективные средства реализации таких требований. Мы уже смогли убедиться в чрезвычайной ценности цикла **if** в том случае, когда необходимо повторить некоторую операцию несколько раз. В языке Си, кроме того, реализовано еще два вида циклов: цикл **for** и цикл **do ... while**. В данной главе рассматриваются принципы работы управляющих структур и даются рекомендации, каким образом лучше всего применять каждую из них. Обсудим операторы **break**, **continue**, **goto** и операцию "запятая" все они могут использоваться для управления ходом выполнения программы. Кроме того, мы расскажем вам еще немного о свойствах, которые часто используются вместе с циклами.

ЦИКЛ while

Далее Содержание

В предшествующих главах мы интенсивно пользовались этой формой цикла, сейчас же хотим рассмотреть его работу в случае простой, возможно, даже примитивной программы, угадывающей число.

```
/* угадывание числа1 */
/* неэффективный способ угадывания */
#inlude
main( )
{
int guess = 1;
char response;
printf(" Задумайте целое число от 1 до 100. Я попробую угадать");
```

```
printf(" его.\n Отвечайте д, если моя догадка правильна и");
printf(" \n н, если я ошибаюсь. \n");
pintf("Итак ... ваше число %d?\n" , guess);
while((response = getchar( )) != 'д') /* получение ответа*/
    if (response != ' \n') /* пропуск символа "новая  строка" */
        printf(" Ну, тогда оно равно %d?\n", ++guess);
printf(" Я знала, что смогу сделать это!\n");
}
```

Обратите внимание на логику программы. Если вы отвечаете **д**, в программе осуществляется выход из цикла и переход на завершающий оператор печати. Программа просит вас отвечать и в случае, если ее догадка неверна, но фактически любой ответ, отличный от символа **д**, приводит к тому, что программа входит в цикл. Однако если введен символ "новая строка", то при данном прохождении тела цикла никаких операций не производится. Получение другого символа приводит к очередной попытке угадывания целого числа. (Что произошло бы в случае использования операции **guess++** вместо операции **++guess?**).

Строка **if(response ! = '\n')** позволяет программе игнорировать поступление постороннего символа "новая строка", когда вы нажмете клавишу [ввод].

В этом случае тело цикла **while** не нужно заключать в фигурные скобки, поскольку оператор **if**, хотя он и занимает две строки в программе, рассматривается как один оператор.

Вы, по-видимому, уже заметили, что это довольно "тупая" программа. Она написана правильно и решает поставленную задачу, но делает это крайне неэффективно. Данный пример показывает нам, что правильность написания - не единственный критерий, по которому необходимо оценивать программу. При этом очень важна ее эффективность. Мы вернемся к данной программе несколько позже и попытаемся ее немного улучшить.

В общем виде цикл **while** записывается так:

while(*выражение*) *оператор*

В наших примерах в качестве выражении использовались условные выражения, но, вообще говоря, это могут быть выражения произвольного типа. В качестве оператора можно использовать простой оператор с символом "точка с запятой" в конце или составной оператор, заключенный в фигурные скобки. Если выражение истинно (т.е в общем случае не равно нулю), то оператор, входящий в цикл **while** выполняется один раз, а затем выражение проверяется снова, а последовательность действий, состоящая из проверки и выполнения оператора, периодически повторяется до тех пор, пока выражение не станет ложным (или в общем случае равным нулю). Такой шаг называется "итерация". Данная структура аналогична структуре оператора **if**. Основное отличие заключается в том, что в операторе **if** проверка условия и (возможное) выполнение оператора осуществляется только один раз, а в цикле **while** эти действия производятся, вообще говоря, неоднократно.

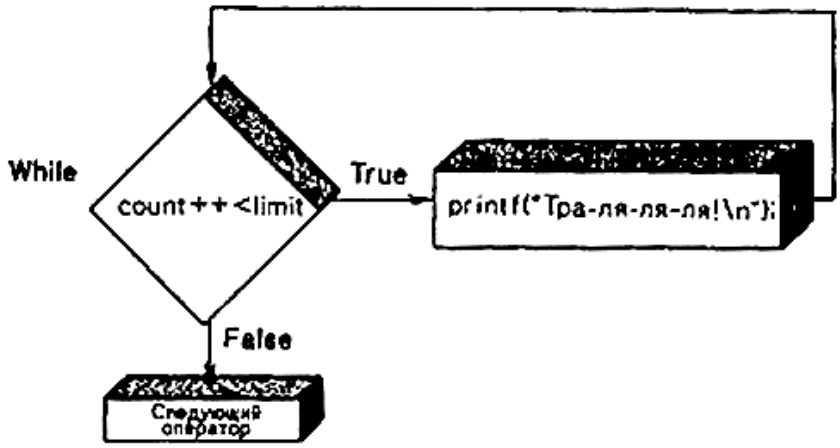


РИС. 8.1. Структура цикла **while**.

Завершение цикла while

Далее Содержание

Мы подошли к самому существенному моменту рассмотрения циклов **while**. При построении цикла **while** вы должны включить в него какие-то конструкции, изменяющие величину проверяемого выражения так, чтобы в конце концов оно стало ложным. В противном случае выполнение цикла никогда не завершится. Рассмотрим следующий пример:

```
index = 1;
while(index < 5)
printf("Доброе утро!\n");
```

Данный фрагмент программы печатает это радостное сообщение бесконечное число раз, поскольку в цикле отсутствуют конструкции, изменяющие величину переменной **index**, которой было присвоено значение 1.

```
index = 1;
while(--index < 5)
printf("Как колеблются старые атомы!\n");
```

И этот фрагмент программы работает ненамного лучше. Значение переменной **index** в нем изменяется, но в "неправильном" направлении! Единственным утешением здесь служит тот факт, что выполнение данного куска программы в конце концов завершится. Это произойдет, когда величина переменной **index** станет меньше наименьшего отрицательного числа, допустимого в системе.

Цикл **while** является "условным" циклом, использующим предусловие (т.е. условие на входе). Он называется условным, потому что выполнение оператора зависит от истинности условия, описываемого с помощью выражения. Действительно ли значение переменной **index** меньше 5? Является ли последний введенный символ признаком **EOF**? Подобное выражение задает предусловие, поскольку выполнение этого условия должно быть проверено *перед* началом выполнения тела цикла. В ситуации, аналогичной приведенной ниже, тело цикла не выполнится ни разу, потому что используемое условие с самого начала является ложным.

```
index = 10;
while(index++ < 5)
printf(" Желаю хорошо провести день.\n");
```

Измените первую строку на

```
index = 3;
```

и вы получите работающую программу.

АЛГОРИТМЫ И ПСЕВДОКОД

Далее Содержание

А теперь вернемся к нашей "тупоумной" программе, угадывающей число. Недостаток этой программы кроется не в программировании самом по себе, а в "алгоритме", т.е. методе, используемом для отгадывания числа. Этот метод можно описать следующим образом: попросите пользователя задумать число компьютер начинает угадывание с 1 до тех пор пока догадка неверна, предлагаемое значение увеличивается на 1.

Эта запись, между прочим, служит примером "псевдокода" представляющего собой способ выражения смысла программ на разговорном языке и являющегося некоторым аналогом языка машины. Псевдокод очень эффективен при разработке логики программы. После того как логика покажется вам правильной, вы можете обратить основное внимание на детали перевода псевдокода на реальный язык программирования. Преимущество использования псевдокода состоит в том, что он позволяет сконцентрироваться на логике и структуре программы, не заботясь пока о способе перевода этих идей на язык машины. Если мы хотим улучшить программу, нам в первую очередь необходимо улучшить алгоритм. Один из методов заключается в том, чтобы выбрать число где-нибудь посередине между 1 и 100 (50 нам вполне подходит) и попросить пользователя ответить больше ли это число задуманного, меньше его или равно ему. Если он сообщает, что данное число слишком велико, то тем самым из рассмотрения немедленно исключаются все числа между 50 и 100. Следующей догадкой программы является число, выбранное где-то посередине между 1 и 49. И снова ответ на вопрос, велико или мало это число, позволит исключить из рассмотрения половину оставшихся возможных чисел; программа продолжает указанный процесс, быстро сужая поле поиска до тех пор, пока задуманное число не будет угадано. Давайте запишем эти логические рассужденя на псевдокоде. Пусть **highest** - максимально возможная величина отгадываемого числа, а **lowest** - его минимально возможное значение. Вначале этими величинами будут соответственно 100 и 1, поэтому алгоритм запишется следующим образом:

установить **highest** равным 100
установить **lowest** равным 1
попросить пользователя задумать число
предложенное значение (**guess**) равно **(highest + lowest)/2**
пока догадка неверна, делать следующее:
{если предложенное значение велико, установить **highest** равным этому предложенному значению минус 1
если предложенное значение мало, установить **lowest** равным этому предложенному значению плюс 1
новое предложенное значение равно **(highest + lowest)/2** }

Обратите внимание на логику алгоритма: если предложенное значение, равное 50, велико, то максимально возможная величина задуманного числа будет равна 49. Если же значение 50 мало, то минимально возможная величина числа будет равна 51.

Сейчас мы переведем текст, указанный выше, на язык Си. Полученная программа представлена на рис. 8.2.

```
/* угадывание числа2 */
/* более эффективный способ угадывания*/
#include
#define HIGH 100
#define LOW 1
main( )
{
int guess = (HIGH + LOW)/2;
int highest = HIGH;
int lowest = LOW;
char response;
```

```
printf(" Задумайте число от %d до %d. Я попробую", LOW, HIGH);
printf(" угадать его.\n Отвечайте д, если моя догадка правильна,");
printf(" б, если \n больше, и м, если");
printf(" меньше.\n");
printf(" Итак ... ваше число %d?\n" , guess);
while((response = getchar( )) != 'д')
{ if( response != '\n')
{
if (response == 'б')
{ /* уменьшение верхнего предела,
 если предложенное значение слишком велико */
 highest = guess - 1;
 guess = (highest + lowest)/2;
 printf(" Гм ... слишком велико. Ваше число %d?\n", guess); }
 else if(response == 'м')
{ /* увеличение нижнего предела,
если предложенное значение слишком мало*/
lowest = guess + 1;
guess = (highest + lowest)/2;
printf(" Гм ... слишком мало. Ваше число %d?\n" , guess); }
else
{ /* подводите пользователя к правильному ответу */
printf(" Я не понимаю; введите, пожалуйста, д,б");
printf ("или м.\n");
} }
printf("Я знала, что смогу сделать это!\n"); }
```

РИС. 8.2. Программа, угадывающая число.

Наличие в программе завершающей части **else** предоставляет пользователю дополнительную возможность правильно ответить на стандартный "отклик" программы. Заметим также, что мы использовали символические константы, чтобы сделать процесс изменения диапазона чисел достаточно простым. Работает ли данная программа? Ниже приводятся результаты этого прогона. Задуманное число - 71.

```
Задумайте число от 1 до 100. Я попробую угадать его
Отвечайте д, если моя догадка правильна б, если
больше, и м, если меньше.
Итак ..., ваше число 50?

Я не понимаю: введите, пожалуйста, д,б или м.
м
Гм ... слишком мало. Ваше число 75?
б
Гм ... слишком велико. Ваше число 62?
м
Гм ... слишком мало. Ваше число 68?
м
Гм ... слишком мало. Ваше число 71?
д
Я знала, что смогу сделать это!
```

Что может быть неправильного в этой программе? Мы реализовали в ней защиту от ошибок, вызванных тем, что пользователи могут указывать неверные символы, поэтому здесь не должно быть никаких проблем. Единственное, что может повлиять на правильность работы программы: если вы вместо **м** укажете **б**, или наоборот. К сожалению, не существует способа заставить пользователя говорить правду и не делать ошибок. Тем не менее, если вы заинтересованы в этом, можете предпринять некоторые шаги. (Например, если захотите поразить свою шестилетнюю племянницу.) Во-первых, обратите внимание на то, что наш способ требует самое большее семи попыток для угадывания любого числа. (Каждая попытка уменьшает число возможностей наполовину. За семь попыток можно угадать любое число в диапазоне от 1 до 27- 1, или 127, что вполне достаточно для работы в диапазоне или 1 до 100.) Вы можете модифицировать программу так, чтобы она подсчитывала число попыток, и если окажется, что оно превышает 7, то тогда можно

вывести на печать сообщение с выражением недовольства, а затем восстановить первоначальные значения переменных **highest**, **lowest** и счетчика. Дополнительные изменения, которые можно внести в программу, заключаются в такой модификации операторов **if**, в результате которой допускался бы ввод как прописных, так и строчных букв.

Резюме: оператор **while**

Оператор **while** определяет операции, которые циклически выполняются до тех пор, пока проверяемое выражение не станет ложным, или равным нулю. Оператор **while** - это цикл с предусловием; решение, выполнять ли в очередной раз тело цикла, принимается перед началом его прохождения. Поэтому вполне возможно, тело цикла не будет выполнено ни разу. Оператор, образующий тело цикла, может быть либо простым, либо составным.

while(*выражение*) *оператор*

Выполнение оператора циклически повторяется до тех пор, пока выражение не станет ложным, или равным нулю.

ПРИМЕРЫ

```
while(n++ < 100) printf(" %d %d \n",n, 2*n + 1);
```

```
while(fargo < 1000)
 { fargo = fargo + step;
   step = 2 * step;
 }
```

В нашем последнем примере в цикле **while** используется "неопределенное " условие: мы не знаем заранее, сколько раз выполнится тело цикла перед тем, как выражение станет ложным. Во многих наших предыдущих примерах, однако, циклы **while** использовались для подсчета числа выполнении тела цикла. Ниже приведен краткий пример[1)] , который содержит подсчитывающий цикл

```
main( )
{ int = 1;                      /* инициализация */
  while (count <= NUMBER)    /* проверка     */
    { printf(" Будь моим Валентином !\n");   /*действие */
      count++;                  /* увеличение счетчика */
    }
}
```

Хотя цикл подобного типа прекрасно работает, это не лучший вариант его записи, поскольку операции, реализующие цикл, не собраны вместе. Рассмотрим этот вопрос более подробно. При организации цикла, когда его тело должно быть выполнено фиксиррованное число раз, осуществляются три операции: инициализация счетчика, сравнение его величины с некоторым граничным значением и увеличение значения счетчика при каждом прохождении тела цикла. Условное выражение, имеющееся в цикле **while**, берет на себя заботу о сравнении, а приращение значения счетчика осуществляется с помощью операции увеличения. Так же как это делалось раньше, можно объединить эти два действия в одно выражение, используя запись **count++ <= NUMBER**. Но инициализация счетчика осуществляется вне цикла, как, например, в нашем примере оператором **count = 1;**. При этом можно забыть о необходимости инициализации счетчика, а то, что может случиться, рано или поздно случается. Сейчас мы рассмотрим управляющий оператор, использование которого позволяет избежать этих проблем.

ЦИКЛ for

Далее Содержание

В цикле **for** все три вышеуказанных действия собраны вместе. Используя цикл **for**, фрагмент, приведенный выше, можно записать в виде одного оператора:

```
for(count = 1; count <= NUMBER; count++)
    printf(" Будь моим Валентином! \n ");
```

В круглых скобках содержатся три выражения, разделенные символом "точка с запятой". Первое из них служит для инициализации счетчика. Она осуществляется только один раз - когда цикл **for** начинает выполняться. Второе выражение - для проверки условия; она производится перед каждым возможным выполнением тела цикла. Когда выражение становится ложным (или в общем случае равным нулю), цикл завершается. Третье выражение вычисляется в конце каждого выполнения тела цикла. Ранее мы использовали его для увеличения значения счетчика **count**, но, вообще говоря, его использование этим не ограничивается. За заголовком цикла **for** следует простой или составной оператор. Рис. 8.3 служит иллюстрацией структуры цикла **for**.

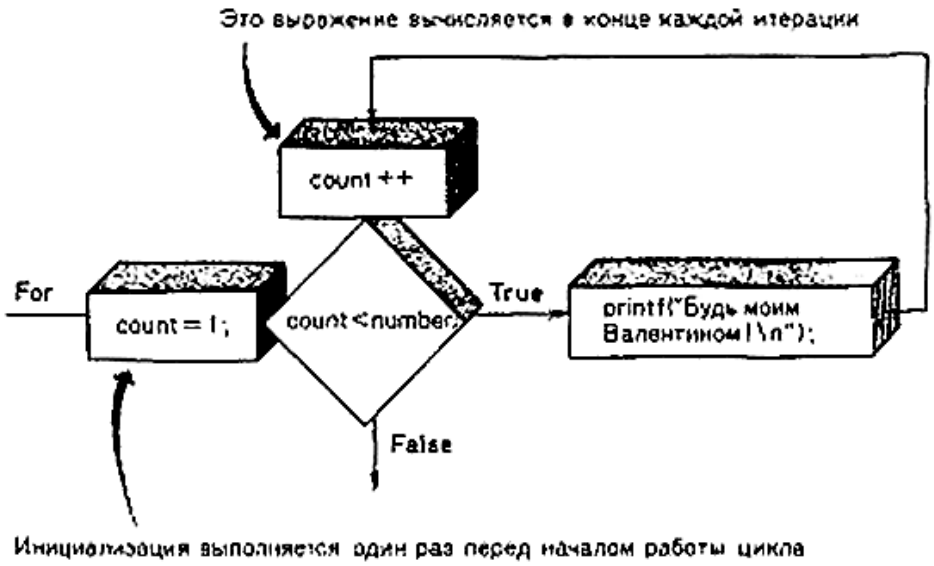


РИС. 8.3. Структура цикла **for**.

Сейчас мы продемонстрируем, как цикл **for** используется в программе, печатающей таблицу кубов целых чисел:

```
/* таблица кубов */
main( )
{ int num;
  for(num=1;num<=6;num++)
    printf(" %5d %5d \n", num, num*num*num);
}
```

программа выводит на печать числа от 1 до 6 и их кубы:

```
1
8
27
64
125
216
```

Из первой строки цикла **for** мы сразу можем узнать всю информацию о параметрах цикла: начальное значание переменной **num**, конечное значение, а также насколько увеличивается значение переменной **num** при каждом выполнении тела цикла. Цикл **for** часто используется для реализации в программе временной задержки с целью согласования скорости реагирования (в

даном случае замедления) машины с возможностями восприятия человека.

```
for(n=1; n<= 10000; n++)
;
```

Этот цикл заставляет машину считать до 10000. Единственный символ "точка с запятой", расположенный во второй строке, интересует нас о том, что никаких других действий в этом цикле не производится. Такой уединенный символ "точка с запятой" можно представлять себе как "пустой" оператор, т. е. оператор, который не выполняет никаких действий.

Гибкость конструкции for

Далее Содержание

Хотя цикл **for** на первый взгляд очень похож на цикл **DO** в Фортране, цикл **FOR** в Паскале и цикл **FOR ... NEXT** в Бейсике, **for** в Си является гораздо более гибким средством, чем любой из упомянутых. Эта гибкость - следствие способа использования упомянутых выше трех выражений в спецификации цикла **for**. До сих пор первое выражение применялось для инициализации счетчика, второе - для задания его граничного значения, а третье - для увеличения его текущего значения на 1. Использованный таким образом оператор **for** в языке Си совершенно аналогичен упомянутым выше соответствующим операторам в других языках. Но, кроме описанной, существует еще и много других возможностей его применения, девять из которых мы приводим ниже.

1. Можно применять операцию уменьшения для счета в порядке убывания вместо счета в порядке возрастания:

```
for(n = 10; n > 0; n--)
        printf(" %d секунд!\n", n);
 printf(" Пуск!\n");
```

2. При желании вы можете вести счет двойками, десятками и т. д.

```
for (n = 2; n & 60; n = n + 13)
printf(" %d\n", n);
```

В этом операторе значение переменной n будет увеличиваться на 13 при каждом выполнении тела цикла; будут напечатаны числа 2, 15, 28, 41 и 54.

Заметим, между прочим, что в языке Си имеется и другая сокращенная форма записи для увеличения переменной на фиксированную величину. Вместо выражения **n = n + 13** можно воспользоваться записью **n + = 13**.

Знак **+=** определяет "аддитивную операцию присваивания", в результате выполнения которой величина, стоящая справа, прибавляется к значению переменной, расположенной слева. Дополнительные детали, относящиеся к этой операции, приведены ниже.

3. Можно вести подсчет с помощью символов, а не только чисел.

```
for(ch = 'a' ; ch <= 'z'; ch++)
    printf(" Величина кода ASCII для %c равна %d.\n" , ch, ch);
```

При выполнении этого оператора будут выведены на печать все буквы от **a** до **z** вместе с их кодами ASCII. Этот оператор "работает", поскольку символы в памяти машины размещаются в виде чисел и поэтому в данном фрагменте счет ведется на самом деле с использованием целых чисел.

4.Можно проверить выполнение некоторого произвольного условия, отличного от условия, налагаемого на число итераций. В нашей программе таблица кубов вы могли бы заменить спецификацию

```
for(num = 1; num <= 6; num ++)
```

на

```
for(num = 1; num *num *num <= 216; num++)
```

Это было бы целесообразно в случае, если бы нас больше занимало ограничение максимального значения диапазона кубов чисел, а не количества итераций.

5. Можно сделать так, чтобы значение некоторой величины возрастало в геометрической, а не в арифметической прогрессии, т. е. вместо прибавления фиксированного значения на каждом шаге цикла, выполнялось бы умножение:

```
for(debt = 100.0; debt < 150.0; debt = debt*l.l)
    printf(" Ваш долг теперь $%.2f.\n", debt);
```

В этом фрагменте программы значение переменной **debt** умножается на 1.1 на каждом шаге цикла, что увеличивает ее на 10%. Результат выглядит следующим образом:

```
Ваш долг теперь $100.00
Ваш долг теперь $110.00
Ваш долг теперь $121.00
Ваш долг теперь $133.10
Ваш долг теперь $146.41
```

Как вы уже смогли догадаться, для умножения **debt** на 1.1 также существует сокращенная запись. Мы могли бы использовать выражение

```
debt * = 1.1
```

для получения того же результата. Знак ***=** определяет "мультипликативную операцию присваивания", при выполнении которой значение переменной, расположенной слева, умножается на величину, стаящую справа. (См. пояснения ниже, на с. 226.)

6. В качестве третьего выражения можно использовать любое правильно составленное выражение. Какое бы выражение вы ни указали, его значение будет меняться при каждой итерации.

```
for(x = 1; y <= 75; y = 5*x++ + 10);
    printf("%10d  %10d\n", x, y);
```

В этом фрагменте выводятся на печать значения переменной **x** и алгебраического выражения **5*x + 10**. Результат будет выглядеть так:

```
1 55
2 60
3 65
4 70
5 75
```

Обратите внимание, что в спецификации цикла проверяется значение **y**, а не **x**. В каждом из трех выражений, управляющих работой цикла **for**, могут использоваться любые переменные.

Хотя этот пример и правилен, он не может служить иллюстрацией хорошего стиля программирования. Программа выглядела бы гораздо понятнее, если бы мы не смешали процесс изменения переменной цикла с алгебраическими вычислениями.

7. Можно даже опустить одно или более выражений (но при этом нельзя опустить символы "точка с запятой"). Необходимо только включить в тело цикла несколько операторов, которые в конце концов приведут к завершению его работы.

```
ans = 2;
for (n = 3; ans <= 25;) ans = ans*n;
```

При выполнении этого цикла величина **n** останется равной 3. Значение переменной **ans** вначале будет равно 2, потом увеличится до 6, 18, а затем будет получена окончательная величина 54. (18 меньше 25, поэтому в цикле **for** выполняется еще одна итерация, и 18 умножается на 3, давая результат 54). Тело цикла

```
for(; ;)
    printf(" я хочу сделать что-нибудь\n");
```

будет выполняться бесконечное число раз, поскольку пустое условие всегда считается истинным.

8. Первое выражение не обязательно должно инициализировать переменную. Вместо этого, например, там мог бы стоять оператор **printf( )** некоторого специального вида. Необходимо помнить только, что первое выражение вычисляется только один раз перед тем, как остальные части цикла начнут выполняться.

```
for(printf("Запоминайте введенные числа!\n"); num == 6;)
scanf(" %d", &num);
printf("Это как раз то, что я хочу!\n");
```

В этом фрагменте первое сообщение оказывается выведенным на печать один раз, а затем осуществляется прием вводимых чисел до тех пор, пока не помтупит число 6.

9. Параметры, входящие в выражения, находящиеся в спецификации цикла, можно изменить при выполнении операций в теле цикла. Предположим, например, что у вас есть цикл со спецификацией следующего вида:

```
for(n = 1; n < 1000; n + = delta)
```

И если после нескольких итераций ваша программа решает, что величина параметра **delta** слишком мала или велика, оператор **if** внутри цикла может изменить значение параметра. В диалоговой программе пользователь может изменить этот параметр в процессе выполнения цикла.

Короче говоря, большая свобода выбора вида выражений, управляющих работой цикла **for**, позволяет с помощью этой конструкции делать гораздо больше, чем просто выполнять фиксированное число итераций. Возможности цикла **for** могут быть еще более расширены путем использования операций, которые мы вкратце обсудим ниже.

Резюме: оператор for

I. КЛЮЧЕВОЕ СЛОВО: **FOR**

II. ОБЩИЕ ЗАМЕЧАНИЯ:

В операторе **for** используются три выражения, управляющие работой цикла; они разделены символами "точка с запятой". *Инициализирующее* выражение вычисляется только один раз до начала выполнения какого-нибудь из операторов цикла. Если **проверяемое** выражение оказывается истинным (или не равным нулю), тело цикла выполняется один раз. Затем вычисляется величина **корректируемого** выражения, и значение **проверяемого** выражения определяется вновь. Оператор **for** - это цикл с *предусловием*: решение, выполнить в очередной раз тело цикла или **нет**, принимается *до начала* его прохождения. Поэтому может случиться так, что тело цикла не будет выполнено ни разу. *Оператор*, образующий тело цикла, может быть как простым, так и составным.

III. ФОРМА ЗАПИСИ:

```
for(инициализация; проверка условия; коррекция)  оператор
```

Тело цикла выполняется до тех пор, пока проверяемое условие не станет ложным или равным нулю

III.ПРИМЕР

```
for(n = 0; n < 10; n++)
printf(" %d %d\n", n, 2*n + 1);
```

Выше уже упоминалось о том, что в языке Си имеется несколько операций *присваивания*. Важнейшей из них является, конечно, операция **=**, при использовании которой значение выражения справа от знака присваивается переменной слева от него. Остальные операции присваивания корректируют значения переменных В записи каждой из них имеются имя переменной, стоящее слева от знака операции, и выражение справа от него Переменной присваивается новое значение, равное старому, скорректированному с помощью величины выражения, стоящего справа. Результат зависит от используемой операции. Например: **scores+= 20** то же самое, что **scores = scores + 20**, **dimes -= 20** то же самое, что **dimes = dimes - 2**, **bunnies *= 2** то же самое, что **bunnies = bunnies * 2**, **time /= 2.73** то же самое, что **time = time / 2.73**, **reduce %= 3** то же самое, что **reduce = reduce % 3**.

Правые части здесь являются обыкновенными числами, но мы могли бы использовать и более сложные выражения

```
x*= 3*y + 12
```

то же самое, что и

```
x = x*(3*y + 12)
```

Этим операциям присваивания назначен тот же низкий приоритет, что и обычной операции **=**, т.е. меньший, чем операциям **+** или *****. Это и отражено в последнем примере. Вам совершенно не обязательно использовать все эти формы. Однако они более компактны, и при трансляции обычно позволяют получить более эффективный машинный код, чем традиционная, более длинная запись. Они бывают особенно полезны в том случае, когда вы хотите поместить некоторое выражение в спецификацию цикла **for**.

Операция "запятая"

Далее Содержание

Операция "запятая" увеличивает гибкость использования цикла **for**, позволяя включать в его спецификацию несколько инициализирующих или корректирующих выражений. Например, ниже приводится программа, которая выводит на печать величины почтовых тарифов первого класса обслуживания. (Во время написания этой книги почтовые тарифы были такими: 20 центов за первую унцию и по 17 центов за каждую следующую.)

```
/* почтовые тарифы */
#define FIRST 20
#define NEXT 17
main( )
{
int ounces, cost;
printf(" унции стоимость \n");
for(ounces = 1, cost = FIRST; ounces <= 16; ounces++, cost+ = NEXT)
    printf(" %3d %7d\n" , ounces, cost);
}
```

Первые четыре строки результата работы программы будут выглядеть следующим образом:

```
унции           стоимость
1               20
2               37
3               54
```

Мы воспользовались операцией "запятая" в первом и третьих выражениях: в первом случае она позволяет инициализировать переменные **ounces** и **cost**; во втором - на каждой итерации увеличивать значение **ounces** на 1, а **cost** на 17 (величину константы **NEXT**). Все вычисления осуществляются в спецификации цикла **for**. Применение операции "запятая" не ограничено только циклами **for** но именно в них она используется особенно часто. Операция обладает одним дополнительным свойством: при ее использовании гарантируется, что выражения, к которым она применяется (т. е. выражения, разделенные запятой), будут вычисляться слева направо. Поэтому переменная ounces будет инициализирована до переменной **cost**. В данном примере это не имеет значения, но порядок инициализации мог бы оказаться существенным, если выражение, соответвующее **cost**, содержало бы переменную ounces. Символ "запятая" также используется как разделитель. Поэтому запятые в операторах: **char ch, date;**

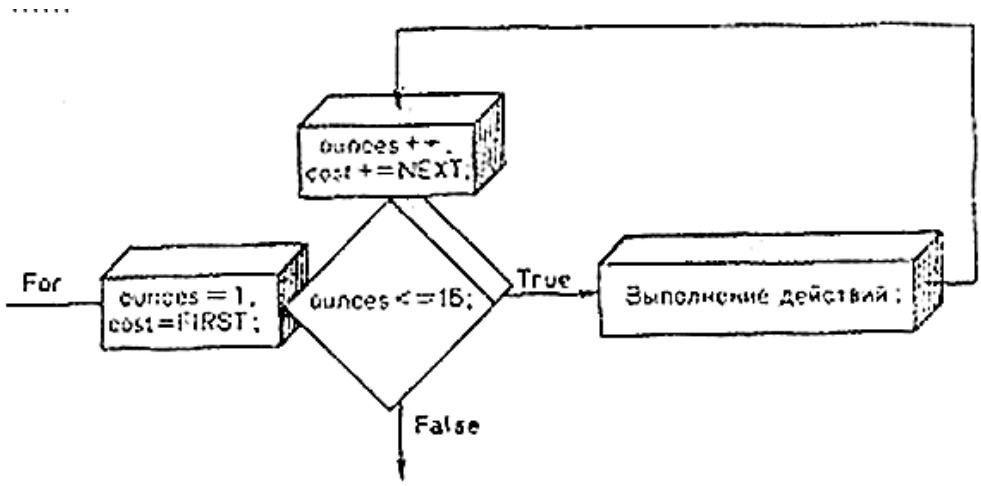


РИС. 8.4. Операция "запятая" и цикл **for**

ИЛИ

```
printf(" %d  %d\n", chimps, chumps);
```

являются разделителями, а не знаками операции "запятая".

резюме: наши новые операции

I. ОПЕРАЦИЯ ПРИСВАИВАНИЯ

Каждая из этих операций корректирует значение переменной слева от знака с помощью величины справа от него, в соответствии с указанной операцией. Ниже мы используем обозначение п.ч. для правой части, а л.ч. для левой части.

```
+=  прибавляет величину п.ч. к переменной л.ч.
-=  вычитает величину п.ч. из переменной л.ч.
*=  умножает неременную л.ч. на величину п.ч.
/=  делит переменную л.ч. на величину п.ч.
%=  дает остаток от деления переменной л.ч. на величину и.ч.
```

ПРИМЕР:

rabbits *= 1.6; то же самое, что и **rabbits * 1.6;**

II. ДОПОЛНИТЕЛЬНЫЕ ОПЕРАЦИИ:ОПЕРАЦИЯ "ЗАПЯТАЯ"

Операция "запятая" связывает два выражения в одно и гарантирует, что самое левое выражение будет вычисляться первым. Обычно она используется для включения дополнительной информации в спецификацию цикла **for**.

Пример:

```
for(step == 2, fargo = 0; fargo < 1000; step *= 2)
    fargo + = step;
```

Философ Зенон[2)] и цикл for

Далее Содержание

Посмотрим, как с помощью операции "запятая" можно разрешить старый парадокс. Греческий философ Зенон утверждал, что пущенная стрела никогда не достигнет цели. Сначала, говорил он, стрела пролетит половину расстояния до цели. После этого ей останется пролететь половину всего расстояния, но сначала она должна будет пролететь половину того, что ей осталось пролететь, и т. д. до бесконечности. Поскольку расстояние полета разбито на бесконечное число частей, для достижения цели стреле может потребоваться бесконечное время. Мы сомневаемся, однако, что Зенон вызвался бы стать мишеныо для стрелы, полагаясь только на убедительность своего аргумента. Применим количественный подход и предположим, что за одну секунду полета стрела пролетает первую половину расстояния. Тогда за последующую 1/2 секунды она пролетит половину того, что осталось от половины, за 1/4 - половину того, что осталось после этого, и т д. Полное время полета представляется в виде суммы бесконечного ряда 1 + 1/2 + 1/4 + 1/8 + 1/16 -1- ... . Мы можем написать короткую программу для нахождения суммы первых нескольких членов.

```
/* Зенон */
#define LIMIT 15
main ( )
{
int count;
float sum, x;
for(sum = 0.0, x = 1.0, count = 1; count <= LIMIT; count++, x *= 2.0)
{ sum + = 1.0/x;
  printf(" sum = %f когда count = %d.\n" , sum, count);
}
}
```

В результате выполнения программы получим значения сумм, соответствующих первым 15 членам ряда:

```
sum = 1.000000 когда count = 1.
sum = 1.500000 когда count = 2.
sum = 1.750000 когда count = 3.
sum = 1.875000 когда count = 4.
sum = 1.937500 когда count = 5.
sum = 1.968750 когда count = 6.
sum = 1.984375 когда count = 7.
sum = 1.992188 когда count = 8.
sum = 1.996094 когда count = 9.
sum = 1.998047 когда count = 10.
sum = 1.999023 когда count = 11.
sum = 1.999512 когда count = 12.
sum = 1.999756 когда count = 13.
sum = 1.999878 когда count = 14.
sum = 1.999939 когда count = 15.
```

Можно видеть, что, хотя мы и добавляем новые члены, сумма, по-видимому, стремится к какому-то пределу. И действительно, математики показали, что при стремлении числа членов к бесконечности сумма ряда сходится к 2,0, что и демонстрируется нашей программой. Какая

радость! Если бы Зенон оказался прав, движение было бы невозможно. (Но если бы движение было невозможно, то не было бы Зенона[3]).

Что можно сказать по поводу самой программы? В ней показано, что в одном выражении можно использовать более, чем одну операцию "запятая". В спецификации цикла мы инициализировали переменные **sum**, **x** и **count**. После задания условий выполнения цикла оставшаяся часть программы оказывается очень короткой.

ЦИКЛ С УСЛОВИЕМ НА ВЫХОДЕ: do while

Далее Содержание

Оба цикла, **while** и **for**, являются циклами с предусловиями. Проверка истинности условия осуществляется перед началом каждой итерации цикла. В языке Си имеется также конструкция цикла с постусловием (условием на выходе), где истинность условия проверяется после выполнения каждой итерации цикла. Этот подход реализуется с помощью цикла **do while**, который иллюстрируется следующим примером.

```
do
{
ch = getchar( );
putchar(ch);
} while(ch != ' \n')
```

Это сильно отличается от записи, например, такого вида

```
while((ch = getchar( )) != '\n') putchar(ch);
```

Различие начинается с того момента, когда прочитан символ "новая строка". Цикл **while** печатает все символы *вплоть* до появления первого символа "новая строка", а цикл **do while** - все символы *вплоть до символа "новая строка" включительно*. Только после печати этого символа в цикле производится проверка, является ли последний прочитанный символ символом "новая строка" В цикле **while** эти действия осуществляются перед проверкой истинности условия. В общем виде цикл **do while** записывается следующим образом:

```
do
оператор
while(выражение);
```

Такой оператор может быть как простым, так и составным.

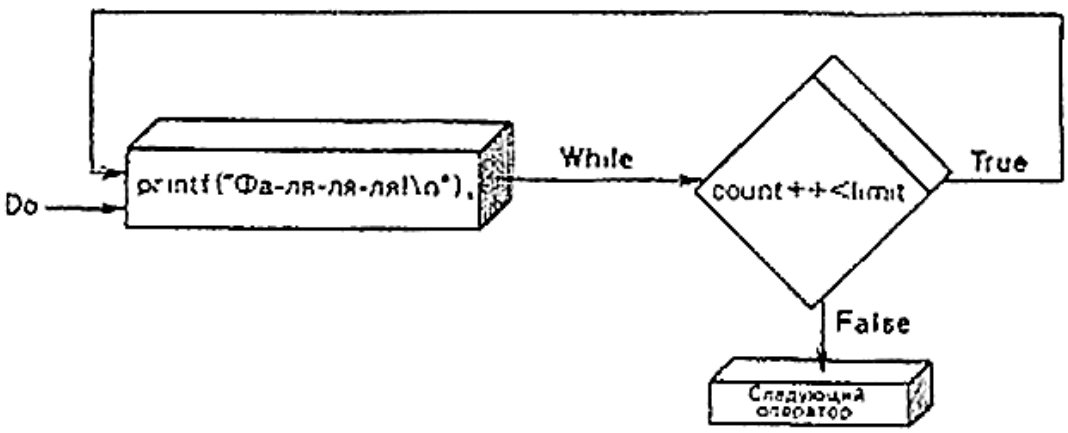


РИС. 8.5. Структура цикла **do while**.

Тело цикла **do while** всегда выполняется по крайней мере один раз, поскольку проверка осуществляется только после его завершения. Тело цикла **for** или **while**, возможно, не будет выполнено ни разу, поскольку проверка осуществляется перед началом его выполнения. Использовать цикл **do while** лучше всего в тех случаях когда должна быть выполнена по крайней мере одна итерация, к примеру, мы могли бы применить цикл **do while** в нашей программе угадывания числа. На псевдокоде алгоритм работы программы можно тогда записать следующим образом:

```
do
{
    выдвиньте предположение получите ответ вида д, б, или м }
while(ответ не совпадает с д)
```

Вы должны избегать использования цикла **do while**, структура которого аналогична представленной ниже в записи на псевдокоде.

```
спросите пользователя, хочет ли он продолжать
do
некоторый умный вздор
while(ответ будет да)
```

В данном случае, после того как пользователь ответит "нет", "некоторый умный вздор" будет выполнен, поскольку проверка осуществляется слишком поздно.

Резюме: оператор do while

I. Ключевые слова: **do while**

II. Общие замечания:

Оператор **do while** определяет действия, которые циклически выполняются до тех пор, пока проверяемое *выражение* не станет ложным, или равным нулю. Оператор **do while** - это цикл с *постусловием*; решение, выполнять или нет в очередное раз тело цикла, принимается *после* его прохождения. Поэтому тело цикла будет выполнено по крайней мере один раз. *Оператор*, образующий тело цикла, может быть как простым, так и составным.

III. Форма записи

```
do оператор
while(выражение);
```

Выполнение *оператора* повторяется до тех пор, пока *выражение* не станет *ложным*, или равным нулю.

IV. Пример

```
do
        scanf(" %d" , amp;number);
while(number != 20);
```

КАКОЙ ЦИКЛ ЛУЧШЕ? Далее Содержание

После того как вы решили, что вам необходимо использовать оператор цикла, возникает вопрос: циклом какого вида лучше всего воспользоваться? Во-первых, решите, нужен ли вам цикл с

предусловием или же с постусловием. Чаще вам нужен будет цикл с предусловием. По оценкам Кернигана и Ритчи; в среднем циклы с постусловием (**do while**) составляют только 5% общего числа используемых циклов. Существует несколько причин, по которым программисты предпочитают пользоваться циклами с. предусловием; в их числе один общий принцип, согласно которому лучше посмотреть, куда вы прыгаете, до прыжка, а не после. Вторым моментом является то, что программу легче читать, если проверяемое условие находится в начале цикла. И наконец, во многих случаях важно, чтобы тело цикла игнорировалось полностью, если условие вначале не выполняется.

Положим, вам необходимо использовать цикл с предусловием. Что лучше: цикл **for** или цикл **while**? Отчасти это дело вкуса, поскольку все, что вы можете сделать с помощью одного, вы можете сделать и с помощью другого. Для превращения цикла **for** в цикл **while** нужно опустить первое и третье выражения:

```
for(; проверка условия;)
```

Такая спецификация эквивалентна записи

```
while(проверка условия)
```

Для превращения цикла **while** в цикл **for** необходимо предварительно осуществить инициализацию некоторых выбранных переменных и включить в тело цикла операторы, корректирующие их значения:

```
инициализация;
while (проверка условия)
{тело;
коррекция;}
```

Данная запись по своим функциональным возможностям эквивалентна следующей:

```
for(инициализация; проверка условия, коррекция) тело;
```

Исходя из соображений стиля программирования, применение цикла **tor** представляется более предпочтительным в случае, когда в цикле используется инициализация и коррекция переменной, а применение цикла **while** - в случае, когда этого нет. Поэтому использование цикла **while** вполне оправданно в случае

```
while((ch = getchar( )) != EOF)
```

Применение цикла **for** представляется более естественным в случаях, когда в циклах осуществляется счет прохождений с обновлением индекса:

```
for (count = 1; count <= 100; count++)
```

ВЛОЖЕННЫЕ ЦИКЛЫ

Далее Содержание

Вложенным называется цикл, находящийся внутри другого цикла. В этом разделе рассматривается пример, в котором вложенные циклы используются для нахождения всех простых чисел, не превышающих данного значения. Простое число - это такое число, которое делится нацело только на 1 и само на себя. Первыми простыми числами будут 2, 3, 5, 7 и 11.

Самый легкий способ узнать, является ли число простым, состоит в делении его на все числа между 1 и им самим. Если оно делится нацело на какое-нибудь число из этого ряда, то оно - не простое. Мы воспользуемся операцией деления по модулю (%) для проверки, выполнялось ли деление нацело. (Вы не забыли еще, конечно, операцию деления по модулю? Ее результатом

является остаток от деления первого операнда на второй. Если одно число делится на другое нацело, результатом операции деления помодулю будет 0.) При обнаружении какого-нибудь одного делителя числа дальнейшие проверки потеряют смысл. Поэтому в программе процесс проверки данного числа завершается после того, как найден его делитель.
Начнем с программы, проверяющей делимость одного числа. В ней имеется всего один оператор цикла.

```
/* простое число1 */
main( )
{
int number, divisor;
printf(" О каком числе  вы      хотите знать, простое ли оно?\n");
scanf(" %d" , &number); /* получение ответа */
while(number <2)        /* число отвергается */
{       printf(" Извините, мы не принимаем чисел меньше 2.\n");
        printf(" Пожалуйста, попробуйте еще раз.\n");
        scanf(" %d" , &number);
}
for(divisor = 2; number % divisor != 0; divisor++)
        ; /* проверка, простое число или нет,
        осуществляется внутри спецификации цикла */
if (divisor == number) /* выполняется после завершения цикла */
        printf(" %d - простое число.\n", number);
else printf(" %d - не простое число.\n", number);
}
```

Мы воспользовались структурой цикла **while**, чтобы избежать ввода значений, которые могли бы привести к аварийному завершению программы.

Обратите внимание, что все вычисления выполняются внутри спецификации цикла **for**. Величина переменной **number** последовательно делится на возрастающие значения делителей до тех пор, пока не произойдет деление нацело (т. е. **number % divisor** станет равным **0**). Если первым делителем, который приведет к такому результату окажется само это число, то значение переменной **number** - простое число. В противном случае данное число будет иметь меньший делитель, и это приведет к тому, что цикл завершится раньше.

Для нахождения всех простых чисел, меньших некоторой заданной величины, нам нужно будет заключить наш цикл **for** в некоторый другой цикл. На псевдокоде это будет выглядеть следующим образом:

```
для числа (number)=1 до верхнего предела limit проверять, является ли число простым
```

Вторая строка представляет собой нашу предыдущую программу.
Переводя эту запись на язык Си, получим программу:

```
/* простые числа2 */
main( )
{
int number, divisor, limit;
int count = 0;

printf(" Укажите, пожалуйста, верхний предел для поиска простых чисел.\n");
printf(" Верхний предел должен быть 2 или больше.\n");
scanf(" %d", &limit);
while(limit < 2) /* вторая попытка, если ошибка при вводе */
{       printf(" Вы были невнимательны! Попробуйте еще раз. \n");
        scanf(" %d", &limit);}
printf(" Сейчас будут печататься простые числа!\n");
for(number = 2; number <= limit; number++) /* внешний цикл */
{  for(divisor =2; number % divisor != 0; divisor++)
        ;
   if(divisor == number)
```

```
        { printf(" %5d", number);
          if(++count % 10 == 0)
                printf(" \n"); /* новая строка начинается
                                  через каждые 10 простых чисел */
        }
}
printf(" \n Вот и все!\n");
}
```

Во внешнем цикле каждое число, начиная с 2 и кончая величиной **limit**, последовательно берется для проверки. Указанная проверка осуществляется во внутреннем цикле. Мы использовали переменную **count** для хранения счетчика получаемых простых чисел. При печати каждое одиннадцатое простое число мы начинаем с новой строки. Ниже приводится пример результатов, получаемых с помощью такой программы:

```
Укажите, пожалуйста, верхний предел для поиска простых чисел.
Верхний предел должен быть 2 или больше.
250
Сейчас будут печататься простые числа!
2    3    5    7    11   13   17   19   23   29
31   37   41   43   47   53   59   61   67   71
73   79   83   89   97   101  103  107  109  113
127  131  137  139  149  151  157  163  167  173
179  181  191  193  197  199  211  223  227  229   233  239  241
Вот и все!
```

Этот способ решения довольно прост, но он не является самым эффективным. Например, если вы хотите узнать, является ли число 121 простым или нет, нет необходимости проверять, существуют ли у него делители, превышающие 11. Дело в том, что если у данного числа существует делитель, больший 11, то результатом деления будет число, меньшее 11; тогда этот делитель был бы обнаружен раньше. Поэтому требуется проверять только делители, не превышающие величину квадратного корня из числа, но в данном случае программа будет несколько сложнее. Мы оставляем ее в качестве упражнения любознательному читателю. (*Указание*: вместо того чтобы сравнивать делитель с величиной квадратного корня из числа, сравнивайте квадрат делителя с самим числом.)

ДРУГИЕ УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ: break, continue, goto

Далее Содержание

Операторы, определяющие циклические вычисления, которые только что обсуждались, и условные операторы (**if, if-else, switch**) являются важнейшими средствами управления выполнением программы на языке Си. Они должны использоваться для реализации общей структуры программы. Три оператора, рассматриваемые ниже, обычно применяются реже, поскольку слишком частое их использование ухудшает читаемость программы, увеличивает вероятность ошибок и затрудняет ее модификацию.

break:

Важнейшим из этих трех управляющих операторов является оператор **break**, который уже встречался нам при изучении оператора **switch**. Он может использоваться в операторе **switch**, где часто это просто необходимо, а также в циклах любого из трех типов. Когда в ходе выполнения программы встречается указанный оператор, его выполнение приводит к выходу из конструкций **switch**, **for**, **while** или **do while**, в которых он содержится, и переходу к следующему оператору программы. Если оператор **break** находится внутри некоторой совокупности вложенных структур, его действие распространяется только на самую внутреннюю структуру, в которой он непосредственно содержится.

Бывает, что **break** используется для выхода из цикла в тех случаях, когда заданы два разных условия прекращения его работы. Ниже приводится цикл, реализующий эхо-печать символов и завершающийся при чтении либо признака **EOF**, либо символа **"новая строка"**:

```
while((ch = getchar( ))!= EOF)
{
        if(ch == '\n') break;
        putchar(ch);
}
```

Мы сделаем логику этою фрагмента программы более понятной, если объединим обе проверки в одном выражении:

```
while((ch = getchar( )) != EOF && ch != '\n') putchar(ch);
```

Если вы обнаружите, что **break** является частью оператора **if**, посмотрите, нельзя ли по-другому выразить это условие (как мы только что сделали), чтобы необходимость его использования отпала.

continue:

Этот оператор может использоваться во всех трех типах циклов, но не в операторе **switch**. Как и в случае оператора **break**, он приводит к изменению характера выполнения программы. Однако вместо завершения работы цикла наличие оператора **continue** вызывает пропуск "оставшейся" части итерации и переход к началу следующей. Заменим оператор **break** в последнем фрагменте на **continue**:

```
while((ch = getchar( ))!==EOF)
{ if(ch == '\n')
        continue;
  putchar(ch); }
```

В версии, использующей оператор **break**, работа цикла полностью прекращается, как только при вводе встречается символ "новая строка". В версии с оператором **continue** просто пропускаются символы **"новая строка"**, а выход из цикла происходит, только когда читается признак **EOF**.
Этот фрагмент, конечно, более компактно можно записать следующим образом:

```
while((ch=getchar( ))!= EOF)
         if(ch != '\n') putchar(ch);
```

Очень часто, так же как и в данном случае, изменение условия в операторе **if** на обратное позволяет исключить необходимость введения в цикл оператора **continue**.

С другой стороны, оператор **continue** помогает иногда сократить некоторые программы, особенно если они включают в себя вложенные операторы **if else**.

goto:

Оператор **goto** - одно из важнейших средств Бейсика и Фортрана - также реализован и в Си. Однако на этом языке в отличие от двух других можно программировать, совершенно не используя указанное средство. Керниган и Ритчи считают оператор **goto** "чрезвычайно плохим" средством и предлагают "применять его как можно реже или не применять совсем".

Сначала мы покажем, как его использовать, а затем объясним, почему этого не нужно делать.

Оператор **goto** состоит из двух частей - ключевого слова **goto** и имени метки. Имена меток образуются по тем же правилам, что и имена переменных. Приведем пример записи оператора

```
goto pait2;
```

Чтобы этот оператор выполнился правильно, необходимо наличие другого оператора, имеющего метку **part2;** в этом случае запись оператора начинается с метки, за которой следует двоеточие.

```
part2: printf(" Уточненный анализ:\n");
```

Избегайте использовать goto

Далее Содержание

В принципе вы никогда не обязаны пользоваться оператором **goto** при программировании на Си. Но если ваш предыдущий опыт связан с работой на Фортране или Бейсике, в каждом из которых требуется его использовать, то у вас могли выработаться навыки программирования, основанные на применении данного оператора. Чтобы помочь вам преодолеть эту привычку, ниже вкратце приводится несколько знакомых вам ситуации, реализуемых с помощью goto, а затем показывается, как это можно осуществить другими средствами, в большей степени соответствующими духу языка Си.

1. Работа в ситуации, когда в операторе **if** требуется выполнить более одного оператора:

```
if(size > 12)
goto a;
goto b;
a: cost = cost * 1.05;
flag = 2;
b: bill = cost * flag;
```

(В стандартных Бейсике и Фортране только один оператор, непосредственно следующий за **if**-условием, считается относящимся к оператору **if**. Мы выразим это с помощью эквивалентного фрагмента на Си.)

Обычный подход, применяемый в языке Си и заключающийся в использовании составного оператора, или блока, упрощает понимание смысла программы:

```
if(size > 12);
{ cost = cost * 1.05;
        flag = 2; }
bill = cost * flag;
```

2. Осуществление выбора из двух вариантов:

```
if(size > 14) goto a;
sheds = 2;
goto b;
a: sheds = 3;
b: help = 2 * sheds;
```

Наличие в языке Си структуры **if-else** позволяет реализовать такой выбор более наглядно:

```
if(ibex > 14)
        sheds = 3;
else
        sheds = 2;
help = 2 * sheds;
```

3. Реализация бесконечного цикла:

```
readin: scanf(" %d", &score);
if(score < 0) goto stagc2;
```

```
большое количество операторов;
goto readin:
stagc2: дополнительная чепуха;
}
```

Эквивалентный фрагмент, в котором используется цикл **while**, выглядит так:

```
scanf(" %d", &score);
whilet(score >= 0)
{ большое количество операторов;
  scanf("%d", &score);
}
дополнительная чепуха;
```

4. Пропуск операторов до конца тела цикла: используйте оператор **continue**.

5. Выход из цикла: используйте оператор **break**. Фактически **break** и **continue** являются специальными формами оператора **goto**. Преимущество их использования заключается в том, что, во-первых, названия этих операторов говорят об осуществляемых ими функциях, а во-вторых, поскольку они не используют меток, отсутствует опасность пометить не тот оператор программы.

6. Выполнение переходов к различным частям программы непредсказуемым образом: так программировать нельзя!

Существует один случай, когда использование оператора **goto** допускается опытными программистами, работающими на языке Си,- это выход из вложенного набора циклов при обнаружении каких-то ошибок. (Применение оператора **break** даст возможность осуществить выход только из самого внутреннего цикла.)

```
while(funct > 0)
{ for(i = 1; i < 100; i++)
        { for(j = 1; j<= 50; j++)
                { большое число операторов;
                  if(большая ошибка)
            goto help;
          операторы;  }
          еще некоторое количество операторов;  }
   и еще некоторое количество операторов;  }
и еще операторы;
help: устранение ошибки;
```

Как вы можете заметить из наших примеров, альтернативные формы представления программ более понятны, чем формы, использующие оператор **goto**. Эти различия станут еще большими, если вы объедините некоторые из рассмотренных случаев. Какие операторы goto используются при реализации операторов **if**, какие моделируют конструкции **if-else**, какие управляют работой циклов, а какие появляются лишь потому, что пользователь написал программу так, что не может без них обойтись? Чрезмерное увеличение оператором **goto** приводит к созданию лабиринта в логике программы. *Дадим вам совет:* если вы совсем не знакомы с оператором **goto**, то не применяйте его вовсе; если вы привыкли пользоваться им, попытайтесь отучить себя от этого. Ирония заключается в том, что в языке Си, который вовсе не нуждается в операторе **goto**, его структура оказывается лучшей, чем в большинстве других языков программирования, поскольку в качестве меток можно использовать смысловые имена, а не числа.

Резюме: переходы в программах

I. Ключевые слова: **break, continue, goto**

II. Общие замечания

Выполнение каждого из этих трех операторов языка вызывает скачкообразное изменение процесса выполнения программы, т. е. переход от одной команды программы к другой (не следующий за ней непосредственно).

III. break

Оператор **break** можно использовать внутри любой из трех форм цикла и конструкции **switch**. Его выполнение приводит к тому, что управление программой, минуя оставшуюся часть тела цикла или конструкцию **switch**, содержащую данный оператор, передается на следующую (за этим циклом или за конструкцией **switch**) команду.

Пример:

```
switch(number)
{ case 4: printf(" Это хороший выбор.\n");
  break;
  case 5: printf(" Это неплохой выбор.\n");
  break;
  default: printf(" Это плохой выбор.\n");
}
```

IV. Continue

Оператор **continue** может использоваться в любой из трех форм циклов, но не в операторе **switch**. Его выполнение приводит к такому изменению логики программы, что остальные операторы тела цикла пропускаются. Для циклов **while** или **for** вслед за этим начинается новый шаг, а для цикла **do while** проверяется условие на выходе, и затем, если оно оказывается истинным, выполняется следующая итерация.

Пример

```
while((ch = gctchar( ))!= EOF)
{ if(ch == ' ') continue;
  putchar(ch);
  chcount++; }
```

В этом фрагменте осуществляется эхо-печать читаемых символов и подсчитывается число символов, отличных от пробела.

V. goto

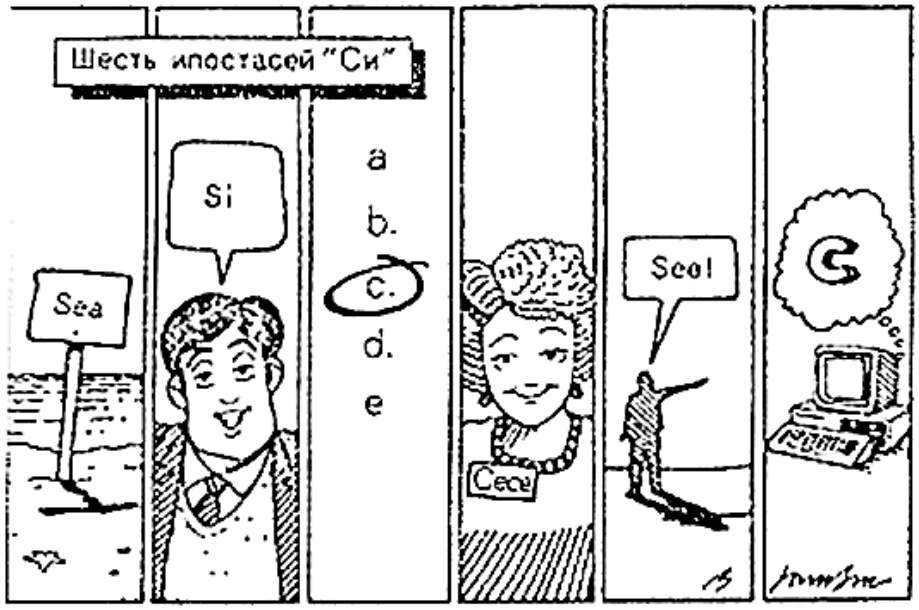
Выполнение оператора **goto** вызывает передачу управления в программе оператору, помеченному указанной меткой. Для отделения оператора от соответствующей ему метки используется двоеточие. Имена меток образуются но тем же правилам, что и имена переменных. Помеченный оператор может появиться в программе текстуально до или после **goto**.

Форма:

```
goto метка;
...
метка: оператор
```

Пример

```
top : ch  = getchar( );
...
if (ch !=  'y')
goto top;
```



МАССИВЫ

Далее Содержание

Массивы являются важнейшим средством языка, применяемым во многих программах. Их использование позволяет удобным способом размещать в памяти большое количество необходимой информации. Позже мы посвятим массивам целую главу, но, поскольку они очень тесно связаны с циклами, мы хотим начать их использовать уже сейчас.

Массив - это набор переменных, имеющих одно и то же базовое имя и отличающихся одна от другой числовым признаком. Например, с помощью описания

```
float debts [20];
```

объявляется, что **debts** - массив, состоящий из двадцати членов, или "элементов". Первый элемент массива называется **debts[0]**, второй - **debts[l]**, и т. д. вплоть до **debts[19]**. Заметим, что перечисление элементов массива начинается с **0**, а не с **1**. Поскольку мы объявили, что массив имеет тип **float**, каждому его элементу можно присвоить величину типа **float**. К примеру, можно писать так:

```
debts[5] = 32.54;
debts[6] = 1.2e+21;
```

Массивы могут быть образованы из данных любого тина:

```
int nannies[22];  /* масснв, содержащий 22 целых чнсла */
char alpha[26];   /* масснв, содержащий 26 символов */
long big[500];    /*массив, содержащий 500 целых чисел типа long */
```

Раньше, например, мы говорили о строках, являющихся частным случаем массива типа **char**. (В общем массив типа **char** - это массив, элементами которого являются величины типа **char**. Строка - массив типа **char**, в котором нуль-символ **'\0'** используется для того, чтобы отмсчать конец строки.)

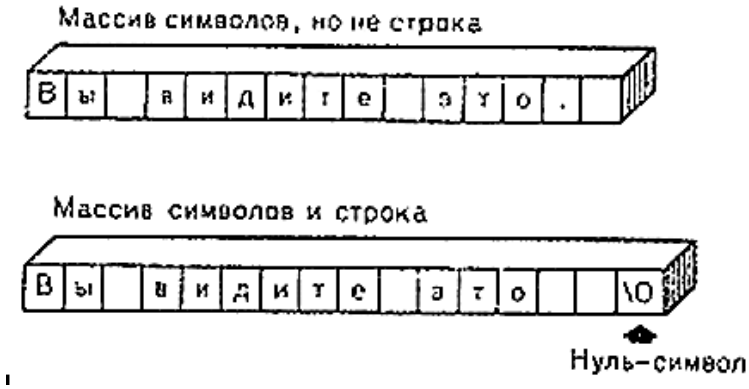


РИС. 8.6. Массивы символов и строки

Числа, используемые для идентификации элементов массива, называются *"подстрочными индексами"* или просто *"индексами"*. Индексами должны быть целые числа, и, как уже упоминалось, индексирование начинается с **0**. Элеметы массива размешаются в памяти последовательно, друг за другом, как показано на рис. 8.6.

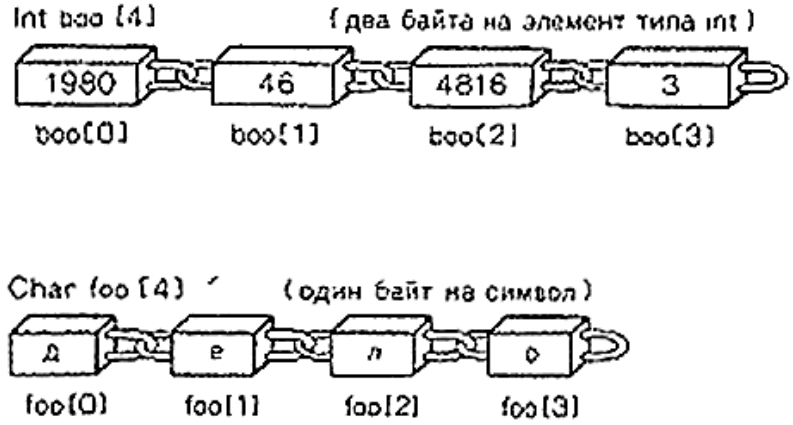


РИС. 8.7. Размещение в памяти массивов с элементамн типа **char** и **int**

Существует огромное количество возможностей применения массивов. Ниже приводится сравнительно простой пример. Предположим, вы хотите написать программу, осуществляющую ввод 10 различных результатов спортивной игры, которые будут обрабатываться позже. Используя понятие массива, вы сможете избежать придумывания 10 различных имен переменных - по одной для каждого счета. Кроме того, для чтения данных вы можете воспользоваться циклом **for**:

```
/* ввод счета */
main( )
{
 int i, score[10];

 for (i = 0; i <= 9; i++)
        scanf(" %d", &a[i]); /* ввод десяти результатов */
 printf(" Введены следующие результаты :\n" );
 for (i = 0; i <= 9; i++)
        printf(" %5d", a[i]); /* проверка правильности ввода */
 printf("\n"); }
```

В понятие хорошего стиля программирования входит эхо-печать только что введенных величин. Она даст уверенность, что в программе будут обрабатываться те данные, для которых она предназначена.

Применяемый здесь способ гораздо более удобен, чем использование 10 различных операторов **scanf()** и 10 отдельных операторов **printf()** для ввода и проверки десяти результатов, определяющих число набранных очков. Цикл **for** обеспечивает очень простои и удобный способ использования индексов массивов.

Какого типа операции могли бы мы выполнить над этими данными? Мы могли бы найти их среднее, величину стандартного отклонения (мы знаем даже, как это сделать) и максимальное значение счета, а также произвести их сортировку в некотором порядке. Займемся двумя простейшими задачами: нахождением среднего и максимального результатов.

Чтобы вычислить среднее, мы можем добавить к нашей программе приведенный ниже фрагмент:

```
int sum, average;
for(i = 0, sum = 0; i <= 9; i++) /* две инициализации */
        sum + = a[i]; /* суммирование элементов массива */
average = sum/10;  /*классический метод усреднения */
printf(" Средний результат равен %d.\n", average);
```

Для нахождения максимального результата к программе можно добавить следующий фрагмент:

```
int highest;
for(highest = a[0], i = 1; i <= 9; i++)
if(a[i] > highest) highest = a[i];
printf ("Максимальный результат равен %d.\n", highest);
```

Здесь мы начинаем с того, что полагаем переменную **highest** равной величине **a[0]**. Затем производится сравнение значения **highest** с каждым элементом массива. Когда обнаруживается, что некоторая величина больше текущего значения переменной **highest**, ей присваивается эта новая большая величина.

Теперь объединим все части программы. На псевдокоде алгоритм ее работы можно записать следующим образом:

```
ввод результатов.
эхо-печать результатов.
вычисление и печать среднего значения.
вычисление и печать максимального значения.
```

Кроме того, мы несколько обобщим нашу программу:

```
/* результаты */
#define NUM 10
main( )
{
int i, sum, average, highest, score [NUM];

printf(" Укажите 10 результатов. \n");
for(i = 0; i < NUM; i++)
        scanf(" %d" , &scorc[i]); /* ввод десяти результатов */
printf(" Введены следующие результаты:\n");
for(i = 0; i<NUM; i++)
        printf("%5d", score[i]);/* проверка правильности ввода */
printf("\n");
tor(i = 0, sum = 0; i < NUM; i++)
        sum + = score[i]; /* суммирование элементов массива */
average = sum/NUM; /* классический метод усреднения */
printf(" Средний результат равен %d.\n", average);
for(highest = score[0], i = 1; i < NUM; i++)
        if(score[i] > highest) /* какая из величин больше */
                highest = score[i];
printf(" Максимальный результат равен %d.\n", highest);
```

```
}
```

Мы заменили число **10** символической константой и воспользовались тем, что выражения **i <=(NUM-1)** и **i < NUM** эквивалентны.

Давайте сначала посмотрим, как это программа работает, а затем сделаем несколько замечаний.

```
Укажите  10 результатов:
76  85  62  48  98  71  66  89  70  77
Введены следующие результаты:
76  85  62  48  98  71  66  89  70  77
Средний результат равен 74. Максимальный результат равен 98.
```

Первый момент, который необходимо отметить, состоит в том, что мы использовали четыре различных цикла **for**. Вас может заинтересовать вопрос: является ли это на самом деле необходимым или мы можем некоторые из данных операций объединить в одном цикле? Такая возможность существует, и она позволила бы сделать программу более компактной. Однако мы побоялись следовать такому подходу (видите, какие мы впечатлительные люди!), поскольку это противоречит принципу модульности. Смысл, заключенный в данной фразе, состоит в том, что программа должна быть разбита на отдельные единицы, или "модули", причем каждый из них должен выполнять одну задачу. (Наша запись на псевдокоде отражает деление программы на четыре модуля.) Такое разбиение облегчает чтение текста программы. Возможно, еще более важным является то, что если отдельные части программы не перемешаны, ее коррекция или модификация упрощаются. Для этого необходимо только исключить из программы требуемый модуль, заменить его новым, а оставшуюся часть программы не изменять.

Второй момент, на который необходимо обратить внимание, состоит в том, что не очень удобно иметь программу, которая обрабатывает ровно 10 чисел. Что произойдет, если кто-то выйдет из игры и будет получено только 9 результатов? Используя символическую константу для обозначения числа 10, мы упростили внесение изменении в программу, по все равно должны произвести ее компиляцию заново. Существуют ли для этого другие возможности? Мы рассмотрим их ниже.

ПРОБЛЕМА ВВОДА

Далее Содержание

Существует несколько способов последовательного ввода набора данных, скажем чисел. Мы обсудим здесь некоторые из них, переходя от менее удобных к более удобным.

Вообще говоря, наименее удобный способ - это тот, который мы только что использовали; написание программы, допускающей ввод фиксированного числа элементов данных. (Такой способ, однако, прекрасно подходит для тех ситуаций, когда число входных данных никогда не изменяется.) Если число входных элементов данных изменяется, необходимо осуществить повторную компиляцию программы.

Следующий шаг состоит в том, чтобы спросить у пользователя, сколько элементов данных будет введено. Так как размер массива в программе фиксирован, она должна проверить, не превышает ли величина, содержащаяся в ответе пользователя, размер массива. Затем пользователь может начать ввод данных. Тогда начало нашей программы можно переписать следующим образом:

```
printf(" Сколько элементов данных вы будете вводить ?\n");
scanf(" %d", &nbr);
while(nbr > NUM)
{
printf("Я смогу обрабатывать не больше %d  элементов; пожалуйста, укажите");
printf("меньшую величину.\n", NUM);
scanf("%d", &nbr); }  /* гарантирует,
что nbr <= NUM - максимального размера массива */
for(i = 0; i <nbr; i++)
        scanf("%d", &score[i]);
```

Мы можем продолжить движение в этом направлении, заменяя в каждом случае символическую константу **NUM** в программе (исключая наличие ее в директиве **#define** и в описании массива) переменной **nbr**. При таком способе различные операции будут выполняться только над теми элементами массива, в которые введены данные.

Недостатком указанного подхода является лежащее в его основе предположение, что пользователь не ошибается при подсчете элементов; если же при программировании полагаться на то, что пользователь всегда все делает правильно, программы оказываются ненадежными.

Это подводит нас к следующему методу, при котором в программе осуществляется подсчет количества вводимых чисел. После всего сказанного выше очевидно, что у компьютеров имеются для этого вес возможности. Основная проблема здесь состоит в том, как сообщить компьютеру о завершении ввода чисел. Один из методов - дать пользователю возможность вводить специальный признак, указывающий на конец ввода. Признак должен принадлежать к *данным того же типа*, что и остальные вводимые данные, так как он должен быть прочитан тем же оператором программы. Но при этом он должен *отличаться* от обычных данных. К примеру, если бы мы вводили результаты игры, чтобы узнать, кто набрал от **0** до **100** очков, мы не могли бы выбрать число **74** в качестве такого признака, потому что оно может соответствовать некоторому возможному результату. С другой стороны, например, число **999** или - **3** вполне могло бы подойти в качестве такого признака, поскольку оно не соответствует требуемому результату. Ниже приводится программа, являющаяся реализацией этого метода:

```
#define STOP 999 /* признак завершения ввода */
#define NUM 50
main( )
{
int i, count, temp, score [NUM];
printf(" Начните ввод результатов. Введите 999 для указания \n");
printf(" конца ввода. Максимальное число результатов, которое вы\n");
printf(" можете ввести.- это %d.\n", NUM);
count = 0;
scanf(" %d", &temp); /* ввод величины */
while(temp != STOP && count <= NUM) /* проверка наличия признака STOP */
{ /* и проверка, не произошло ли превышения размера массива */
score[count++] = temp;
/* запись величины в память и коррекция счетчика */
if(count < NUM + 1)
        scanf("%d", &temp); /* ввод очередного результата */
else
        printf("Я не могу принять больше данных.\n");
}
printf("Вы ввели %d результатов, а именно:\n", count);
for(i = 0; i < count; i++)
        printf("%5d\n", scorc[i]);
}
```

Мы вводим данные во временную переменную **temp** и присваиваем ее значение соответствующему элементу массива только в том случае, если оно не является признаком конца ввода. Совершенно не обязательно реализовывать все именно так; мы просто считаем, что указанный способ делает процесс проверки несколько более наглядным.

Обратите внимание на то, что проверяется выполнение двух условий: прочитан ли признак конца ввода и есть ли место в массиве для следующего числа. Если мы заполнили массив данными до того, как указали признак конца ввода, программа вежливо сообщает нам об этом и прекращает ввод данных.

Заметьте также, что мы воспользовались постфиксной формой операции увеличения. Поэтому, когда значение **count** равно **0**, элементу массива **score[0]** присваивается величина переменной **temp**, а затем **count** возрастает на **1**. После каждой итерации цикла **while** величина счетчика **count** становится на единицу больше последнего использованного индекса массива. Это как раз то, что

нам нужно, поскольку **score[0]** - первый элемент, **score[20]** - 2-й элемент и т. д. Когда работа цикла в программе завершается, значение **count** оказывается равным полному числу прочитанных элементов данных. Затем величина **count** используется в качестве верхней границы числа итераций для последующих циклов.

Этот алгоритм хорошо работает, пока у нас имеется запас таких чисел, которые никогда не будут вводиться как данные. Но что делать, если мы захотим иметь программу, допускающую ввод в качестве данных любых чисел, относящихся к некоторому определенному типу? В таком случае мы не сможем использовать ни одно из чисел как признак конца ввода.

Мы столкнулись с аналогичной проблемой, когда искали подходящий символ для признака **End-of-File**. Тогда было принято решение использовать для ввода символов специальную функцию(**getchar( )**), которая при обращении к ней фактически возвращала величину типа **int**. Это позволяло функции читать "символ" **EOF**, который на самом деле не был обычным символом. В рассматриваемом нами примере полезной оказалась бы функция, которая осуществляла бы ввод целых чисел, могла бы, кроме того, читать данные не только целого типа, но и использовать их в качестве признака конца ввода.

Мы можем одновременно и обрадовать и огорчить вас: такое решение оказывается возможным, но вы должны узнать несколько больше о работе функций; поэтому обсуждение данной идеи откладывается до гл. 10.

РЕЗЮМЕ

Далее Содержание

Основной темой данной главы было обсуждение возможностей управления ходом выполнения программы. Язык Си предоставляет много средств для структурирования программ. С помощью операторов **while** и **for** реализуются циклы с предусловием. Второй оператор особенно подходит для циклов, включающих в себя инициализацию и коррекцию переменной. Использование операции "запятая" в цикле **for** позволяет инициализировать и корректировать более одной переменной. Для тех редких случаев, когда требуется использовать цикл с постусловием, язык Си предоставляет оператор **do while**. Операторы **break**, **continue** и **goto** обеспечивают дополнительные возможности управления ходом выполнения программы.

Мы обсудили здесь также и понятие массива. Массивы в программе описываются так же, как обычные переменные, но при этом в квадратных скобках указывается число элементов. Первому элементу массива присваивается номер **0**, второму - номер **1** и т. д. Индексы, используемые для нумерации элементов массива, могут обрабатываться обычным образом при помощи циклов.

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Три типа циклов в языке Си: **while**, **for** и **do while**.
Различие между циклами с предусловием и с постусловием.
Почему циклы с предусловием используются гораздо чаще, чем циклы с постусловием.
Дополнительные операции присваивания: += -= *= /= %=.
Как пользоваться операцией "запятая".
Когда использовать операторы **break** и **continue**: по возможности редко.
Когда использовать оператор **goto**: когда вы хотите иметь неудобные, трудные для понимания программы.
Как использовать оператор **while** для защиты программы от ошибок при вводе данных.
Что такое массив и как его описать: **long arms[8]**.

ВОПРОСЫ И ОТВЕТЫ

Далее Содержание

Вопросы

1. Определите значение переменной **quack** после выполнения каждого оператора из приведенной ниже их послесдовательности.

```
int quack = 2;
quack + = 5;
quack * = 10;
quack - = 6;
quack / = 8;
quack % = 3;
```

2. Что будет получено на выходе в результате работы следующего цикла?

```
for(value = 36; value > 0; value /= 2) printf("%3d", value);
```

3. Как можно модифицировать операторы **if** в программе **угадывание числа2** чтобы был возможен ввод как прописных, так и строчных букв?

4. Мы подозреваем, что следующая программа не совсем правильная. Какие ошибки вы сможете в ней обнаружить?

```
main( )        /* строка 1 */
{        /* строка 2 */
int i, j, lisl[10];        /* строка 3 */
for (i = 1, i < = 10, i++ )        /* строка 5 */
{        /* строка 6 */
list[i] = 2*i + 3;        /* строка 7 */
for(j = 1, j >= i, j++ )        /* строка 8 */
printf(" %d \n", lisl[j]);        /* строка 9 */
}        /* строка 10 */
```

5. Воспользуйтесь пложенными циклами при написании программы, выводящей на печать следующую фигуру:

```
$$$$$$$$ $$$$$$$$ $$$$$$$$ $$$$$$$$
```

6. Напишите программу, которая создает массив из 26 элементов и помещает в него 26 строчных букв.

Ответы

1. 2, 7, 70, 64, 8, 2

2. 36 18 9 4 2 1.

Вспомните, как выполняется деление целых чисел. Результатом деления **1** на **2** будет **0**, поэтому работа цикла завершится после того, как переменная **value** станет равной **1**.

3. **if(response == 'б' || response == 'Б')**.

4. строка 3: должно быть list[10].

строка 5: вместо запятых должны стоять символы "точка с запятой".

строка 5: переменная **i** должна изменяться в диапазоне от **0** до **9**, а не от **1** до **10**.

строка 8: вместо запятых должны стоять символы "точка с запятой".

строка 8: знак **>=** должен быть заменен на **<=**. В противном случае при значении **i**, равном **1**, цикл никогда не завершится.

строка 10: между строками **9** и **10** должна находиться еще одна закрывающая фигурная скобка. Одна скобка закрывает составной оператор, а другая тело программы.

5.
```
main( )
{ int i, j;
for(i = 1; i <= 4; i++)
{
for( j = 1; j <= 8; j++)
printf("$");
}
printf("\N");
}
```

6.
```
main( )
{ int i;
char ch, alpha[26];
for(i = 0, ch = 'a'; i <= 26; i++, ch++)
alpha[i] == ch;
}
```

УПРАЖНЕНИЯ Содержание

1. Модифицируйте программу **угадывание числа2** в соответствии с нашими предположениями об улучшении ее работы.

2. Реализуйте наше предложение о повышении эффективности работы программы **нахождения простых чисел**.

3. Воспользуйтесь вложенными циклами при написании программы, выводящей на печать следующую фигуру:

```
$
$$
$$$
$$$$
$$$$$
```

[1)] Валентин(а) - возлюбленный (возлюбленная), выбираемые в день св. Валентина, 14 февраля.- *Прим. перев.*

[2)] Зеион Элейскии (ок. 490-430 до н.э.) - древнегреческий философ. Известен своими знаменитыми парадоксами (апориями).- *Прим. перев.*

[3)] Читатель должен понимать, что философский смысл апории Зенона о стреле не исчерпывается приведенными выше рассуждениями.- *Прим. перев.*

---

9. Как правильно пользоваться функциями

ФУНКЦИИ
СТРОИТЕЛЬНЫЕ БЛОКИ ПРОГРАММЫ
СВЯЗЬ МЕЖДУ ФУНКЦИЯМИ: АРГУМЕНТЫ, УКАЗАТЕЛИ, ВОЗВРАТ ЗНАЧЕНИЯ
ТИПЫ ФУНКЦИЙ

КЛЮЧЕВОЕ СЛОВО
return

Принципы программирования на языке Си основаны на понятии функции. В представленных ранее примерах программирования мы уже воспользовались несколькими функциями: **printf( ), scanf( ), getchar( ), putchar( )** и **strlen( )**. Эти функции являются системными, однако мы создали и несколько своих собственных функций под общим именем **main( )**. Выполнение программы всегда начинается с команд, содержащихся в функции **main( )**, затем последняя вызывает другие функции, например **getchar( )**. Теперь мы переходим к вопросу о том, как создавать свои собственные функции н делать их доступными для функции **main( )**, а также друг для друга.

Во-первых, что такое функция? Функция - самостоятельная единица программы, спроектированная для реализации конкретной задачи. Функции в языке Си играют ту же роль, какую играют функции, подпрограммы и процедуры в других языках, хотя детали их структуры могут быть разными. Вызов функции приводит к выполнению некоторых действии. Например, при обращении к функции **printf( )** осуществляется вывод данных на экран. Другие же функции позволяют получать некоторую величину, используемую затем в программе. К примеру, функция **strlen( )** "сообщает" программе длину конкретной строки. В общем функции могут выполнять действия и получать значения величин, используемых в программе.

Почему мы пользуемся функциями? Во-первых, они избавляют нас от повторного программирования. Если конкретную задачу необходимо выполнить в программе несколько раз, мы напишем соответствующую функцию только один раз, а затем будем вызывать ее всегда, когда это требуется. Во-вторых, мы можем применять одну функцию, например **putchar( )**, в различных программах. Даже в том случае, если некоторая задача выполняется только в одной программе, лучше оформить ее решение в виде функции, поскольку функции повышают уровень модульности программы и, следовательно, облегчают ее чтение, внесение изменений и коррекцию ошибок. Предположим, например, что мы хотим написать программу, которая делает следующее:вводит набор чисел сортирует, эти числа, находит их среднее, выводит на печать гистограмму.
Соответствующую программу можно записать так:

```
main( )
{
float list [50];
readlist(list);
sort(list);
average(list);
bargrapli(list);
}
```

Разумеется, мы должны были бы запрограммировать четыре функции **readlist( ), sort( ), average( ) и bargraph( )**, но... это уже детали. Используя смысловые имена функции, мы четко определяем, что программа делает и как она организована. После этого можно заниматься каждой функцией отдельно и совершенствовать ее до тех пор, пока она не будет правильно выполнять требуемую задачу. Дополнительное преимущество указанного подхода заключается в том, что если мы создадим функции достаточно общего вида, то их можно будет использовать и в других программах.

Многие программисты предпочитают думать о функции, как о "черном ящике"; они задают ее через поступающую информацию (вход) и полученные результаты (выход). Все, что происходит внутри черного ящика, их не касается до тех пор, пока не нужно писать программу, реализующую

эту функцию. Когда мы используем, например, функцию **printf( )**, мы знаем, что должны передать ей управляющую строку и возможно, несколько аргументов. Мы знаем также результат вы зова функций **printf( )**. Не нужно полагать, что при программнровании вам придется заниматься созданием функции **printf( )**. Использование функций указанным выше способом позволяет сконцентрироват внимание на обшей структуре программы, а не на деталях.

Что нам требуется знать о функциях? Нужно знать, как их можно определять, как к ним обращаться и как устанавливать связи между функцией и программой, ее вызывающей. Чтобы изучить это, мы рассмотрим очень простои пример, а затем будем обобщать его, вводя дополнительные характеристики до тех пор, пока не получим полную и ясную картину.

СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ПРОСТОЙ ФУНКЦИИ

Далее Содержание

Наша первая скромная цель - создание функции, которая печатает 65 символов ***** в ряд. Чтобы эта функция выполнялась в некотором контексте, мы включили ее в программу, которая печатает простой титул фирменного бланка. Ниже приведена полная соответствуюшая программа. Она состоит из функции **main( )** и **starbar( )**.

```
/* титул фирменного бланка! */
#define NAME "MEGATHINK, INC."
#define ADDRESS "10 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
main( )
{
starbar( );
printf("%s\n", NAME);
printf(" %s\n", ADDRESS);
printf("%s\n", PLACE);
starbar( );
}
/* далее следует функция starbar( ) */
#include
#define LIMIT 65
starbar( );
{ int count;
for (count = 1; count <= LIMIT; count++)
putchar('*');
putchar('\n');
}
```

Результат работы программы выглядит так:

```
*************************************************************
MEGATHINK, INC 10 Megabuck Plaza Megapolis, CA 94904
*************************************************************
```

При рассмотрении этой программы необходимо обратить внимание на следующие моменты:

1. Мы вызвали функцию **starbar( )** (или, можно сказать, обратились к ней) из функции **main( )**, используя только ее имя. Это несколько напоминает заклинание, вызывающее злого духа, но, вместо того чтобы чертить пятиугольник, мы помещаем вслед за именем функции точку с запятой, создавая таким образом оператор: **starbar( );**

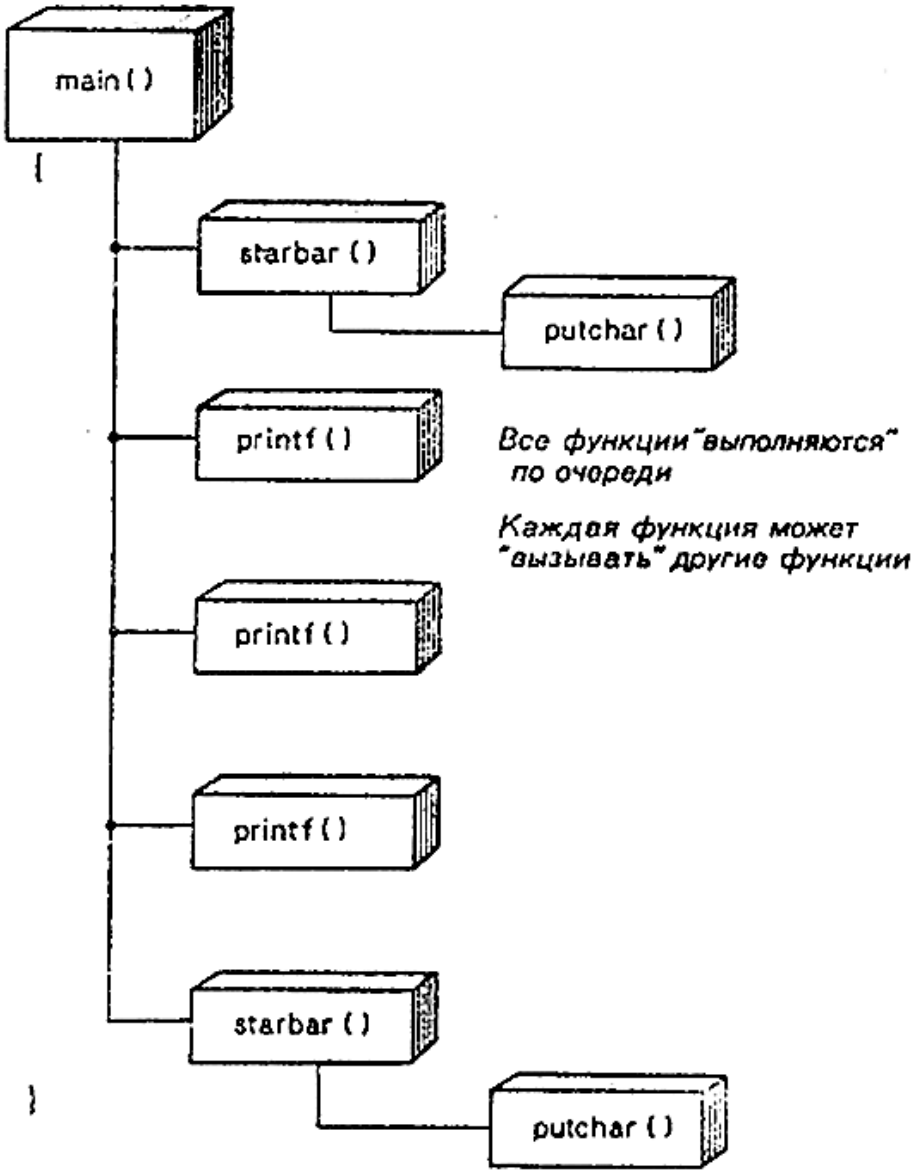


РИС. 9.1. Схема выполнения операторов программы титул "фирменною бланка 1".

Это одна из форм вызова функции, но далеко не единственная. Когда в процессе выполнения программы компьютер достигает оператора **starbar( )**, он находит указанную функцию, после чего начинает выполнять соответствующие ей команды. Затем управление возвращается следующей строке "вызывающе" программы" - в данном случае **main( )**.

2. При написании функции **starbar( )** мы следовали тем же правилам, что и при написании **main( )**: вначале указывается имя, затем идет открывающая фигурная скобка, приводится описание используемых переменных, даются операторы, определяющие работу функции, и, наконец, закрывающая фигурная скобка. Мы даже поместили перед описанием функции **starbar( )** директивы **#define** и **#include**, требующиеся для нее, а не для функции **main( )**.

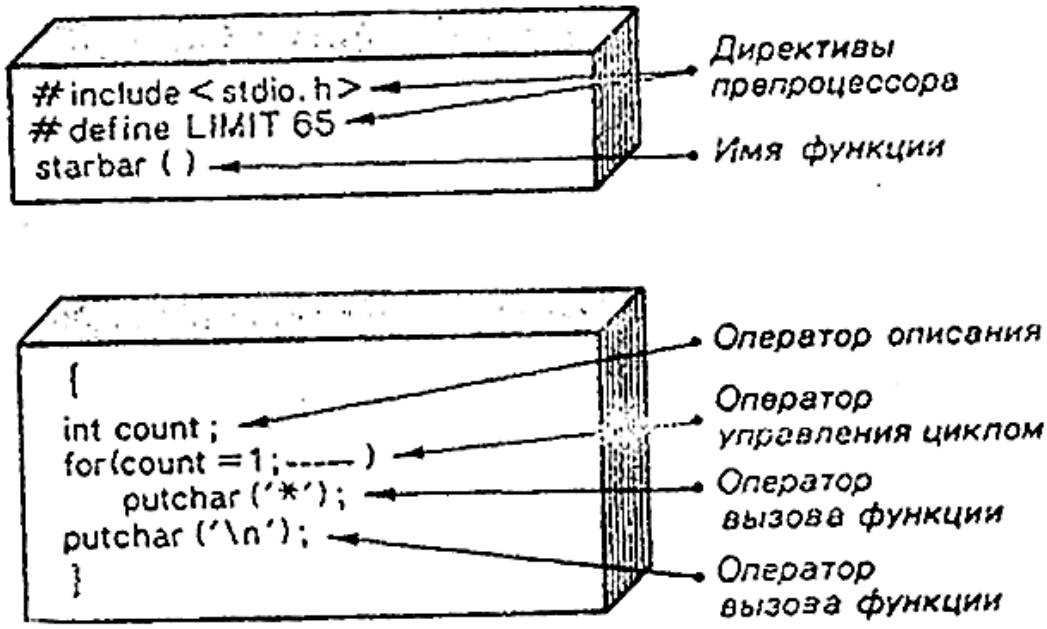


РИС. 9.2. Структура простой функции.

3. Мы включили функции **starbar()** и **main()** в один файл. Вообще говоря, можно было создать два отдельных файла. Один файл несколько упрощает компиляцию, а два отдельных файла облегчают использование одной функции в разных программах. Случай двух и более файлов мы обсудим позже, а пока будем держать все наши функции в одном месте. Закрывающая фигурная скобка функции **main( )** указывает компилятору на се конец. Круглые скобки в имени **starbar( )** говорят о том, что **starbar( )** - это функция. Обратите внимание, что здесь за именем **starbar( )** не следует символ "точка с запятой"; его отсутствие служит указанием компилятору, что мы *определяем* функцию **starbar( )**, а не используем ее.

Если рассматривать функцию **starbar( )** как черный ящик, то ее выход - это напечатанная строка, состоящая из символов *****. Какие бы то ни было данные на входе у нее отсутствуют, потому что ей не нужно использовать информацию из вызывающей программы. Вообще, этой функции не требуется связь с вызывающей программой. Обратимся к случаю, когда такая связь необходима.

АРГУМЕНТЫ ФУНКЦИИ

Далее Содержание

Титул фирменного бланка выглядел бы несколько лучше, если бы текст был сдвинут к центру. Мы сможем поместить текст в центре, если напечатаем нужное число пробелов перед выводом требуемой строки. Воспользуемся некоторой функцией для печати пробелов. Наша функция **space( )** (давайте назовем ее так) будет очень напоминать функцию **starbar( )**, за исключением того, что на этот раз между функцией **main( )** и функцией **space( )** должна быть установлена связь, так как необходимо сообщить последней функции о требуемом числе пробелов.

Рассмотрим это более конкретно. В строке, состоящей из звездочек, 65 символов, а в строке MEGATHINK, INC. - 15. Поэтому в нашем первом варианте программы вслед за этим сообщением шло 50 пробелов. Чтобы сместить текст к центру, нужно сначала напечатать 25 пробелов, а потом текст, в результате чего слева и справа от данной фразы окажется по 25 пробелов. Следовательно, необходимо иметь возможность передать величину "25" функции, печатающей пробелы. Мы применяем тот же способ, что и при передаче символа '*' функции **putchar( )**: используем аргумент. Тогда запись **space(25)** будет означать, что необходимо напечатать 25 пробелов. 25 - это аргумент. Мы будем вызывать функцию **space( )** три раза: один раз для каждой строки адреса. Вот как

выглядит эта программа:

```
/* титул фирменного бланка2 */
#define NAME "MEGATHINK, INC."
#define ADDRESS "10 Megabuck Plaza"
#define PLACE "Mcgapolis, CA 94904"
main( )
{
int spaces;

starbar( );
space(25); /* space( ) использует в качестве аргумента константу */
printf("%s\n", NAME);
spaces = (65 - strlen(ADDRESS))/2;
/* мы заставляем программу вычислять, сколько пропустить пробелов */
space(spaces);  /* аргументом является переменная */
printf("%s\n", ADDRESS);
space((65 - strlen(PLACE))/2); /* аргументом является выражение */
printf(" %s \n", PLACE);
starbar( );
}
/ *определение функции starbar( ) */
#include
#define LIMIT 65
starbar( )
{
int count;
for (count = 1;count <= LIMIT;count++) putchar('*');
putchar('\n');
}
/* определение функции space( ) */
space(number)
int number; /* аргумент описывается перед фигурной скобкой */
{
int count /* дополнительная переменная описывается
                                                    после фигурной скобки */
for (count = 1;count <= number;count++) putchar(' ');
}
```

РИС. 9.3. Программа, печатающая титул фирменного бланка.

Обратите внимание на то, как мы экспериментировали при вы зовах функции **space( ):** мы задавали аргумент тремя различными способами. Являются ли все они работоспособными? Да - и вот доказательство.

```
**********************************************************************

MEGATHINK, INC.
10 Megabuck Plaza
Megapolis, CA 94904

**********************************************************************
```

Рассмотрим сначала, как определить функцию с одним аргументом, после чего перейдем к вопросу о том, как она используется.

Далее Содержание

Определение функции с аргументом: формальные аргументы

Определение нашей функции начинается с двух строк:

```
space(number)
```

```
int number;
```

Первая строка информирует компилятор о том, что у функции **space( )** имеется аргумент и что его имя **number**. Вторая строка - описание, указывающее компилятору, что аргумент **number** имеет тип **int**. Обратите внимание: аргумент описывается *перед* фигурной скобкой, которая отмечает начало тела функции. Вообще говоря, вы можете объединить эти две строки в одну:

```
space(int number;)
```

Независимо от формы записи переменная **number** называется "формальным" аргументом. Фактически это новая переменная, и в памяти компьютера для нее должна быть выделена отдельная ячейка. Посмотрим, как можно пользоваться этой функцией.

Вызов функции с аргументом: фактические аргументы

Далее Содержание

Задача в данном случае состоит в том, чтобы присвоить некоторую величину формальному аргументу **number**. После того как эта переменная получит свое значение, программа сможет выполнить свою задачу. Мы присваиваем переменной **number** значение фактического аргумента при вызове функции. Рассмотрим наш первый случай использования функции **space( )**:

```
space(25);
```

Фактический аргумент здесь 25, и эта *величина* присваивается формальному аргументу - переменной **number**, т. е. вызов функции оказывает следующее действие:

```
number = 25;
```

Короче говоря, формальный аргумент - переменная в вызываемой программе, а фактический аргумент - конкретное значение, при своенное этой переменной вызывающей программой. Как было показано в нашем примере, фактический аргумент может быть константой, переменной или даже более сложным выражением. Независимо от типа фактического аргумента он вначале вычисляется, а затем его величина (в данном случае некоторое целое число) передается функции. Рассмотрим, например, наше последнее обращение к функции **space( )**.

```
space((65- strlen(PLACE))/2);
```

Сначала было вычислено значение длинного выражения, образующего фактический аргумент; оно оказалось равным 26. Затем величина 26 присваивается переменной **number**. Функция не знает и не "хочет" знать, является ли поступившее число константой, значением некоторой переменной или более общего выражения. В заключение повторим снова, что фактический аргумент - это конкретное значение, которое присваивается переменной, называемой формальным аргументом.

Функция как "черный ящик"

Далее Содержание

Рассматривая функцию **space( )** как черный ящик, можно сказать, что ее вход - это число пропущенных позиций, а выход -j фактический пропуск позиций. Вход связан с функцией через аргумент.

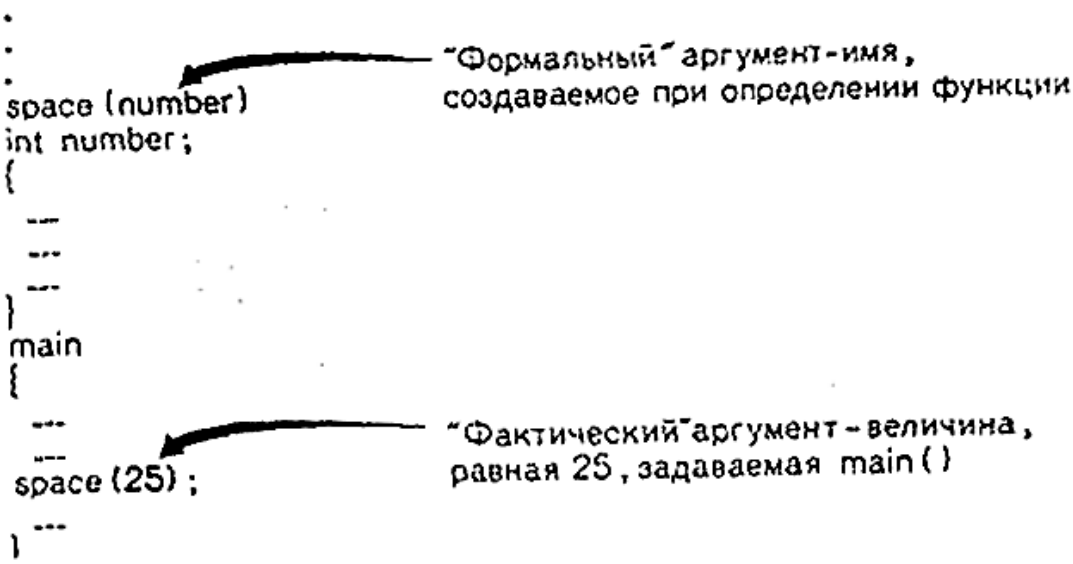


РИС. 9.4. Фактические аргументы и формальные аргументы.

С помощью аргумента обеспечивается связь между функциями **main( )** и **space( )**. В то же время переменная **count** описана в теле функции, и другие функции ничего не знают о ней. Указанная переменная является частью механизма, скрытого внутри черного ящика. Это не та же переменная, что **count** в **starbar()**.

Наличие нескольких аргументов

Далее Содержание

Если для связи с некоторой функцией требуется более одного аргумента, то наряду с именем функции можно задавать список аргументов, разделенных запятыми, как показано ниже.

```
printnum(i,j) int i, j;
{ printf(" Новых точек = %d. Всего точек = %d.\n", i, j); }
```

Мы уже знаем, как передавать информацию из вызывающей программы в вызываемую функцию. Можно ли передавать информацию каким-нибудь другим способом? Этот вопрос послужит нам темой следующего обсуждения.

ВОЗВРАЩЕНИЕ ЗНАЧЕНИЯ ФУНКЦИЕЙ: ОПЕРАТОР return

Далее Содержание

Создадим функцию, вычисляющую абсолютную величину числа. Абсолютная величина числа - это его значение (если отбросить знак). Следовательно, абсолютная величина 5 равна 5, а абсолютная величина -3 равна 3. Мы назовем эту функцию **abs( )**. Входом для **abs()** может быть любое число, для которого мы хотим найти абсолютную величину. Выходом функции будет соответствующее неотрицательное число. Входная величина может обрабатываться благодаря наличию аргумента; выходная величина возвращается (т. е. выдается), как вы увидите ниже, при помощи ключевого слова языка Си - **return**. Поскольку функция **abs( )** должна быть вызвана другой функцией, мы создадим простую программу **main( )**, основной целью которой будет проверка, работает ли функция **abs( )**. Программа, спроектированная для того, чтобы проверять работу функции именно таким образом, называется "драйвером". Драйвер подвергает функцию последовательным проверкам. Если результаты оказываются удовлетворительными, то ее можно поместить в программу, заслуживающую большего внимания. (Термин "драйвер" обычно относится к программам, управляющим работой устройств.) Приведем далее наш драйвер и функцию,

вычисляющую абсолютную величину числа:

```
/* abs.драйвер */
main( )
{
int a = 10, b = 0, c = -22;
int d, e, f;

d = abs(a);
c = abs(b);
f = abs(c);
printf(" °%d %d %d\n" , d, e, f);
}

/* функция, вычисляющая величину числа */
abs(x) int x;
{
int y;
y = (x < 0) ? -x : x; /* вспомните операцию  ?: */
return (y ); /* возвращает значение y вызывающей программе */
}
```

Результат работы программы выглядит так:

```
10 0 22
```

Сначала вспомним операцию условия **?:**. Эта операция в функции **abs( )** выполняется следующим образом: если **x** меньше **0**, **y** полагается равным **-x**; в противном случае y полагается равным **x**. Это как раз то, что нам нужно, поскольку если **x** равен **-5**, то **y** равен **-(-5)**, т. е. **5**.

Ключевое слово **return** указывает на то, что значение выражения, заключенного в круглые скобки, будет присвоено функции, содержащей это ключевое слово. Поэтому, когда функция **abs( )** впервые вызывается нашим драйвером, значением **abs(a)** будет число **10**, которое затем присваивается переменной **d**.

Переменная **y** является внутренним объектом функции **abs()**, но значение **y** передается в вызывающую программу с помощью оператора **return**. Действие, оказываемое оператором

```
d = abs(a);
```

по-другому можно выразить так:

```
abs(a);
d = y;
```

Можно ли в действительности воспользоваться такой записью? Нет, так как вызывающая программа даже не подозревает о том, что переменная **y** существует.

Возвращаемое значение можно присвоить переменной, как в нашем примере, или использовать как часть некоторого выражения, например, следующим образом:

```
answer = 2*abs(z) + 25;
printf("  %d\n" , abs(-32 + answer));
```

Оператор **return** оказывает и другое действие. Он завершает выполнение функции и передает управление следующему оператору в вызывающей функции. Это происходит даже в том случае, если оператор **return** является не последним оператором тела функции. Следовательно, функцию **abs( )** мы могли бы записать следующим образом:

```
/* функция, вычисляющая абсолютную величину числа,
вторая версия */
abs(x) int x;
```

```
{ if(x < 0)
        return(-x);
else
        relurn(x);
}
```

Эта версия программы проще, и в ней не используется дополнительная переменная **y**. Для пользователя, однако, обе версии неразличимы, поскольку у них имеется один и тот же вход и они обеспечивают один и тот же выход. Только внутренние структуры обеих функций различны. Даже версия данной программы, приведенная ниже, работает точно так же:

```
/* функция, вычисляющая абсолютную величину числа,
третья версия */
abs(x) int(x);
{ if (x < 0)
        return(-x);
else
        return(x);
printf(" Профессор Флеппард - болван. \n");
}
```

Наличие оператора **return** препятствует тому, чтобы оператор печати **printf( )** когда-нибудь выполнился в программе. Профессор Флеппард может пользоваться в своих программах объектным кодом, полученным в результате компиляции данной функции, и никогда не узнает об истинных чувствах своего студента-программиста.

Вы можете также использовать просто оператор **return**;
Его применение приводит к тому, что функция, в которой он содержится, завершает свое выполнение и управление возвращается в вызывающую функцию. Поскольку у данного оператора отсутствует выражение в скобках, никакое значение при этом не передается функции.

ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ

Далее Содержание

Мы уже несколько раз касались вопроса о том, что переменные в функции являются ее внутренними переменными и "не известны" вызывающей функции. Аналогично переменные вызывающей функции не известны вызываемой функции. Вот почему для связи с ней, т. е. для передачи значений в нее и из нее, мы пользуемся аргументами и оператором **return**.

Переменные, известные только одной функции, а именно той, которая их содержит, называются "*локальными*" переменными. До сих пор это был единственный вид переменных, которыми мы пользовались, но в языке Си допускается наличие переменных, известных нескольким функциям. Такие нелокальные переменные называются "*глобальными*", и мы вернемся к ним позже. Теперь же мы хотим подчеркнуть, что локальные переменные являются действительно локальными. Даже в том случае, если мы используем одно и то же имя для переменных в двух различных функциях, компилятор (и, таким образом, компьютер "считает" их разными переменными. Мы можем показать это, используя операцию **&** (не путайте с операцией **&&**).

НАХОЖДЕНИЕ АДРЕСОВ: ОПЕРАЦИЯ &

Далее Содержание

В результате выполнения операции & определяется адрес ячейки памяти, которая соответствует переменной. Если **pooh** - имя переменной, то &**pooh** - ее адрес. Можно представить себе адрес как ячейку памяти, но можно рассматривать его и как метку, которая используется компьютером, для идентификации переменной. Предположим, мы имеем оператор

```
pooh = 24;
```

Пусть также адрес ячейки, где размещается переменная **pooh** - 12126. Тогда в результате выполнения оператора

```
printf(" %d %d\n" , pooh, &pooh);
```

получим

```
24  12126
```

Более того, машинный код, соответствующий первому оператору, словами можно выразить приблизительно так: "Поместить число 24 в ячейку с адресом 12126".

Воспользуемся указанной выше операцией для проверки того, в каких ячейках хранятся значения переменных, принадлежащих разным функциям, но имеющих одно и то же имя.

```
/* контроль адресов */
main( )
{ int pooh = 2, bah = 5;
printf(" в main( ), pooh = %d и &pooh = %u \n" , pooh, &pooh);
printf('В main( ), bah = %d и &bah = %u\n>/, bah, &bah);
mikado(pooh);
}
mikado(bah) int bah;
{
 int pooh = 10;
 printf("В mikado( ), pooh = %d и &pooh = %u\n, pooh, &pooh);
 printf(" в mikado( ), bah = %d и &bah = %u\n" , bah, &bah);
}
```

Мы воспользовались форматом **%u** (целое без знака) для вывода на печать адресов на тот случай, если их величины превысят максимально возможное значение числа типа **int**. В нашей вычислительной системе результат работы этой маленькой программы выглядит так:

```
в main( ), pooh = 2 и &pooh = 56002
В main( ), bah = 5 и &bah = 56004
В mikado( ), pooh = 10 и &pooh =55996
в mikado( ), bah = 2 и &bah = 56000.
```

О чем это говорит? Во-первых, две переменные **pooh** имеют различные адреса. То же самое верно и относительно переменных **bah**. Следовательно, как и было обещано, компьютер рассматривает их как четыре разные переменные. Во-вторых, при вызове **mikado(pooh)** величина (**2**) фактического аргумента (**pooh** из **main( )**) передастся формальному аргументу (**bah** из **mikado( )**). Обратите внимание, что было передано только значение переменной. Адреса двух переменных (**pooh** в **main( )** и **bah** в **mikado( )**) остаются различными.

Мы коснулись второго вопроса потому, что этот факт оказывается неверным для всех других языков. В той или иной процедуре Фортрана, например, можно использовать переменные вызывающей программы. Кроме того, в такой процедуре переменные могут иметь различные имена, но адреса их при этом будут совпадать. В языке Си подобные механизмы отсутствуют. Каждая функция использует свои собственные переменные. Это более предпочтительно, потому что "исходные" переменные не будут таинственным образом изменяться из-за того, что вызванная функция обладает побочным эффектом. Но это может также приводить и к некоторым трудностям, о чем и будет рассказано и следующем разделе.

ИЗМЕНЕНИЕ ПЕРЕМЕННЫХ В ВЫЗЫВАЮЩЕЙ ПРОГРАММЕ

Далее Содержание

Иногда требуется, чтобы одна функция могла изменять переменные, относящиеся к другой. Например, в задачах сортировки часто бывает необходимо осуществлять обмен значениями между двумя переменными. Предположим, у нас есть две переменные **x** и **y** и мы хотим, чтобы они обменялись своими значениями. Простая последовательность операторов

```
x = y;
y = x;
```

не является решением поставленной задачи, потому что к тому моменту, когда начнет выполняться оператор во второй строке, первоначальное значение переменной **x** будет потеряно. Чтобы сохранить это первоначальное значение, необходимо дополнить данный фрагмент еще одной строкой:

```
temp = x;
x = y;
y = temp;
```

Теперь у нас есть требуемый метод; реализуем его в виде некоторой функции, а также создадим драйвер для ее проверки. Чтобы сделать более ясным, какая переменная принадлежит функции **main( )**, а какая - функции **interchange( )**, мы будем использовать переменные **x** и **y** в первой из них, и **u** и **v** - во второй.

```
/* обмен1 */
main( )
{
int x = 5, y = 10;
printf(" Вначале x = %d  и y = %d.\n" , x, y);
interchange(x, y);
prinlf(" Теперь x = %d и y = %d.\n" , x, y);
}
```

```
interchangce(u, v) int u, v;
{
int temp;
temp = u;
u = v;
v = temp;
}
```

Попробуем выполнить эту программу. Результаты будут выглядеть следующим образом:

```
Вначале x = 5 и y = 10.
Теперь x = 5 и y = 10.
```

Не может быть! Значения переменных не поменялись местами! Вставим в программу **interchange( )** несколько операторов печати, чтобы понять причину допущенной ошибки.

```
/* обмен2 */
main( )
{
int x = 5, y = 10;
printf(" Вначале x = %d и y = %d.\n", x,y);
interchange(x, y);
printf(" Теперь x = %d и y = %d.\n", x, y);
}
interchange(u, v)
int u, v;
{ int temp;
printf(" Вначале u = %d и v = %d.\n",  u, v);
temp = u;
u = v;
v = temp;
printf(" Теперь u = %d и  v = %d.\n", u, v);
}
```

Результат работы этой программы выглядит так:

```
Вначале x  = 5 и y = 10.
Вначале u  = 5 и v = 10.
Вначале u  = 10 и  v = 5.
Вначале x  = 5 и y = 10.
```

Отсюда видно, что ничего неправильного в работе функции **interchange( )** нет; она осуществляет обмен значениями между переменными **u** и **v**. Проблема состоит в передаче результатов обратно в функцию **main( )**. Как мы уже указывали, функции **interchange( )** и **main()** используют различные переменные, поэтому обмен значениями между переменными **u** и **v** не оказывает никакого влияния на **x** и **y**! А нельзя ли каким-то образом воспользоваться оператором **return**? Мы могли бы, конечно, завершить тело функции **interchange( )** строкой

```
return(u);
```

и изменить форму вызова в функции **main( )** следующим образом:

```
x = interchange(x, y);
```

В результате такого обращения к функции переменная **x** получит новое значение, но **y** при этом не изменится.

С помощью оператора **return** *в вызывающую программу можно передать только одну величину*. Но нам нужно передать две величины. Это оказывается вполне осуществимым! Для этого нужно лишь воспользоваться "*указателями*".

Указатели: первое знакомство

Далее Содержание

Указатели? Что это такое? Вообще говоря, указатель - некоторое символическое представление адреса. Например, ранее мы воспользовались операцией получения адреса для нахождения адреса переменной **pooh**. В данном случае **&pooh** означает "указатель на переменную **pooh**". Фактический адрес - это число (в нашем случае 56002), а символическое представление адреса **&pooh** является *константой* типа указатель. После всего сказанного выше становится очевидным, что адрес ячейки, отводимой переменной **pooh**, в процессе выполнения программы не меняется.

В языке Си имеются и *переменные* типа указатель. Точно так же как значением переменной типа **char** является символ, а значением переменной типа **int** - целое число, значением переменной типа указатель служит адрес некоторой величины. Если мы дадим указателю имя **ptr**, то сможем написать, например, такой оператор

```
ptr = &pooh;  /* присваивает адрес pooh переменной ptr */
```

Мы говорим в этом случае, что **ptr** "указывает на" **pooh**. Различие между двумя формами записи: **ptr** и **&pooh**, заключается в том, что **ptr** - это переменная, в то время как **&pooh** - константа. В случае необходимости мы можем сделать так, чтобы переменная **ptr** указывала на какой-нибудь другой объект:

```
ptr = &bah; /* ptr указывает на bah, а не на pooh */
```

Теперь значением переменной **ptr** является адрес переменной **bah**.

Операция косвенной адресации: *

Далее Содержание

Предположим, мы знаем, что в переменной **ptr** содержится ссылка на переменную **bah**. Тогда для доступа к значению этой переменной можно воспользоваться операцией "косвенной адресации" (*****). (Не путайте эту **унарную** операцию косвенной адресации с **бинарной** операцией умножения *****).

```
val = *ptr;  /* определение значения, на которое указывает ptr */
```

Последние два оператора, взятые вместе, эквивалентны следующему:

```
val = bah;
```

Использование операций получения адреса и косвенной адресации оказывается далеко не прямым путем к результату; отсюда и появление слова "косвенная" в названии операции.

Резюме: операции, связанные с указателями

I. Операция получения адреса &

Когда за этим знаком следует имя переменной, результатом операции является адрес указанной переменной.

Пример:
&**nurse** дает адрес переменной **nurse**.

II. Операция косвенной адресации
* Когда за этим таком следует указатель на переменную, результатом операции является величнна, помещенная и ячейку с указанным адресом.

Пример:

```
nurse = 22;
pir = &nurse; /* указатель на nurse */ val = *ptr;
```

Результатом выполнения этого фрагмента является присваивание значения 22 переменной **val**.

Описание указателей

Далее Содержание

Мы знаем, как описывать переменные типа **int** и других типов. Но как описать переменную типа "указатель"? На первый взгляд это можно сделать так:

```
pointer ptr;       /* неправильный способ описания указателя */
```

Почему нельзя использовать такую запись? Потому что недостаточно сказать, что некоторая переменная является указателем. Кроме этого, необходимо сообщить еще, на переменную какого типа ссылается данный указатель! Причина заключается в том, что переменные разных типов занимают различное число ячеек, в то время как для некоторых операций, связанных с указателями, требуется знать объем отведенной памяти. Ниже приводятся примеры правильного описания указателей:

```
int *pi;             /* указатель на переменную типа целого */
char *pc;              /* указатель на символьную переменную */
float *pf, *pg;        /* указатели на переменные с плавающей точкой */
```

Спецификация типа задает тип переменной, на которую ссылается указатель, а символ звездочка (*****) определяет саму переменную как указатель. Описание вида **int *pi;** говорит, что **pi** - это указатель и что ***pi** - величина типа **int**.

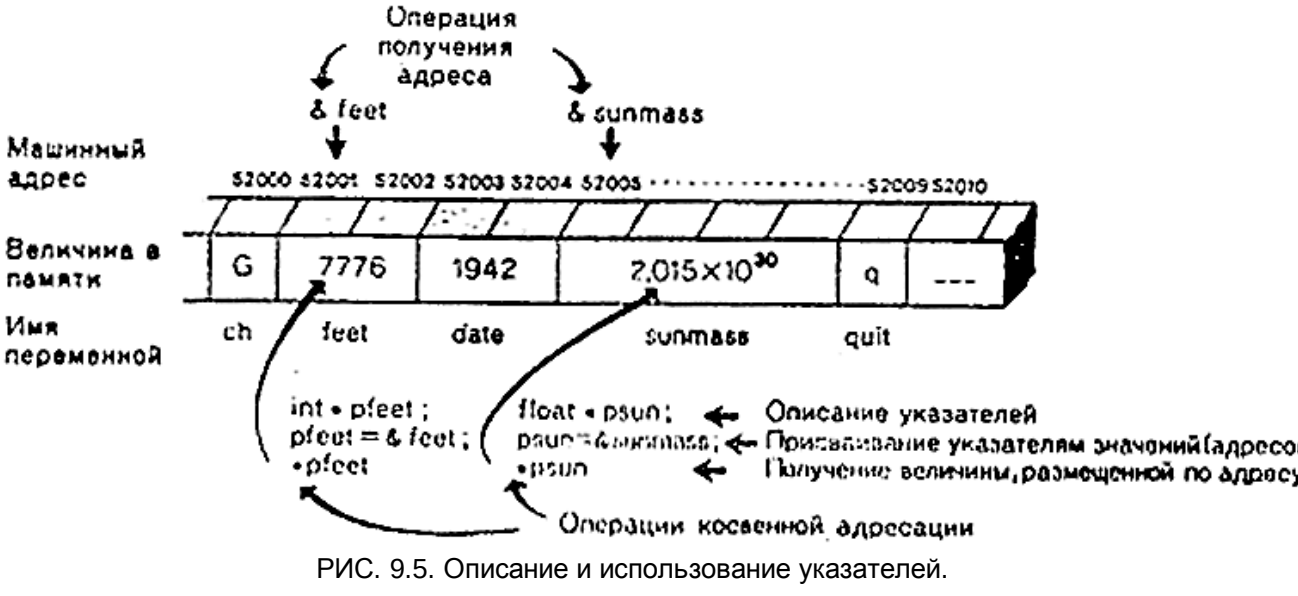


РИС. 9.5. Описание и использование указателей.

Точно так же величина (***pc**), на которую ссылается переменна **pc**, имеет тип **char**. Что можно сказать о самой переменной **pc**? Мы считаем, что она имеет тип "указатель на переменную типа **char**". Ее величина, являющаяся адресом,- это целое число без знака, поэтому при выводе на печать значения переменной **pc** мы будем пользоваться форматом **%u**.

Использование указателей для связи между функциями

Далее Содержание

Мы только прикоснулись к обширному и увлекательному миру указателей. Сейчас нашей целью

является использование указателей для решения задачи об установлении связи между функциями. Ниже приводится программа, в которой указатели служат средством, обеспечивающим правильную работу функции, которая осуществляет обмен значениями переменных. Посмотрим, как она выглядит, выполним ее, а затем попытаемся понять, как она работает.

```
/* обмен3 */
main( )
{
int x = 5, y = 10;
printf(" Вначале x = %d и y = %d.\n" , x, y);
interchange(&x,&y); /* передача адресов функции */
printf(" Теперь x = %d и y = %d.\n", x, y);
}
interchange(u, v)
int *u, *v; /* u и v являются указателями */
{
int temp;

temp = *u; /* temp присваивается значение, на которое указывает u */
*u = *v;
*v = temp;
}
```

После всех встретившихся трудностей, проверим, работает ли этот **вариант 1**

```
Вначале x = 5 и y = 10.
Теперь x = 10 и y = 5.
```

Да программа работает. Посмотрим, как она работает. Во-первых, теперь вызов функции выглядит следующим образом:

```
interchange(&x, &y);
```

Вместо передачи *значений* **x** и **y** мы передаем их *адреса*. Это означает, что формальные аргументы **u** и **v**, имеющиеся в спецификации:

```
interchange(u,v)
```

при обращении будут заменены адресами и, следовательно, они должны быть описаны как указатели. Поскольку **x** и **y** - целого типа, **u** и **v** являются указателями на переменные целого типа, и мы вводим следующее описание:

```
int *u, *v;
```

Дале в теле функции оператор описания:

```
int temp;
```

используется с целью резервирования памяти. Мы хотим поместить значение переменной **x** в переменную **temp**, поэтому пишем:

```
temp = *u;
```

Вспомните, что значение переменной **u** - это **&x**, поэтому переменная **u** ссылается на **x**. Это означает, что операция ***u** дает значение **x**, которое как раз нам и требуется. Мы не должны писать, например, так:

```
temp = u; /* неправильно */
```

поскольку при этом происходит запоминание *адреса* переменной **x**, а не ее *значения*; мы же пытаемся осуществить обмен значениями, а не адресами.

Точно так же, желая присвоить переменной **y** значение переменной **x**, мы пользуемся оператором:

```
*u = *v;
```

который соответствует оператору

```
x = y;
```

Подведем итоги. Нам требовалась функция, которая могла бы изменять значения переменных **x** и **y**. Путем передачи функции адресов переменных **x** и **y** мы предоставили ей возможность доступа к ним. Используя указатели и операцию ***, функция смогла извлечь величины, помещенные в соответствующие ячейки памяти, и поменять их местами.

Вообще говоря, при вызове функции информация о переменной может передаваться функции в двух видах. Если мы используем форму обращения:

```
function1(x);
```

происходит передача значения переменной **x**. Если же мы используем форму обращения:

```
function2(&x);
```

происходит передача *адреса* переменной **x**. Первая форма обращения требует, чтобы определение функции включало в себя формальный аргумент того же типа, что и **x**:

```
functionl(num)
int num;
```

Вторая форма обращения требует, чтобы определение функции включало в себя формальный аргумент, являющийся указателем на объект соответствующего типа:

```
function2(ptr)
int *ptr;
```

Пользуйтесь первой формой, если входное значение необходимо функции для некоторых вычислений или действий, и второй формой, если функция должна будет изменять значения переменных в вызывающей программе. Вторая форма вызова уже применялась при обращении к функции **scanf( )**. Когда мы хотим ввести некоторое значение в переменную **num**, мы пишем **scanf("%d, &num)**. Данная функция читает величину, затем, используя адрес, который ей дается, помещает эту величину в память.

Указатели позволяют обойти тот факт, что переменные функции **interchange( )** являются локальными. Они дают возможность нашей функции "добраться" до функции **main( )** и изменить величи ны описанных в ней объектов.

Программисты, работающие на языке Паскаль, могут заметить, что первая форма вызова аналогична обращению с параметром-значением, а вторая - с параметром-переменной. У программистов, пишущих на языке Бейсик, понимание всей этой методики может вызвать некоторые затруднения. В этом случае если материал данного раздела покажется вам поначалу весьма не обычным, не сомневайтесь, что благодаря некоторой практике, все обсуждаемые средства станут простыми, естественными и удобными.

Переменные: имена, адреса и значения

Наше обсуждение указателей строится на рассмотрении связей между именами, адресами и значениями переменных; дальше мы продолжим обсуждение этих вопросов.

При написании программы мы представляем себе переменную как объект, имеющий два атрибута: имя и значение. (Кроме указанных существуют еще и другие атрибуты, например тип, но это уже другой вопрос.) После компиляции программы и загрузки в память "с точки зрения машины" данная переменная имеет тоже два атрибута: адрес и значение. Адрес - это машинный вариант имени.

Во многих языках программирования адрес объекта скрыт от программиста и считается относящимся к уровню машины. В языке Си благодаря операции **&** мы имеем возможность узнать и использовать адрес переменной:
&barn - это адрес переменной **barn**.

Мы можем получить значение переменной, соответствующее данному имени, *используя* только *само имя*:

```
printf(" %d\n", barn)          печатает значение переменной barn
```

Мы можем также получить значение переменной, исходя из ее адреса, при помощи операции ***:

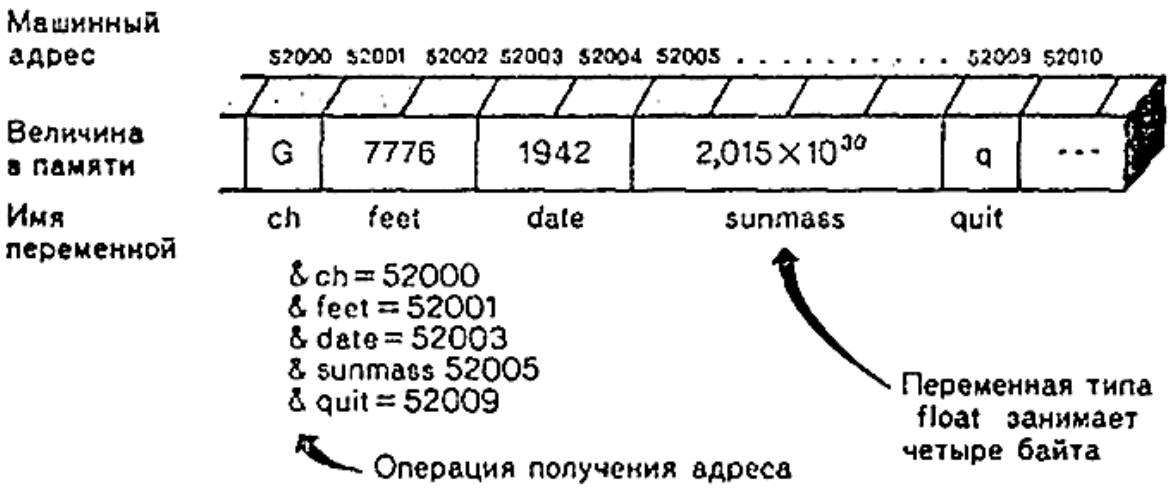


РИС. 9.6. Имена, адреса и величнны в системе с "байтовой адресацией" тина IBM PC.

Дано **pbarn** = **&barn;** тогда ***pbarn** - это величина, помещенная по адресу **&barn**. Хотя мы и можем напечатать адрес переменной для удовлетворения своего любопытства, это не основное применение операции *&*. Более важным является то, что наличие операций **&, *** и указателей позволяет обрабатывать адреса и их содержимое в символическом виде, чем мы и занимались в программе **обмен3**.

ИСПОЛЬЗОВАНИЕ НАШИХ ЗНАНИЙ О ФУНКЦИЯХ

Далее Содержание

Теперь, когда мы знаем о функциях немного больше, соберем вместе несколько поучительных примеров, но сначала решим, чем мы будет заниматься.

Что вы скажете насчет функции возведения в степень, которая дает возможность возводить 2 в 5-ю степень или 3 в 3-ю и т. д.? Во-первых, необходимо решить, что будет служить входом программы. Это понятно: Си требуется знать число, возводимое в степень, и показатель степени. Достичь этого можно путем введения двух аргументов:

```
power(base, exp)
int base, exp;
```

(Мы ограничились здесь целыми числами, а также тем предположением, что результат будет

сравнительно невелик.)

Далее требуется решить, что будет выходом функции. Ответ, конечно, тоже очевиден. Выходом должно быть одно число, являющееся значением переменной **answer**. Мы можем реализовать это с помощью оператора

```
return(answcr);
```

Теперь для получения требуемого результата выберем алгоритм:

установим переменную **answer** равной 1,

умножим **answer** на **base** столько раз, сколько указывает **exp**.

Возможно, не совсем ясно, как осуществить второй шаг, поэтому разобьем его дальше на более мелкие шаги:

умножим **answer** на **base** и уменьшим на 1, остановимся, когда **exp** станет равной 0.

Если значение **exp** равно, скажем 3, тогда использование такого алгоритма приведет к трем умножениям; поэтому данный способ кажется вполне разумным.

Очень хорошо. Выразим теперь этот алгоритм в виде программы на языке Си.

```
/* возводит основание в степень */
power(base, exp)
int base, exp;
{
int answer;
for (answer = 1; exp > 0; exp--)
        answer = answer* base;
return(answer);
}
```

Теперь проверим ее работу с помощью драйвера.

```
/* проверка возведения в степень */
main( )
{
int x;
x = power(2,3);
printf(" %d\n", x);
x = power(-3,3);
prinif(" %d\n", x);
x = power(4, -2);
printf(" %d\n", x);
x = power(5, 10);
printf(" %d\n", x);
}
```

Объединим указанные две функции, проведем компиляцию и выполним данную программу. Результаты оказываются следующими:

```
8
-27
1
761
```

Итак, 2 в 3-й степени - это 8, а - 3 в 3-й равно -27. Пока все правильно. Но 4 в степени -2 равно 1/16, а не 1. А 5 в 10-й степени, если память нам не изменяет,- это 9 765 625. В чем дело? Во-первых, программа не предназначалась для обработки отрицательных степеней, поэтому она и не смогла справиться с этой задачей. Во-вторых, в нашей системе величины типа **int** не могут превосходить 65 535.

Можно расширить программу путем включения в нее обработки отрицательных степеней и использования чисел с плавающей точкой для представления переменных **base** и **answer**. В любом случае показатель степени должен выражаться целым числом, потому что это число выполняемых

умножений; нельзя произвести 2,31 умножения.

```
/* возводит основание в степень*/
double power(base, exp)
double, base;
int exp;
{ double answer;
if(exp > 0)
{
for(answer = 1.0; exp > 0; exp --) answer * = base;
return(answer);
} else if(base != 0)
{ for(answer = 1.0; exp < 0; exp++ ) answer /= base;
return(answer);
} else /* base = 0 и exp <= 0 */
{ printf(" Нельзя возводить 0 в %d степень!\n", exp);
return(0); }
}
```

Необходимо отметить здесь несколько моментов.
Первый: самым главным является то, что мы должны описать тип функции! Переменная **answer** имеет тип **double**, следовательно, сама функция **power()** тоже должна быть типа **double**, так как ей присваивается величина, возвращаемая оператором **return**. Почему, спросите вы, мы не описывали тип функции раньше? Дело в том, что по умолчанию в языке Си функция имеет тип **int** (для большинства функций это так), если не указано что-то иное.
Второй: мы хотели показать, что не забыли те новые операции присваивания, которые ввели в гл. 8.
Третий: в соответствии с алгебраическими правилами возведение в отрицательную степень было преобразовано в деление. Это внесло опасность деления на нуль, но в данном случае мы предусмотрели выдачу сообщения об ошибке и возврат значения 0, чтобы работа программы не прекращалась.

Мы можем воспользоваться тем же драйвером при условии, что тип функции **power( )** там тоже описан.

```
/* проверка возведения в степень        */
main( )
{
double x;
double power( ); /* это пример  описания  функции */
x = power(2.0, 3);
printf(" %.0f \n", x);
x = power(-3.0, 3);
printf(" %.0f\n" , x);
x = power(4.0, -2);
printf(" %.4f\n", x);
x = power(5.0, 10);
print f ("%.0f \n", x);
}
```

На этот раз результаты работы программы выглядят вполне удовлетворительно.

```
8
-27
0.0625
9765625
```

Данный пример побуждает нас ввести следующий короткий раздел.

ОПИСАНИЕ ТИПОВ ФУНКЦИЙ

Далее Содержание

Тип функции определяется типом возвращаемого ею значения, а не типом ее аргументов. Если указание типа отсутствует, то по умолчанию считается, что функция имеет тип **int**. Если значения функции не принадлежат типу **int**, то необходимо указать ее тип в двух местах.

1. Описать тип функции в ее определении:

```
char pun(ch, n) /* функция возвращает символ */
int n;
char ch;

float raft(num) /* функция возвращает величину типа float */
int num;
```

2. Описать тип функции также в вызывающей программе. Описание функции должно быть приведено наряду с описаниями переменных программы; необходимо только указать скобки (но не аргументы) для идентификации данного объекта как функции.

```
main( )
{ char rch, pun( );
float raft; }
```

Запомните! Если функция возвращает величину не типа **int**, указывайте тип функции там, где она определяется, и там, где она используется.

Резюме: функции

1. Форма записи
Типичное определение функции имеет следующий вид:

```
имя (список аргументов)
описание аргументов
тело функции
```

Наличие списка аргументов и описаний не является обязательным. Переменные, отличные от аргументов, описываются внутри тела, которое заключается в фигурные скобки.

Пример:

```
diff(x, y)                  /* имя функции и список аргументов */
ini x, y;                  /* описание аргументов */
 {                         /* начало тела функции */
int z;                 /* описание локальной переменной */
z = x - y;
return(z);
}                        /* конец тела функции */
```

II. Передача значений функции:
Аргументы используются для передачи значений из вызывающей программы и функцию. Если значения переменных **a** и **b** будут 5 и 2, то при вызове

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

осуществляется передача этих значений переменным **x** и **y**. Значения 5 и 2 называют ся фактическими аргументами, а переменные **x** и **y**, указанные в описании функции: **diff( )** - формальными аргументами.

Использование ключевого слова **return** позволяет передавать в вызывающую программу одно значение из вызываемой функции. В нашем примере переменной с присваивается значение переменной **z**, равное 3.

Обычно выполнение функции не оказывает никакого влияния на значения переменных вызывающей программы. Чтобы иметь возможность непосредственно изменять значения переменных вызывающей программы, необходимо использовать указатели в качестве аргументов. Это может оказаться необходимым в случае, если в вызывающую программу требуется передать более чем одно значение.

III. Тип функции
Функции должны иметь тот же тип, что и значения, которые они возвращают в качестве результатов. По умолчанию предполагается, что функции имеют тип **int**. Если функция имеет другой тип, он должен быть указан и в вызывающей программе, и в самом определении функции.

Пример

```
main( )
{
float q, x, duff( ); /* описание в вызывающей программе */
int n;
...
q = duff(x, n);
...
}
float duff(u, k); /* описание в определении функции */
float u;
int k;
{
float tor;
...
return(tor); /* возвращает значение типа float */
}
```

В ЯЗЫКЕ СИ ВСЕ ФУНКЦИИ РАВНОПРАВНЫ

Далее Содержание

Все функции в программе, написанной на языке Си, равноправны: каждая из них может вызывать любую другую функцию и в свою очередь каждая может быть вызвана любой другой функцией. Это делает функции языка Си несколько отличными от процедур Паскаля, поскольку процедуры в Паскале могут быть вложены в другие процедуры (причем, процедуры, содержащиеся в одном гнезде, являются недоступными для процедур, расположенных в другом).

Нет ли у функции **main( )** какой-то специфики? Безусловно, есть; она заключается в том, что после "сборки" программы, состоящей из нескольких функций, ее выполнение начинается с первого оператора функции **main( )**. Но этим ее исключительность и граничивается. Даже функция **main( )** может быть вызвана другими функциями, как показывает приведенный ниже пример:

```
/* вызов функции main( ) */
#include
main( )
{
char ch;
printf (" Укажите произвольный символ. Q - признак конца работы. \n");
ch = getchar( );
printf ("Так! Вы указали %c!\n", ch);
if(ch != 'Q') more( );
} more( );
{
main( );
}
```

Функция **main( )** вызывает **more()**, а функция **more()** вызывает **main()**! После вызова функции **main( )** ее выполнение начинается с самого начала; мы организовали цикл с взаимным вызовом.

Функция может даже вызывать сама себя. Упростим предыдущий пример следующим образом:

```
/* main.main */
#include
main( )
{
char ch;
printf (" Укажите произвольный символ. Q - признак конца работы.\n");
ch = getchar( );
printf ("Так! Вы указали %c!\n", ch);
if(ch != 'Q') main( );
}
```

Ниже приводятся результаты одного прогона программы, показывающие, что она работает. Обратите внимание на то, как обрабатывается символ "новая строка", который передается программе при нажатии клавиши [**ввод**].

```
Введите произвольный символ. Q - признак конца работы.
I
Так! Вы указали I !
Введите произвольный символ. Q - признак конца работы.
!
Так! Вы указали ! !
Введите произвольный символ. Q - признак конца работы.
Q
Так! Вы указали Q !
```

Действие, состоящее в том, что функция вызывает сама себя, называется "рекурсией". Цикл, который мы создали, используя рекурсию, отличается от циклов **while** и **do while**. Когда функция **main( )** вызывает сама себя, не происходит передачи управления на ее начало. Вместо этого в памяти машины создаются копни всего набора переменных функции **main( )**. Если вы выведете на печать адреса переменных в обычном цикле, то увидите, что эти адреса не изменяются от итерации к итерации. Что же касается рассматриваемого здесь цикла, то в нем адрес используемой переменной меняется, поскольку при каждом выполнении тела цикла создается новая копия переменной **ch**. Если программа циклически выполняется 20 раз, то будет создано 20 различных копий переменной, каждая из которых носит имя **ch**, но имеет свой собственный адрес.

Компиляция программ, состоящих из двух или более функций

Простейший способ использования нескольких функций в одной программе заключается в том, чтобы поместить их в один файл, после чего осуществить компиляцию программы, содержащейся в этом файле так, как будто она состояла из одной функции.

Второй способ заключается и применении директивы **#include**. Если одна функция содержится в файле с именем **file1.c**, а вторая и файле **file2.c**, поместите эту директиву в файл **filel.c**:

```
#include "file2.c"
```

Дополнительная информация о директиве **#include** находится п гл. 11. Другие возможные способы являются в большей степени системнозависимыми. Вот некоторые из них:

ОС UNIX

Предположим, **file1.c** и **file2.c** - два файла, содержащие программные тексты, соответствующие функциям языка Си. В результате выполнения команды

```
cc file1.c  file2.c
```

будет осуществлена компиляция функций, содержащихся в обоих файлах, и получен файл выполняемого кода с именем **a.out**. Кроме того, будут созданы два файла с "объектным" кодом - **file1.0** и **file2.0**. Если позже вы измените текст, содержащийся в файле с именем **filel.c**, а второй файл оставите без изменений, то сможете осуществить компиляцию первого файла, а затем объединить полученный объектный код с объектным кодом, соответствующим второму файлу, при помощи команды

```
cc file1.c  file2.0
```

Компиляторы Lattice C и MICROSOFT C

Выполните раздельную компиляцию функции, содержащихся в файлах **filel.c** и **file2.c;** в результате будут получены два файла с объектным кодом - **file1.obj** и **file2.obj**. Используйте системный редактор связей для объединения их друг с другом и со стандартным объектным модулем **c.obj**:

```
link c filel file2
```

Системы, построенные на основе трансляции в ассемблкрный код

Некоторые из таких систем позволяют компилировать функции, содержащиеся в нескольких файлах, сразу так же, как в ОС UNIX с помощью команды:

```
cc filel.c  file2.c
```

или какого-то ее эквивалента. В некоторых случаях вы можете получить отдельные модули с кодом ассемблера, а затем объединить их, используя процесс ассемблирования.

РЕЗЮМЕ

Далее Содержание

Для создания больших программ вы должны использовать функции в качестве "строительных блоков". Каждая функция должна выполнять одну вполне определенную задачу. Используйте аргументы для передачи значений функции и ключевое слово **return** для передачи результирующего значения в вызывающую программу. Если возвращаемое функцией значение не принадлежит типу **int**, вы должны указать тип функции в ее определении и в разделе описаний вызывающей программы. Если вы хотите, чтобы при выполнении функции происходило изменение значении переменных в вызывающей программе, вы должны пользоваться адресами и указателями.

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Как определять функцию.
Как передавать функции информацию: при помощи аргументов.
Различие между формальным и фактическим аргументами: первый является переменной, используемой функцией, а второй - значением, поступающим из вызывающей функции.
Где необходимо описывать аргументы: после имени функции и перед первой фигурной скобкой.
Где необходимо описывать остальные локальные переменные: после первой фигурной скобки.
Когда и как использовать оператор **return**.

Когда и как использовать адреса и указатели для доступа к объектам.

ВОПРОСЫ И ОТВЕТЫ

Далее Содержание

Вопросы

1. Напишите функцию, возвращающую сумму двух целых чисел.

2. Какие изменения должны были бы произойти с функцией из вопроса 1, если вместо целых складывались бы два числа типа **float**?

3. Напишите функцию **alter( )**, которая берет две переменные **x** и **y** типа **int** и заменяет соответственно на их сумму и разность.

4. Проверьте, все ли правильно в определении функции, приведенной ниже?

```
salami(num)
{
int num, count;
for(count = 1; count <= num; num++) printf(" О салями!\n");
}
```

Ответы

1.

```
sum(j,k) int j, k;
{ return(j+k); }
```

2.

```
float sum(j,k) float j,k;
```

Необходимо также привести описание функции **float sum( )** и вызывающей программе.

3. Поскольку мы хотим изменить две переменные в вызывающей программе, можно воспользоваться адресами и указателями. Обращение к функции будет выглядеть так: **alter(&x,&y)**. Возможное решение имеет следующий вид:

```
alter(px, py)
int *px, *py; /* указатели на x и y*/
{
int sum, diff;
sum = *px + *py; /* складывает содержимое двух переменных, определяемых адресами */
diff = *px - *py;
*px = sum;
*py = diff;
}
```

4. Нет; переменная **num** должна быть описана перед первой фигурной скобкой, а не после нее. Кроме того, выражение **num++** необходимо заменить на **count++**.

УПРАЖНЕНИЯ

1. Напишите функцию **max(x, y)**, возвращающую большее из двух значении.

2. Напишите функцию **chIlne(ch, i, j)**, печатающую запрошенный символ с **i**-й по **j**-ю позиции. Смотри программу **художник-график**, приведенную в гл. 7.

---

10. Классы памяти и разработка программ

ЛОКАЛЬНЫЕ И ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ
КЛАССЫ ПАМЯТИ
ФУНКЦИЯ ПОЛУЧЕНИЯ СЛУЧАЙНЫХ ЧИСЕЛ
ПРОВЕРКА ОШИБОК
МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ
СОРТИРОВКА

КЛЮЧЕВЫЕ СЛОВА
auto, extern, static, register

Одно из достоинств языка Си состоит в том, что он позволяет управлять ключевыми механизмами программы. Классы памяти языка Си - пример такого управления; они дают возможность определить, с какими функциями связаны какие переменные и как долго переменная сохраняется в программе. Классы памяти - первая тема данной главы.

Программирование, точно так же как написание романа (или даже письма),- это не просто знание языковых правил - это нечто большее. В данной главе мы рассмотрим несколько полезных функций. При этом попытаемся привести некоторые соображения, используемые при конструировании функций. В частности, сделаем упор на значение модульного подхода, разбивающего программы на выполнимые задачи. Сначала, однако, обсудим классы памяти.

КЛАССЫ ПАМЯТИ И ОБЛАСТЬ ДЕЙСТВИЯ

Далее Содержание

Мы уже упоминали раньше, что локальные переменные известны только функциям, содержащим их. В языке Си предполагается также, что о глобальных переменных "знают" сразу несколько функций. Предположим, например, что и **main( )**, и **critic( )** имеют доступ к переменной **units**. Это будет иметь место, если отнести **units** к "внешнему" классу памяти, как показано ниже:

```
/* глобальная переменная units */
int units;  /* внешняя переменная */
main( )
{
extern int units;
printf (" Сколько фунтов масла находится в бочонке?\n");
scanf (" %d" , &units);
while (units != 56) critic( );
printf(" Вы должны поискать в справочнике !\n");
} critic( )
{
extern int units;
printf (" Не повезло, дружок. Попытайся снова.\n");
scanf (" %d" , &units);
}
```

Вот полученный результат:

```
Сколько фунтов масла находится в бочонке?
14
Не повезло, дружок. Попытайся снова.
56
```

```
Вы должны поискать в справочнике!
```

(Мы сделали это.)

Обратите внимание, что второе значение **units** было прочитано функцией **critic( )**, однако **main()** также "узнала" это новое значение, когда оно вышло из цикла **while**.

Мы сделали переменную **units** внешней, описав ее вне любого определения функции. Далее, внутри функций, использующих эту переменную, мы объявляем ее внешней при помощи ключевого слова **extern**, предшествующего спецификации типа переменной. Слово **extern** предлагает компьютеру искать определение этой переменной вне функции. Если бы мы опустили ключевое слово **extern** в функции **critic( )**, то компилятор создал бы в функции **critic** новую переменную и тоже назвал бы ее **units**. Тогда другая переменная **units()** [которая находится в **main()**] никогда не получила бы нового значения.

Каждая переменная, как мы знаем, имеет тип. Кроме того, каждая переменная принадлежит к некоторому классу памяти. Есть четыре ключевых слова, используемые для описания классов памяти: **extern** (для внешнего), **auto** (для автоматического), **static** и **register**. До сих пор мы не обращали внимание на классы памяти, так как переменные, описанные внутри функции, считались относящимися к классу **auto**, если они не описывались иначе (по умолчанию они относились к классу **auto**).

Определение класса памяти переменной зависит от того, гд переменная описана и какое ключевое слово (если оно есть) используется.

Класс памяти позволяет установить два факта. Во-первых, определить, какие функции имеют доступ к переменной. (Пределы, до которых переменная доступна, характеризуют ее "область действия".) Во-вторых, определить, как долго переменная находится в памяти. Теперь перейдем к свойствам каждого типа.

Автоматические переменные

Далее Содержание

По умолчанию переменные, описанные внутри функции, являются автоматическими. Можно, однако, это подчеркнуть явно с помощью ключевого слова **auto**:

```
main( )
{
auto int plox;
```

Так поступают, если хотят, например, показать, что определение переменной не нужно искать вне функции.

Автоматические переменные имеют локальную область действия. Только функция, в которой переменная определена, "знает" ее. (Конечно, можно использовать аргументы для связи значения и адреса переменной с другой функцией, однако это частичное и косвенное "знание".) Другие функции могут использовать переменные с тем же самым именем, но это должны быть независимые переменные, находящиеся в разных ячейках памяти.

Автоматическая переменная начинает существовать при вызове функции, содержащей ее. Когда функция завершает свою работу и возвращает управление туда, откуда ее вызвали, автоматическая переменная исчезает. Ячейка памяти может снова использоваться для чего-нибудь другого.

Следует еще сказать об области действия автоматической переменной: область действия ограничена блоком ({ }), в котором переменная описана. Мы всегда должны описывать наши переменные в начале тела функции (блока), так что областью действия их является вся функция.

Однако в принципе можно было бы описать переменную внутри подблока. Тогда переменная будет известна только в этой части функции. Обычно при создании программы, программисты редко принимают во внимание упомянутое свойство. Но иногда торопливые программисты пользуются такой возможностью, особенно когда пытаются быстрее внести коррективы.

Внешние переменные

Далее Содержание

Переменная, описанная вне функции, является внешней. Внешнюю переменную можно также описать в функции, которая использует ее, при помощи ключевого слова **extern**. Описания могут выглядеть примерно так:

```
int errupt;   /* Три переменные, описанные вне функции */
char coal;
double up;
main( )
{
extern int errupt;       /* объявлено, что 3 переменные */
extern char coal;       /* являются внешними */
extern double up;
```

Группу **extern**-описаний можно совсем опустить, если исходные определения переменных появляются в том же файле и перед функцией, которая их использует. Включение ключевого слова **extern** позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом файле. (Оба файла, конечно, должны быть скомпилированы, связаны или собраны в одно и то же время.)

Если слово **extern** не включено в описание внутри функции, то под этим именем создается новая автоматическая переменная. Вы можете пометить вторую переменную как "автоматическую" с помощью слова auto и тем самым показать, что это ваше намерение, а не оплошность. Три примера демонстрируют четыре возможных комбинация описаний:

```
/* Пример1 */
int hocus;
main( ) {
extern int hocus;       /*      hocus   описана внешней */
...
}
magic( ) {
extern int      hocus;
...
}
```

Здесь есть одна внешняя переменная **hocus**, и она известна обеим функциям **main( )** и **magic( )**.

```
/* Пример2 */
int hocus ;
main( )
{
extern int hocus;   /* hocus описана внешней    */
...
}
magic( )
{
/* hocus не описана совсем */
...
}
```

Снова есть одна внешняя переменная **hocus**, известная обеим функциям. На этот раз она известна функции **magic( )** по умолчанию.

```
/* Пример3 */
int hocus;
main( )
{
int hocus;   /* hocus описана и
        является автоматической по умолчанию */
...
}
magic( )
 {
auto int hocus; /* hocus описана автоматической */
...
}
```

В этом примере созданы три разные переменные с одинаковым именем. Переменная **hocus** в функции **main( )** является автоматической по умолчанию и локальной для **main( )**, в функции **magic( )** она явно описана автоматической и известна только для **magic( )**. Внешняя переменная **hocus** не известна ни **main( )**, ни **magic( )**, но обычно известна любой другой функции в файле, которая не имеет своей собственной локальной переменной **hocus**.

Эти примеры иллюстрируют область действия внешних переменных. Они существуют, пока работает программа, и так как эти переменные доступны любой функции, они не исчезнут, если какая-нибудь одна функция закончит свою работу.

Статические переменные

Далее Содержание

Название раздела не следует понимать буквально, т. е. считать, что такие переменные не могут изменяться. В действительности слово "статические" здесь означает, что переменные остаются в работе. Они имеют такую же область действия, как автоматические переменные, но они не исчезают, когда содержащая их функция закончит свою работу. Компилятор хранит их значения от одного вызова функции до другого. Следующий пример иллюстрирует это и показывает, как описать статическую переменную.

```
/* статическая переменная */
main( )
{
int count;
for (count = 1; count <= 3; count++)
{
printf(" Итерация %d:\n", count);
trystat( );
}
}
trystat( )
{
int fade = 1;
static int stay;        = 1;
printf("fade =  %d и stay =     %d\n", fade++, stay++);
}
```

Заметим, что функция **trystat( )** увеличивает каждую переменную после печати ее значения. Работа этой программы даст следующие результаты:

```
Итерация 1:
fade = 1 и staly = 1
Итерация 2:
fade = 1 и stay = 2
Итерация 3:
fade = 1 и stay = 3
```

Статическая переменная **stay** "помнит", что ее значение было увеличено на 1, в то время как для переменной **fade** начальное значение устанавливается каждый раз заново. Это указывает на разницу в инициализации: **fade** инициализируется каждый раз, когда вызывается **trystat( )**, в то время как **stay** инициализируется только один раэ при компиляции функции **trystat( )**.

Внешние статические переменные

Далее Содержание

Вы можете также описать **статические** переменные вне любой функции. Это создаст **"внешнюю статическую"** переменную. Разница между внешней переменной и внешней статической переменной заключается в области их действия. Обычная внешняя переменная может использоваться функциями в любом файле, в то время как внешняя статическая переменная может использоваться только функциями того же самого файла, причем после определения переменной. Вы описываете внешнюю статическую переменную, располагая ее определение вне любой функции.

```
static randx = 1;
rand( )
{
```

Немного позже мы приведем пример, в котором будет необходим этот тип переменной.

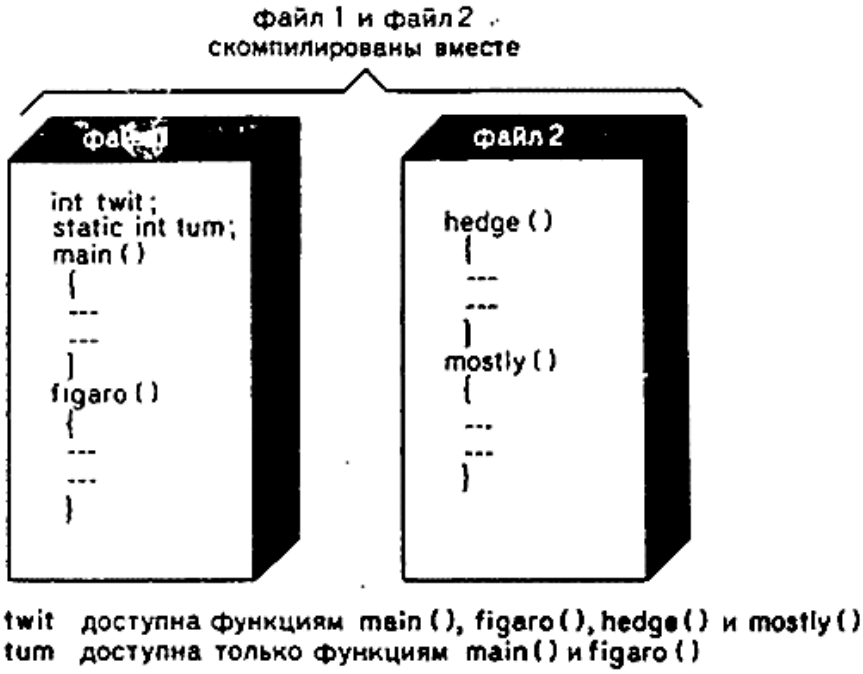


РИС. 10.1. Внешние и внешние статические переменные.

Регистровые переменные

Далее Содержание

Обычно переменные хранятся в памяти машины. К счастью, регистровые переменные запоминаются в регистрах центрального процессора, где доступ к ним и работа с ними выполняются гораздо быстрее, чем в памяти. В остальном регистровые переменные аналогичны автоматическим переменным. Они создаются следующим образом:

```
main( )
{
register int quick;
```

Мы сказали "к счастью", потому что описание переменной как регистровой, является скорее просьбой, чем обычным делом. Компилятор должен сравнить ваши требования с количеством доступных регистров, поэтому вы можете и не получить то, что хотите. В этом случае переменная становится простой автоматической переменной.

Какой класс памяти применять? Далее Содержание

Ответ на вопрос почти всегда один - "автоматический". В конце концов почему этот класс памяти выбран по умолчанию? Мы знаем, что на первый взгляд использование внешних переменных очень соблазнительно. Опишите все ваши переменные как внешние, и у вас никогда не будет забот при использовании аргументов и указателей для связи между функциями в прямом и обратном направлениях. К сожалению, у вас возникнет проблема с функцией С, коварно изменяющей переменные в функции А, а это совсем не входит в паши интересы. Неоспоримый совокупный опыт использова-ния машин, накопленный в течение многих лет, свидетельствует о том, что такая проблема значительно перевешивает кажущуюся привлекательность широкого использования внешних переменных.

Одно из золотых правил защитного программирования заключается в соблюдении принципа "необходимо знать только то, что нужно". Организуйте работу каждой функции автономно, насколько это возможно, и используйте глобальные переменные только тогда, когда это действительно необходимо.

Иногда полезны и другие классы памяти. Но прежде чем их использовать, спросите себя, необходимо ли это.

Резюме: Классы памяти

I. Ключевые слова: **auto, extern, static, register**

II. Общие замечания:
Класс памяти определяет область действия переменной и продолжительность ее существования в памяти. Класс памяти устанавливается при описании переменной с соответствующим ключевым словом. Переменные, определенные вне функции, являются внешними и имеют глобальную область действия. Переменные, определенные внутри функции, являются автоматическими и локальными, если только не используются другие ключевые слова. Внешние переменные, определенные раньше функции, доступны ей, даже если не описаны внутри ее.

III. Свойства

КЛАСС ПАМЯТИ	КЛЮЧЕВОЕ СЛОВО	ПРОДОЛЖИТЕЛЬНОСТЬ СУЩЕСТВОВАНИЯ	ОБЛАСТЬ ДЕЙСТВИЯ
Автоматический Регистровый Статический	auto register static	Временно Временно Постоянно	Локальная Локальная Локальная
Внешний Внешний статический	extern static	Постоянно Постоянно	Глобальная (все файлы) Глобальная (один файл)

1. Разделим случайное число на 32768. В результате получим число **x** в диапазоне **- 1 <= x < 1**. (Мы должны превратить его в тип **float**, чтобы иметь десятичные дроби.)
2. Добавим 1. Наше новое число удовлетворяет отношению **0 < = x < 2**.
3. Разделим на 2. Теперь имеем **0 <= x < 1**.
4. Умножим на 6. Имеем **0 <= x < 6**. (Близко к тому, что нужно, но 0 не является возможным

значением.)
5. Прибавим 1: **1 <= x < 7**. (Заметим, что эти числа все еще являются десятичными дробями.)
6. Преобразуем в целые числа. Теперь мы имеем целые в диапазоне от 1 до 6.
7. Для обобщения достаточно заменить значение 6 в п. 4 на число сторон.
Вот функция, которая выполняет эти действия:

```
/* электронное бросание костей */
#define SCALE 32768.0
rollem(sides) float sides;
{
float roll;
roll = ((float)rand( )/SCALE + 1.0) * sides/2.0 + 1.0;
return((int)roll);
}
```

Мы включили в программу два явных описания типа, чтобы показать, где выполняются преобразования типов. Обратимся к программе, которая использует эти средства:

```
/* многократное бросание кости */
main( )
{
int dice, count, roll, seed;
float sides;
printf(" Введите, пожалуйста, значение зерна. \n");
scanf(" %d, &seed);
srand(seed);
printf(" Введите число сторон кости, 0 для завершения. \n");
scanf(" %d", &sides);
while(sides > 0)
{ printf(" Сколько костей?\n");
scanf(" %d", &dice);
for( roll = 0, count = 1; count <= dice;        count++)
roll + = rollem(sides); /* бросание всех костей набора */
printf(" У вас выпало %d, для %d  %.0f-сторонних костей.\n", roll, dice, sides);
printf(" Сколько сторон? Введите 0 для завершения.\n");
scanf(" %f", &sides);
} printf(" Удачи вам!\n");
}
```

Теперь давайте используем эту программу:

```
Введите значение зерна
1
Введите число сторон  кости,  0 для завершения.
6
Сколько костей?
2
У вас выпало 4 для 2 6-сторонних костей.
Сколько сторон ? Введите  0 для завершения.
6
Сколько костей ?
2
У вас выпало 7 для 2  6-сторонних костей.
Сколько сторон? Введите  0 для завершения.
0
Удачи Вам!
Спасибо.
```

Вы можете использовать функцию **rollem( )** по-разному. Пусть число сторон (**sides**) равно двум, тогда бросание) монеты даст следующий результат: "орел" выпал 2 раза, а "решка" - один (или наоборот, смотря, что вы предпочитаете). Можно легко модифицировать программу, чтобы показать как отдельные результаты, так и итог. Или вы можете построить имитатор игры "крапс". Если вам нужно большое число бросаний, вы можете легко модифипировать свою программу и

получить результат, подобный следующему:

```
Введите значение зерна.
10
Введите количество ходов; введите 0 для завершения.
18
Сколько сторон и сколько костей? 6   3
Здесь 18 ходов для 3  6-сторонних костей.
7 5  9 7 12 10 7 12 10 14
9 8 13 9 10 7 16 10
Сколько ходов? Введите 0 для завершения. 0
```

Использование функции **rand( )** [но не **rollem( )**] изменило бы вашу программу угадывания чисел: компьютер стал бы выбирать, а вы угадывать, вместо того чтобы сделать наоборот.

Разработаем еще некоторые функции. Сначала мы хотим создать функцию, которая читает целые числа.

ФУНКЦИЯ ПОЛУЧЕНИЯ ЦЕЛЫХ ЧИСЕЛ: getint( )

Далее Содержание

Вероятно, наш проект покажется вам очень простым. В конце концов мы можем использовать функцию **scanf( )** с форматом **%d**, если хотим прочитать целое число. Такой подход очень прост, но он имеет один большой недостаток. Если вы ошибочно напечатали, скажем **T** вместо **6**, **scanf( )** попытается интерпретировать **T** как целое число. Мы хотим создать функцию, которая проверяет вводимую информацию и предупреждает вас, если введено нецелое число. Теперь, может быть, наш проект уже не кажется таким простым. Однако не беспокойтесь - мы хорошо начали: у нас есть имя для новой функции. Мы назовем ее **getint( )**.

План

Далее Содержание

К счастью, мы уже выработали стратегию. Во-первых, заметим, что любую вводимую информацию можно читать как строку символов. Целое число 324, например, можно прочитать как строку из трех символов: символ **'3'**, символ **'2'** и символ **'4'**. Это подсказывает нам следующий план:
1. Прочитать вводимую информацию как символьную строку.
2. Проверить, состоит ли строка только из символов цифр и стоит ли перед ними знак плюс или минус.
3. Если все это имеет место, превратить символьную строку в правильное числовое значение.
4. Если нет, выдать предупреждение.

Этот план так хорош, что он должен работать. (Тот факт, что он представляет собой стандартный подход, существовавший на протяжении многих лет, придает нам также некоторую уверенность в возможности его выполнения. Но, прежде чем начать программировать, нужно подумать, что будет делать наша функция.

В частности, до того как мы займемся содержанием нашей функции **getint( )**, нужно точно решить, как она должна взаимодействовать со своим окружением: с какой информацией? Какую информацию она должна получать от вызывающей программы? Какую информацию должна возвращать? В каком виде должна быть эта информация? Снова мы рассматриваем функцию как черный ящик. Мы хотим знать, что входит в функцию и что выходит из нее и, наконец, что находится внутри ее. Этот подход помогает обеспечивать более однородное взаимодействие между различными частями программы. Иначе вы можете оказаться в положении человека, пытающегося установить трансмиссию автомашины "Волво" в автомобиль "Тойота". Сама функция правильная, но интерфейс является проблемой.

Какой выход должна иметь наша функция? Во-первых, несомненно, что она должна была бы выдавать значение прочитанного числа. Конечно, функция **scanf( )** уже делает так. Во-вторых, и это очень существенно, мы собираемся создать такую функцию, которая будет выдавать сообщения о состоянии, т. е. найдено или нет целое число. Чтобы функция была действительно полезной, она должна также сообщать о нахождении ею символа **EOF**. Тогда мы могли бы использовать функцию **getint( )** в цикле **while**, который читает целые числа до тех пор, пока не обнаружит символ **EOF**. Короче говоря, нам нужно, чтобы **getint( )** возвращала два значения: целое числе и состояние.

Так как мы хотим иметь два значения, то с одной функцией **return** с этой задачей нам не справиться. Нам следует иметь два указателя. Однако полное решение задачи мы получим, если используем указатели для выполнения основной работы функции и функцию **return** для выдачи некоторого кода состояния. Именно это и делает функция **scanf( )**. Она возвращает количество символов, которые нашла, и символ **EOF**, если встречает его. Ранее мы не пользовались такой возможностью, но могли бы, если бы применяли вызов вида

```
status = scanf(" % d", &number);
```

Теперь будем это делать. Тогда наш вызов функции выглядел бы следующим образом:

```
status = getint(&number);
```

Правая часть равенства использует адрес переменной **number**, чтобы получить ее значение, а **return** применяется для получения значения переменной **status**.

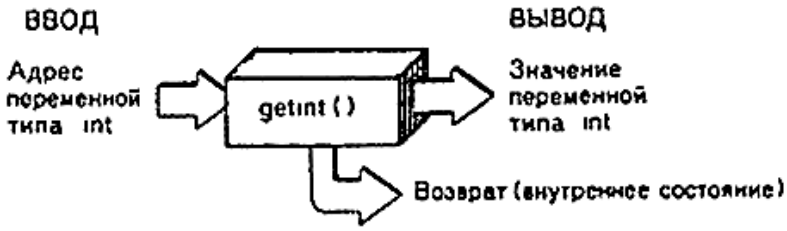


РИС. 10.2. Создание функции getint( )

Мы должны выбрать коды для выдачи сообщения о состоянии. Так как предполагается, что неописанные функции имеют тип **int**, наши коды должны состоять из целых чисел. Используем следующие коды для сообщения о состоянии:

-1 означает, что найден символ EOF.
1 означает, что найдена строка, содержащая не цифры.
0 означает, что найдена строка, содержащая только цифры.

Нашу функцию **getint( )** можно представить себе (рис. 10.2) как имеющую один вход и два выхода. На ее вход поступает адрес целой переменной, значение которой считывается. На первом выходе имеем значение считанного целого, полученного через указатель. (Таким образом, аргумент-указатель является двусторонним каналом передачи информации.) На втором выходе получаем код состояния, что обеспечивается функцией **return**. Отсюда следует, что "скелет" нашей функции должен выглядеть примерно так:

```
getint(ptint)
int *ptint;    /* указатель     на целое число */
{
int status;
...
return (status);
}
```

Замечательно! Теперь мы должны просто заполнить внутренность функции.

Содержание getint( ) Далее Содержание

Наш основной план для **getint( )** в общих чертах на псевдокоде выглядит примерно так:

читаем на входе информацию в виде символов
помещаем символы в строку, пока не встретим символ **EOF**
если встретился символ **EOF**, устанавливаем состояние в **STOP**
в противном случае проверяем строку,
преобразуем символы в целое число, если возможно, и выдаем сообщение о состоянии (**YESNUM** или **NONUM**).

Здесь мы используем **STOP**, **YESNUM** и **NONUM** как символические константы, равные **-1**, **0** и **1**, как описано выше.

Рассмотрим еще некоторые вопросы. Как функция будет решать, что она достигла конца входной строки? Должны ли мы ограничивать длину строки?

Мы вошли в область, где нам предстоит решать, что предпочесть: удобство программиста или удобство пользователя. Самым простым было бы предложить пользователю нажимать на клавишу [**ввод**], когда строку надо закончить. Это означало бы один ввод на строку. Для пользователя все же было бы приятнее, если бы функция могла размещать несколько чисел в одной и той же строке:

```
2   34    4542    2    98
```

Мы решили предоставить привилегию пользователю. Пусть функция будет считать, что строка начинается с символа, не являющегося **пробелом** или **"новой строкой"**, и заканчивается символом **пробела** или **"новой строкой"**. Такой ввод может производиться в одну строку или в несколько строк.

Мы ограничим вводимую строку 80 символами. Так как строки заканчиваются **нуль-символом**, нам нужен массив из 81 символа для включения в него этого символа. Это слишком щедро, потому что нам нужно только 6 символов для 16-разрядного целого числа и знака. Вы можете вводить более длинные числа, но их размер будет сокращен до размера строки.

Чтобы сделать программу более модульной, мы поручим преобразование "строка в целое число" другой функции, и назовем ее **stoi( )**. У нас будет также возврат функцией **stoi( )** соответствующего кода состояния в функцию **getint( )**, а **getint( )**, в свою очередь, может передать код состояния своей вызывающей программе. Функция **stoi( )** выполнит последние две строки нашего плана (на псевдокоде).

Рис. 10.3 представляет программу для функции **getint( );**. Функция **stoi( )** будет показана позже:

```
/* getint( ) */
#include
#define LEN 81 /* максимальная длина строки */
#define STOP-1 /* коды состояний */
#define NONUM 1
#define YESNUM O
getint(ptint)
int  *ptint; /* указатель на вывод целого числа */
{
char intarr[LEN]; /* запоминание вводимой строки */
int ch;
int ind = 0; /* индекс массива */
```

```
while((ch = getchar( )) = = '\n' || ch == '  ' || ch == '\t');
/* обход начальных символов "новая строка", пробелов и табуляций */
while(ch != EOF && ch != '\n' && ch != '   ' && ind < LEM)
{
intarr[ind++] = ch; /* запись символа в массив */
ch = getchar( ); /* получение очередного символа */
}
intarr[ind] = '\0'; /* конец массива по нуль-символу */
if(ch == EOF)
return(STOP);
else
return(stoi(intarr, ptint) ); /* выполнение преобразования */
}
```

РИС. 10.3. Программа функции getint( )

Мы получаем символ **ch**. Если он является символом пробела, или "новой строки", или табуляции, мы берем следующий символ и так продолжаем до тех пор, пока не получим символ, отличающийся от перечисленных. Затем, если этот символ не **EOF**, помещаем его в массив. Продолжаем брать символы и помещать их в массив, пока не найдем запрещенный символ или не достигнем предельного размера строки. Далее помещаем нуль-символ (**' \0'**) в следующую позицию массива, чтобы отметить конец строки. Таким образом, мы создали массив в виде стандартной символьной строки. Если **EOF** был последним прочитанным символом, возвращаем **STOP;** иначе идем дальше и пытаемся преобразовать строку. Мы вызываем новую функцию **stoi( )**, чтобы выполнить эту работу. Что делает **stoi( )**? При вводе она берет символьную строку и указатель на целую переменную, использует указатель для присваивания значения самой переменной, а также **return** для пересылки сообщения о состоянии, которое **getint( )** передает затем функции **getarray( )**. Поразительно! Двойная игра! Вот менее компактный способ использования функции **stoi( ):**

```
status = stoi(intarr, print);
return (status);
```

Здесь **status** была бы переменной типа **int**. Первый оператор дает значение, на которое указывает **ptint**; она также присваивает значение переменной **status**. Второй оператор возвращает это значение программе, которая вызвала **getint( )**. Наша единственная строка программы имеет точно такой же эффект, за исключением того, что нам не нужна промежуточная переменная **status**. Теперь напишем функцию **stol( )**.

Преобразование строки в целое: stoi( )

Далее Содержание

Сначала опишем, каким должен быть вход и выход у этой функции. Вход будет символьной строкой, поэтому **stoi( )** будет иметь символьную строку в качестве аргумента. На выходе должно быть получено два значения: состояние и преобразованное целое число. Мы применяем **return** для состояния и поэтому должны использовать указатель для возврата другого значения. Таким образом, появится второй аргумент - указатель на целое число. Скелет нашей функции будет выглядеть примерно так:

```
stoi(string, intptr)
char string[ ];  /* вводимая строка * /
int *intptr; /* указатель на переменную, получающую целое значение*/
{
int status;
...
return(status);
}
```

Прекрасно, а что можно сказать об алгоритме выполнения преобразования? На некоторое время проигнорируем знак и предположим, что строка содержит только цифры. Возьмем первый символ и

преобразуем его в числовой эквивалент. Предположим, это символ **'4'**. Он имеет в коде ASCII числовое значение 52 и в таком виде запоминается. Если мы из него вычтем 48, то получим 4, т. е.

```
'4' - 48 = 4
```

Но 48 - это ASCII-код символа '0', поэтому

```
'4' - '0' =4
```

Действительно, этот последний оператор был бы справедлив в любом коде, в котором используются последовательные числа для представления последовательных цифр. Поэтому если **num** - числовое значение, а **chn** - символ цифры, то

```
num = chn - '0';
```

Итак, мы используем этот метод для преобразования первой цифры в число. Теперь возьмем следующий элемент массива. Если он **'\0'**, то у нас была только одна цифра, и мы закончили работу. Предположим, однако, что этот элемент **'3'**. Превратим его в числовое значение 3. Но если оно равно 3, то 4 должно было быть числом 40, а оба числа вместе 43:

```
num = 10 * num + chn - '0';
```

Теперь просто продолжим этот процесс, умножая старое значение num на 10 каждый раз, когда мы находим следующую цифру. Наша функция будет использовать этот метод.

Вот определение функции **stoi( )**. Мы храним ее в том же файле, что и **getint( )**, так что она может использовать те же самые директивы #**define**.

```
/* превращает строку в целое число и сообщает о состоянии */
stoi(string, intptr)
char  string[ ];  /* строка, подлежащая преобразованию в целое*/
int *intptr;   /* значение целого */
{
int sign = 1; /* проверяет наличие знака +  или - */
int index = 0;
if(string[index] == '-' ||  string[index] == '+')
sign = (string[index ++] == '-') ? -1 : 1; /* установить знак */
*intptr = 0; /* начальное значение */
while(string[index] >= '0' && string[index] <= '9')
*intptr = 10 * (*intptr) + strmg[index++] - '0';
if(string[index] == '\0')
{
*intptr = sign * (*intptr);
 return(YESNUM); }
else /* найден символ, отличный от цифры, знака или ' \0'  */
return(NONUM);
}
```

Оператор **while** продолжает работу, преобразуя цифры в числа, пока не достигнет нецифрового символа. Если это символ **'\0'**, все прекрасно, потому что он означает конец строки. Любой другой нецифровой символ отсылает программу к **else** для сообщения об ошибке.

Стандартная библиотека языка Си содержит функцию **atoi( )** (перевод кода ASCII в целое число), очень похожую на **stoi( )**. Основная разница заключается в том, что **stoi( )** проверяет на нецифровые строки, а **atoi( )** использует **return** вместо указателя, для возврата числа, и пропускает пробел, как мы это делали в **getint()**. Можно было бы осуществить все проверки состояния в **getint( )** и использовать **atoi( )** вместо **stoi( )**, но мы полагаем, что было бы интереснее разработать нашу собственную версию.

Так ли уж правильны наши рассуждения? Давайте проверим нашу функцию на учебной программе:

```
/* проверка функции getint( )*/
#define STOP - 1
#define NONUM 1
#define YESNUM 0
main( )
{
int num, status;

printf(" Программа прекращает считывание чисел, если встречает EOF. \n" );
while((status = getint(&num)) != STOP)
if(status = = YESNUM)
printf(" Число %d принято. \n", num);
else
printf(" Это не целое число! Попытайтесь снова. \n");
printf("Это оно. \n");
}
```

Вот пример работы программы:

```
Программа прекращает считывание чисел, если встречает EOF.
100   -23
Число 100 принято.
Чмсло -23 принято.
+892.
Число 892 принято.
wonk
Это не целое число! Попытейтесь снова.
23skidoo
Это не целое число! Попытайтесь снова.
775
Число 775 принято.
Клавиша [control z] (посылает символ EOF в нашу программу).
Это оно.
```

Как видите, программа выполняется. Обратите внимание на то, как мы сумели организовать цикл для неограниченного считывания целых чисел до тех пор, пока не будет введен символ **EOF**. Это удобное свойство.

Есть ли здесь ошибки? По меньшей мере одна. Если непосредственно за числом следует символ **EOF** без разделяющего пробела или символа новой строки, ввод прекращается, и это число не принимается во внимание:

```
706 EOF /* 706 принято*/
706 EOF /* 706 не принято*/
```

Мы не хотели делать пример слишком сложным, поэтому допустили возможность этой ошибки. Дальнейшую разработку программы, как мы думаем, может осуществить сам читатель в качестве упражнения.

Теперь, когда у нас есть удобная функция для получения целых чисел, займемся новой задачей, в которой она будет использоваться.

СОРТИРОВКА ЧИСЕЛ

Одним из наиболее распространенных тестов для машин является сортировка. Мы хотим

разработать программу для сортировки целых чисел. Снова применим принцип черного ящика и подумаем в терминах ввода и вывода. Наш общий замысел, показанный на рис. 10.4, довольно прост.

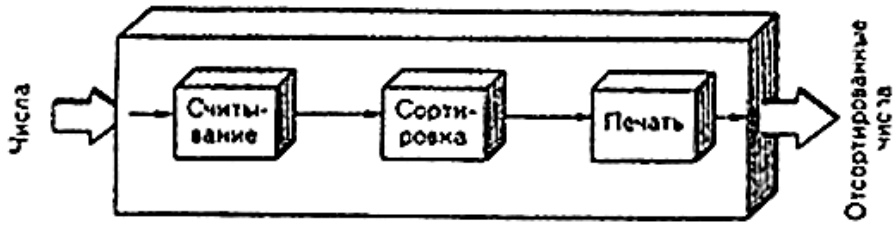


РИС. 10.4. Программа сортировки, рассматриваемая как черный яшик

На данном этапе программа еще недостаточно определена, чтобы ее кодировать. Следующий шаг заключается в определении главных задач, которые должна выполнить программа для достижения поставленных целей. Их три:
1. Считывание чисел.
2. Сортировка чисел.
3. Печать отсортированных чисел. На рис. 10.5 показано это разделение при движении от верхнего уровня конструкции вниз к более детальному.

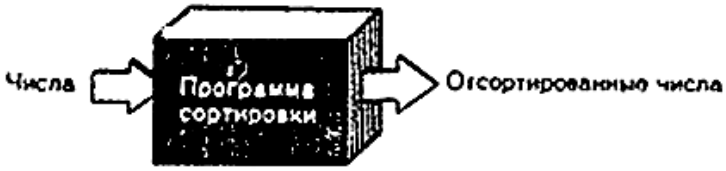


РИС. 10.5 Программа сортировки: содержание

Теперь мы имеем три черных ящика, каждый со своим входом и выходом. Можно передать каждую часть разным группам программистов, если мы уверены в том, что числа, выводимые блоком "считывание", представлены в той же самой форме, которую использует блок "сортировка" для ввода.

Как вы можете видеть, делается упор на модульность. Мы разделили исходную задачу на три более мелких, но лучше управляемых.

Что дальше? Теперь приложим наши усилия к каждому из трех модулей в отдельности, разделяя их на более простые элементы до тех пор, пока не достигнем момента, когда программа станет очевидной. Делая это, обратим внимание на такие важные проблемы, как выбор представления данных, локализация ошибок и передача информации. Вернемся к нашему примеру и займемся сначала задачей считывания.

Считывание числовых данных

Далее Содержание

Многие программы включают считывание чисел, поэтому идеи, которые мы развиваем здесь, будут полезны везде. Общий вид первой части программы ясен: использовать цикл для считывания чисел до тех пор, пока все числа не будут считаны. Но в этом есть нечто большее, чем вы можете себе представить!

Выбор представления данных

Далее Содержание

Как мы представляем группу чисел? Можно использовать группу переменных, по одной на каждое число. Об этом даже страшно подумать. Можно использовать массив, по одному элементу на каждое число. Это значительно лучше, поэтому давайте использовать массив.

Однако какого типа будет массив? Типа **int**? Типа **double**? Нам нужно знать, как такую программу можно будет применять. Предположим, что она должна работать с целыми числами. (А что если она должна применять и целые и нецелые числа? Это возможно, но потребуется работы больше, чем нам бы хотелось сейчас.) Будем использовать массив целых чисел для запоминания чисел, которые мы считываем.

Завершение ввода

Далее Содержание

Как программа "узнает", сколько ей нужно считать чисел? В гл. 8 мы обсудили несколько решений этой проблемы, большинство из которых были неудовлетворительны. Однако теперь, когда есть функция **getint( )**, у нас нет проблем. Вот один подход:

читаем число до тех пор пока не встретится символ **EOF**
заносим число в массив и
читаем следующее число, если массив не заполнен

Заметим, что здесь есть два разных условия, приводящих к завершению этой части программы: символ **EOF** и заполнение массива.

Дальнейшие рассуждения

Далее Содержание

Прежде чем реализовать все это на языке Си, нам нужно еще решить, что будем делать с проверкой ошибок ? Должны ли мы превратить эту часть программы в функцию?

Сначала мы решим, что делать, если пользователь вводит ошибочные данные, скажем букву вместо целого числа? Без функци **getint( )** мы полагались бы на "гипотезу идеального пользователя", согласно которой пользователь не делает ошибок при вводе. Однако мы считаем, что эту гипотезу нельзя применять ник одному пользователю, кроме себя. К счастью, можно использовать способность функции **getint()** сообщать о состоянии, кто поможет нам выйти из затруднительного положения.

Теперь займемся программированием, которое можно легко реализовать в **main( )**. Для соблюдения модульности следует использовать разные функции для каждой из трех основных частей программы, что мы как раз и сделаем. Входом для этой функции будут числа с клавиатуры или файл, а выходом - массив, содержащий неотсортированные числа. Было бы хорошо, если бы такая функция помогла основной программе узнать, сколько было считано чисел, поэтому предусмотрим это для выхода. Вконце концов нужно подумать и о пользователе. Мы заставим фикцию печатать сообщение, указывающее ее пределы, и, осуществлять эхо-печать входной информации.

Вызовем нашу функцию **getarray( )**, предназначенную для считывания. Мы определили эту функцию в терминах ввода и вывода и наметили в общих чертах схему на псевдокоде. Теперь давайте напишем функцию и покажем, как она включается в основную программу:

Сначала напишем **main( )**:

```
/* сортировка 1 */
#define MAXSIZE 100 /* предельное количество сортируемых целых чисел */
main( )
{
int numbers [MAXSIZE]; /* массив для ввода */
int size; /* количество вводимых чисел */
size = getarray(numbers, MAXSIZE); /* запись чисел в массив */
sort(numbers, size); /* сортировка массива */
print(numbers,/size); /* печать отсортированного массива */
}
```

Это общий вид программы. Функция **getarray()** размещает введенное числа в массиве **numbers** и выдает сообщение о том, сколько значений было считано; эта величина записывается в **size**. Затем идя функции **sort( )** и **print( )**, которые мы еще должны написать; они сортируют массив и печатают результаты. Включая в них **size**, мы облегчаем им работу и избавляем от необходимости выполнять самим подсчет. Мы также снабдили **getarray( )** переменной **MAXSIZE**, которая сообщает размер массива, необходимого для запоминания.

Теперь, когда мы добавили **size** к передаваемой информации, нужно модифицировать рисунок нашего черного ящика. См. рис. 10.6.

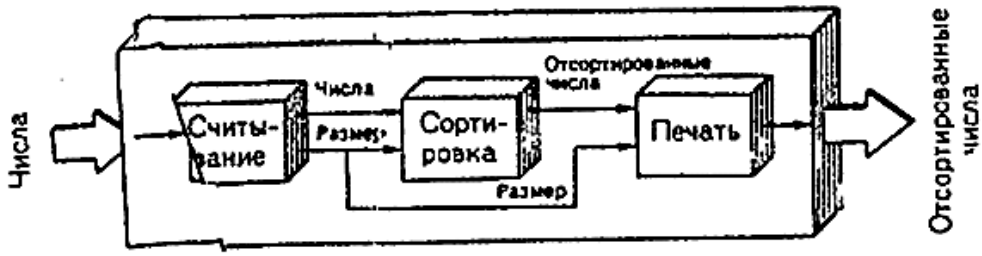


РИС. 10.6. Программ: сортировки, дополнительные детали.

Теперь рассмотрим функцию **getarray( )**:

```
/* getarray( ), использующая getint( ) */
#define STOP  -1 /* признак EOF */
#define NONUM 1 /* признак нецифровой строки */
#define YESNUM 0 /* признак строки цифр */
getarray(array, limit);
int array[ ], limit;
{
int num, status;
int index = 0; /* индекс массива */
printf(" Эта программа прекращает считывание чисел после %d значений. \n", limit);
printf(" или если введен символ EOF.\n");
while(index < limit && (status = getint(&num)) != STOP)
{  /* прекращает считывание после достижения limit или EOF */
if(status == YESNUM)
{ array[index++] = num;
printf(" число %d принято.\n", num);
} else if(status == NONUM)
printf(" Это было не целое число! Попытайтесь снова. \n");
```

```
else
printf(" Этого не может быть! Что-то неправильно. \n");
if(index == limit) /* сообщить, если массив заполнен */
printf(" Все %d элементов массива заполнены. \n ", limit);
return(index);
}
```

Это значительная часть программы, и у нас есть немало замечаний.

Разъяснения

Далее Содержание

Так как немного трудно вспомнить значение, скажем кода -1, мы используем мнемонические символические константы для представления кодов состояния.

Применяя эти коды, мы создаем **getarray( )** для управления каждым из возможных значений состояния. Состояние **STOP** вызывает прекращение цикла чтения, если **getint( )** находит на своем "пути" **EOF**. Состояние **YESNUM** говорит о запоминании числа в предлагаемом массиве. Кроме того, отсылается "эхо-число" пользователю, чтобы он знал, что оно принято. Состояние **NONUM** предписывает пользователю попытаться выполнить задачу еще раз. (Это признак "дружелюбия").

У нас есть еще оператор **else**. Единственный путь достижения этого оператора возможен, если **getint( )** возвращает значение, отличное от -1, 0 или 1. Однако это единственные значения, которые могут быть возвращены, поэтому else является, по-видимому бесполезным оператором. Почему он включен в программу? Мы вставили его как пример "защитного программирования", как способ защиты программы от будущей ошибки.

Когда-нибудь мы (или кто-нибудь еще), может быть, решим обратиться к функции **getint( )** и добавить в ее репертуар немного больше возможных значений состояния. Наиболее вероятно, что мы забудем (а они могут никогда не узнать), что **getarray( )** предполагает только три возможных ответа. Поэтому мы включаем это последнее **else**, чтобы "поймать" любые новые ответы, которые появятся, и значительно упростить будущую отладку.

Размер массива устанавливается в **main()**. Поэтому мы не задаем его, когда описываем аргумент-массив в **getarray()**. Мы ставим только квадратные скобки в оператор, чтобы указать, что аргумент является массивом.

```
int numbers [MAXSIZE]; /* размер задается в main */
int array[ ] /* нет определения размера в вызвавшей функции */
```

Использование массивов в функциях обсудим в гл. 12. Мы решили применить ключевое слово **return** для возврата числа прочитанных элементов. Таким образом, вызов нашей функции:

```
size = getarray(numbers);
```

присваивает значение переменной **size** и дает значения массиву **numbers**. Вы можете спросить, почему мы не использовали указатели в вызове

```
size = getarray (numbers);
```

ведь у нас функция изменяет значение чего-то (массива) в вызывающей программе? Ошибаетесь - мы использовали указатель! В языке Си имя массива является также указателем на первый элемент массива, т. е.

```
numbers == &numbers[0]
```

Когда функция **getarray()** создает массив **array**, то адрес элемента **array[0]** совпадает с адресом элемента **numbers[0]** и т. д. для всех других индексов. Поэтому все манипуляции, которые

выполняет **qetarray( )** с **array[ ]**, фактически выполняются с **numbers[ ]**. Мы будем более подробно говорить о связи между указателями и массивами в гл. 12. Теперь же нам нужно усвоить, что функция воздействует на массив в вызывающей программе, если мы используем массив в качестве аргумента функции.

В функциях, содержащих счетчики и пределы, таких как **getarray( )**, наиболее вероятным местом появления ошибок являются "граничные условия", где значения счетчиков достигают своих пределов. Мы собираемся прочитать максимальное количество чисел, указанное в **MAXSIZE**, или же мы намерены ограничиться одним? Хотим обратить внимание на детали, такие, как **++index** или **index++** и **<** или **<=**. Мы также должны помнить, что у массивов индексы начинаются с 0, а не с 1.

Проверьте вашу программу и посмотрите, работает ли она так, как должна. Самое простое - предположить, что **limit** равен 1 и пройти по программе шаг за шагом.

Обычно наиболее трудной частью программы является обеспечение ее взаимодействия с пользователем удобным и надежным способом. Это относится и к нашей программе. Теперь, когда мы справились с **getarray( )**, находим, что функция **sort( )** должна быть проще и **print( )** - тоже. Теперь перейдем к функции **sort( )**.

Сортировка данных

Далее Содержание

Рассмотрим еще раз функцию **main( )**:


```
main(  )
{
int numbers[MAXSIZE]; /* массив для ввода */
int size;     /* количество введенных элементов */
size = getarray(numbers, MAXSIZE); /* помещает ввод в массив */
sort(numbers, size); /* сортировка массива */
printf(numbers, size); /* печать отсортированного массива       */
}
```

Мы видим, что функция **sort()** имеет на входе массив целых чисел, предназначенных для сортировки, и счетчик количества элементов, подлежащих сортировке. На выходе получается массив, содержащий отсортированные числа. Мы все еще не решили, как выполнять сортировку, поэтому мы должны дополнительно уточнить это описание.

Очевидно, в начале трудно определить направление сортировки. Собираемся ли мы вести сортировку от большего к меньшему, или наоборот? Мы свободны в выборе и допустим, что хотим сортировать от большего к меньшему. (Можно сделать программу, работающую любым из этих методов, но тогда нам нужно придумать способ сообщить ей о своем выборе.)

Рассмотрим теперь метод, который будем использовать для сортировки. В настоящее время разработано много алгоритмов сортировки; возьмем один из самых простых.

Вот наш план на псевдокоде:
от **n** = первому элементу до **n** = ближайшему - **к**- последнему элементу находим самое большое из оставшихся чисел и помещаем его в **n**-ю позицию.

Он выполняется примерно так. Сначала пусть **n = 1**. Мы просматриваем весь массив, находим самое большое число и помещаем его в первый элемент. Затем **n = 2**, и мы опять просматриваем весь массив, кроме первого элемента, находим самое большое из оставшихся чисел и помещаем его во второй элемент. Продолжаем этот процесс до тех пор, пока не достигнем ближайшего - **к** - последнему элементу. Теперь осталось только два элемента. Мы сравниваем эти числа и помещаем большее в элемент, ближайший - **к** - последнему. Оставшееся самое меньшее из всех чисел помещаем в последний элемент.

Это выглядит очень похоже на задачу с циклом **for**, но мы все же должны описать процесс "найти и поместить" более детально. Как сделать так, чтобы мы находили каждый раз самое большое из оставшихся чисел? Вот один способ. Сравните первый и второй элементы оставшегося массива. Если второй больше, поменяйте их местами. Теперь сравните первый элемент с третьим. Если третий больше, поменяйте местами эти два. Каждый раз больший элемент перемещается вверх. Продолжаем таким образом, пока не сравним первый элемент с последним. Если мы дошли до конца, самое большое число теперь будет в первом элементе оставшегося массива. По существу мы имеем отсортированный массив для первого элемента, но остаток массива находится в беспорядке. На псевдокоде это можно выразить так:

для **n** = от второго до последнего элемента сравниваем **n**-й элемент с первым; если **n**-й больше, меняем их местами.

Это напоминает еще один цикл **for**. Его следует вставить в первый цикл **for**. Внешний цикл показывает, какой элемент массива должен быть заполнен, а внутренний цикл находит значение, которое следует поместить туда. Записывая обе части на псевдокоде и переводя их на язык Си, мы получаем следующую функцию:

```
/* сортировка массива целых чисел в порядке убывания */
sortarray(array, limit)
int array[ ], limit;
{
int top, search;
for(top = 0; top < limit - 1; top++)
for(search = top + 1; search < limit; search++)
if(array [search] > array[top]
interchange(&array[search], &array[top] );
}
```

Мы помним, что первый элемент имеет индекс **0**. Кроме того, еще в гл. 9 была создана функция обмена, поэтому мы использовали ее здесь. Так как функция **interchange** "работает" с двумя элементами массива, а не со всем массивом, мы должны использовать адреса только двух интересующих нас элементов. (В то время как имя array является указателем на весь массив, нам нужно применить операцию **&**, чтобы указывать на отдельные элементы.)

Мы использовали **top** в качестве индекса для элемента массива, который следует заполнить, так как он является вершиной неотсортированной части массива. Индекс **search** перемещает по массиву в порядке убывания текущий элемент. Большинство текстов использует обозначения **i** и **j** для этих индексов, однако это осложняет ситуацию, если нужно посмотреть, что происходит.

Этот алгоритм иногда называют "пузырьковой сортировкой", так как самое большое значение медленно поднимется вверх. Именно теперь мы должны написать функцию **print( )**.

Печать данных Далее Содержание

Эта программа достаточно проста:

```
/* печать массива */
print(array, limit)
int array[ ], limit;
{
int index;
for(index = 0; index <= limit; index ++)
printf(" %d\n", array[index]);
}
```

Если мы хотим сделать что-нибудь другое, например печатать по строкам, а не в столбец, можно всегда вернуться и изменить эту функцию, оставив другие функции неизменными. Аналогично, если мы нашли алгоритм сортировки, который нам больше нравится, можно. заменить этот модули программы. Это один из приятных моментов модульной программы.

Результаты

Далее Содержание

Давайте скомпилируем и протестируем нашу программу сортировки. Чтобы упростить проверку граничных условий, временно изменим **MAXSIZE** на **5**.

В нашем первом тесте будем снабжать программу числами до тех пор, пока она не откажется их принимать.

Эта программа прекращает считывание чисел после **5** значений, или если встретился символ **EOF**.

```
12 34 54 23 67
Все 5 элементов массива заполнены.
67
54
34
23
12
```

Программа считала 5 чисел и отсортировала их. Теперь посмотрим, как будет выглядеть результат, если она остановится, встретив символ **EOF**.

Эта программа прекращает считывание чисел после **5** значений, или если встретился символ **EOF**.

```
456 928
-23  +16
Клавиша [control -z] (передает EOF в нашу систему)
928
456
16
-23
```

Быстрее чем вы сможете сказать "экология это наука о домашнем хозяйстве", целый огромный массив отсортирован.

Успех! Это было не просто, но не невозможно. Разделив задачу на небольшие части и продумав, какой информацией должна обмениваться каждая из них, мы свели задачу к частям, поддающимся управлению. Кроме того, отдельные модули, которые мы создали, можно использовать как части подобных программ.

Этим завершаются наши примеры в данной главе. Давайте теперь вернемся немного назад и сделаем обзор главы.

ОБЗОР

Далее Содержание

Чего же мы достигли? Если посмотреть с практической стороны, то мы разработали генератор случайных чисел и программу сортировки целых чисел. При этом создали функцию **getint( )**, которую можно использовать в других программах. Если посмотреть с педагогической точки зрения, то мы проиллюстрировали несколько основных принципов и концепций, полезных при создании программ.

Следует обратить внимание на самый существенный момент:
программы нужно *проектировать*, а не создавать их методом проб и ошибок. Вы должны внимательно подумать о форме и содержании ввода и вывода для программы. Необходимо разделить программу на хорошо определенные задачи, затем раздельно запрограммировать, принимая во внимание взаимодействие их друг с другом. Идея заключается в достижении модульности. Если необходимо, разбивайте модули на еще более мелкие модули. Используйте функции для повышения степени модульности и простоты программы.
При проектировании программы, попытайтесь предвидеть, что может идти неправильно и программируйте, исходя из этого. Используйте локализацию ошибок, чтобы контролировать действия в местах потенциальных затруднений, или по крайней мере предупреждать пользователя, что может возникнуть осложнение. Гораздо лучше дать пользователю еще одну возможность ввести данные, чем продолжать выполнять программу и прийти к аварийной ситуации.

Если создается функция, сначала определите, как она будет взаимодействовать с вызывающей программой. Решите также, какая информация будет входить в нее, а какая выходить. Какими должны быть аргументы? Хотите ли вы использовать указатели, или возврат, или то и другое? Если вы примете во внимание все эти параметры, то можете обратить внимание на работу самой функции.

Используйте эти идеи, и ваша программа будет более надежной и менее подверженной аварийным ситуациям. Вы получаете тело функции, которое можете применять в других программах. Программирование в таком случае потребует меньше времени. Вообще все это похоже на хороший рецепт здорового программирования.

Не забывайте о классах памяти. Переменные можно определять вне функции; в этом случае их называют внешними (или глобальными) и они доступны более чем для одной функции. Переменные, определенные внутри функции, являются локальными для нее и не известны другим функциям. Если можно, используйте автоматическую разновидность локальных переменных. Они охраняют переменные одной функции от воздействия других функций.

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Как представлять функцию: черный ящик с информационным потоком.
Что такое "проверка ошибок" и почему эта процедура хороша.
Алгоритм сортировки.
Как заставить функцию изменять массив: **function(array)**.
Как преобразовать строку цифр в число.
Классы памяти: **auto**, **extern**, **static** и **register**.
Область действия каждого класса памяти.
Какой класс памяти использовать: обычно **auto**.

ВОПРОСЫ И ОТВЕТЫ

Далее Содержание

Вопросы

1. Что может сделать наш алгоритм сортировки неэффективным?
2. Как следует изменить нашу программу сортировки, чтобы она сортировала и по рядке возрастания, а не убывания?
3. Измените функцию **print( )** таким образом, чтобы она печатала по 5 чисел в строке.
4. Как следует изменить функцию **stoi( )**, чтобы обрабатывать строки, представляющие восьмеричные числа?
5. Какие функции "знают" каждую переменную из описанных ниже? Есть ли здесь какие-нибудь ошибки?

```
/* файл1 */
int  daisy;
main( )
{
int lily;
}
petal( ) {
extern int daisy, lily;
}
/* файл2 */
static int lily;
int rose;
stem( ) {
int rose;
}
root( )
{
extern int daisy;
}
```

Ответы

1. Предположим, что вы сортируете 20 чисел. Программа производит 19 сравнений, чтобы найти одно самое большое число. Затем делается 18 сравнений, чтобы найти следующее самое большое. Вся информация, полученная во время первого поиска "забывается" за исключением самого большого числа, поставленного на первое место. Второе самое большое число можно временно поместить на место с номером 1, а затем при сортировке опустить вниз. Много сравнений, выполнявшихся в первый раз, повторяется второй, третий раз и т. д.

2. Замените **array[search] > array[top]** на **array[search] < array[top]**

3.

```
/* печать массива */
print(array, limit)
int array[ ], limit);
{
int index;
for(index = 0; index < limit; index++)
{ printf("%10d", array[index]);
if(index % 5 == 4) primf("\n" );
} printf("\n");
}
```

4. Во-первых, обеспечьте, чтобы разрешенные символы были только цифрами от 0 до 7. Во-вторых, умножайте на 8 вместо 10 каждый раз, когда обнаружите новую цифру.

5. **daisy** известна функции **main( )** по умолчанию и функциям **petal( )** и **roo1( )** благодаря **extern**-описанию. Она не известна функции **stem( )**, потому что они находятся в разных файлах.
Первая **lily** локальна для **main**: ссылка на **lily** в **petal( )** является ошибочной, потому что в каждом из этих файлов нет внешней **lily**.
Есть внешняя статическая **lity**, но она известна только функциям второго файла. Первая, внешняя **rose**, известна функции **root( )**, а функция **stem( )** отменила ее своей собственной локальной **rose**.

УПРАЖНЕНИЯ Содержание

1. Некоторые пользователи, возможно испугаются, если их попросить ввести символ **EOF**.
   а. Модифицируйте **getarray( )** и вызываемые ею функции так, чтобы использовать символ # вместо **EOF**.
   б. Модифицируйте затем их так, чтобы можно было использовать либо **EOF**, либо #.

2. Создайте программу, которая сортирует числа типа **float**.

3. Создайте программу, превращающую смешанный текст из прописных и строчных букв в текст, состоящий только из прописных букв.

4. Создайте программу, которая удваивает пробелы в тексте с одиночными пробелами.

---

11. Препроцессор языка Си

ДИРЕКТИВЫ ПРЕПРОЦЕССОРА
СИМВОЛЬНЫЕ КОНСТАНТЫ

МАКРООПРЕДЕЛЕНИЯ И "МАКРОФУНКЦИИ"
ПОБОЧНЫЕ ЭФФЕКТЫ МАКРООПРЕДЕЛЕНИИ
ВКЛЮЧЕНИЕ ФАЙЛОВ
УСЛОВНАЯ КОМПИЛЯЦИШЯ

ДИРЕКТИВЫ ПРЕПРОЦЕССОРА
#define, #include, #undef, #if, #ifdef, #ifndef, #else, #endif

Язык Си был разработан в помощь работающим программистам, а им нравится его препроцессор. Этот полезный помощник просматривает программу до компилятора (отсюда и термин "препроцессор") и заменяет символические аббревиатуры в программе на соответствующие директивы. Он отыскивает другие файлы, необходимые вам, и может также изменить условия компиляции. Однако эти слова не отражают истинную пользу и значение препроцессора, поэтому обратимся к примерам. Конечно, до сих пор мы снабжали все примеры директивами **#define** и **#include**, но теперь мы можем подытожить все, что изучили, и развить тему дальше.

СИМВОЛИЧЕСКИЕ КОНСТАНТЫ: #define

Далее Содержание

Директива **#define**, подобно всем директивам препроцессора, начинается с символа # в самой левой позиции. Она может появиться в любом месте исходного файла, а даваемое ею определение имеет силу от места появления до конца файла. Мы активно используем эту директиву для определения символических констант в наших программах, однако она имеет более широкое применение; что мы и покажем дальше. Вот пример, иллюстрирующий некоторые возможности и свойства директивы **#define**:

```
/* простые примеры директивы препроцессора */
#define TWO 2 /* по желанию можно использовать комментарии */
#define MSG "Старый серый кот поет веселую песню."
/* обратная косая черта продолжает определение на следующую строку */
#define FOUR TWO *TWO
#define PX printf("X равен %d.\n", x)
#define FMT "X равен %d.\n"
main( )
{
int x = TWO;
PX;
x = FOUR;
printf(FMT, x);
printf( "%s\n", MSG);
printf("FTWO: MSG\n");
}
```

Каждая строка состоит из трех частей. Первой стоит директива **#define**. Далее идет выбранная нами аббревиатура, известная у программистов как "макроопределение". Макроопределение не должно содержать внутри себя пробелы. И наконец, идет строка (называемая *"строкой замещения"*), которую представляет макроопределение. Когда препроцессор находит в программе одно из ваших макроопределений, он почти всегда заменяет его строкой замещения. (Есть одно исключение, которое мы вам сейчас покажем.) Этот процесс прохождения от макроопределения до заключительной строки замещения называется "макрорасширением". Заметим, что при стандартной форме записи на языке Си можно вставлять комментарии; они будут игнорироваться препроцессором. Кроме того, большинство систем разрешает использовать обратную косую черту (**'\'**) для расширения определения более чем на одну строку.
"Запустим" наш пример, и посмотрим за его выполнением.

```
X равен 2.
X равен 4.
Старый серый кот поет веселую песню. TWO: MSG
```

Вот что произошло. Оператор **int x = TWO;** превращается **в int x = 2;**.

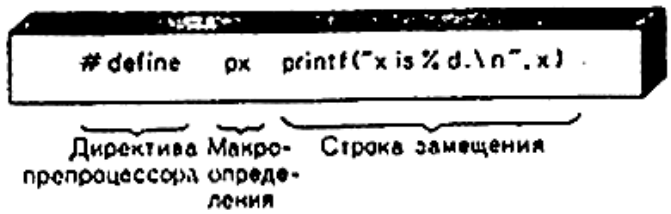


РИС. 11.1. Части макроопределения.

т. е. слово **TWO** заменилось цифрой **2**. Затем оператор **PX;** превращается в **printf("X равно %d. \n", x);** поскольку сделана полная замена. Это новшество, так как до сих пор мы использовали макроопределения только для представления констант. Теперь же мы видим, что макроопределение может представлять любую строку, даже целое выражение на языке Си. Заметим, однако, что это константная строка; **PX** напечатает только переменную, названную **x**.
Следующая строка также представляет что-то новое. Вы можете подумать, что **FOUR** заменяется на **4**, но на самом деле выполняется следующее:

```
x = FOUR;
```

превращается в **x = TWO *TWO;** превращается в **x = 2*2;** и на этом все заканчивается. Фактическое умножение имеет место не во время работы препроцессора и не при компиляции, а всегда без исключения при работе программы. Препроцессор не выполняет вычислений; он только очень точно делает предложенные подстановки.

Заметим, что макроопределение может включать другие определения. (Некоторые компиляторы не поддерживают это свойство "вложения".)
В следующей строке **printf(FMT, x);** превращается в **printf(" X равно %d.\n", x)** когда **FMT** заменяется соответствующей строкой. Этот подход может оказаться очень удобным, если у вас есть длинная строка, которую вы используете несколько раз.
В следующей строке программы **MSG** заменяется соответствующей строкой. Кавычки делают замещающую строку константой символьной строки; поскольку программа получает ее содержимое, эта строка будет запоминаться в массиве, заканчивающемся нуль-символом. Так, **#define HAL 'Z'** определяет символьную константу, а **#define HAP "Z"** определяет символьную строку: **Z\0**.

Обычно препроцессор, встречая одно из ваших макроопределений в программе, очень точно заменяет их эквивалентной строкой замещения. Если эта строка также содержит макроопределения, они тоже замешаются. Единственным исключением при замене является макроопределение, находящееся внутри двойных кавычек. Поэтому **printf("TWO: MSG");** печатает буквально **TWO: MSG** вместо печати следующей строки:

```
2: "Старый серый кот поет веселую песню."
```

Если вам нужно напечатать эту строку, можно использовать оператор

```
printf("%d: %s\n", TWO, MSG);
```

потому что здесь макроопределения находятся вне кавычек.

Когда следует использовать символические константы? Вероятно, вы должны применять их для большинства чисел. Если число является константой, используемой в вычислениях, то символическое имя делает яснее ее смысл. Если число - размер массива, то символическое имя упрощает изменение вашей программы при работе с большим массивом. Если число является системным кодом, скажем для символа **EOF**, то символическое представление делает вашу программу более переносимой; изменяется только определение **EOF**. Мнемоническое значение, легкость изменения, переносимость: все это делает символические константы заслуживающими

внимания. Легко ли этого достичь? Рискнем и рассмотрим простую функцию, т. е. макроопределение с аргументами.

ИСПОЛЬЗОВАНИЕ АРГУМЕНТОВ С #define

Далее Содержание

Макроопределение с аргументами очень похоже на функцию, поскольку аргументы его заключены в скобки. Ниже приведено несколько примеров, иллюстрирующих, как определяется и используется такая "макрофункция". В некоторых примерах к тому же указаны возможные ловушки, поэтому читайте их внимательно.


```
/* макроопределение с аргументами */
#define SQUARE (x) x*x
#define PR(x) printf("x равен %d.\n" ,x)
main( )
{
int x = 4;
int z;

z = SQUARE(x);
PR(z);
z = SQUARE(2);
PR(z);
PR(SQUARE(x));
PR(SQUARE(x + 2));
PR(100/SQUARE(2));
PR(SQUAREC(++x));
}
```

Всюду, где в вашей программе появляется макроопределение **SQUARE(x)**, оно заменяется на **x*x**. В отличие от наших прежних примеров при использовании этого макроопределения мы можем совершенно свободно применять символы, отличные от **x**. В макроопределении **'x'** замещается символом, использованным в макровызове программы. Поэтому макроопределение **SQUARE(2)** замещается на **2*2**. Таким образом, **x** на самом деле действует как аргумент. Однако, как вы вскоре увидите, аргумент макроопределения не "работает" точно так же, как аргумент функции. Вот результаты выполнения программы. Обратите внимание, что некоторые ответы отличаются от тех, которые вы могли бы ожидать.

```
z равно 16.
z равно 4.
```

```
SQUARE(x) равно 16.
SQUARE(x + 2) равно 14.
100/SQUARE(2) равно 100.
SQUAREC(++ x) равно 30.
```

Первые две строки предсказуемы. Заметим, однако, что даже внутри двойных кавычек в определении **PR** переменная замещается соответствующим аргументом. Все аргументы в этом определении замещаются.
Третья строка представляет интерес:

```
PR(SQUARE(x));
```

она становится следующей строкой:

```
printf("SQUARE(x) равно %d.\n", SQUARE(x));
```

после первого этапа макрорасширения. Второе **SQUARE(x)** расширяется, превращаясь в **x*x**, а первое остается без изменения, потому что теперь оно находится внутри двойных кавычек в операторе программы, и таким образом защищено от дальнейшего расширения. Окончательно строка программы содержит

```
printf(" SQUARE(x) равно %d.\n", x*x);
```

и выводит на печать

```
SQUARE(x) равно 16.
```

при работе программы.

Давайте еще раз проверим то, что заключено в двойные кавычки. Если ваше макроопределение включает аргумент с двойными кавычками, то аргумент будет замещаться строкой из макровызова. Но после этого он в дальнейшем не расширяется, даже если строка является еще одним макроопределением. В нашем примере переменная **x** стала макроопределением **SQUARE(x)** и осталась им.

Теперь мы добрались до несколько специфических результатов. Вспомним, что **x** имеет значение **b**. Это позволяет предположить, что **SQUARE(x + 2)** будет равно **6*6** или **36**. Но напечатанный результат говорит, что получается число **14**, которое, несомненно, никак не похоже на квадрат целого числа! Причина такого вводящего в заблуждение результата проста, и мы уже об этом говорили: препроцессор не делает вычислений, он только замещает строку. Всюду, где наше определение указывает на **x**, препроцессор подставит строку **x + 2**. Таким образом, **x*x** становится **x + 2*x + 2.**
Единственное умножение здесь **2*x**. Если **x** равно **4**, то получается следующее значение этого выражения:

```
4+2*4+2=4+8+2= 14.
```

Этот пример точно показывает очень важное отличие между вызовом функции и макровызовом. Вызов функции передает *значение* аргумента в функцию во время выполнения программы. Макровызов передает строку аргументов в программу до ее компиляции; это другой процесс, происходящий в другое время.

Можно ли ваше определение переделать так, чтобы **SQUARE(x + 2)** было равно **36**? Конечно. Нам просто нужно больше скобок:

```
#define SQUARE(x) (x)*(x)
```

Тогда **SQUARE(x + 2)** становится **(x + 2)*(x + 2)**, и мы получаем наше желанное умножение, так как перенесли скобки в строку замещения.

Однако это не решает всех наших проблем. Рассмотрим случаи, которые приводят к следующей строке на выходе: **100/SQUARE(2)** превращается в **100/2*2** .

Вычисления следует вести слева направо, т. е. **(100/2)*2** или **50*2** или **100**.

Эту путаницу можно исправить, определив **SQUARE(x)** следующим образом:

```
#define SQUARE(x) (x*x)
```

Это даст **100/(2 *2)**, что в конечном счете эквивалентно **100/4** или **25**.

Чтобы выполнить два последних примера, нам необходимо определение

```
#define SQUARE(x) ((x)*(x))
```

Это урок использования необходимого количества скобок для гарантии, что операции и соединения выполняются в правильном порядке.

Даже эти предосторожности не спасают последний пример от беды: **SQUARE(++ x)** превращается в **++x * ++x** и **x** увеличивается дважды - один раз до умножения и один раз после: **++x * ++x = 5*6 = 30**.

(Так как порядок выполнения операций не установлен, то некоторые компиляторы превратят это в **6*5**, но конечный результат будет тем же самым.)

Единственное лекарство в этом случае - не использовать **++x** в качестве аргумента для макроопределения. Заметим, что **++x** обычно *работает* как аргумент *функции*, так как ему присваивается значение **5**, и затем это значение **5** передается функции.

МАКРООПРЕДЕЛЕНИЕ ИЛИ ФУНКЦИЯ?

Далее Содержание

Многие задачи можно решать, используя макроопределение с аргументами или функцию. Что из них следует применять нам? На этот счет нет строгих правил, но есть некоторые соображения.

Макроопределения должны использоваться скорее как хитрости, а не как обычные функции: они могут иметь нежелательные побочные эффекты, если вы будете неосторожны. Некоторые компиляторы ограничивают макроопределения одной строкой, и, по-видимому, лучше соблюдать такое ограничение, даже если ваш компилятор этого не делает.

Выбор макроопределения приводит к увеличению объема памяти, а выбор функции - к увеличению времени работы программы. Так что думайте, что выбрать! Макроопределение создает "строчный" код, т. е. вы получаете оператор в программе. Если макроопределение применить 20 раз, то в вашу программу вставится 20 строк кода. Если вы используете функцию 20 раз, у вас будет только одна копия операторов функции, поэтому получается меньший объем памяти. Однако управление программой следует передать туда, где находится функция, а затем вернуться в вызывающую программу, а на это потребуется больше времени, чем при работе со "строчными" кодами.

Преимущество макроопределений заключается в том, что при их использовании вам не нужно беспокоиться о типах переменных (макроопределения имеют дело с символьными строками, а не с фактическими значениями). Так наше макроопределение **SQUARE(x)** можно использовать одинаково хорошо с переменными типа **int** или **float**.

Обычно программисты используют макроопределения для простых действии, подобных следующим:

```
#define MAX(X,Y)     ( (X) > (Y) ? (X) : (Y))
```

```
#define ABS(X)       ( (X) < 0 ? -(X) : (X))
#define ISSIGN(X)    ( (X) == '+' || (X) == '-' ? 1 : 0)
```

(Последнее макроопределение имеет значение **1** (истинно), если **X** является символом алгебраического знака.) Отметим следующие моменты:

1. В макроопределении нет пробелов, но они могут появиться в замещающей строке. Препроцессор "полагает", что макроопределение заканчивается на первом пробеле, поэтому все, что стоит после пробела, остается в замещающей строке.

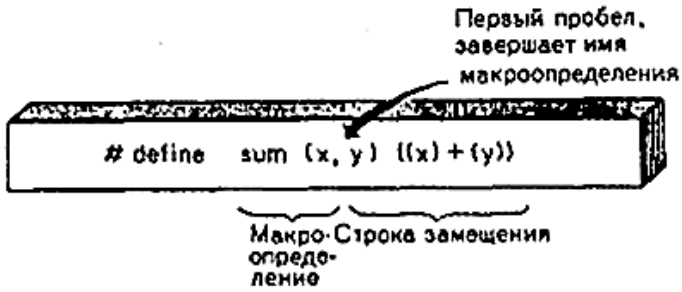


РИС. 11.2. Ошибочный пробел и макроопределении.

2. Используйте круглые скобки для каждого аргумента и всего определения. Это является гарантией того, что элементы будут сгруппированы надлежащим образом в выражении, подобном **forks = 2*MAX(guests + 3, last);**

3. Для имен макрофункций следует использовать прописные буквы. Это соглашение не распространяется так широко, как соглашение об использовании прописных букв для макроконстант. Применение их предостережет вас от возможных побочных эффектов макроопределений.

Предположим, что вы разработали несколько макрофункций по своему усмотрению. Должны ли вы их переопределять каждый раз, когда пишете новую программу? Нет, если вы вспомните о директиве **#include.** Теперь рассмотрим ее.

ВКЛЮЧЕНИЕ ФАЙЛА: #include

Далее Содержание

Когда препроцессор "распознает" директиву **#include**, он ищет следующее за ней имя файла и включает его в текущий файл. Директива выдается в двух видах:

```
#include <stdio.h>      имя файла в угловых скобках
#include "mystuff.h"        имя файла в двойных кавычках
```

В операционной системе UNIX угловые скобки сообщают препроцессору, что файл следует искать в одном или нескольких стандартных системных каталогах. Кавычки говорят ему, что сначала нужно смотреть в вашем каталоге (или в каком-то другом, если вы определяете его именем файла), а затем искать в "стандартных" местах.

```
#include <stdio.h>            ищет в системном каталоге
#include "hot.h"             ищет в вашем текущем рабочем каталоге
#include "/usr/biif/p.h"      ищет в каталоге /usr/biff
```

В типичной микропроцессорной системе эти две формы являются синонимами, и препроцессор ведет поиск на указанном диске.

```
#include "stdio.h"            ищет на стандартном диске
#include <stdio.h>            ищет на стандартном диске
#include "a : stdio.h"          ищет на диске a
```

Зачем включают файлы? Потому что они несут нужную вам информацию. Файл **stdio.h**, например, обычно содержит определения **EOF**, **getchar( )** и **putchar( )**. Две последние Функции определены как макрофункции.

По соглашению суффикс **.h** используется для *"заголовочных"* файлов, т. е. файлов с информацией, которая располагается в начале вашей программы. Заголовочные файлы обычно состоят из операторов препроцессора. Некоторые файлы, подобно **stdio.h**, включены в систему, но вы можете создать и свой собственный.

Заголовочные файлы: Далее Содержание

Пример:
Предположим, например, что вам правится использовать булевы переменные, т. с. вместо того чтобы иметь **1** как "истину" и **0** как "ложь", хотели бы использовать слова **TRUE** и **FALSE**. Можно создать файл, названный, скажем, **bool.h**, который содержал бы эти определения:

```
/* файл bool.h */
#define BOOL int;
#define TRUE 1
#define FALSE 0
```

Вот пример программы, использующей этот заголовок:

```
/* считает пустые символы */
#include <stdio.h>
#include "bool.h"
main( )
{
int ch;
int count = 0;
BOOL whitesp( );
while((ch = getchar( )) != EOF)
    if(whitesp(ch)) count++;
printf(" Имеется %d пустых символов. \n", count);
}
BOOL whitesp(c)
char c;
if(c == '   ' || c == '\n' || c == '\t')
return(TRUE);
else
return(FALSE);
}
```

Замечания по программе Далее Содержание

1. Если две функции в этой программе (**'main( )'** и **'whitesp( )'**) следовало скомпилировать раздельно, то нужно было бы использовать директиву **#include "bool.h"** с каждой из них.

2. Выражение **if(whitesp(ch))** аналогично **if(whitesp(ch)) == TRUE**, так как сама функция **whitesp(ch)** имеет значение **TRUE** или **FALSE**.

3. Мы не создали новый тип **BOOL**, так как **BOOL** уже имеет тип **int**. Цель наименования функции **BOOL** - напомнить пользователю, что функция используется для логических вычислений (в противоположность арифметическим).

4. Использование функции в предусматриваемых логических сравнениях может сделать программу яснее. Она может также сэкономить наш труд, если сравнение встречается в программе более одного раза.

5. Мы могли бы использовать макроопределение вместо функции для задания **whitesp( ).**

Многие программисты разрабатывают свои стандартные заголовочные файлы, чтобы использовать их в программах. Некоторые файлы могут создаваться для специальных целей, другие можно использовать почти в каждой программе. Так как включенные файлы могут содержать директивы **#include**, можно, если хотите, создать сжатый, хорошо организованный заголовочный файл. Рассмотрим этот пример:

```
/*заголовочный файл mystuff.h*/
#include
#include "bool.h"
#include "funct.h"
#define YES 1
#define NO 0
```

Во-первых, мы хотим напомнить вам, что препроцессор языка Си распознает комментарии, помеченные **/* и */**, поэтому мы можем включать комментарии в эти файлы.
Во-вторых, мы включили три файла. По-видимому, третий содержит некоторые макрофункции, которые используются часто.
В-третьих, мы определили **YES** как **1**, тогда как в файле **bool.h** мы определили **TRUE** как **1**. Но здесь нет конфликта, и мы можем использовать слова **YES** и **TRUE** в одной и той же программе. Каждое из них будет заменяться на **1**. Может возникнуть конфликт, если мы добавим в файл строку

```
#define TRUE 2
```

Второе определение вытеснило бы первое, и некоторые препроцессоры предупреждали бы вас, что **TRUE** переопределено.

Директива **#include** не ограничивается заголовочными файлами. Если вы записали нужную функцию в файл **sort.c**, то можно использовать

```
#include "sort.c"
```

чтобы скомпилировать его совместно с вашей текущей программой.

Директивы **#include** и **#define** являются наиболее активно используемыми средствами препроцессора языка Си. Рассмотрим более сжато другие его директивы.

ДРУГИЕ ДИРЕКТИВЫ: #undef, #if, #ifdef, #ifndef, #else И #endif

Далее Содержание

Эти директивы обычно используются при программировании больших модулей. Они позволяют приостановить действие более ранних определений и создать файлы, каждый из которых можно компилировать по-разному.

Директива **#undef** отменяет самое последнее определение поименованного макроопределения.

```
#define BIG 3
#define HUGE 5
#undef BIG              /* BIG теперь не определен */
#define HUGE 10        /* HUGE переопределен как 10 */
#undef HUGE           /* HUGE снова равен 5*/
#undef HUGE           /* HUGE теперь не определен */
```

Очевидно (мы надеемся), вы не будете компилировать файл, как в этом примере. Предположим, что у вас есть большой стандартный файл, определенный директивой **#include**, который вы хотели бы использовать, но для этого некоторые из его определений для одной из функций программы нужно будет временно изменить, Вместо того чтобы иметь дело с этим файлом, вы можете просто

включить его, а затем окружить такую выделяющуюся функцию соответствующими директивами **#define** и **#undef**.

Или, предположим, что вы работаете с большой системой программ. Вы хотите задать макроопределение, но не уверены, использует ли ваше определение другие определения откуда-нибудь из другого места системы. В этом случае просто отмените ваше макроопределение в том месте, где оно вам больше не нужно, а оригинал этого макроопределения, если он есть, будет по-прежнему оставаться в силе для остальной части системы.

Другие упомянутые нами директивы позволяют выполнять условную компиляцию. Вот пример:

```
#ifdef MAVIS
#include " horse.h" /* выполнится, если MAVIS определен */
#define STABLES 5
#else
#include "cow.h" /*выполнится, если MAVIS не определен */
#define STABLES 15
#endif
```

Директива **#ifdef** сообщает, что если последующий идентификатор (**MAVIS**) определяется препроцессором, то выполняются все последующие директивы вплоть до первого появления **#else** или **#endif**. Когда в программе есть **#else**, то программа от **#else** до **#endif** будет выполняться, если идентификатор не определен.

Такая структура очень напоминает конструкцию **if-else** языка Си. Основная разница заключается в том, что препроцессор не распознает фигурные скобки **{}**, отмечающие блок, потому что он использует директивы **#else** (если есть) и **#endif** (которая должна быть) для пометки блоков директив.

Эти условные конструкции могут быть вложенными.

Директивы **#ifdef** и **#if** можно использовать с **#else** и **#endif** таким же образом. Директива **#ifndef** опрашивает, является ли последующий идентификатор *неопределенным*; эта директива противоположна **#ifdef**. Директива **#if** больше похожа на обычный оператор **if** языка Си. За ней следует константное выражение, которое считается истинным, если оно не равно нулю:

```
#if SYS == "IBM"
#include "ibm.h"
#endif
```

Одна из целей использования "условной компиляции" - сделать программу более мобильной. Изменяя несколько ключевых определений в начале файла, вы можете устанавливать различные значения и включать различные файлы для разных систем.

Эти краткие примеры иллюстрируют удивительную способность языка Си изощренно и строго управлять программами.

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Как определять символьные константы директивой **#define**: **#define FINGERS 10**
Как включать другие файлы: **#include "albanian.h"**
Как определить макрофункцию: **#define NEG(X) (-(X))**
Когда использовать символические константы: часто.
Когда использовать макрофункции: иногда.
Опасность применения макрофункций: побочные эффекты.

Вопросы

1. Ниже приведены группы операторов, содержащих по одному и более макроопре делений, за которыми следуют строки исходных кодов, использующих эти макро определения. Какой результат получается в каждом случае? Правилен ли он?

а. #define FPM 5280 /* футов в миле */
dist = FPM * miles;

б. #define FEET 4
#define POD FEET + FEET
plort = FEET * POD;

в. #define SIX = 6;
nex = SIX;

г. #define NEW(X) X + 5
y = NEW(y);
berg = NEW(berg) * lob;
est = NEW(berg) / NEW(y);
nilp = lob * NEW(-berg);

2. Подправьте определение в вопросе 1.г, чтобы сделать его более надежным.

3. Определите макрофункцию, которая возвращает минимальное из двух значений.

4. Задайте макроопределение, в котором есть функция whitesp(c) считающая в программе пустые символы.

5. Определите макрофункцию, которая печатает представления значения двух целых выражений.

Ответы

1.
а. dist = 5280 * miles; правильно.

б. plot = 4 * 4 + 4; правильно, но если пользователь на самом деле хотел иметь 4 * (4 + 4), то следовало применять директиву #define POD (FEET + FEET).

в. nex = = 6; неправильно; очевидно, пользователь забыл, что он пишет для препроцессора, а не на языке Си.

г. y - y + 5; правильно.
berg = berg + 5 * lob; правильно, но, вероятно, это нежелательный результат.
est = berg + 5/y + 5; то же самое.
nilp = lob * -berg + 5; то же самое.

2. #define NEW(X) ((X) + 5)

3. #deline MIN(X,Y) ((X) < (Y) ? (X) : (Y))

4. #define WHITESP(C) ((C) == ' ' || (C) == '\n' || (C)) == '\t')

5. #define PR2(X,Y) printf(" X равно %d и Y равно %d.\n", X, Y)
Так как в этом макроопределении **X** и **Y** никогда не используются никакими другими операциями (такими, как умножение), мы не должны ничего заключать в скобки.

УПРАЖНЕНИЕ

1. Создайте заголовочный файл определений препроцессора, которые вы хотите применять.

---

12. Массивы и указатели

МАССИВЫ
МНОГОМЕРНЫЕ МАССИВЫ
ИНИЦИАЛИЗАЦИЯ МАССИВОВ
УКАЗАТЕЛИ И ОПЕРАЦИИ НАД УКАЗАТЕЛЯМИ
СВЯЗЬ МЕЖДУ МАССИВОМ И УКАЗАТЕЛЕМ

ОПЕРАЦИИ
& * (унарные)

Между массивами и указателями существует очень тесная связь, поэтому обычно их рассматривают вместе. Но, прежде чем исследовать эту связь, давайте проверим наши знания о массивах и пополним их, а уж после этого перейдем к изучению связи между массивами и указателями.

МАССИВЫ

Далее Содержание

Вы уже знаете, что массив представляет собой группу элементов одного типа. Когда нам требуется для работы массив, мы сообщаем об этом компилятору при помощи операторов описания. Для создания массива компилятору необходимо знать тип данных и требуемый класс памяти, т. е. то же самое, что и для простой переменной (называемой "скалярной"). Кроме того, должно быть известно, сколько элементов имеет массив. Массивы могут иметь те же типы данных и классы памяти, что и простые переменные, и к ним применим тот же принцип умолчания. Рассмотрим примеры, различных описаний массивов:

```
/* несколько описаний массивов */
int temp[365]; /* внешний массив из 365 целых чисел */
main( )
{
float rain[365];  /* автоматический массив из 365 чисел типа
float */

static char code[12];  /* статический массив из 12 символов */
extern temp[ ]; /* внешний массив; размер указан выше */
}
```

Как уже упоминалось, квадратные скобки ([ ]) говорят о том, что **temp** и все остальные идентификаторы являются именами массивов, а число, заключенное в скобки, указывает количество элементов массива. Отдельный элемент массива определяется при помощи его номера, называемого также индексом. Нумерация элементов начинается с нуля, поэтому **temp[0]** является первым, а **temp[364]** последним 365-элементом массива **temp**. Но все это вам уже должно быть известно, поэтому изучим что-нибудь новое.

Для хранения данных, необходимых программе, часто используют массивы. Например, в массиве из 12 элементов можно хранить информацию о количестве дней каждого месяца. В подобных случаях желательно иметь удобный способ инициализации массива перед началом работы программы. Такая возможность, вообще говоря, существует, но только для статической и внешней памяти. Давайте посмотрим, как она используется.

Мы знаем, что скалярные переменные можно инициализировать в описании типа при помощи таких выражений, как, например:

```
int fix = 1;
float flax = PI*2;
```

при этом предполагается, что **PI** - ранее введенное макроопределение. Можем ли мы делать что-либо подобное с массивом? Ответ не однозначен: и да, и нет.

Внешние и *статические* массивы *можно* инициализировать.

Автоматические и *регистровые* массивы инициализировать *нельзя*.

Прежде чем попытаться инициализировать массив, давайте посмотрим, что там находится, если мы в него ничего не записали.

```
/* проверка содержимого массива */
main( ) {
int fuzzy[2]; /*автоматический массив */
static int wuzzy[2]; /* статический массив */
printf("%d %d\n", fuzzy[1], wuzzy[1];
}
```

Программа напечатает

```
525 0
```

Полученный результат иллюстрирует следующее правило:
Если ничего не засылать в массив перед началом работы с ним, то внешние и статические массивы инициализируются нулем, а автоматические и статические массивы содержат какой-то "мусор", оставшийся в этой частя памяти.

Прекрасно! Теперь мы знаем, что нужно предпринять для обнуления статического или внешнего массива - просто ничего не делать. Но как быть, если нам нужны некоторые значения, отличные от нуля, например количество дней в каждом месяце. В этом случае мы можем делать так:

```
/* дни месяца */
int days[12]=[31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];
main( )
{
int index;
extern int days[ ]; /* необязательное описание */
for(index = 0; index < 12; index++)
printf(" Месяц %d имеет %d дней.\n", index + 1, days[index]);
}
```

Результат:

```
Месяц 1 имеет 31 дней. Месяц 2 имеет 28 дней. Месяц 3 имеет 31 дней.
Месяц 4 имеет 30 дней. Месяц 5 имеет 31 дней. Месяц 6 имеет 30 дней.
Месяц 7 имеет 31 дней. Месяц 8 имеет 31 дней. Месяц 9 имеет 30 дней.
```

```
Месяц 10  имеет 31 дней. Месяц 11 имеет 30 дней. Месяц 12 имеет 31 дней.
```

Программа не совсем корректна, поскольку она выдает неправильный результат для второго месяца каждого четвертого года.

Определив массив **days[ ]** вне тела функции, мы тем самым сделали его внешним. Мы инициировали его списком, заключенным в скобки, используя при этом запятые для разделения элементов списка.

Количество элементов в списке должно соответствовать размеру массива. А что будет, если мы ошиблись в подсчете? Попробуйте переписать последний пример, используя список, который короче, чем нужно (на два элемента):

```
/* дни месяца */
int days[12]=[31, 28, 31, 30, 31, 30, 31, 31, 30, 31];
main( )
{
int index;
extern int days[  ]; /* необязательное описание */
for(index = 0; index < 12; index++)
printf(" Месяц %d имеет %d дней.\n", index + 1, days[index]);
}
```

В этом случае результат оказывается иным:

```
Месяц 1 имеет 31 дней.
Месяц 2 имеет 28 дней.
Месяц 3 имеет 31 дней.
Месяц 4 имеет 30 дней.
Месяц 5 имеет 31 дней.
Месяц 6 имеет 30 дней.
Месяц 7 имеет 31 дней.
Месяц 8 имеет 31 дней.
Месяц 9 имеет 30 дней.
Месяц 10 имеет 31 дней.
Месяц 11 имеет 0 дней.
Месяц 12 имеет 0 дней.
```

Можно видеть, что у компилятора не возникло никаких проблем: просто когда он исчерпал весь список с исходными данными, то стал присваивать всем остальным элементам массива нулевые значения.

Однако в случае излишне большого списка компилятор будет уже не столь "великодушен" к вам, поскольку посчитает выявленную избыточность ошибкой. Поэтому нет никакой необходимости заранее подвергать себя "насмешкам" компилятора. Надо просто выделить массив, размер которого будет достаточен для размещения списка:

```
/* дни месяца */
int days[ ]=[31, 28, 31, 30, 31, 30, 31, 31, 30, 31];
main( )
{
int index;
extern int days[ ]; /* необязательное описание */
for(index = 0; index < sizeof days/(sizeof (int)); index++)
printf(" Месяц %d имеет %d дней.\n", index + 1, days [index]);
}
```

К этой программе следует сделать два существенных замечания:
Первое: если вы используете пустые скобки для инициализации массива, то компилятор сам определит количество элементов в списке и выделит для него массив нужного размера.

Второе: оно касается добавления, сделанного п управляющем операторе **for**. Не полагаясь (вполне

обоснованно) на свои вычислительные способности, мы возложили задачу подсчета размера массива на компилятор. Оператор **sizeof** определяет размер в байтах объекта или типа, следующего за ним. (Мы. уже упоминали об этом в гл. 3.) В нашей вычислительной системе размер каждого элемента типа **int** равен двум байтам, поэтому для получения количества элементов массива мы делим общее число байтов, занимаемое массивом, на 2. Однако в других системах элемент типа **int** может иметь иной размер. Поэтому в общем случае выполняется деление на значение переменной **sizeof** (для элемента типа **int**). Ниже приведены результаты работы нашей программы:

```
Месяц 1 имеет 31 дней.
Месяц 2 имеет 28 дней.
Месяц 3 имеет 31 дней.
Месяц 4 имеет 30 дней.
Месяц 5 имеет 31 дней.
Месяц 6 имеет 30 дней.
Месяц 7 имеет 31 дней.
Месяц 8 имеет 31 дней.
Месяц 9 имеет 30 дней.
Месяц 10 имеет 31 дней.
```

Ну вот, теперь мы получаем точно 10 значений. Наш метод, позволяющий программе самой находить размер массива, не позволил нам напечатать конец массива.

Существует и более короткий способ инициализации массивов, но поскольку он применим только к *символьным* строкам, мы рассмотрим его в следующей главе.

В заключение мы покажем, что можно *присваивать* значения элементам массива, относящегося к любому классу памяти. Например, в приведенном ниже фрагменте программы присваиваются четные числа элементам автоматического массива:

```
/* присваивание значений массиву */
main( )
{
int counter, evens [50];
for(counter = 0; counter < 50; counter++)
evens[counter] = 2 * counter;
...
}
```

УКАЗАТЕЛИ МАССИВОВ

Далее Содержание

Как было сказано в гл. 9, указатели позволяют нам работать с символическими адресами. Поскольку в реализуемых аппаратно командах вычислительной машины интенсивно используются адреса, указатели предоставляют возможность применять адреса примерно так, как это делается в самой машине, и тем самым повышать эффективность программ. В частности, указатели позволяют эффективно организовать работу с массивами. Действительно, как мы могли убедиться, наше обозначение массива представляет собой просто скрытую форму использования указателей.

Например, имя массива определяет также его первый элемент, т. е. если **flizny[]** - массив, то

```
flizny == &flizny[0]
```

и обе части равенства определяют адрес первого элемента массива. (Вспомним, что операция **&** выдает адрес.) Оба обозначения являются *константами* типа указатель, поскольку они не изменяются на протяжении всей программы. Однако их можно присваивать (как значения) *переменной* типа указатель и изменять значение переменной, как показано в ниже следующем

примере. Посмотрите, что происходит со значением указателя, если к нему прибавить число.

```
/* прибавление к указателю */
main( )
{
int dates[4], *pti, index;
float bills [4], *ptf;
pti = dates; /* присваивает адрес указателю массива */
ptf = bills;
for(index = 0; index < 4; index++)
printf(" указатели + %d:  %10  u  %10u \n", index, pti + index, ptf + index);
}
```

Вот результат:

```
указатели + 0    56014    56026
указатели + 1     56016   56030
указатели + 2    56018    56034
указатели + 3    56020    56038
```

Первая напечатанная строка содержит начальные адреса двух массивов, а следующая строка - результат прибавления единицы к адресу и т. д. Почему так получается?

```
56014 + 1 = 56016? 56026 + 1 = 56030?
```

Не знаете, что сказать? В нашей системе единицей адресации является байт, но тип **int** использует два байта, а тип **float** - четыре. Что произойдет, если вы скажете: "прибавить единицу к указателю?" Компилятор языка Си добавит *единицу памяти*. Для массивов это означает, что мы перейдем к адресу следующего *элемента*, а не следующего байта. Вот почему мы должны специально оговаривать тип объекта, на который ссылается указатель; одного адреса здесь недостаточно, так как машина должна знать, сколько байтов потребуется для запоминания объекта. (Это справедливо также для указателей на скалярные переменные; иными словами, при помощи операции ***pt** нельзя получить значение.)

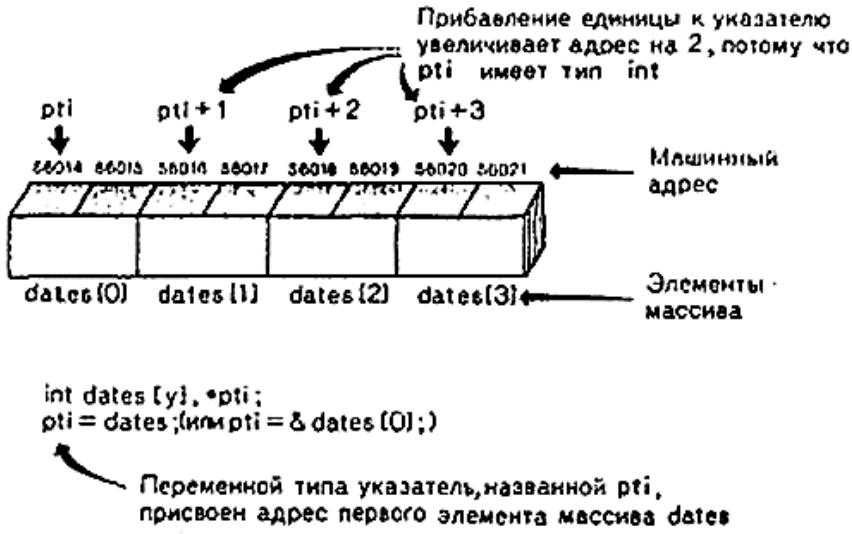


РИС. 12.1. Увеличение указателя массива.

Благодаря тому что компилятор языка Си умеет это делать, мы имеем следующие равенства:

```
dates + 2 == &dates[2] /* один и тот же адрес */
*(dates + 2) == dates[2] /* одно и то же значение */
```

Эти соотношения суммируют тесную связь между массивами и указателями. Они показывают, что можно использовать указатель для определения отдельного элемента массива, а также для

получения его значения. По существу мы имеем два различных обозначения для одного и того же. Действительно, компилятор превращает обозначение массива в указатели, поэтому метод указателей более предпочтителен.

Между прочим, постарайтесь различать выражения ***(dates + 2)**, и ***dates + 2**. Операция (***) имеет более высокий приоритет, чем **+**, поэтому последнее выражение означает

```
(*dates) + 2:
*(dates + 2) /* значение 3-го элемента массива dates */
*dates +2    /* 2 добавляется к значению 1-го элемента массива */
```

Связь между массивами и указателями часто позволяет нам применять оба подхода при создании программ. Одним из примеров этого является функция с массивом в качестве аргумента.

ФУНКЦИИ, МАССИВЫ И УКАЗАТЕЛИ

Далее Содержание

Массивы можно использовать в программе двояко. Во-первых, их можно описать в теле функции. Во-вторых, они могут быть аргументами функции. Вес, что было сказано в этой главе о массивах, относится к первому их применению; теперь рассмотрим массивы в качестве аргументов.

Об этом уже говорилось в гл. 10. Сейчас, когда мы познакомились с указателями, можно заняться более глубоким изучением массивов-аргументов. Давайте проанализируем скелет программы, обращая внимание на описания:

```
/* массив-аргумент */
main( )
{
int ages[50]; /* массив из 50 элементов */
convert(ages);
...
}
convert (years);
int years [ ]; /* каков размер массива? */
{
...
}
```

Очевидно, что массив **ages** состоит из 50 элементов. А что можно сказать о массиве **years**? Оказывается, в программе нет такого массива. Описатель

```
int years[ ];
```

создает не *массив*, а *указатель* на него. Посмотрим, почему это так.
Вот вызов нашей функции:

```
convert(ages);
```

ages - аргумент функции **convert**. Вы помните, что имя **ages** является *указателем* на первый элемент массива, состоящего из 50 элементов. Таким образом, оператор вызова функции передает ей указатель, т. е. адрес функции **convert()**. Это значит, что аргумент функции является указателем, и мы можем написать функцию **convert()** следующим образом:

```
convert (years);
int *years;
{
}
```

Действительно, операторы

```
int years[ ];
```

```
int *years;
```

синонимы. Оба они объявляют переменную **years** указателем массива целых чисел. Однако главное их отличие состоит в том, что первый из них напоминает нам, что указатель **years** ссылается на массив.

Как теперь связать его с массивом **ages**? Вспомним, что при использовании указателя в качестве аргумента, функция взаимодействует с соответствующей переменной в вызывающей программе, т. е. операторы, использующие указатель **years** в функции **convert()**, фактически работают с массивом **ages**, находящимся в теле функции **main()**.

Посмотрим, как работает этот механизм. Во-первых, вызов функции инициализирует указатель **years**, ссылаясь на **ages[0]**. Теперь предположим, что где-то внутри функции **convert( )** есть выражение **years[3]**. Как вы видели в предыдущем разделе, оно аналогично ***(years + 3)**. Однако если years указывает на **ages[0]**, то **years+3** ссылается на **ages[3]**. Это приводит к тому, что ***(years+3)** означает **ages[3]**. Если внимательно проследить данную цепочку, то мы увидим, что **years[3]** аналогично ***(years + 3)**, которое в свою очередь совпадает с **ages[3]**. Что и требовалось доказать, т. е. операции над указателем years приводят к тем же результатам, что и операции над массивом **ages**.

Короче говоря, когда имя массива применяется в качестве аргумента, функции передается указатель. Затем функция использует этот указатель для выполнения изменений в исходном массиве, принадлежащем программе, вызвавшей функцию. Рассмотрим пример.

ИСПОЛЬЗОВАНИЕ УКАЗАТЕЛЕЙ ПРИ РАБОТЕ С МАССИВАМИ

Далее Содержание

Попробуем написать функцию, использующую массивы, а затем перепишем ее, применяя указатели.

Рассмотрим простую функцию, которая находит (или пытается найти) среднее значение массива целых чисел. На входе функции мы имеем имя массива и количество элементов. На выходе получаем среднее значение, которое передастся при помощи оператора **return**. Оператор вызова функции может выглядеть следующим образом:

```
printf("Среднее из заданных значений %d.\n", mean(numbs,size));

/* находит среднее значение массива из n целых чисел */
int mean(array, n);
int array[ ], n;
{
int index;
long sum; /* Если целых слишком много, их можно
             суммировать в формате long int */
if(n > 0)
{
for(index = 0, sum = 0; index < n; index++)
    sum + = array[index];
return((int)(sum/n)); /* возвращает int * / }
else {
printf("Нет массива. \n");
return(0); }
}
```

Эту программу легко переделать, применяя указатели. Объявим **pa** указателем на тип **int**. Затем заменим элемент массива **array[index]** на соответствующее значение: ***(pa + index)**.

```
/* Использование указателей для нахождения
        среднего значения массива n целых чисел */
```

```
int mean(pa, n) int opa, n;
{
int index;
long sum; /*Если целых слишком много,
    их можно суммировать в формате long int */
if(n > 0)
{
for(index=0, sum=0; index < n; index++)
sum + = *(pa + index);
return((int)(sum/n)); /* Возвращает     целое */ }
else {
printf("Нет массива.\n");
return(0); }
}
```

Это оказалось несложным, но возникает вопрос: должны ли мы изменить при этом вызов функции, в частности **numbs**, который был именем массива в операторе **mean(numbs, size)**? Ничего не нужно менять, поскольку имя массива *является* указателем. Как мы уже говорили в предыдущем разделе, операторы описания:

```
int pa[ ];
```

и

```
int *pa;
```

идентичны по действию: оба объявляют ра указателем. В программе можно применять любой из них, хотя до сих пор мы использовали второй в виде ***(pa + index)**.

Понятно ли вам, как работать с указателями? Указатель устанавливается на первый элемент массива, и значение, находящееся там, добавляется в **sum**. Затем указатель передвигается на следующий элемент (к указателю прибавляется единица), и значение, находящееся в нем, также прибавляется к **sum** и т. д. Это похоже на механизм работы с массивом, где индекс действует как стрелка часов, показывающая по очереди на каждый элемент массива.

Теперь у нас есть два подхода; какой же из них выбрать? Во-первых, хотя массивы и указатели тесно связаны, у них есть отличия. Указатели являются более общим и широко применяемым средством, однако многие пользователи (по крайней мере начинающие) считают, что массивы более привычны к понятны. Во-вторых, при использовании указателей у нас нет простого эквивалента для задания размера массива. Самую типичную ситуацию, в которой можно применять указатель, мы уже показали: это функция, работающая с массивом, который находится где-то в другой части программы. Мы предлагаем использовав любой из подходов по вашему желанию. Однако несомненное преимущество использования указателей в приведенном выше примере должно научить вас легко применять их, когда в этом *возникает* необходимость.

ОПЕРАЦИИ С УКАЗАТЕЛЯМИ

Далее Содержание

Что же мы теперь умеем делать с указателями? Язык Си предлагает пять основных операций, которые можно применять к указателям, а нижеследующая программа демонстрирует эти возможности. Чтобы показать результаты каждой операции, мы будем печатать значение указателя (являющегося адресом, на который ссылается указатель), значение, находящееся по этому адресу, и адрес самого указателя.

```
/* операции с указателями */
#define PR(X)
printf("X = %u,*X = %d, &X = %u\n",X, *X,&X);
/* печатает значение указателя (адрес),
            значение, находящееся по */
/* этому адресу, и адрес самого указателя */
```

```
main( )
static int urn[ ] = [100, 200, 300];
int *ptrl, *ptr2;
{
ptrl = urn; /* присваивает адрес указателю */
ptr2 = &urn [2]; /* то же самое */
PR(ptrl); /* см. макроопределение, указанное выше */
ptrl++; /* увеличение указателя */
PR(ptrl);
PR(ptr2);
++ptr2; /* выходит за конец массива */
PR(ptr2);
printf("ptr2 - ptrl = %u\n", ptr2 - ptrl);
}
```

В результате работы программы получены следующие результаты:

```
ptrl = 18, *ptrl = 100, &ptrl = 55990
ptrl = 20, *ptrl = 200, &ptrl = 55990
ptr2 =22, *ptr2 = 300, &ptr2 = 55992
ptr2 =24, *ptr2 = 29808, &ptr2 = 55992
ptr2 - ptrl = 2
```

Программа демонстрирует пять основных операций, которые можно выполнять над переменными типа указатель.

1. ПРИСВАИВАНИЕ. Указателю можно присвоить адрес. Обычно мы выполняем это действие, используя имя массива или операцию получения адреса (**&**). Программа присваивает переменной **ptrl** адрес начала массива **urn;** этот адрес принадлежит ячейке памяти с номером 18. (В нашей системе статические переменные запоминаются в ячейках оперативной памяти.) Переменная **ptr2** получает адрес третьего и последнего элемента массива, т. е. **urn[2]**.

2. ОПРЕДЕЛЕНИЕ ЗНАЧЕНИЯ. Операция выдает значение, хранящееся в указанной ячейке. Поэтому результатом операции ***ptrl** в самом начале работы программы является число 100, находящееся в ячейке с номером **18**.

3. ПОЛУЧЕНИЕ АДРЕСА УКАЗАТЕЛЯ. Подобно любым переменным переменная типа указатель имеет адрес и значение. Операция **&** сообщает нам, где находится сам указатель. В нашем примере указатель **ptrl** находится в ячейке с номером **55990**. Эта ячейка содержит число **18**,

являющееся адресом начала массива **urn**.

4. УВЕЛИЧЕНИЕ УКАЗАТЕЛЯ. Мы можем выполнять это действие с помощью обычной операции сложения либо с помощью операции увеличения. Увеличивая указатель, мы перемещаем его на следующий элемент массива. Поэтому операция **ptr1++** увеличивает числовое значение переменной **ptrl** на **2** (два байта на каждый элемент массива целых чисел), после чего указатель **ptrl** ссылается уже на **urn[l]** (рис. 12.2). Теперь **ptrl** имеет значение **20** (адрес следующего элемента массива), а операция ***ptrl** выдает число **200**, являющееся значением элемента **urn[1]**. Заметим, что адрес самой ячейки **ptrl** остается неизменным, т.е. **55990**. После выполнения операции сама переменная не переместилась, потому что она только изменила значение!

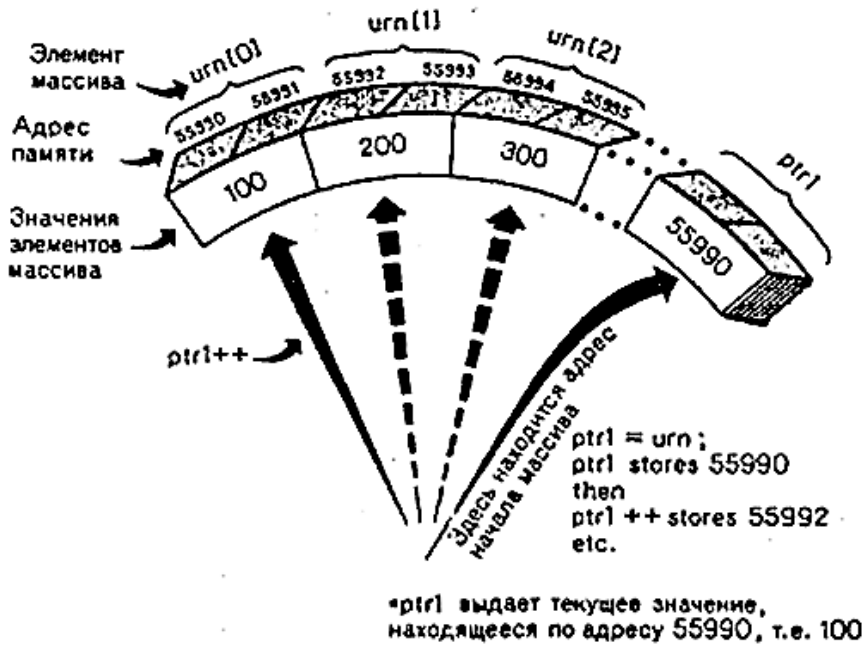


РИС. 12.2. Увеличение указателя типа int.

Аналогичным образом можно и уменьшить указатель. Однако при этом следует соблюдать осторожность. Машина не следит, ссылается ли еще указатель на массив или уже нет. Операция **++ptr2** перемещает указатель **ptr2** на следующие два байта, и теперь он ссылается на некоторую информацию, расположенную в памяти за массивом.

Кроме того, оператор увеличения можно использовать для переменной типа указатель, но не для констант этого типа подобно тому, как вы не можете применять оператор увеличения для обычных констант. Для переменных и констант типа указатель можно использовать простое сложение:

Правильно	*Неправильно*
ptr1++;	urn ++;
x ++;	3++;
ptr2 = ptr1 + 2;	ptr2 = urn++;
ptr2 = urn + 1;	x = y + 3++;

5. РАЗНОСТЬ. Можно находить разность двух указателей. Обычно это делается для указателей, ссылающихся на элементы одного и того же массива, чтобы определить, на каком расстоянии друг от друга находятся элементы. Помните, что результат имеет тот же тип, что и переменная, содержащая *размер* массива.

Перечисленные выше операции открывают большие возможности. Программисты на языке Си создают массивы указателей, указатели на функции, массивы указателей на указатели, массивы

указателей на функции и т. д. Мы будем придерживаться основных применений, которые уже упоминались. Первое из них - передача информации в функцию и из нее. Мы использовали указатели, когда хотели, чтобы функция изменила переменные, находящиеся в вызывающей программе. Второе - использование указателей в функциях, работающих с многомерными массивами.

МНОГОМЕРНЫЕ МАССИВЫ

Далее Содержание

Темпест Клауд, метеоролог, занимающаяся явлением перисто-сти облаков, хочет проанализировать данные о ежемесячном количестве осадков на протяжении пяти лет. В самом начале она должна решить, как представлять данные. Можно использовать 60 переменных, по одной на каждый месяц. (Мы уже упоминали о таком подходе ранее, но в данном случае он также неудачен.) Лучше было бы взять массив, состоящий из 60 элементов, но это устроило бы нас только до тех пор, пока можно хранить раздельно данные за каждый год. Мы могли бы также использовать 5 массивов по 12 элементов каждый, но это очень примитивно и может создать действительно большие неудобства, если Темнеет решит изучать данные о количестве осадков за 50 лет вместо пяти. Нужно придумать что-нибудь получше.

Хорошо было бы использовать массив массивов. Основной массив состоял бы тогда из 5 элементов, каждый из которых в свою очередь был бы массивом из 12 элементов. Вот как это записывается:

```
static float rain[5][12];
```

Можно также представить массив **rain** в виде двумерного массива, состоящего из 5 строк и 12 столбцов.

При изменении второго индекса на единицу мы передвигаемся вдоль строки, а при изменении первого индекса на единицу, передвигаемся вертикально вдоль столбца. В нашем примере второй индекс дает нам месяцы, а первый - годы.

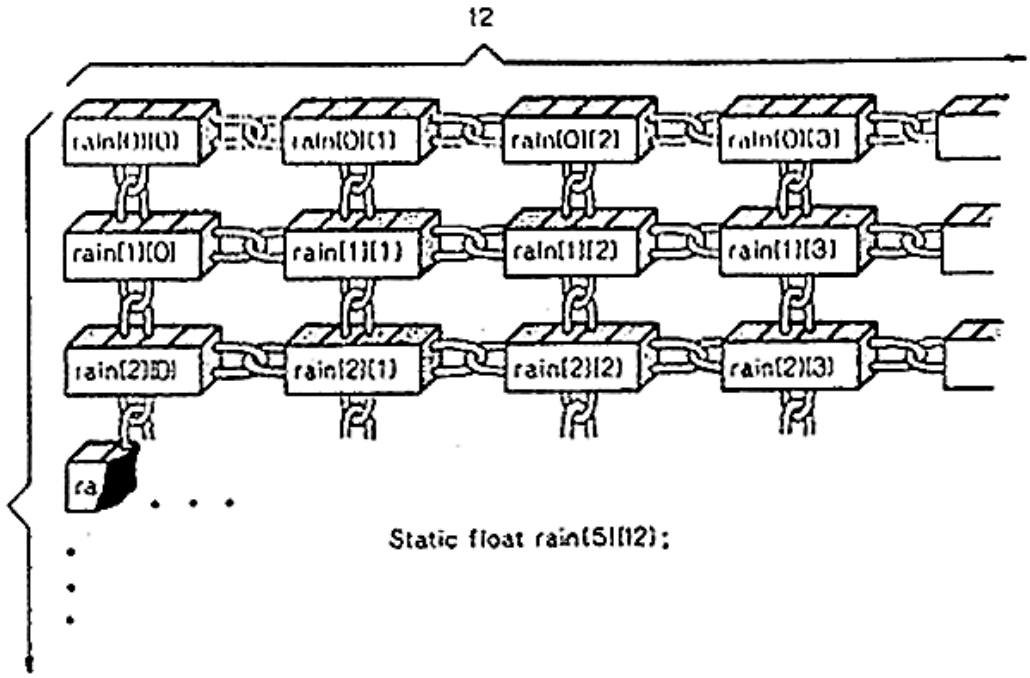


РИС. 12.3. Двумерный массив.

Используем этот двумерный массив в метеорологической программе. Цель нашей программы -

найти общее количество осадков для каждого года, среднегодовое количество осадков и среднее количество осадков за каждый месяц. Для получения общего количества осадков за год следует сложить все данные, находящиеся в нужной строке. Чтобы найти среднее количество осадков за данный месяц, мы сначала складываем все данные в указанном столбце. Двумерный массив позволяет легко представить и выполнить эти действия. Рис. 12.4 содержит программу.

```
/* найти общее количество осадков для каждого года, среднего */
/* довое, среднемесячное количество осадков, за несколько лет */
#define TWLV 12 /* число месяцев в году */
#define YRS 5 /* число лет */
main( )
{
static float rain [YRS][TWLV] = {
{10.2, 8.1, 6.8, 4.2, 2.1, 1.8, 0.2, 0.3, 1.1, 2.3, 6.1, 7.4},
{9.2, 9.8, 4.4, 3.3, 2.2, 0.8, 0.4, 0.0, 0.6, 1.7, 4.3, 5.2},
{6.6, 5.5, 3.8, 2.8, 1.6, 0.2, 0.0, 0.0, 0.0, 1.3, 2.6, 4.2},
{4.3, 4.3, 4.3, 3.0, 2.0, 1.0, 0.2, 0.2, 0.4, 2.4, 3.5, 6.6},
{8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9, 0.3, 0.9, 1.4, 7.2}
};
/* инициализация данных по количеству осадков за 1970-1974 */
int year, month;
float subtot, total;
printf("ГОД КОЛИЧЕСТВО ОСАДКОВ (дюймы)\n\n");
for(year = 0, total = 0; year < YRS; year++)
{ /* для каждого года, суммируем количество осадков для каждого месяца */
for(month = 0, subtot = 0; month < TWLV; month++)
    subtot + = rain [year][month];
printf("%5d %15.lf\n", 1970 + year, subtot);
total + = subtot; /* общее для всех лет */
}
printf(" \n среднегодовое количество осадков
        составляет %.lf дюймов. \n \n ", total/YRS );
printf(" Янв. Фев. Мар. Апр.Май Июн.Июл. Авг.Сент.");
printf(" Окт. Нояб. Дек.\n" );
for(month = 0; month < TWLV; month++)
{   /* для каждого месяца, суммируем
    количество осадков за все годы */
for(year = 0, subtot = 0; year < YRS; year++)
subtot += rain[year][month];
printf(" %4.lf ", subtot/YRS); }
printf(" \n");
}
```

РИС. 12.4. Метеорологическая программа.

```
ГОД     КОЛИЧЕСТВО ОСАДКОВ (дюймы)
1970    50.6
1971    41.9
1972    28.6
1973    32.3
1974    37.8

Среднегодовое количество осадков составляет 38.2 дюйма.

ЕЖЕМЕСЯЧНОЕ КОЛИЧЕСТВО:
Янв. Фев. Мар. Апр. Май. Июн. Июл. Авг. Сент. ОКТ. Нояб. Дек.
7.8  7.2  4.1  3.0  2.1  0.8  1.2  0.3  0.5       1.7  3.6       6.l
```

В этой программе следует отметить два основных момента: инициализацию и вычисления. Инициализация сложнее, поэтому мы сначала рассмотрим вычисления.

Чтобы найти общее количество осадков за год, мы не изменяем **year**, а заставляем переменную **month** пройти все свои значения. Так выполняется внутренний цикл **for**, находящийся в первой части программы. Затем мы повторяем процесс для следующего значения **year**. Это внешний цикл

первой части программы. Структура вложенного цикла, подобная описанной, подходит для работы с двумерным массивом. Один цикл управляет одним индексом, а второй цикл - другим.

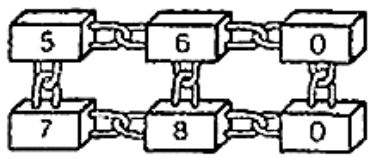
Вторая часть программы имеет такую же структуру, но теперь мы изменяем **year** во внутреннем цикле, а **month** во внешнем. Помните, что при однократном прохождении внешнего цикла внутренний цикл выполняется полностью. Таким образом, программа проходит в цикле через все годы, прежде чем изменится месяц, и дает нам общее количество осадков за пять лет для первого месяца, затем общее количество за пять лет для второго месяца и т. д.

Инициализация двумерного массива

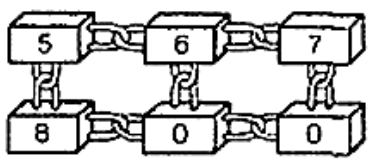
Далее Содержание

Для инициализации массива мы взяли пять заключенных в скобки последовательностей чисел, а все эти данные еще раз заключили в скобки. Данные, находящиеся в первых внутренних скобках, присваиваются первой строке массива, данные во второй внутренней последовательности - второй строке и т. д. Правила, которые мы обсуждали раньше, о несоответствии между размером массива и данных применяются здесь для каждой строки. Если первая последовательность в скобках включает десять чисел, то только первым десяти элементам первой строки будут присвоены значения. Последние два элемента в этой строке будут, как обычно, инициализированы нулем по умолчанию. Если чисел больше, чем нужно, то это считается ошибкой; перехода к следующей строке не произойдет.

Мы могли бы опустить все внутренние скобки и оставить только две самые внешние. До тех пор пока мы будем давать правильное количество входных данных, результат будет тем же самым. Однако, если данных меньше, чем нужно, массив заполняется последовательно (не обращается внимание на разделение по строкам), пока не кончатся все данные. Затем оставшимся элементам будут присвоены нулевые значения. См. рис. 12.5.




```
static int sq [2][3] ={
                        {5,6},
                        {7,8},
                       };
```




```
static int sq [2][3] = {5,6,7,8};
```

РИС. 12.5. Два метода инициализации массива.

Все, что мы сказали о двумерных массивах, можно распространить и на трехмерные массивы и т. д. Трехмерный массив описывается следующим образом:

```
int solido[10][20][30];
```

Вы можете представить его в виде десяти двумерных массивов (каждый 20x30), поставленных

друг на друга, или в виде массива из массивов. То есть это массив из 10 элементов, и каждый его элемент также является массивом. Все эти массивы в свою очередь имеют по 20 элементов, каждый из которых состоит из 30 элементов. Преимущество этого второго подхода состоит в том, что можно довольно просто перейти к массивам большей размерности, если окажется, что вы не можете представить наглядно четырехмерный объект! Мы же останемся верны двум измерениям.

УКАЗАТЕЛИ И МНОГОМЕРНЫЕ МАССИВЫ

Далее Содержание

Как создать указатели для многомерных массивов? Чтобы найти ответ на этот вопрос, рассмотрим несколько примеров.

Предположим, что у нас есть описания

```
int zippo[4][2];  /* массив типа int
                  из 4 строк и 2 столбцов */
int *pri;    /* указатель на целый тип */
```

Тогда на что **pri = zippo;** указывает? На первый столбец первой строки:

```
zippo == &zippo[0][0]
```

А на что указывает **pri + 1**? На **zippo[0][l]**, т.е. на 1-ю строку 2-го столбца? Или на **zippo[l][0]**, элемент, находящийся во второй строке первого столбца? Чтобы ответить на поставленный вопрос, нужно знать, как располагается в памяти двумерный массив. Он размещается, подобно одномерным массивам, занимая последовательные ячейки памяти. Порядок элементов определяется тем, что самый правый индекс массива изменяется первым, т. е. элементы массива располагаются следующим образом:

```
zippo[0][0] zippo[0][1] zippo[1][0] zippo[1][1] zippo[2][0]
...
```

Сначала запоминается первая строка, за ней вторая, затем третья и т. д. Таким образом в нашем примере:

```
pri == &zippo[0][0] /* 1-я строка, 1 столбец */
pri + 1 == &zippo[0][1] /* 1-я строка, 2 столбец */
pri + 2 == &zippo[1][0] /* 2-я строка, 1 столбец */
pri + 3 == &zippo[1][1] /* 2-я строка, 2 столбец */
```

Получилось? Хорошо, а на что указывает **pri + 5**? Правильно, на **zippo[2][l]**.

Мы описали двумерный массив как массив массивов. Если **zippo** является именем нашего двумерного массива, то каковы имена четырех строк, каждая из которых является массивом из двух элементов? Имя первой строки **zippo[0]**, имя четвертой строки **zippo[3]**; вы можете заполнить пропущенные имена. Однако имя массива является также указателем на этот массив в том смысле, что оно ссылается на первый его элемент. Значит,

```
zippo[0] == &zippo[0][0]
zippo[1] == &zjppo[1][0]
zippo[2] == &zippo[2][0]
zippo[3] == &zippo[3][0]
```

Это свойство является более, чем новшеством. Оно позволяет использовать функцию, предназначенную для одномерного массива, для работы с двумерным массивом! Вот доказательство (хотя мы надеемся, что теперь вы бы поверили нам и так) использования двумерного массива в нашей программе нахождения среднего значения:

```
/* одномерная функция, двумерный массив */
main( )
```

```
{
static int junk[3][4] = {
{2, 4, 6, 8},
{100, 200, 300, 400},
{10, 40, 60, 90} };
int row;
for(row = 0; row < 3; row ++)
printf(" Среднее строки %d равно %d.\n", row, mean(junk[row],4));
/* junk [row] - одномерный массив из четырех элементов */
}
/* находит среднее в одномерном массиве */
int mean(array,n)
int array[ ], n;
{
int index;
long sum;
if(n > 0) {
for(index = 0, sum = 0; index < n; index++)
    sum += (long)array[index];
return((int)(sum/n)); }
else {
printf(" Нет массива. \n");
return(0); }
}
```

Результат работы программы:

```
Среднее строки  0 равно 5.
Среднее строки  1 равно 250.
Среднее строки  2 равно 50.
```

Функции и многомерные массивы Далее Содержание

Предположим, что вы хотите иметь функцию, работающую с двумерным массивом, причем со всем целиком, а не с частями. Как вы запишите определения функции и ее описания? Подойдем к этому более конкретно и скажем, что нам нужна функция, управляющая массивом **junk[ ][ ]** в нашем последнем примере. Пусть функция **main( )** выглядит так:

```
/* junk в main */
main( )
{
static int junk[3][4] = {
{2, 4, 5, 8},
{100, 200, 300, 400}
{10, 40, 60, 90} };
stuff(junk);
}
```

Функция **stuff( )** использует в качестве аргумента **junk**, являющийся указателем на весь массив. Как написать заголовок функции, не зная, что делает **stuff( )**?
Попробуем написать:

```
stuff(junk) int junk[ ];
```

или

```
stuff(junk) int junk[ ][ ];
```

Нет и нет. Первые два оператора еще будут работать некоторым образом, но они рассматривают **junk** как одномерный массив, состоящий из 12 элементов. Информация о расчленении массива на

строки отсутствует.

Вторая попытка ошибочна, потому что хотя оператор и указывает что **junk** является двумерным массивом, но нигде не говорится, из чего он состоит. Из шести строк и двух столбцов? Из двух строк и шести столбцов? Или из чего-нибудь еще? Компилятору недостаточно этой информации. Ее дают следующие операторы:

```
stuff(junk)
int junk[ ][4];
```

Они сообщают компилятору, что массив следует разбить на строки по четыре столбца. Массивы символьных строк являются особым случаем, так как у них нулевой символ в каждой строке сообщает компилятору о конце строки. Это разрешает описания, подобные следующему:

```
char *list[  ];
```

Символьные строки представляют одно из наиболее частых применений массивов и указателей; мы вернемся к этой теме в гл. 13.

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Как объявить одномерный массив: **long id_no[200];**
Как объявить двумерный массив: **short chess[8][8];**
Какие массивы можно инициализировать: внешние и статические.
Как инициализировать массив: **static int hats[3]=[10,20,15];**
Другой способ инициализации: **static int caps[ ]=[3,56,2];**
Как получить адрес переменной: использовать операцию **&**.
Как получить значение, ссылаясь на указатель: использовать операцию *****.
Смысл имени массива: **hats == &hats[0]**.
Соответствие массива и указателя: если **ptr = hats;** то **ptr + 2 == &hat[2];** и ***(ptr+2) == hat[2];**
Пять операций, которые можно применять для переменных типа указатель: см. текст.
Метод указателей для функций, работающих с массивами.

ВОПРОСЫ И ОТВЕТЫ

Далее Содержание

Вопросы

1. Что напечатается в результате работы этой программы?

```
#define PC(X, Y)
printf(" %c %c \n", X, Y)
char ref[ ] = { D, O, L, T};
main( )
{
char *ptr;
int index;
for(index =0; ptr = ref; index < 4; index++, ptr++)
    PC(ref[indcx], *ptr);
}
```

2. Почему в вопросе 1 массив **ref** описан до оператора **main( )**?

3. Определите значение ***ptr** и ***(ptr + 2)** в каждом случае:

```
а. int *ptr;
    static int boop[4] = {12, 21, 121, 212};
    ptr = boop;
б. float *ptr;
    static float awk[2][2] = { {1.0, 2.0}, {3.0, 4.0}};
    ptr = awk[0];
в. int *ptr;
    static int jirb[4] = {10023, 7};
    ptr = jirb;
г. int = *ptr;
    static int torf[2][2] = {12, 14, 16};
    ptr = torf[0];
д. int *ptr;
    static int fort[2][2] = { { 12}, {14, 16} };
    ptr = fort[0];
```

4. Предположим, у нас есть описание **static int grid[30][100];**

а. Выразите адрес grid [22][56] иначе.
б. Выразите адрес grid[22][0] двумя способами.
в. Выразите адрес grid[0][0] тремя способами.

Ответы

1.
D D
O O
L L
T T

2. По умолчанию такое положение **ref** относит его к классу памяти типа **extern**, а массивы этого класса памяти можно инициализировать.

3.
а. 12 и 121
б. 1.0 и 3.0
в. 10023 и 0 (автоматическая инициализация нулем)
г. 12 и 16
д. 12 и 14 (именно 12 появляется в первой строке из-за скобок).

4.
а. &grid[22][56]
б. &grid[22][01 и grid[22]
в. &grid[ ][ ] и grid[0] и grid

УПРАЖНЕНИЕ

1. Модифицируйте нашу метеорологическую программу таким образом, чтобы она выполняла вычисления, используя указатели вместо индексов. (Вы по-прежнему должны объявить и инициализировать массив.)

[Содержание] [Вверх]

Язык Си Массивы и указатели

13. Символьные строки и функции над строками

СИМВОЛЬНЫЕ СТРОКИ
ИНИЦИАЛИЗАЦИЯ СИМВОЛЬНЫХ СТРОК
ВВОД-ВЫВОД СТРОК
ИСПОЛЬЗОВАНИЕ ФУНКЦИЙ, РАБОТАЮЩИХ СО СТРОКАМИ
АРГУМЕНТЫ КОМАНДНЫХ СТРОК

Символьные строки представляют один из наиболее полезных и важных типов данных языка Си. Хотя до сих пор все время применялись символьные строки, мы еще не все знаем о них. Конечно, нам уже известно самое главное: символьная строка является массивом типа **char**, который заканчивается нуль-символом (**'\0'**). В этой главе мы больше узнаем о структуре строк, о том, как описывать и инициализировать строки, как их вводить или выводить из программы, как работать со строками.

На рис. 13.1 представлена работающая программа, которая иллюстрирует несколько способов создания строк, их чтения и вывода на печать. Мы используем две новые функции: **gets( )**, которая получаст строку, и **puts( )**, которая выводит строку. (Вы, вероятно, заметили сходство их имен с функциями **getchar( )** и **putchar( )**.) В остальном программа выглядит достаточно привычно.

```
/* работа со строками */
#include <stdio.h>
#deline MSG  "У вас, наверное, много талантов.
                  Расскажите о некоторых" .
/* константа символьной строки */
#define NULL 0
#define LIM 5
#define LINLEN 81 /* максимальная длина строки + 1 */
char ml[ ] = " Только ограничьтесь одной строкой.";
/* инициализация внешнего символьного массива */
char *m2 = " Если вы не можете вспомнить что-нибудь, придумайте.";
/* инициализация указателя внешнего символьного массива */
main( )
{
char name[LlNLEN];
static char talents[LINLEN];
int i;
int count = 0;
char *m3 = " \n Достаточно обо мне -- Как вас зовут?";
/* инициализация указателя */
static char *mytal[LlM] = ("Быстро складываю числа",
                           "Точно умножаю",
                           "Записываю данные",
                           "Правильно выполняю команды",
                           "Понимаю язык Си"};
/* инициализация массива строк */
printf("Привет! Я Клайд, компьютер.
           У меня много талантов.\n");
printf("%s \n", "Позвольте рассказать о некоторых из них.");
puts(" Каковы они? Ах да, вот их неполный перечень.");
for(i = 0; i<LIM; i++)
puts(mytal[i]); /* печатает перечень талантов компьютера */
```

```
puts(m3);
gets(name);
printf(" Хорошо, %s, %s\n" , name, MSG);
printf(" %s\n %s\n", m1, m2);
gets(talents);
puts(" Давайте, посмотрим, получил ли я этот перечень:");
puts(talents);
printf(" Спасибо за информацию, %s \n" , name);
}
```

РИС. 13.1. Программа, использующая строки.

Чтобы помочь вам разобраться в том, что делает эта программа, мы приводим результат ее работы:

```
Привет, я Клайд, компьютер. У меня много  талантов.
Позвольте рассказать о некоторых из них.
Каковы они? Ах  да, вот их неполный перечень.
Быстро складываю числа.
Точно умножаю.
Записываю данные.
Правильно выполняю команды команды.
Понимаю язык Си.
Достаточно обо мне - Как вас зовут? Найджел Барнтвит
Хорошо, Найджел Барнтвит, у вас, наверное, много талантов.
Расскажите о некоторых.
Только ограничтесь одной строкой.
Если вы не можете вспомнить что-нибудь, придумайте.
Фехтование, пение тирольских песен, симуляция, дегустация сыра.
Давайте посмотрим, получил ли я этот перечень.
Фехтование, пение тирольских песен, симуляция, дегустация сыра.
Спасибо за информацию, Найджел Барнтвит.
```

Тщательно исследуем программу. Но вместо того чтобы просматривать строку за строкой, применим более общий подход. Сначала рассмотрим способы определения строк в программе. Затем выясним, что нужно для чтения строки в программе. И наконец, изучим способы вывода строки.

ОПРЕДЕЛЕНИЕ СТРОК В ПРОГРАММЕ

Далее Содержание

Вы, вероятно, заметили, когда читали программу, что есть много способов определения строк. Попытаемся теперь рассмотреть основные: использование строковых констант, массивов типа **char**, указателей на тип **char** и массивов, состоящих из символьных строк. В программе должно быть предусмотрено выделение памяти для запоминания строки, и мы еще вернемся к этому вопросу.

Строковые константы

Далее Содержание

Всякий раз, когда компилятор встречается с чем-то, заключенным в двойные кавычки, он определяет это как строковую константу. Символы, заключенные в кавычки, плюс завершающий символ **'\0'**, записываются в последовательные ячейки памяти. Компилятор подсчитывает количество символов, поскольку ему нужно знать размер памяти, необходимой для запоминания строки. Наша программа использует несколько таких строковых констант, чаще всего в качестве аргументов функций **printf( )** и **puts( )**. Заметим также, что мы можем определять строковые константы при помощи директивы **#define**.

Если вы хотите включить в строку символ двойной кавычки, ему должен

предшествовать символ обратной дробной черты:

```
printf("\"Бегн, Спот, беги!\" - сказал Дик.\n");
```

В результате работы этого оператора будет напечатана строка:

```
"Беги, Спот, беги! - "сказал Дик.
```

Строковые константы размещаются в *статической памяти*. Вся фраза в кавычках является указателем на место в памяти, где записана строка. Это аналогично использованию имени массива, служащего указателем на расположение массива. Если это действительно так, то как выглядит оператор, который выводит строку?

```
/* строки в качестве указателей */
main( )
{
printf("%s, %u, %c \n", "We", "love", *"figs");
```

Итак, формат **%s** выводит строку **We**. Формат **%u** выводит целое без знака. Если слово **"love"** является указателем, то выдается его значение, являющееся адресом первого символа строки. Наконец, ***"figs"** должно выдать значение, на которое ссылается адрес, т. е. первый символ строки **"figs"**. Произойдет ли это на самом деле? Да, мы получим следующий текст:

```
We, 34, f
```

Ну, вот! Давайте теперь вернемся к строкам, находящимся в *символьных массивах*.

Массивы символьных строк и их инициализация

Далее Содержание

При определении массива символьных строк необходимо сообщить компилятору требуемый размер памяти. Один из способов сделать это - инициализировать массив при помощи строковой константы. Так как *автоматические массивы* нельзя инициализировать, необходимо для этого использовать *статические* или *внешние массивы*. Например, оператор

```
char m1[ ] = "Только ограничьтесь одной строкой.";
```

инициализировал внешний (по умолчанию) массив **m1** для указанной строки. Этот вид инициализации является краткой формой стандартной инициализации массива

```
char m1[ ] = {'Т', 'о', 'л', 'ь', 'к', 'о', ' ',
                   'о', 'г', 'р', 'а', 'н', 'и', 'ч',
                   'ь', 'т', 'е', 'с', 'ь', ' ', 'о',
                   'д', 'н', 'о', 'й', ' ', 'с', 'т',
                   'р' 'о', 'к', 'о', 'й', ' .', '\0'};
```

(Обратите внимание на замыкающий нуль-символ. Без него мы имеем массив символов, а не строку.) Для той и другой формы (а мы рекомендуем первую) компилятор подсчитывает символы и таким образом получает размер массива.

Как и для других массивов, имя **m1** является указателем на первый элемент массива:

m1 == &m1[0], *m1 == 'Т', и *(m1 + l) == m1[1] == 'о',

Действительно, мы можем использовать указатель для создания строки. Например:

char *m3 = " \n Достаточно обо мне - как вас зовут?";

Это почти то же самое, что и

static char m3[ ] = "\n Достаточно обо мне - как вас зовут?" ;

Оба описания говорят об одном: **m3** является указателем строки со словами " Как вас зовут?" . В том и другом случае сама строка определяет размер памяти, необходимой для ее размещения. Однако вид их не идентичен.

Массив или указатель

Далее Содержание

В чем же тогда разница между этими двумя описаниями? Описание с массивом вызывает создание в статической памяти массива из 38 элементов (по одному на каждый символ плюс один на завершающий символ '**\0**'. Каждый элемент инициализируется соответствующим символом. В дальнейшем компилятор будет рассматривать имя **m3** как синоним адреса первого элемента массива, т. е. &**m3[0]**. Следует отметить, что **m3** является *константой* указателя. Вы не можете изменить **m3**, так как это означало бы изменение положения (адрес) массива в памяти. Можно использовать операции, подобные **m3+1**, для идентификации следующего элемента массива, однако не разрешается выражение **++m3**. Оператор увеличения можно использовать с именами переменных, но не констант.

Форма с указателем также вызывает создание в статической памяти 38 элементов для запоминания строки. Но, кроме того, выделяется еще одна ячейка памяти для *переменной* **m3**, являющейся указателем. Сначала эта переменная указывает на начало строки, но ее значение может изменяться. Поэтому мы можем использовать операцию увеличения; **++m3** будет указывать на второй символ строки (**Д**). Заметим, что мы не объявили ***m3** статической неременной, потому что мы инициализировали не массив из 38 элементов, а одну переменную типа указатель. Не существует ограничений на класс памяти при инициализации обычных переменных, не являющихся массивом.

Существенны ли эти отличия? Чаще всего нет, но все зависит от того, что вы пытаетесь делать. Посмотрите несколько примеров, а мы возвращаемся к вопросу выделения памяти для строк.

Массив и указатель: различия

В нижеследующем тексте мы обсудим различия в использовании описаний этих двух видов:

```
static char heart[ ] ="Я люблю Тилли !";
char *head ="Я люблю Милли!";
```

Основное отличие состоит в том, что указатель **heart** является константой, в то время как указатель **head** - переменной. Посмотрим, что на самом деле даст эта разница.

Во-первых, и в том и в другом случае можно использовать операцию сложения с указателем.

```
for(i = 0; i < 6; i++ )
putchar(*(heart + i));
putchar('\n');
for(i = 0; i < 6; i++ )
putchar(*(head + i));
putchar('\n');
```

в результате получаем

```
я люблю я люблю
```

Но только в случае с указателем можно использовать операцию увеличения:

```
while( *(head) != '\0') /* останов и конце строки */
putchar(*(head++ )); /* печать символа и перемещение указателя */
```

дают в результате:

```
я люблю милли!
```

Предположим, мы хотим заменить **head** на **heart**. Мы можем сказать

```
head = heart /* теперь head указывает на массив heart */
```

но теперь мы можем сказать

```
heart =  head; /* запрещенная конструкция */
```

Ситуация аналогична **x = 3** или **3 = x;** левая часть оператора присваивания должна быть именем переменной. В данном случае **head = heart;** не уничтожит строку **Милли**, а только изменит адрес, записанный в **head**. Вот каким путем можно изменить обращение к **heart** и проникнуть в сам массив:

```
heart[8] = 'М';
```

или

```
*(heart + 8) = 'М';
```

Элементы массива (но не *имя*) являются переменными.

Явное задание размера памяти

Далее Содержание

Иной путь выделения памяти заключается в явном ее задании. Во внешнем описании мы могли бы скачать:

```
char m1[44] = "Только ограничьтесь одной строкой.";
```

вместо

```
char m1[ ] = "Только ограничьтесь одной строкой.";
```

Можно быть уверенным, что число элементов по крайней мере на один (это снова нуль-символ) больше, чем длина строки. Как и в других статических или внешних массивах, любые неиспользованные элементы автоматически инициализируются нулем (который в символьном виде является нуль-символом, а не символом цифры нуль).

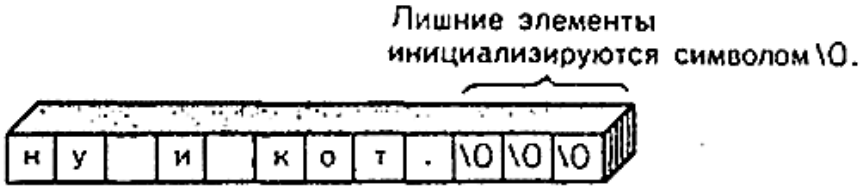


РИС. 13.2. Инициализация массива.

Отметим, что в нашей программе массиву **name** задан размер:

char name [81];

Поскольку массив **name** должен читаться во время работы программы, у компилятора нет другого способа узнать заранее, сколько памяти нужно выделить для массива. Это нс символьная константа, в которой компилятор может посчитать символы. Поэтому мы предположили, что 80 символов будет достаточно, чтобы поместить в массив фамилию пользователя.

Массивы символьных строк

Далее Содержание

Обычно бывает удобно иметь массив символьных строк. В этом случае можно использовать индекс для доступа к нескольким разным строкам. Покажем это на примере:

```
static char *mytal[LIM] = {"Быстро складываю числа",
                           "Точно умножаю",
                           "Записываю данные",
                           "Правильно выполняю команды",
                           "Понимаю язык Си"};
```

Разберемся в этом описании. Вспомним, что **LIM** имеет значение 5, мы можем сказать, что **mytal** является массивом, состоящим из пяти указателей на символьные строки. Каждая строка символов, конечно же, представляет собой символьный массив, поэтому у нас есть пять указателей на массивы. Первым указателем является **mytal[0]**, и он ссылается на первую строку. Второй указатель **mytal[1]** ссылается на вторую строку. Каждый указатель, в частности, ссылается на первый символ своей строки:

```
*mytal[0] == 'Б', *mytal[1] == 'Т', mytal[2] == 'З'
```

и т. д.

Инициализация выполняется по правилам, определенным для массивов. Тексты в кавычках эквивалентны скобочной записи
{{...}, {...}, ..., {...}};

где многоточия подразумевают тексты, которые мы поленились напечатать. В первую очередь мы хотим отметить, что первая последовательность, заключенная в двойные кавычки, соответствует первым парным скобкам и используется для инициализации первого указателя символьной строки. Следующая последовательность в двойных кавычках инициализирует второй указатель и т. д. Запятая разделяет соседние последовательности.

Кроме того, мы могли бы явно задавать размер строк символов, используя описание, подобное такому:

```
static char mytal[LIM][LINLIM];
```

Разница заключается в том, что второй индекс задает "прямоугольный" массив, в котором все "ряды" (строки) имеют одинаковую длину. Описание

```
static char *mytal [LIM]
```

однако, определяет "рваный" массив, где длина каждого "ряда" определяется той строкой, которая этот "ряд" инициализировала. Рваный массив не тратит память напрасно.

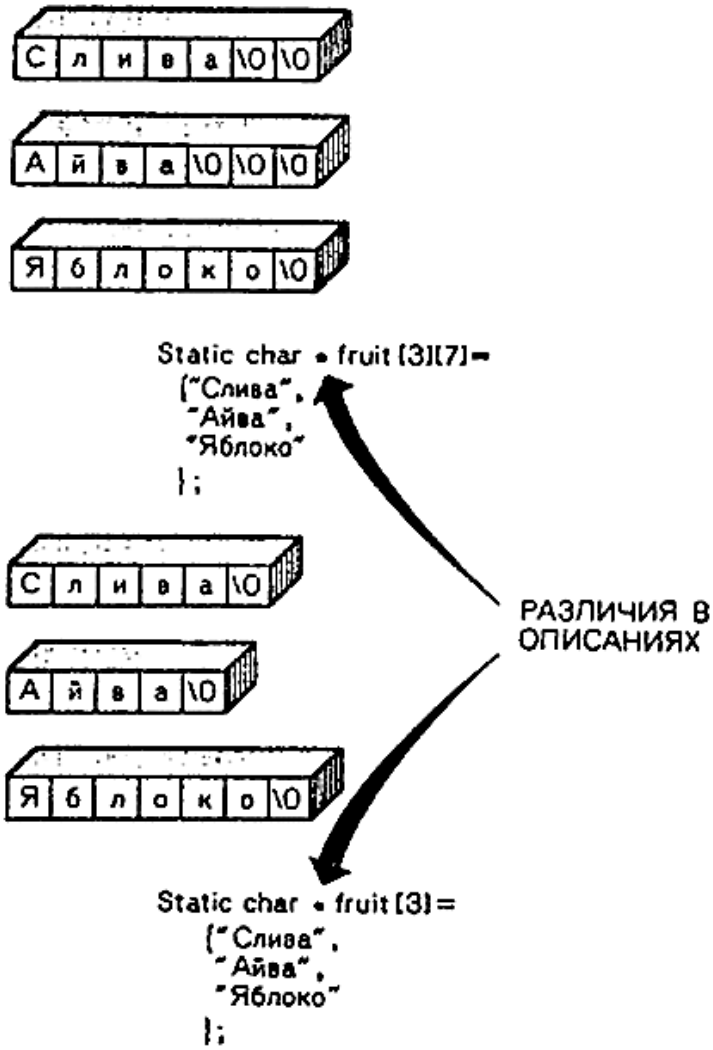


РИС. 13.3. Прямоугольный массив или рваный.

Указатели и строки

Далее Содержание

Возможно, вы заметили периодическое упоминание указателей в нашем рассказе о строках. Большинство операции языка Си, имеющих дело со строками, работает с указателями. Например, рассмотрим приведенную ниже бесполезную, но поучительную программу

```
/* указатели и строки */
#define PX(X) printf("X = %s; значение = %u; &X = %u\n", X, X, &X)
main( ) {
static char *mesg = "Не делай глупостей!";
static char *copy;
copy = mesg;
printf(" %s \n" , copy);
PX(mesg);
PX(copy);
}
```

Взглянув на эту программу, вы можете подумать, что она копирует строку "Не делай глупостей!", и при беглом взгляде на вывод вам может показаться правильным это

предположение:

```
Не делай глупостей!
mesg = Не делай глупостей!;  значение = 14; &mesg  = 32
copy = Не делай глупостей!; значение = 14; &copy = 34
```

Но изучим вывод **PX()**. Сначала **X**, который последовательно является **mesg** и **copy**, печатается как строка (**%s**). Здесь нет сюрприза. Все строки содержат "Не делай глупостей!".

Далее ... вернемся к этому несколько позднее.

Третьим элементом в каждой строке является **&X**, т. е. адрес **X**. Указатели **mesg** и **copy** записаны в ячейках 32 и 34 соответственно.

Теперь о втором элементе, который мы называем *значением*. Это сам **X**. Значением указателя является адрес, который он содержит. Мы видим, что **mesg** ссылается на ячейку 14, и поэтому выполняется **copy**.

Смысл заключается в том, что сама строка никогда не копируется. Оператор **copy=mesg;** создаст второй указатель, ссылающийся на ту же самую строку.

Зачем все эти предосторожности? Почему бы не скопировать всю строку? Хороню, а что эффективнее - копировать один адрес или, скажем, 50 отдельных элементов ? Часто бывает, что адрес это все, что необходимо для выполнения работы.

Теперь, когда мы обсудили определение строк в программе, давайте займемся вводом строк.

ВВОД СТРОК

Далее Содержание

Процесс ввода строки выполняется за два шага: выделение памяти для запоминания строки и применение функции ввода для получения строки.

Выделение памяти

Далее Содержание

Сначала следует определить место для размещения строки при вводе. Как было отмечено раньше, это значит, выделить память, достаточную для размещения любых строк, которые мы предполагаем читать. Не следует надеяться, что компьютер подсчитает длину строки при ее вводе, а затем выделит для нес память. Он не будет этого делать (если только вы не напишите программу, которая должна это выполнять). Если вы попытаетесь сделать что-то подобное

```
static char *name;
scanf(" %s", name);
```

компилятор, вероятно, выполнит нужные действия. Но при вводе имя будет записываться на данные или текст вашей программы. Большинство программистов считает это очень забавным, но только в чужих программах. Проще всего включить в описание явный размер массива:

```
char name[81];
```

Можно также использовать библиотечные функции языка Си, которые распределяют намять, и мы рассмотрим их в гл. 15.

В нашей программе для **name** использовался *автоматический массив*. Мы смогли это сделать, потому что не требовалось инициализации массива.

Как только выделена память для массива, можно считывать строку. Мы уже упоминали, что программы ввода не являются частью языка. Однако большинство систем имеют две библиотечные функции **scanf( )** и **gets( )**, которые могут считывать строки. Чаще всего используется функция **gets( )**, поэтому мы вначале расскажем о ней.

Функция gets( )

Далее Содержание

Эта функция считывания строки очень удобна для диалоговых систем. Она получает строку от стандартного устройства ввода вашей системы, которым, как мы предполагаем, является клавиатура. Поскольку строка не имеет заранее заданной длины, функция **gets( )** должна знать, когда ей прекратить работу. Функция читает символы до тех пор, пока ей не встретится символ новой строки (**'\n'**), который вы создаете, нажимая клавишу [**ввод**]. Функция берет все символы до (но не включая) символа новой строки, присоединяет к ним нуль-символ (**'\0'**) и передает строку вызывающей программе. Вот простой способ использования функции.

```
/* получение имени1 */
main( )
{
char name[81]; /* выделение памяти */
printf(" Привет, как вас зовут?\n");
gets(name);  /* размещение введенного имени в строку "name" */
printf(" Хорошее имя, %s. \n" , name);
}
```

Функция примет любое имя (включая пробелы) длиной до 80 символов. (Не забудьте запасти один символ для **'\0'**.)

Отметим, что мы хотели при помощи функции **gets( )** воздействовать на нечто (**name**) в вызывающей программе. Значит, нужно использовать указатель в качестве аргумента; а имя массива, конечно, *является* его указателем.

Функция **gets( )** обладает большими возможностями, чем показано в последнем примере. Взгляните на эту программу:

```
/* получение имени2 */
main( )
{
char name [80];
char *ptr, *gets( );

printf(" Привет, как вас зовут?\n");
ptr = gets(name);
printf(" %s? Ах! %s!\n", name, ptr);
}
```

Получился диалог:

```
Привет, как вас зовут?
Тони де Туна
Тони де Туна? Ах! Тони де Туна!
```

Функция **gets( )** предоставляет вам два способа ввода строки!

1. Использует метод указателей для передачи строки в **name**.
2. Использует ключевое слово **return** для возврата строки в **ptr**.

Напомним, что **ptr** является указателем на тип **char**. Это означает, что **gets( )** должна вернуть значение, которое является указателем на тип **char**. И в приведенном выше изложении вы можете увидеть, что мы так и описали **gets( )**.

Описание вида

```
char *gets( );
```

говорит о том, что **gets( )** является функцией (отсюда круглые скобки) типа "указатель на тип char" (поэтому ***** и **char**). В примере **получение имени1** мы обходились без этого описания, потому что мы никогда не пытались использовать возвращенное значение функции **gets( )**.

Между прочим, вы можете также описать указатель на функцию. Это выглядело бы следующим образом:

```
char (*foop)( );
```

и **foop** был бы указателем на функцию типа **char**. Мы расскажем немного подробнее о таких причудливых описаниях в гл. 14.

Структура функции **gets( )** выглядела бы примерно так:

```
char *gets(s);
char *s;
{
char *p;
return(p);
}
```

На самом деле структура немного сложнее, и для **gets( )** есть две возможности возврата. Если все идет хорошо, она возвращает считанную строку, как мы уже сказали. Если что-то неправильно или если **gets( )** встречает символ **EOF**, она возвращает **NULL**, или нулевой адрес. Таким образом **gets( )** включает разряд проверки ошибки. Поэтому данная функция удобна для использования в конструкциях, подобных

```
while(gets(name) != NULL)
```

где **NULL** определен в файле **stdio.h** как **0**. При помощи указателя массиву name присваивается значение. Наличие **возврата** позволяет присваивать значение всей **gets(name)** и выполнять проверку на **EOF**. Этот двоякий подход более компактен, чем использование функции **getchar( )**, которая имеет **возврат** без аргумента.

```
while((ch = getchar( )) != EOF)
```

Функция scanf( )

Далее Содержание

Мы уже использовали ранее функцию **scanf( )** и формат **%s** для считывания строки. Основное различие между **scanf( )** и **gets( )** заключается в том, как они определяют, что достигли конца строки: **scanf( )** предназначена скорее для получения слова, а не строки. Функция **gets( )**, как мы уже видели, принимает все символы до тех пор, пока нс встретит первый символ "новая строка". Функция **scanf( )** имеет два варианта. Для любого из них

строка начинается с первого встретившегося непустого символа. Если вы используете формат **%s**, строка продолжается до (но не включая) следующего пустого символа (пробел, табуляция или новая строка). Если вы определяете размер поля как **%10s**, то функция **scanf( )** считает не более 10 символов или же считает до любого пришедшего первым пустого символа.

Функция **scanf( )** возвращает целое значение, равное числу счи танных символов, если ввод прошел успению, или символ **EOF**, ее ли он встретился.

```
/* scanf( ) и подсчет количества */
main( )
{
static char name1[40], name2[11];
int count;

printf(" Введите, пожалуйста, 2 имени.\n");
count = scanf(" %s  %10s", name1, name2);
printf(" Я считал %d имен %s и %s.\n", count, name1, name2);
}
```

Вот два примера работы программы:

```
Введите, пожалуйста, два имени.
Джсссика  Джукс.
Я считал два имени
Джсссика  и  Джукс.
Введите, пожалуйста, 2 имени.
Лиза Апплеботтхэм
Я считал 2 имени Лиза и Апплеботтхэм.
```

Во втором примере были считаны только первые 10 символов от Апплеботтхэм, так как мы использовали формат **%10s**.

Если вы получаете только текст с клавиатуры, лучше применять, функцию **gets( )**. Она проще в использовании, быстрее и более компактна. Функция **scanf( )** предназначена в основном для ввода смеси типов данных в некоторой стандартной форме. Например, если каждая вводимая строка содержит наименование инструмента, количество его на складе и стоимость каждого инструмента, вы можете использовать функцию **scanf( )** или можете создать свою собственную функцию, которая выполняет проверку некоторых ошибок при вводе.
Теперь давайте рассмотрим процесс вывода строк.

ВЫВОД СТРОК

Далее Содержание

Опять мы должны полагаться на библиотечные функции, которые могут немного изменяться от системы к системе. Функции **puts( )** и **printf( )** - две рабочие лошадки, используемые при выводе строк.

Функция puts( )

Далее Содержание

Это очень простая функция; у нее есть только один аргумент, являющийся указателем строки. Нижеследующий пример иллюстрирует некоторые из многих способов ее применения.

```
/* простые выдачи */
#include <stdio.h>
#define DEF  "Я строка #define."
```

```
main( )
{
static char str1[ ] = "Массив инициализирован мной.";
static char *str2 = "Указатель инициализирован мной.";

puts(" Я аргумент функции puts( )." );
puts(DEF);
puts(str1);
puts(str2);
puts(&str1[4]);
puts(str2 + 4);
}
```

В результате работы программы получаем

```
Я аргумент функции puts( ).
Я строка #define.
Массив инициализирован мной.
Указатель инициализирован мной.
ив инициализирован мной.
атель инициализирован мной.
```

Этот пример напоминает нам, что фразы в кавычках и имена строк символьных массивов являются указателями. Обратите внимание на два последних оператора. Указатель **&strl[4]** ссылается на пятый элемент массива **str1**. Этот элемент содержит символ 'и', и функция **puts( )** использует его в качестве начальной точки. Аналогично **str2 + 4** ссылается на ячейку памяти, содержащую 'а' в "указателе", и с нее начинается вывод строки.

Как **puts( )** узнает, когда остановиться? Она прекращает работу, если встречает нуль-символ, поэтому лучше, чтобы он был. Не пытайтесь делать так!

```
/* нет строки! */
main( )
{
static char dont[ ] = (' Н', ' Г , ' ! ', ' ! ');
puts(dont);  /* dont не является строкой */
}
```

Поскольку в **dont** отсутствует завершающий нуль-символ, она не является строкой. Так как нуль-символ отсутствует, **puts( )** не знает, когда ей останавливаться. Она будет просто перебирать ячейки памяти, следующие за **dont** до тех пор, пока не найдет где-нибудь нуль-символ. Если повезет, она, может быть, найдет его в ближайшей ячейке, но может и не повезти.

Обратите внимание, что любая строка, вводимая функцией **puts( )**, начинается с новой строки. Если **puts( )** в конце концов находит завершающий нуль-символ, она заменяет его символом "новой строки" и затем выводит строку.

Функция printf( )

Далее Содержание

Мы уже обсуждали функцию **printf( )** довольно основательно. Подобно **puts( )**, она использует указатель строки в качестве аргумента. Функция **printf( )** менее удобна, чем **puts( )**, но более гибка.

Разница заключается в том, что **printf( )** не выводит автоматически каждую строку текста с новой строки. Вы должны указать, что хотите выводить с новых строк. Так,

```
printf(" %s\n" , string);
```

дает то же самое, что и

```
puts(string);
```

Вы можете видеть, что первый оператор требует ввода большего числа символов и большего времени при выполнении на компьютере. С другой стороны, **printf( )** позволяет легко объединять строки для печати их в одной строке. Например:

```
printf(" Хорошо, %s, %s \n", name, MSG);
```

объединяет " Хорошо" с именем пользователя и с символьной строкой **MSG** в одну строку.

СОЗДАНИЕ СОБСТВЕННЫХ ФУНКЦИЙ

Далее Содержание

Не ограничивайте себя при вводе и выводе только этими библиотечными функциями. Если у вас нет нужной функции, или она вам не нравится, можно создавать свои собственные версии, используя для этого **getchar( )** и **putchar( )**.

Предположим, у вас нет функции **puts( )**. Вот один из путей ее создания:

```
/* put1  - печатает строку */
put1(string);
char *string;
{
while(*string != '\0') putchar(*string++);
putchar('\n');
}
```

Символьный указатель **string** вначале ссылается на первый элемент вызванного аргумента. После печати его содержимого указатель увеличивается и ссылается уже на следующий элемент. Это продолжается до тех пор, пока указатель не дойдет до элемента, содержащего нуль-символ. Затем в конце строки будет поставлен символ новой строки.

Предположим, у вас есть **puts( )**, но вам нужна функция, которая, кроме того, сообщает, сколько напечатано символов. Эту возможность легко добавить:

```
/* put2- - печатает строку и считывает символы */
put2 (string);
char *string;
{
int count = 0;
while(*string != '\0') {
putchar(* string++);
count++;
putchar('\n');
return(count);
}
```

Вызов:

```
put2(" пицца" );
```

печатает строку пицца, в то время как оператор

```
num = puts(" пицца");
```

передаст, кроме того, количество символов в **num;** в данном случае это число 5. Вот несколько более сложный вариант, показывающий вложенные функции:

```
/* вложенные функции */
#include <stdio.h>
main( )
{
put1("Если бы я имел столько денег, сколько могу потратить,");
printf("Я считаю %d символа.\n",
put2(" Я никогда бы не жаловался, что приходится чинить старые стулья.");
}
```

(Мы включили в программу при помощи директивы **#include** файл **stdio.h**, потому что в нашей системе в нем определена функция **putchar( )**, а она используется в нашей новой функции.)

Да-а, мы используем функцию **printf( )** для печати значения **put2( )**, но в процессе нахождения значения **put2( )** компьютер должен сначала заставить ее поработать - напечатать строку. Вот что получается при этом:

```
Если бы я имел столько денег, сколько могу потратить,
Я никогда бы не жаловался, что приходится чинить старые стулья.
Я считаю 63 символа.
```

Теперь вы можете построить работающую версию функции **gets( );** она должна быть похожа на нашу функцию **getint( )** из гл. 10, но гораздо проще ее.

ФУНКЦИИ, РАБОТАЮЩИЕ СО СТРОКАМИ

Далее Содержание

Большинство библиотек языка Си снабжено функциями, работающими со строками. Рассмотрим четыре наиболее полезных и распространенных: **strlen( ), strcat( ), strcmp( ) и strcpy( )**.

Мы уже применяли функцию **strlen( )**, которая находит длину строки. Используем ее в нижеследующем примере функции, укорачивающей длинные строки.

Функция strlen( )

Далее Содержание

```
/* Функция Прокруста */
fit(string, size)
char *string;
int size;
{
if(strlen(string) > size)
    *(string + size) = '\0';
}
```

Проверьте ее в "деле" в этой тестовой программе:

```
/* тест */
main( ) {
static char mesg[ ] = "Ну, теперь держитесь, компьютероманы.";

puts(mesg);
fit(mesg, 10);
puts(mesg);
}
```

Программа выдает:

```
Ну, теперь держитесь, компьютероманы.
```

```
Ну, теперь
```

Наша функция помещает символ **'\0'** в одиннадцатый элемент массива, заменяя символ пробела. Остаток массива остается на старом месте, но **puts( )** прекращает работу на первом нуль-символе и игнорирует остаток массива.

Функция strcat( )

Далее Содержание

Вот что умеет делать функция **strcat( )**:

```
/* объединение двух строк */
#include
< stdio.h>
main( )
{
static char flower [80];
static char addon[  ] = "ы пахнут старыми ботинками.";
puts(" Назовите ваш любимый цветок." );
gets(flower);
strcat (flower, addon);
puts(flower);
puts(addon);
}
```

Получаем на экране:

```
Назовите ваш любимый цветок.
Ирис
Ирисы пахнут старыми ботинками.
ы пахнут старыми ботинками.
```

Очевидно, что **strcat( )** (*string concatenation*) использует в качестве аргументов две строки. Копия второй строки присоединяется к концу первой, и это объединение становится новой первой строкой. Вторая строка не изменяется.

Внимание! Эта функция не проверяет, умещается ли вторая строка в первом массиве. Если вы ошиблись при выделении памяти для первого массива, то у вас возникнут проблемы. Конечно, можно использовать *strlen( )* для определения размера строки до объединения.

```
/* Объединение двух строк, проверка размера первой */
#include <stdio.h>
#define SIZE 80
main( )
{
static char flower[SIZE];
static char addon[ ] = " ы пахнут старыми ботинками." ;
puts(" Назовите ваш любимый цветок. ");
gets(flower);
if((strlen(addon) + strlen(flower) + 1) < SIZE)
    strcat (flower, addon);
puts(flower);
}
```

Мы добавляем 1 к объединенной длине для размещения нуль-символа.

Функция strcmp( )

Далее Содержание

Предположим, что вы хотите сравнить чей-то ответ со строкой, находящейся в

памяти:

```
/* Будет ли это работать? */
#include <stdio.h>
#define ANSWER  " Грант"
main( )
{
char try [40];
puts(" Кто похоронен в могиле Гранта?" );
gets(try);
while(try != ANSWER)
puts(" Нет, неверно. Попытайтесь еще раз." );
gets(try);
} puts(" Правильно.");
}
```

Хотя эта программа и смотрится неплохо, она не будет работать правильно, **try** и **ANSWER** на самом деле являются указателями, поэтому сравнение **(try != ANSWER)** спрашивает не о том, одинаковы ли эти две строки, а одинаковы ли два адреса, на которые ссылаются **try** и **ANSWER**. Так как **ANSWER** и **try** запоминаются в разных ячейках, эти два указателя никогда не могут быть одним и тем же, и пользователю всегда сообщается, что программа неверна. Такие программы обескураживают людей.

Нам нужна функция, которая сравнивает содержимое строк, а не их адреса. Можно было бы придумать ее, но это уже сделала за нас функция **strcmp( )** (*string comparision*).

Теперь исправим нашу программу:

```
/* это будет работать */
#include <stdio.h>
#define ANSWER " Грант"
main( )
{
char try [40];
puts(" Кто похоронен в могиле Гранта?" );
gets(try);
while(strcmp(try, ANSWER) != 0)
{ puts(" Нет, неверно. Попытайтесь еще раз.");
gets(try);
} puts(" Правильно!");
}
```

Так как ненулевые значения интерпретируются всегда как **"true"**, мы можем сократить оператор **while do while(strcmp(try, ANSWER)).**

Из этого примера можно сделать вывод, что **strcmp( )** использует два указателя строк в качестве аргументов и возвращает значение 0, если эти две строки одинаковы. Прекрасно, если вы придете к такому выводу.

Хорошо, что **Strcmp( )** сравнивает строки, а не массивы. Поэтому, хотя массив **try** занимает 40 ячеек памяти, а " Грант" - только 6 (не забывайте, что одна нужна для нуль-символа), сравнение выполняется только с частью **try**, до его первого нуль-символа. Такую функцию **strcmp( )** можно использовать для сравнения строк, находящихся в массивах разной длины.

А что если пользователь ответил " ГРАНТ" или " грант" или "Улиссес С. Грант" ? Хорошо, если пользователю сказали, что он ошибся? Чтобы сделать программу гибкой, вы должны предусмотреть несколько допустимых правильных ответов. Здесь есть некоторые тонкости. Вы могли бы в операторе **#define** определить в качестве ответа " ГРАНТ" и написать функцию, которая превращает любой ответ только в это слово. Это

устраняет проблему накопления, но остаются другие поводы для беспокойства.

Между прочим, какое значение возвращает **strcmp( )**, если строки не одинаковы? Вот пример:

```
/* возвраты функции strcmp */
#include <a: stdio.h>
main( )
{
printf(" %d \n" , strcmp( "A" , " A" ));
printf(" %d \n" , strcmp( "A" , " B" ));
printf(" %d \n" , strcmp( "B" , " A" ));
printf(" %d \n" , strcmp( "C" , "A" ));
printf(" %d \n" , strcmp(" apples", " apple"));
}
```

В результате получаем

```
0 -1
1
2 115
```

Как мы и предполагали, сравнение **"A"** с самим собой возвращает 0. Сравнение **"A"** с **"B"** дает -1, а **"B"** с **"A"** дает 1. Это наводит на мысль, что **strcmp( )** возвращает отрицательное число, если первая строка предшествует второй в алфавитном порядке, или положительное число, если порядок иной. Кроме того, сравнение **"C"** с **"A"** дает 2 вместо 1. Картина проясняется: функция возвращает разницу между двумя символами в коде ASCII. В более общем смысле **strcmp()** передвигается вдоль строк до тех пор, пока не находит первую пару несовпадающих символов; затем она возвращает разницу в кодах ASCII. Например, в самом последнем примере **"apples"** и **"apple"** совпадают, кроме последнего символа **'s'**, в первой строке. Он сопоставляется с шестым символом в **"apple"**, который является нуль-символом (0 в ASCII).

Возвращается значение

```
's' - '\0' = 115 - 0 = 115,
```

где 115 является кодом буквы **'s'** в ASCII.

Обычно вам не нужно точно знать возвращаемое значение. Чаще всего вы только хотите знать, нуль это или нет, т. е. было ли совпадение. Или, может быть, вы пытаетесь отсортировать строки в алфавитном порядке и хотите узнать, в каком случае сравнение дает положительный, отрицательный или нулевой результат.

Можно использовать эту функцию, чтобы проверить, остановится ли программа, читая вводимую информацию:

```
/* Начало какой-то программы */
#include &stdio.h&
#define SIZE 81
#define LIM 100
#define STOP   "  "  /* нулевая строка */
main( )
{
static char input[LIM][SIZE];
int ct = 0;

while(gets(input[ct]) != NUL  && strcmp(input[ct].STOP) !=0
                              && ct++ < LIM)
...
}
```

Программа прекращает чтение вводимой строки, если встречает символ **EOF** [в этом случае **gets( )** возвращает **NULL**], или если вы нажимаете клавишу [**ввод**] в начале строки (т.е. введете пустую строку), или если вы достигли предела **LIM**. Чтение пустой строки дает пользователю простой способ прекращения ввода.
Давайте перейдем к последней из обсуждаемых нами функций, работающих со строками.

Функция strcpy( )

Далее Содержание

Мы уже говорили, что если **pts1** и **pts2** являются указателями строк, то выражение

```
pts2 = ptsl;
```

копирует только адрес строки, а не саму строку. Предположим, что вы все же хотите скопировать строку. В этом случае можно использовать функцию **strcpy( )**. Она работает примерно так:

```
/* демонстрация strcpy( ) */
#include <stdio.h>
#define WORDS  "Проверьте, пожалуйста, вашу последнюю чапись."
main( ) {
static char *orig = WORDS;
static char copy [40];

puts(orig);
puts(copy);
strcpy(copy, orig);
puts(orig);
puts(copy);
}
```

Вот результат:

```
Проверьте, пожалуйста, вашу последнюю запись.
Проперьте, пожалуйста, пашу последнюю запись.
Проверьте. пожалуйста, пашу последнюю запись.
```

Очевидно, что строка, на которую указывает второй аргумент (**orig**) функции **strcpy( )**, скопирована в массив, на который указывает первый аргумент (**copy**). Порядок аргументов функции такой же, как в операторе присваивания: строка, получающая значение, стоит слева. (Пустая строка является результатом печати массива **copy** до копирования, и она говорит о том, что статические массивы .инициализируются нулями, т. е. нуль-символами в символьном виде.)

Нужно обеспечить, чтобы размер массива, принимающего строку, был достаточен для ее размещения. Поэтому мы используем описание

```
static char copy [40];
```

а не

```
static char *copy; /* не выделяет память для строки */
```

Короче говоря, **strcpy()** требует два указателя строк в качестве аргументов. Второй указатель, ссылающийся на исходную строку, может быть объявленным указателем, именем массива или строковой константой. А первый указатель, ссылающийся на копию, должен ссылаться на массив или часть массива, имеющего размер, достаточный для размещения строки.

Теперь, когда мы описали несколько функций, работающих со строками, рассмотрим целую программу, работающую со строками.

ПРИМЕР: СОРТИРОВКА СТРОК

Далее Содержание

Возьмем реальную задачу сортировки строк в алфавитном порядке. Эта задача может возникнуть при подготовке списка фамилий, при создании алфавитного указателя и во многих других ситуациях. В такой программе одним из главных инструментов является функция **strcmp( )**, так как ее можно использовать для определения старшинства двух строк. Последовательность наших действий будет состоять из считывания массива строк, их сортировки и последующего вывода. Совсем недавно мы показали последовательность действий для считывания строк, и сейчас мы начнем программу таким же образом.

```
/* считывает строки и сортирует их */
#include <stdio.h>
#define SIZE 81 /* предельная длина строки, включая \0 */
#define LIM 20 /* максимальное количество считываемых строк */
#define HALT   " " /* нулевая строка для прекращения ввода */
main( )
{
static char input[LIM][SIZE]; /* массив для запоминания вводимых строк */
char *ptstr[LIM];  /* массив переменных типа указатель */
int ct = 0; /* счетчик вводимых строк */
int k;  /* счетчик выводимых строк */
printf(" Введите до %d строк и я их отсортирую.\n" , LIM);
printf(" Для прекращения ввода нажмите клавишу [ввод] в начале строки.\n");
while((gets(input[ct])!= NULL) && strcmp(input[ct], HALT)
                                != 0 && ct++ < LIM)
    ptstr[ct - 1] = input[ct - 1];  /*указывает на еще не
                                     отсортированный ввод */
stsrt(ptstr, ct); /* сортировка строк */
puts(" \n Вот отсортированный список строк:\n");
for(k = 0; k < ct; k++)
puts(ptstr[k]); /* указатели на отсортированные строки */
}

/* функция сортировки-строк-с-использованием-указателей */
stsrt(strings, num)
char *strings[  ];
int num;
{ char *temp;
int top, seek;
for(top = 0; top < num-1; top++)
for(seek = top + 1; seek < num; seek++)
if(strcmp(strings[top], strings[seek]) > 0)
{ temp = strings [top];
strings [top] = strings [seek];
strings [seek] = temp;
} }
```

РИС. 13.4. Программа чтения и сортировки строк.

Вывод строк на печать не составляет проблемы, а для сортировки можно взять тот же алгоритм, который использовался раньше для чисел. Сейчас мы применим один хитрый трюк: посмотрим, сможете ли вы его заметить.
Для проверки возьмем детский стишок.

```
Введите 20 строк, и я их отсортирую.
Для прекращения ввода нажмите клавишу [ввод] в начале строки.
```

```
Жил на свете человек
Скрюченные ножки
И гулял он целый век
По скрюченной дорожке
```

Вот отсортированный список строк

```
Жил на свете человек
И гулял он целый век
По скрюченной дорожке
Скрюченные ножки
```

Детские стишки не кажутся слишком искаженными после сортировки их по алфавиту.

Трюк состоит в том что вместо перегруппировки самих строк мы перегруппировали их *указатели*. Разберемся в этом. В начале **ptrst[0]** ссылается на **input[0]** и т. д. Каждый **input[ ]** является массивом из 81 элемента, а каждый элемент **ptrst[ ]** является отдельной переменной. Процедура сортировки перегруппировывает **ptrst**, не трогая **input**. Если, например, **input[l]** стоит перед **input[0]** по алфавиту, то программа переключает указатели **ptrst**, в результате чего **ptrst[0]** ссылается на **input[1]**, а **ptrst[1]** на **input[0]**. Это гораздо легче, чем, используя **strcpy( )**, менять местами две введенные строки. Просмотрите еще раз этот процесс на рисунке.

И наконец, давайте попытаемся заполнить пробелы, оставшиеся в нашем описании, а именно "пустоту" между скобками в функции **main( )**.

АРГУМЕНТЫ КОМАНДНОЙ СТРОКИ

Далее Содержание

Командная строка - это строка, которую вы печатаете на клавиатуре, чтобы запустить вашу программу. Это нетрудно. Предположим, у нас есть программа в файле с именем **fuss**. В этом случае командная строка выглядела бы так:

%fuss

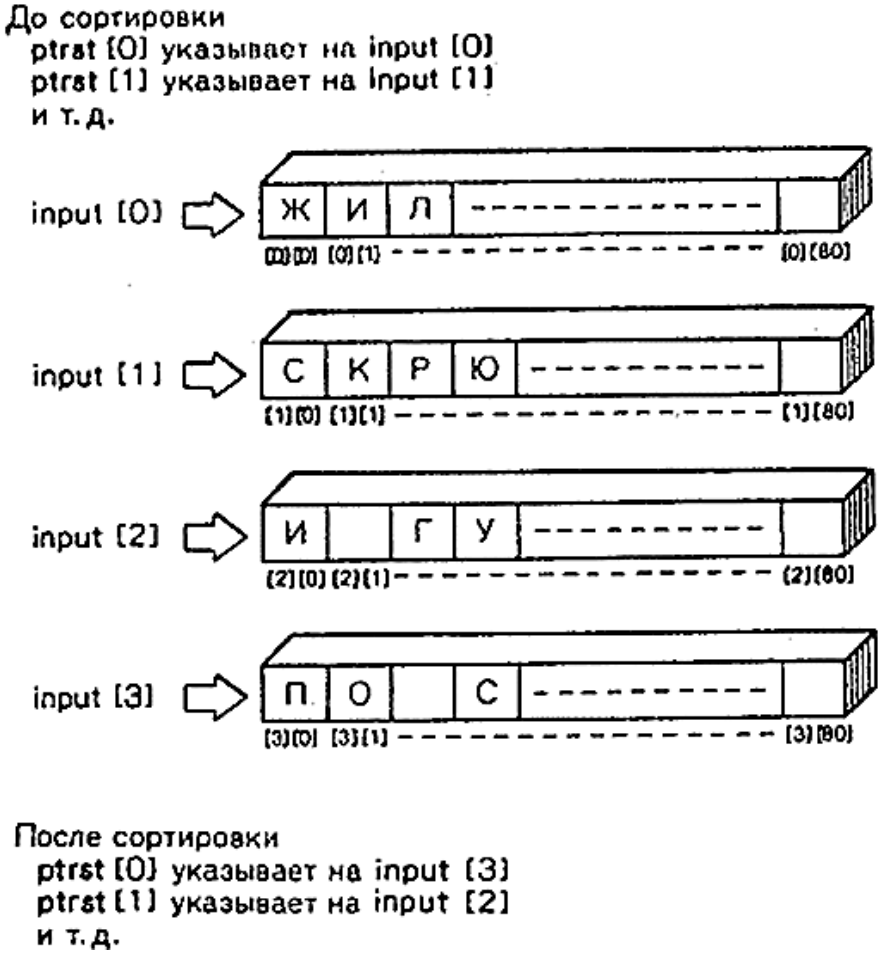


РИС. 13.5. Указатели сортируемых строк.

или, может быть,

```
A > fuss
```

с использованием двух системных приглашений.

Аргументы командной строки являются дополнительными элементами в той же самой строке:

```
%fuss - r Ginger
```

Следует заметить, что программа на языке Си может вводить информацию в эти элементы и применять их для собственных нужд. Этот механизм предназначен для использования аргументов функции **main( )**. Вот типичный пример:

```
/* main( ) с аргументами */
main(argc, argv)
int argc;
char *argv[  ];
{
int count;
for(count = 1; count < argc; count++)
    printf(" %s", argv[count]);
printf("\n");
```

```
}
```

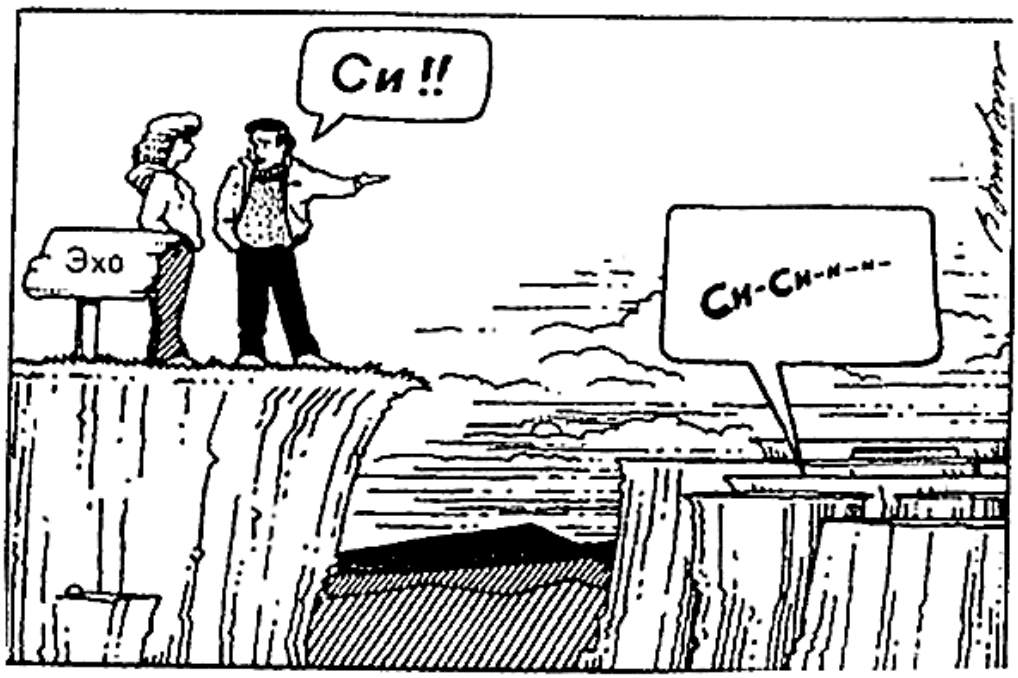
Поместите эту программу в выполняющий файл, названный **echo**, и вот что произойдет:

```
A > echo

Я мог бы воспользоваться небольшой помощью.
Я мог бы воспользоваться небольшой помощью.
```

Вероятно, вы видите, почему функция называется **echo**, но еще не можете понять, как она работает. Может быть, это объяснение поможет вам (мы надеемся).

Компиляторы Си предполагают наличие у **main( )** двух аргументов. Первый аргумент представляет количество строк, следующих за командным словом. Обычно (но не обязательно) этот аргумент типа **int** называется **argc** (*argument count*). Система использует пробелы, чтобы сообщить о конце одной строки и начале следующей. Так, наш пример с **echo** имеет шесть строк, а пример с **fuss** имел две строки. Второй аргумент является массивом указателей строк. Каждой строке, входящей в командную строку, присваивается ее



собственный указатель. По соглашению, этот массив указателей называется **argv** (**argument values**). Если можно (некоторые операционные системы не позволяют этого), элементу **argv[0]** присваивается имя самой программы. В этом случае аргументу **argv[l]** присваивается первая следующая строка и т. д. Для нашего примера имеем

```
argv[0] ссылается на echo (для большинства систем)
argv[1] ссылается на я
argv [2] ссылается на мог
argv [6] ссылается на помощью
```

Поскольку вы используете эти обозначения, то можете легко проследить остаток программы.

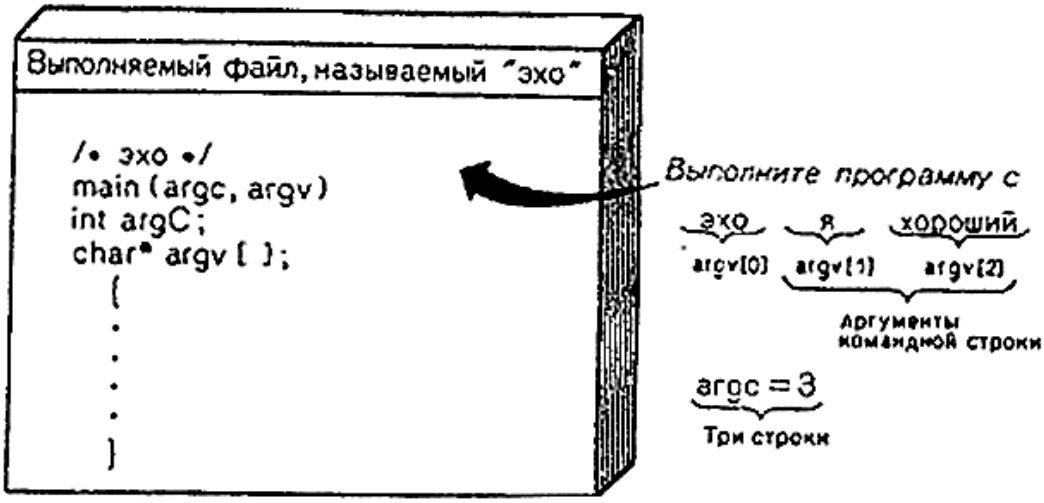


РИС. 13.6. Аргументы командной строки.

Многие программисты используют для **argv** и другие обозначения:

```
main(arge, argv)
int arge;
char **argv;
```

Описание **argv** на самом деле эквивалентно **char *argv[ ];**. Читая его, вы могли бы сказать, что **argv** является указателем на указатель на тип **char**. Наш пример дает то же самое. У нас был массив из семи элементов. Имя массива является указателем на его первый элемент. Поэтому **argv** ссылается на **argv[0]**, а **argv[0]** является указателем на тип **char**. Следовательно, даже с исходным определением **argv** является указателем на указатель на тип **char**. Вы можете использовать любую из этих форм, но видно, что первая проще для понимания.

Очень часто аргументы командных строк используются для указания возможностей программы. Например, можно применять комбинацию символов - **r**, чтобы заставить программу выполнять сортировку в обратном порядке. Обычно альтернативы задаются при помощи дефиса и буквы, как - **r**. Эти "флажки" ничего нс означают в языке Си; вы должны сами запрограммировать их распознавание.
Вот очень простой пример, показывающий, как программа может проверять и использовать флажок.

```
/* обычное начало */
#define YES 1
#define NO  0
main(argc, argv)
int argc;
char *argv[ ];
{
float array[100];
int n;
int flag = NO;
if(argv[l][0] == '-' && argv[l][l] == 'r')
     flag = YES;
...
if flag = NO
     sort1(array, n);
else
sort2(array, n);
...
```

```
}
```

Эта программа проверяет первую строку после имени командного файла, чтобы посмотреть, начинается ли она с дефиса. Затем она проверяет, является ли следующий символ кодом буквы **r**. Если это так, то устанавливается флажок, приводящий к использованию другой программы сортировки. Строки после первой игнорируются. Как мы уже сказали, этот пример достаточно прост.

Если вы использовали систему UNIX, то, вероятно, заметили, что команды UNIX предоставляют разнообразные варианты командной строки и ее аргументов. Эти примеры показывают, как использовать аргументы командной строки языка Си, поскольку большая часть системы UNIX написана на языке Си.

Аргументы командных строк могут быть также именами файлов, и вы можете использовать их вместо операторов переключения, чтобы указать программе, над какими файлами надо работать. Мы покажем вам, как это делается, в гл. 15.

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Как объявить строку символов: **static char fun[ ]** и т. д.
Как инициализировать строку символов: **static char *p0 = "0!"**
Как использовать **gets( )** и **puts( )**
Как использовать **strlen( )**, **strcmp( )**, **strcpy( )** и **strcatf( )**
Как использовать аргументы командной строки.
В чем сходство и различие описателей **char *bliss** и **char bliss[ ]**
Как создать строковую константу: "используя кавычки".

ВОПРОСЫ И ОТВЕТЫ

Далее Содержание

Вопросы

1. Что неправильно в этой попытке описания символьной строки?

```
main( ) {
char name[ ] = {'F', 'c', 's', 's'};
```

2. Что напечатает эта программа?

```
#include <stdio.h>
main( )
{
static char note[ ] = "До встречи в буфете." ;
char *ptr;
ptr = note;
puts(ptr);
puts(++ptr);
note[7] = '\0';
puts(note);
puts(++ptr);
}
```

3. Что напечатает эта программа?

```
main( )
{ static char food[ ] = "Йумми";
```

```
char *ptr;
ptr = food + strlen(food);
while(--ptr >= food) puts(ptr);
}
```

4. Что напечатает нижеследующая программа?

```
main( )
{
static char goldwyn[28] = " аз я считываю"
static char samuel[40] = " Каждый р" ;
char *quote = " часть строки."
strcat(goldwyn, quote);
strcat(samuel, goldwyn);
puts(samuel);
}
```

5. Создайте функцию, которая использует указатель строки в качестве аргумента и возвращает указатель, ссылающийся на первый пробел в строке в указанном месте или после него. Если она не находит ни одного пробела, то пусть возвращает **NULL**-указатель.

Ответы

1. Класс памяти должен быть **extern** или **static;** инициализация должна включать символ **'\0'**.

2.

```
До встречи в буфете.
    о встречи в буфете.
        До вст
    вст
```

3.

```
и
        ми
    мми
        умми
    Йумми
```

4.

```
Каждый раз я считываю   часть строки.
```

5.

```
char *strblk(string)
char *string; {
while(*string != ' ' && *string != '\0')
        string++;  /* останавливается на первом пробеле или нуль-символе */
if(*string == '\0')
        return(NULL);  /* NULL = 0 */
else return(string); }
```

УПРАЖНЕНИЯ

1. Создайте функцию, которая считывает очередные **n** символов при вводе, включая

символы пробелов, табуляции и новой строки.

2. Модифицируйте последнюю функцию таким образом, чтобы она останавливалась после ввода **n** символов или после первого символа пробела, табуляции или но вой строки независимо от того, какой из них идет первым [только не используйте функцию **scanf( )**].

3. Создайте функцию, которая считывает очередное слово при вводе; определите слово как последовательность символов, не включающую символы пробела, табуляции или новой строки.

4. Создайте функцию, которая ищет первое появление определенного символа в определенной строке. Функция должна возвращать указатель, ссылающийся на этот символ, в случае успешного поиска или **NULL**, если символ в строке не найден.

[Содержание] [Вверх]

[Содержание] [Вниз]

14. Структуры и другие типы данных

СТРУКТУРЫ ДАННЫХ
СТРУКТУРНЫЕ ШАБЛОНЫ, ТЕГИ И ПЕРЕМЕННЫЕ
ДОСТУПНЫЕ ЧАСТИ СТРУКТУРЫ
СТРУКТУРНЫЕ УКАЗАТЕЛИ
СТРУКТУРНЫЕ МАССИВЫ
ФУНКЦИИ И СТРУКТУРЫ
ОБЪЕДИНЕНИЯ
СОЗДАНИЕ НОВЫХ ТИПОВ

КЛЮЧЕВЫЕ СЛОВА
struct, union, typedef

ОПЕРАЦИИ
->

Успех программы часто зависит от удачного выбора способа представления данных, с которыми она должна работать. В этом отношении языку Си очень повезло (и не случайно), так как он обладает очень мощными средствами представления сложных данных. Этот тип данных, называемых "структурой", не только достаточно гибок для представления разнообразных данных, но, кроме того, он позволяет пользователю создавать новые типы. Если вы знакомы с "записями" языка Паскаль, вам должны быть удобны структуры.
Посмотрим на конкретном примере, почему структуры нам необходимы и как их создавать и использовать.

Гвен Гленн хочет напечатать опись своих книг. Она хотела бы занести в нее различную информацию о каждой книге: ее название, фамилию автора, издательство, год издания, число страниц, тираж и цену. Теперь некоторые из этих элементов, такие, как название, можно записать в массив строк. Другие элементы требуют массив целого типа или массив типа **float**. Если работать с семью различными массивами и следить за веси содержащейся в них информацией, можно сойти с ума, особенно если Гвен желает иметь несколько списков - список, упорядоченный по названиям, список, упорядоченный по авторам, по цене и т. д. Гораздо лучше было бы использовать один массив, в котором каждый элемент содержал бы всю информацию о книге.

Но какой тип данных может содержать строки и числа одновременно и как-то хранить эту информацию раздельно? Ответом должна быть, конечно, тема данной главы - структура. Чтобы посмотреть, как создается структура и как она работает, начнем с небольшого примера. Для упрощения задачи введем два ограничения: первое - мы включим в опись только название книги, фамилию автора и цену; второе - ограничим опись до одной книги. Если у вас больше книг, не беспокойтесь; мы покажем, как расширить эту программу.
Сначала посмотрите на программу и ее результат, а потом мы рассмотрим основные вопросы.

```
/* инвентаризация одной книги */
#include <stdio.h>
#define MAXTIT 41     /* максимальная длина названия + 1 */
#define MAXAUT 31     /* максимальная длина фамилии автора + 1 */
struct book {         /* шаблон первой структуры: book
                            является именем типа структуры */
char title [MAXTIT];  /* символьный массив для названия */
char author [MAXAUT]; /* символьный массив для фамилии автора */
float value;          /* переменная для хранения цены книги */
};                    /* конец шаблона структуры */
main( )
{
struct book libry;    /* описание переменной типа book */
printf(" Введите, пожалуйста, название книги.\n");
gets(libry. title); /* доступ к элементу title */
printf(" Теперь введите фамилию автора.\n");
gets(libry.author);
printf(" Теперь введите цену.\n");
scanf(" %f ", &libry.value);
printf("%s, %s: %p.2f \n", libry.title, libry.autor,
                                        libry.value);
printf("%s: \" %s \" \(%p.2f\)\n", libry.author,
                            libry.title, libry.value);
}
```

Вот образец работы программы:

```
Введите, пожалуйста,  название книги.
Искусство программирования для ЭВМ
Теперь введите фамилию автора.
Д. Кнут
Теперь введите цену.
5р.67
Искусство программирования для  ЭВМ,  Д. Кнут: 5р.67
Д. Кнут: "Искусство программирования для ЭВМ"  (5р. 67)
```

Созданная нами структура состоит из трех частей: одна для названия, другая для фамилии автора и третья для цены. Мы должны изучить три основных вопроса:
1. Как устанавливать формат или "шаблон" для структуры.

2. Как объявлять переменную, соответствующую этому шаблону.
3. Как осуществлять доступ к отдельным компонентам структурной переменной.

УСТАНОВКА СТРУКТУРНОГО ШАБЛОНА

Далее Содержание

Структурный шаблон является основной схемой, описывающей как собирается структура. Наш шаблон выглядел бы так:

```
struct book {
char title [MAXTIT];
char author [MAXAUT];
float value;
};
```

Этот шаблон описывает структуру, составленную из двух символьных массивов и одной переменной типа **tloat**. Давайте рассмотрим его детально.

Первым стоит ключевое слово **struct**; оно определяет, что все, что стоит за ним, является структурой. Далее следует необязательный "тег" (имя типа структуры) - слово **book**, являющееся сокращенной меткой, которую мы можем использовать позже для ссылки на эту структуру. Поэтому где-нибудь позже у нас будет описание:

```
struct book libry;
```

которое объявляет **libry** структурой типа **book**.

Далее у нас есть список "элементов" структуры, заключенный в парные фигурные скобки. Каждый элемент определяется своим собственным описанием. Например, элемент **title** является символьным массивом, состоящим из **MAXTIT**-элементов. Как мы уже отмечали, элементы могут быть данными любого типа, включая другие структуры! И наконец, мы ставим точку с запятой, завершающую определение шаблона.

Вы можете разместить этот шаблон за пределами любой функции (вне), как мы и сделали, или внутри определения функции. Если мы установили шаблон внутри функции, то он может использоваться только внутри этой функции. Если вне, то шаблон доступен всем функциям программы, следующим за его определением. Например, в другой функции вы можете определить

```
struct book dickens;
```

и эта функция должна иметь переменную **dickens**, которая следует за нашим шаблоном.

Мы сказали, что имя типа структуры необязательно, но его следует использовать, если вы создаете структуру так, как это сделали мы, определив шаблон в одном месте, а фактические переменные в другом. Мы вернемся к этому вопросу после того, как рассмотрим определение структурных переменных.

ОПРЕДЕЛЕНИЕ СТРУКТУРНЫХ ПЕРЕМЕННЫХ

Далее Содержание

Слово "структура" используется двояко. Во-первых, в смысле "структурного шаблона", о котором мы только что рассказали. Шаблон является схемой без содержания; он сообщает компилятору, как делать что-либо, но нс вызывает никаких действий в программе. Следующий шаг заключается в создании "структурной переменной"; это и есть второй смысл слона структура. Строка нашей программы, создающая структурную перемеиную, выглядит так:

```
struct book libry;
```

На основании этого оператора компилятор создаст переменную **libry**. Согласно плану, установленному шаблоном **book**, он выделяет память для символьного массива, состоящего из **MAXTIT**-элементов, для символьного массива из **MAXAUT**-элемсентов и для переменной типа **float**. Эта память объединяется под именем **libry**. (В следующем разделе мы расскажем, как ее "разъединить", если понадобится.)

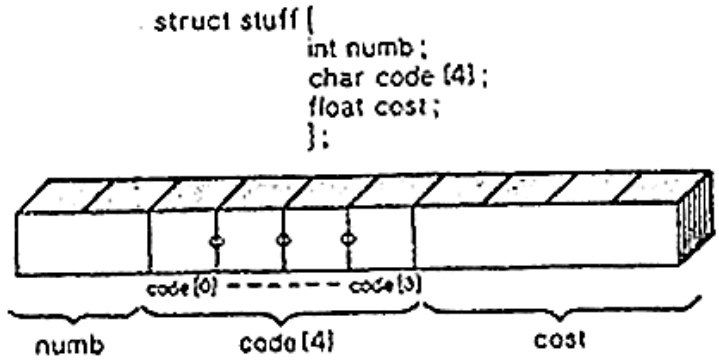


РИС. 14.1. Распределение памяти для структуры.

В этом описании **struct book** играет ту же роль, что и **int** или **float** в своих описаниях. Например, мы могли бы описать две переменные типа **struct book** или даже указатель на этот тип структуры:

```
struct book doyle panshin, *ptbook;
```

Каждая структурная переменная, **doyle** и **panshin**, имела бы части **title**, **author** и **value**. Указатель **ptbook** мог бы ссылаться на **doyle**, **panshin** или любую другую **book**-структуру. Для компьютера оператор нашей программы

```
struct book libry;
```

является сокращенной записью

```
struct book {
char title [MAXTIT];
char author [MAXAUT];
float value;
} libry; /* присоединяет имя переменной к шаблону */
```

Другими словами, процесс определения структурного шаблона и процесс определения структурной переменной можно объединить в один этап. Объединение шаблона и определений переменных является именно тем случаем, когда не нужно использовать имя типа структуры:

```
struct { /* без имени типа структуры */
char title [MAXTIT];
char author [MAXAUT];
float value;
}   libry;
```

Форма с именем типа структуры удобнее, если вы используете структурный шаблон более одного раза.

Есть один аспект определения структурной переменной, который не нашел отражения в нашем примере - инициализация. Теперь мы хотим заняться этим вопросом.

Инициализация структуры

Далее Содержание

Мы видели, как инициализируются переменные и массивы:

```
int count = 0;
static int fibo[ ]={0, 1, 1, 2, 3, 5, 8};
```

Можно ли инициализировать и структурную переменную? Да, если структурная переменная будет внешней или статической. Здесь следует иметь в виду, что принадлежность структурной переменной к внешнему типу зависит от того, где определена *переменная*, а не где определен *шаблон*. В нашем примере шаблон **book** является внешним, а переменная **libry** - внутренней, так как она определена внутри функции и по умолчанию располагается в классе автоматической памяти. Предположим, мы создали такое описание:

```
static struct book libry;
```

В этом случае используется статическая память, и можно инициализировать структуру следующим способом:

```
static struct book libry={"Пират и девица",
                          "Рене Вивот",
                          1p.95 } ;
```

Чтобы сделать ассоциации более явными, мы дали каждому элементу свою собственную строку для инициализации, хотя компилятору требуются только запятые, чтобы отделить инициализацию одного элемента от инициализации следующего.
Продолжим наши разъяснения свойств структуры.

ДОСТУП К ЭЛЕМЕНТАМ СТРУКТУРЫ

Далее Содержание

Структура является разновидностью супермассива, в котором один элемент может быть массивом типа **char**, следующий - **float** и еще один **int**. Обычно можно обращаться к отдельным элементам массива, используя индекс. Как это сделать для отдельных элементов структуры? Для этого мы используем символ "**.**", обозначающий операцию получения элемента структуры. Например, **libry .value** является элементом **value** структуры **libry**. Можно применять **libry.value** точно так же, как вы использовали бы любую другую переменную типа **float**. Можно применять и **libry.title** точно-так же, как массив типа **char**. Поэтому мы могли бы использовать выражения, подобные

```
gets(libry.title)
```

и

```
scanf(" %f ", &libry.value);
```

В сущности **.title**, **.author** и **.value** играют роль индексов для структуры **book**.

Если у вас есть вторая структурная переменная такого же типа, вы могли бы ее использовать точно так же:

```
struct book spiro; gerald;
gets (spiro.title);
gets (gerald.title);
```

.title ссылается на первый элемент структуры **book**.

Посмотрите, как в самой первой программе мы печатали содержимое структурной переменной **libry** в двух различных форматах; она демонстрирует нам возможность использования элементов структуры.

Мы изложили самое основное. Теперь хотелось бы расширите ваш кругозор и рассмотреть некоторые понятия, связанные ее структурами, включая массивы структур, структуры структур, указатели на структуры, а также функции и объединения.

МАССИВЫ СТРУКТУР

Далее Содержание

Настроим нашу программу инвентаризации на обработку, если потребуется, двух или трех (или, может быть, даже большего числа) книг. Очевидно, каждую книгу можно описать структурной переменной типа **book**. Для описания двух книг нам нужны две такие переменные и т. д. Для обработки нескольких книг потребуется массив таких структур, и мы его создали в программе, показанной на рис. 14.2.

```
/* инвентаризация большого количества книг */
#include <stdio.h>
#define MAXTIT 40
#define MAXAUT 40
#define MAXBKS 100 /* максимальное количество книг */
#define STOP  " " /* нулевая строка прекращает ввод  */
struct book { /* создание шаблона типа book */
char title [MAXTIT];
char author [MAXAUT];
float value; };
main ( )
{
struct book libry[MAXBKS]; /* массив структур типа book */
int count = 0;
int index;
printf("Введите, пожалуйста, название книги.\n");
printf(" Нажмите клавишу [ввод] в начале строки для останова.\n");
while(strcmp(gets(libry [count].title), STOP) != 0  &&
                                          count < MAXBKS)
 { printf("Введите теперь фамилию автора.\n");
   gets(libry [count].author);
   printf("Введите теперь цену.\n");
   scanf(" %f", & libry [count++].value);
   while(getchar()!='n'); /* очистите строку ввода */
     if(counts < MAXBKS)
        printf("Введите название следующей книги.\n");
 }  printf ("Вот список ваших книг: \n");
for(index = 0; index < count; index++)
printf("%s, %s: %p.2f\n", libry [index].title, libry[index].author,
                                                 libry[index].value);
}
```

РИС. 14.2. Программа инвентаризации большого количества книг.

Вот пример работы программы:

```
Введите, пожалуйста, название книги.
Нажмите клавишу [ввод]  в начале строки для останова.
Искусство программирования
Введите теперь фамилию автора. Д.Кнут
Введите теперь цену.
5р.67
Введите название следующей книги.
                        ... еще вводы...
```

```
Вот список ваших книг:
Искуство программирования для ЭВМ, Д.Кнут: 5р.67
ПЛ/1 для программистов, Скотт Р., Сондак Н: 1р.08
Программирование на языке Паскаль, П. Грогоно: 1р.30
Язык Фортран 77, Г. Кантан: 0р.80
Трансляция языков программирования, Ф. Вайнгартен: 0р.75
Язык Эсперанто, М.И. Исаев: 0р.60
Ассемблеры и загрузчики, Д.Баррон: 0р.30
Структурное программирование, У. Дал, Э. Дейкстра, К. Хоор: 1р.11
Операционные системы, Г. Катцан: 2р.25
Параллельные вычислительные системы, Б.А.Головкин: 2р.50
```

Следует обратить внимание на два важных момента, относящихся к массивам структур, - как описывать и как обращаться к отдельным их элементам. После разъяснения этих вопросов мы вернемся и сделаем пару замечаний по нашей программе.

Описание массива структур

Далее Содержание

Процесс описания массива структур совершенно аналогичен описанию любого другого типа массива:

```
struct book libry [MAXBKS];
```

Этот оператор объявляет **libry** массивом, состоящим из **MAXBKS**-элементов. Каждый элемент массива представляет собой структуру типа **book**. Таким образом, **libry[0]** является **book**-структурой, **libry[1]** - второй **book**-структурой и т. д. Рис. 14.3 может помочь вам представить это. Имя **libry** само по себе не является именем структуры; это имя массива, содержащего структуры.

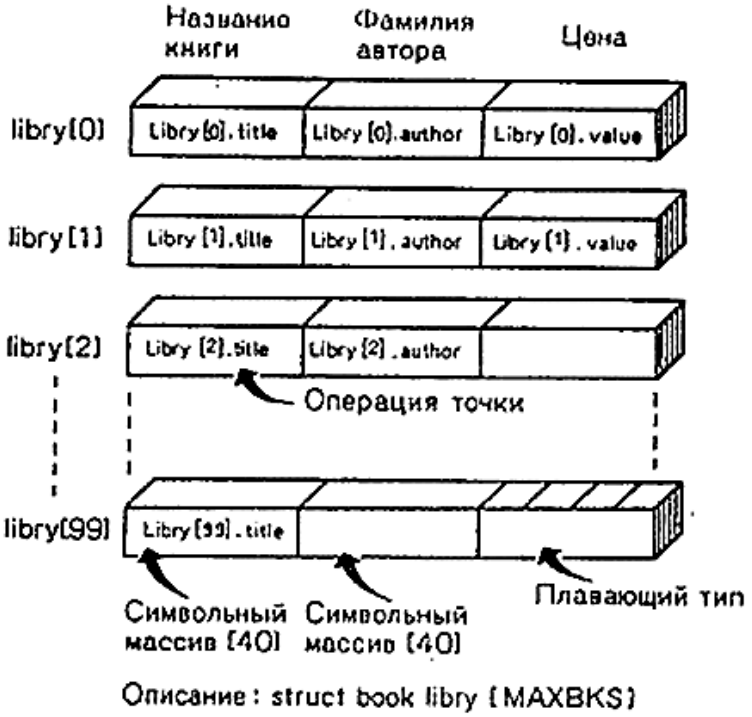


РИС. 14.3. Массив структур.

Определение элементов массива структур

Далее Содержание

При определении элементов массива структур мы применяем те же самые правила, которые используются для отдельных структур: сопровождаем имя структуры операцией получения элемента и именем элемента:

```
libry [0].value  value - первый элемент массива
libry [4].title  title - пятый элемент массива
```

Заметим, что индекс массива присоединяется к **libry**, а не к концу имени:

```
libry. value[2]   /* неправильно */
libry[2].value   /* правильно */
```

Мы используем **libry[2].value**, потому что **libry[2]** *является* именем структурной переменной точно так же, как **libry[l]** является именем другой структурной переменной, а ранее **doyle** было именем структурной переменной.
Между прочим, что бы это значило?

```
libry[2].title[4]
```

Это был бы пятый элемент элемента **title** (т. е. **title[4]**) структуры типа **book**, описанный третьей структурой (т.е. **libry[2]**). В нашем примере им был бы символ **p**. Это означает, что индексы, находящиеся справа от операции "**.**", относятся к отдельным элементам, в то время как индексы, расположенные слева от операции, относятся к массивам структур.
Теперь покончим с этой программой.

Детализация программы

Далее Содержание

Главное отличие ее от нашей первой программы заключается в том, что теперь создается цикл для считывания названий книг. Мы начинаем цикл с **while**-условия:

```
while(strcmp(gets(libry [count].title), STOP) != 0
                            && count < MAXBKS)
```

Выражение **gets(libry [count].title)** считывает вводимую строку, содержащую название книги. Функция **strcmp( )** сравнивает эту строку со **STOP**, которая является " " , т.е. пустой строкой. Если пользователь нажмет клавишу [**ввод**] в начале строки, то перепишется пустая строка и цикл закончится. Мы также должны проверять, не превысило ли число считанных на текущий момент книг предельного размера массива.
В программе есть странная строка **while(getchar ( ) ! = '\n'); /* очистить строку ввода */**

Она включена для того, чтобы использовать особенность функции **scanf( )**, которая игнорирует символы пробела и новой строки. Когда вы отвечаете на запрос об элементе **value** в структуре **book**, то вводите что-нибудь вроде

```
12.50 [ввод]
```

При этом передается последовательность символов

```
12.50\n
```

Функция **scanf( )** собирает символы 1, 2, . , 5, 0, но опускает символ **\n**, стоящий там, и ожидает, что следом придет еще какой-нибудь оператор чтения. Если пропустить нашу странную строку, то следующим оператором чтения будет **gets(libry [count].title)** в операторе управления циклом. Он прочел бы оставшийся символ новой строки как *первый* символ, и программа решила бы, что мы послали сигнал останова. Поэтому мы и

вставили такую странную строку. Если вы поняли это, то увидите, что она "проглатывает" символы до тех пор, пока не найдет и не получит символ новой строки. Функция ничего не делает с данным символом, только принимает его от устройства ввода. Это приводит к тому, что функция **gets( )** снова начинает работать. Вернемся теперь к изучению структур.

ВЛОЖЕННЫЕ СТРУКТУРЫ

Далее Содержание

Иногда бывает удобно, чтобы одна структура содержалась или была "вложена" в другую. Например, Шалала Пироски строит структуру, содержащую информацию о ее друзьях. Один элемент структуры - это, конечно, имя друга. Однако имя можно представить самой структурой с разными элементами для имени и фамилии. На рис. 14.4 приведен сокращенный пример деятельности Шалалы.

```
/* пример вложенной структуры */
#define LEN 20
#define M1   "Спасибо за прекрасный вечер,"
#define M2   "Вы, конечно, правы, что"
#define M3   " -своеобразный парень. Мы должны собраться"
#define M4   " отведать очень вкусный"
#define M5   "и немного повеселиться."
struct names {   /*первый структурный шаблон */
char first[LEN];
char last[LEN], };
struct guy {   /* второй шаблон */
struct names handle; /* вложенная структура */
char favfood[LEN];
char job[LEN];
float income;
};
main( ) {
static struct guy fellow = {  /*инициализация переменной */
{" Франко," " Уотэл"},
" баклажан",
" вязальщик половиков",
15435.00 };
printf("Дорогой %s, \n \n," fellow.handle.first);
printf(" %s %s.\n", M1, fellow.handle.first);
printf(" %s %s\n" , M2, fellow.job);
printf(" %s \n" , M3);
printf(" %s %s %s\n\n", M4, fellow.favfood, M5);
printf(" %40s %s \n", "    " , " До скорой встречи" );
printf(" %40s %s\n", "    ", Шалала");
}
```

РИС. 14.4. Программа вложенной структуры.

Вот результат работы программы:

```
Дорогой Франко,
     Спасибо за прекрасный вечер, Франко.
Вы, конечно, правы. что вязальщик половиков - своеобразный парень.
Мы должны собраться отведать очень вкусный баклажан и немного повеселиться.
                                                    До скорой встречи,
                                                    Шалала
```

Во-первых, следует рассказать о том, как поместить вложенную структуру в шаблон. Она просто описывается точно так же, как это делалось бы для переменной типа **int**:

```
struct names handle;
```

Это означает, что **handle** является переменной типа **struct names**. Конечно, файл должен также содержать шаблон для структуры типа **names**.

Во-вторых, следует рассказать, как мы получаем доступ к элементу вложенной структуры. Нужно дважды использовать операцию "**.**" :

```
fellow.handle.first = = " Франко";
```

Мы интерпретируем эту конструкцию, перемещаясь слева направо;

```
(fellow.handle).first
```

То есть первым находим элемент **fellow**, далее элемент **handle** структуры **fellow**, а затем его элемент **first**. Теперь рассмотрим указатели.

УКАЗАТЕЛИ НА СТРУКТУРЫ

Далее Содержание

Любители указателей будут рады узнать, что указатели можно использовать и для структур. Это хорошо по крайней мере по трем причинам. Во-первых, точно так же как указатели на массивы, они легче в использовании (скажем, в задаче сортировки), чем сами массивы, а указателями на структуры легче пользоваться, чем самими структурами. Во-вторых, структура не может использоваться в качестве аргумента функции, а указатель на структуру может. В-третьих, многие удобные представления данных являются структурами, содержащими указатели к другим структурам.

Следующий короткий пример (рис. 14.5) показывает, как определять указатель на структуру и как использовать его для получения элементов структуры.

```
/* указатель на структуру */
#define LEN 20 struct names {
char first [LEN];
char last [LEN]; };
struct guy {
struct names handle;
char favfood [LEN];
char job [LEN];
float income; };
main( ) {
static struct guy fellow [2] = {
    { "Франко", "Уотэл" }
    "баклажан",
    " вязальщик половиков ",
    15435.00},
    {{"Родней", "Свилбели" },
    "лососевый мусс", "декоратор интерьера",
    35000.00 } };
struct guy *him;  /* ЭТО - - указатель па структуру */
printf("адрес #1:  %u  #2 : %u\n", &fellow[0],
                                    &fellow[1]);
him = &fellow[0]; /* сообщает указателю, на что ссылаться */
printf("указатель #1: %u   #2: %u \n ", him, him + 1);
printf("him  -> доход $ %.2f: (*him).доход $ %.2f \n",
                            him -> доход, (*him).доход);
him++;    /* указывает на следующую структуру */
printf("him -> favfood is %s : him -> names.last is %s\n",
                        him-> favfood, him -> handle.last);}
```

РИС. 14.5. Программа с использованием указателя на структуру.

Вот, пожалуйста, ее выход:

```
адрес #1:  12    #2: 96
указатель #1: 12     #2: 96
him -> доход $15435.00: (*him).доход $15435.00
him -> favfood  лососевый мусс: him -> names.last
                                    - Свилбели
```

Сначала посмотрим, как мы создали указатель на структуру **guy**. Затем научимся определять отдельные элементы структуры при помощи указателей.

Описание и инициализация указателя на структуру

Далее Содержание

Вот самое простое описание, какое только может быть:

```
struct guy *him;
```

Первым стоит ключевое слово **struct**, затем слово **guy**, являющееся именем структурного шаблона, далее ***** и за нею имя указателя. Синтаксис тот же, как для описаний других указателей, которые мы видели.

Теперь можно создать указатель **him** для ссылок на любые структуры типа **guy**. Мы инициализируем **him**, заставляя его ссылаться на **fellow[0]**; заметим, что мы используем операцию получения адреса:

```
him = &fellow[0];
```

Первые две выведенные строки показывают результат этого присваивания. Сравнивая две строки, мы видим, что **him** ссылается на **fellow[0]**, а **him+1** - на **fellow[l]**. Заметим, что добавление 1 к **him** прибавляет 84 к адресу. Это происходит потому, что каждая **guy**-структура занимает 84 байта памяти: первое имя - 20, последнее имя - 20, **favfood** - 20, **job** - 20 и **income** - 4 байта (размер элемента типа **float** в нашей системе).

Доступ к элементу структуры при помощи указателя

Далее Содержание

him ссылается на *структуру* **fellow[0]**. Каким образом можно использовать **him** для получения значения *элемента* структуры fellow[0]? Третья выведенная строка даст для этого два способа.

Первый способ, наиболее общий, использует новую операцию **->**. Она заключается в вводе дефиса (**-**) и следующего за ним символа "больше чем" (**>**). Пример помогает понять смысл сказанного:

```
him -> income - это fellow[0].income,
                если him = &fellow[0]
```

Другими словами, структурный указатель, за которым следует операция **->**, работает так же, как имя структуры с последующей операцией "**.**". (Мы не можем сказать **him.income**, потому что **him** не является именем структуры.)

Очень важно отметить, что **him**-*указатель*, а **him - > income** - *элемент* структуры, на которую делается ссылка. Таким образом, в этом случае **him - > income** является переменной типа **float**.

Второй способ определения значения элемента структуры вытекает из

последовательности:
если **him == &fellow[0]**, то ***him == fellow[0]**. Это так, поскольку **&** и ***** - взаимообратные операции. Следовательно, после подстановки

```
fellow[0].income == (*him).income
```

Круглые скобки необходимы, поскольку операция "**.**" имеет приоритет выше, чем ***.

Отсюда можно сделать вывод, что если **him** является указателем на структуру **fellow[0]**, то следующие обозначения эквивалентны:

```
fellow[0].income == (*him).income == him->income
```

Давайте теперь посмотрим, как взаимодействуют структуры и функции.

Резюме: операции над структурами и объединениями

Эта операция используется с именем структуры или объединения для определения элемента этой структуры или объединения. Если **name** является именем структуры, а **member** - элементом, определенным структурным шаблоном, то **name.member** обозначает этот элемент структуры. Операция получения элемента может использоваться таким же образом для объединений.

Примеры

```
struct {
int code;
float cost;
} item;
item.code = 1265;
```

Данный оператор присваивает значение элементу **code** структуры **item**.

II. ОПЕРАЦИЯ КОСВЕННОГО ПОЛУЧЕНИЯ ЭЛЕМЕНТА: ->

Эта операция используется с указателем на структуру или объединение для определения элемента структуры или объединения. Предположим, что **ptrstr** является указателем на структуру и что **member** элемент, определенный структурным шаблоном. Тогда

```
ptrstr -> member
```

определяет элемент, на который выполняется ссылка. Операцию косвенного обращения к элементу можно использовать таким же образом и для объединений.

Пример

```
struct {
int code
float cost;
} item, *ptrst;
ptrst = &item;
ptrst -> code = 3451;
```

Операторы присваивают значение элементу **code** структуры **item**. Следующие три

выражения эквивалентны:

```
ptrst->code      item.code      (*ptrst).code
```

ПЕРЕДАЧА ИНФОРМАЦИИ О СТРУКТУРАХ ФУНКЦИЯМ

Далее Содержание

Вспомним, что аргументы функции передают *значения* в функцию. Каждое значение является либо числом типа **int** или **float**, либо ASCII-кодом или адресом. Структура гораздо сложнее, чем отдельная переменная, поэтому неудивительно, что саму структуру нельзя использовать в качестве аргумента функции. (Это ограничение снято в некоторых новых реализациях.) Однако есть способы ввести информацию о структуре внутрь функции. Рассмотрим три способа (на самом деле два с вариациями).

Использование элементов структуры

Далее Содержание

Поскольку элемент структуры является переменной с единственным значением (т.е. типа **int** или одного из его "родственников" - **char**, **float**, **double** или указатель), он может быть передан как аргумент функции. Простая программа финансовых расчетов на рис. 14.6, которая прибавляет взнос клиента к его счету, иллюстрирует этот способ. Заметим, между прочим, что мы объединили определение шаблона, описание переменной и инициализацию в один оператор.

```
/* передача элементов структуры как аргументов функции */
struct funds {
char *bank;
float bankfund;
char *save;
float savefund; }
stan =  { " Банк синьора Помидора",
            1023.43,
            " Сбережения и займы Снупи",
            4239.87 };
main( )
{
float total, sum( );
extern struct funds stan;   /* необязательное описание */
printf("У Стэна всего %.2f долл.\n", sum(stan.bankfund,
                                        stan.savefund));
}
/* складывает два числа типа float */
float sum(x, y);
float x, y;
{ return( x + y); }
```

РИС. 14.6. Программа, передающая функции аргументы, являющиеся элементами структуры.

Результат выполнения этой программы:

```
У Стэна всего 5263.30 долл.
```

Вот это да, она работает. Заметим, что функция **sum( )** "не знает", или же ей безразлично, являются ли элементами структуры фактические аргументы; она только "требует", чтобы они имели тип **float**.

Конечно, если вы хотите, чтобы программа воздействовала на значение элемента в

вызывающей программе, можно передать ей адрес этого элемента:

```
modify(&stan.bank fund);
```

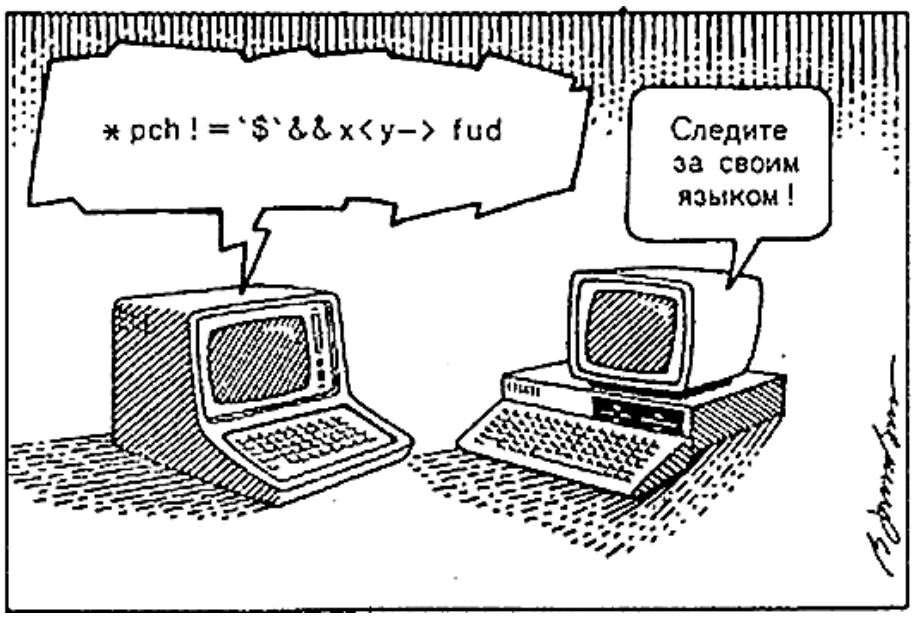
и тогда получилась бы функция, которая изменила бы банковский счет в структуре **stan**.

Второй способ передачи информации о структуре заключается в возможности сообщить суммирующей функции, что она имеет дело со структурой.

Использование адреса структуры

Далее Содержание

Мы будем решать ту же самую задачу, что и прежде, но при этом использовать адрес структуры в качестве аргумента. Это хорошо, поскольку адрес представляет собой только одно число. Так как функция должна работать со структурой **funds**, она тоже должна использовать шаблон **funds**. См. рис. 14.7.




```
/* передача адреса структуры в функцию */
struct funds {
        char *bank;
        float bankfund;
        char *save;
        float savefund;
        } stan = {
        "Банк синьора Помидора" ,
        1023.43,
        " Сбережения    и       займы Снупи" ,
        4239.87
        };
main( )
{
float total, sum( );
printf(" У Стэна всего %.2f  долл.\n", sum(&stan) );
}
float sum (money)
struct funds *money;
}
return( money-> bankfund + money-> savefund);
}
```

РИС. 14.7. Программа, передающая функции адрес структуры. Эта программа тоже выдает

```
У Стэна всего 5263.30 долл.
```

Функция **sum( )** имеет указатель (**money**) на структуру **fund**. Передача адреса **&stan** в функцию заставляет указатель **money** ссылаться на структуру **stan**. Затем используем операцию **- >** для получения значений элементов **stan.bankfund** и **stan.savefund**.

Эта функция также имеет доступ к названиям учреждений, хотя их не использует. Заметим, что мы должны применять операцию **&** для получения адреса структуры. В отличие от имени массива имя структуры само по себе не является синонимом своего адреса.

Наш следующий способ применим к массивам структур и является вариантом данного способа.

Использование массива

Далее Содержание

Предположим, у нас есть массив структур. Имя массива *является* синонимом его адреса, поэтому его можно передать функции. С другой стороны, функции будет необходим доступ к структурному шаблону. Чтобы показать, как такая программа работает (рис. 14.8), мы расширим нашу программу таким образом, чтобы она обслуживала двух человек, а мы, следовательно, имели бы массив двух структур **funds**.

```
/* передача массива структур в функцию */
struct funds {
    char *bank;
    float bankfund;
    char *save;
    float savefund; }
jones[2] ={
            { " Банк синьора Помидора" ,
            1023.43,
            " Сбережения и займы Снупи" ,
            4239.87 },
            { " Банк Честного Джека",
            976.57,
            "Накопления по предварительному плану",
            1760.13 } };
main( )
{
float total, sum( );
printf("Джонсы имеют всего %.2f долл.\n", sum(jones));
}
float sum(money);
struct funds *money;
{
float total;
int i;
        for( i = 0, total = 0; i < 2; i++, money++ )
        total += money- > bankfund + money -> savefund;
        return(total);
}
```

РИС. 14. 8. Программа, передающая функции массив структур.

Программа выдает

```
Джонсы имеют всего  8000.00  долл.
```

(Что за круглая сумма! Можно подумать, что эти цифры взяты с потолка.) Имя массива **jones** является указателем на массив. В частности, оно ссылается на первый элемент массива, который является структурой **jones[0]**. Таким образом, в начале указатель money за дается через

```
money = &jones[0];
```

Затем использование операции **- >** позволяет нам добавить два вклада для первого Джонса. Это действительно очень похоже на последний пример. Далее, цикл **for** увеличивает указатель **money** на 1. Теперь он ссылается на следующую структуру, **jones[1]**, и остаток вкладов может быть добавлен к **total**.

Вот два основных замечания:

1. Имя массива можно использовать для передачи в функцию указателя на первую структуру в массиве.
2. Затем можно использовать арифметическую операцию над указателем, чтобы передвигать его на последующие структуры в массиве. Заметим, что вызов функции

```
sum(&jones[0])
```

дал бы тот же самый эффект, что и применение имени массива, так как оба они ссылаются на один и тот же адрес. Использование имени массива является просто косвенным способом передачи адреса структуры.

СТРУКТУРЫ: ЧТО ДАЛЬШЕ?

Далее Содержание

Мы не будем больше рассказывать о структурах, но хотелось бы отметить одно очень важное использование структур: создание новых типов данных. Пользователи компьютеров разработали новые типы данных, гораздо более эффективные для определенных задач, чем массивы и простые структуры, которые мы описали.

Эти типы имеют такие названия, как очереди, двоичные деревья, неупорядоченные массивы, рандомизированные таблицы и графы. Многие из этих типов создаются из "связанных" структур. Обычно каждая структура будет содержать один или два типа данных плюс один или два указателя на другие структуры такого же типа. Указатели служат для связи одной структуры с другой и для обеспечения пути, позволяющего вам вести поиск по всей структуре. Например, на рис. 14.9 показано двоичное дерево, в котором каждая отдельная структура (или "узел") связана с двумя, расположенными ниже.

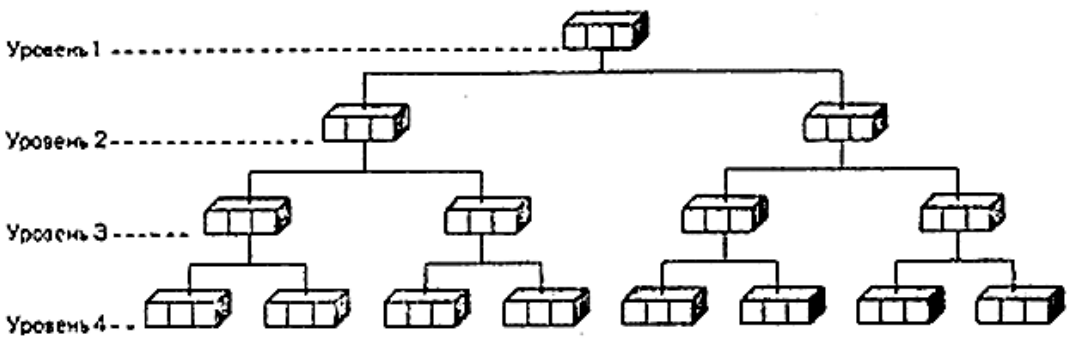


РИС. 14.9. Структура двоичного дерева.

Является ли эта разветвленная конструкция более эффективной чем массив?

Рассмотрим случай дерева с 10 уровнями узлов. Если вы составите его, то найдете 1023 узла, в которых вы можете запомнить, скажем, 1023 слова. Если слова упорядочены, согласно некоторому разумному плану, вы можете начать с верхнего уровня и находить любое слово в лучшем случае за 9 перемещений, если ваш поиск идет сверху вниз с одного уровня на следующий. Если слова находятся в массиве, вам, может быть, придется перебрать все 1023 элемента, прежде чем вы найдете нужное слово.

Когда вас интересуют более прогрессивные структуры данных, обратитесь к литературе по вычислительной технике. Используя структуры языка Си, вы сможете создавать типы, о которых вы прочитали.

Это наше последнее слово о структурах. Далее мы хотим вкратце ознакомить вас с двумя другими средствами языка Си для работы с данными: объединением и функцией **typedef**.

ОБЪЕДИНЕНИЯ - КРАТКИЙ ОБЗОР

Далее Содержание

Объединение - это средство, позволяющее запоминать данные различных типов в одном и том же месте памяти. Типичным использованием его было бы создание таблицы, содержащей смесь типов в некотором порядке, который не является постоянным и не известен заранее. Объединение позволяет создавать массив, состоящий из элементов одинакового размера, каждый из которых может содержать различные типы данных.

Объединения устанавливаются таким же способом, как и структуры. Есть шаблон объединения и переменные объединения. Они могут определяться одновременно или, если используется имя объединения, последовательно за два шага. Вот пример шаблона с именем объединения:

```
union holders {
int digit;
double bigf1;
char letter; };
```

А вот пример определения переменных объединения типа **holdem**:

```
union holdem fit; /* переменная объединения типа holdem */
union holdem save[10]; /* массив из 10 переменных объединения */
union holdem *pu;  /* указатель на переменную типа holdem */
```

Первое описание создаст одну переменную **fit**. Компилятор выделяет достаточно памяти для размещения самой большой из описанных переменных. В этом случае наибольшей из возможных является переменная **double**, для которой требуется в нашей системе 64 разряда или 8 байтов. Массив **save** имел бы 10 элементов, каждый по 8 байтов.
Вот как используется объединение:

```
fit.digit = 23;  /* 23 записывается в fit; используется 2 байта */
fit.double = 2.0; /* 23 стирается, 2.0 записывается; используется 8 байтов */
fit.letter = 'h'; /* 2.0 стирается, h записывается; используется 1 байт */
```

Вы применяете операцию получения элемента, чтобы показать, какие типы данных используются. В каждый момент времени запоминается только одно значение; нельзя записать **char** и **int** одновременно, даже если для этого достаточно памяти.

Вы сами должны следить за типом данных, записываемых в данный момент в объединение; приведенная ниже последовательность операторов показывает, что нельзя делать:

```
fit.letter = 'A';
finum = 3.02*fit.double; /* ОШИБКА ОШИБКА ОШИБКА */
```

Ошибка заключается в том, что записано значение типа **char**, а следующая строка предполагает, что содержимое **fit** имеет тип **double**.

Можно использовать операцию **- >** с объединениями таким же образом, как это делалось для структур:

```
pu = &fit;
x =  pu  -> digit; /* то же, что и x=fit.digit */
```

Рассмотрим теперь еще одно средство языка для работы с данными.

typedef - КРАТКИЙ ОБЗОР

Далее Содержание

Функция **typedef** позволяет нам создать свое собственное имя типа. Это напоминает директиву **#define**, но со следующими тремя изменениями:

1. В отличие от **#define** функция **typedef** дает символические имена, но ограничивается только типами данных.
2. Функция **typedef** выполняется компилятором, а не препроцессором.
3. В своих пределах функция **typedef** более гибка, чем **#define**.

Посмотрим, как она работает. Предположим, вы хотите использовать термин **real** для чисел типа **float**. Тогда вы определяете термин **real**, как если бы он был переменной типа **float**, и перед его определением ставите ключевое слово **typedef**:

```
typedef float real;
```

С этого момента вы можете использовать **real** для определения переменных:

```
real x, y[25], *pr;
```

Область действия такого определения зависит от расположения оператора **typedef**. Если определение находится внутри функции, то область действия локальна и ограничена этой функцией. Если определение расположено вне функции, то область действия глобальна.

Часто в этих определениях используются прописные буквы, чтобы напомнить пользователю, что имя типа является на самом деле символической аббревиатурой:

```
typedef float REAL;
```

В последнем примере можно было бы применить директиву **#define**. А здесь это делать нельзя:

```
typedef char *STRING;
```

Без ключевого слова **typedef** оператор определял бы **STRING** как указатель на тип **char**. С ключевым словом оператор делает **STRING** *идентификатором* указателей на тип **char**. Так,

```
STRING name, sign;
```

означает

```
char *name, *sign;
```

Мы можем использовать **typedef** и для структур. Вот пример:

```
typedef struct COMPLEX {
float real;
float imag; };
```

Кроме того, можно использовать тип **COMPLEX** для представления комплексных чисел.

Одна из причин использования **typedef** заключается в создании удобных, распознаваемых имен для часто встречающихся типов. Например, многие пользователи предпочитают применять **STRING** или его эквивалент, как это мы делали выше. Вторая причина: имена **typedef** часто используются для сложных типов. Например, описание

```
typedef char *FRPTC ( ) [5];
```

приводит к тому, что **FRPTC** объявляет тип, являющийся функцией, которая возвращает указатель на пятиэлементный массив типа **char**. (См. "Причудливые описания".)

Третья причина использования **typedef** заключается в том, чтобы сделать программы более мобильными. Предположим, например, что вашей программе нужно использовать 16-разрядные числа. В некоторых системах это был бы тип **short**, в других же он может быть типом **int**. Если вы использовали в ваших описаниях **short** или **int**, то должны изменить все описания, когда перейдете от одной системы к другой. Вместо этого сделайте следующее, В файле директивы **#include** есть такое определение:

```
typedef short TWOBYTE;
```

Используйте **TWOBYTE** в ваших программах для переменных типа **short**, которые должны быть 16-разрядными. Тогда если вы перемешаете программу туда, где необходимо использовать тип **int**, то следует только изменить одно определение в вашем файле директивы **#include**:

```
typedef int TWOBYTE;
```

Это пример того, что делает язык Си столь мобильным. При использовании **typedef** следует иметь в виду, что он не создаст новых типов, он только создает удобные метки.

ПРИЧУДЛИВЫЕ ОПИСАНИЯ

Язык Си позволяет вам создавать сложные формы данных. Обычно мы придерживаемся более простых форм, но считаем споим долгом указать ни потенциальные возможности языка. При создании описания мы используем имя (или "идентификатор"), которое можно изменять при помощи модификатора:

Модификатор	*значение*
*	указатель
( )	функция
[ ]	массив

Язык Си позволяет использовать одновременно более одного модификатора, что даст возможность создавать множество типов:

```
int board[8] [8]; /* массив массивов типа int */
int **ptr;  /* указатель на указатель на тип int */
int *risks[10];  /* 10-элементный массив указателей на тип int */
int (*wisks) [10];  /* указатель на 10-элемснтный массив типа int */
int *oof[3] [4]: /* 3-элементныи массив указателей на 4-элементный
                                                  массив типа int */
```

```
int (*uuf) [3][4]; /* указатель на массив 3x4 типа int */
```

Для распутывания этих описаний нужно понять, в каком порядке следует применять модификаторы. Три правила помогут вам справиться с этим.

1. Чем ближе кодификатор стоит к идентификатору, тем выше его приоритет.
2. Модификатиры [ ] и ( ) имеют приоритет выше, чем *.
3. Круглые скобки используются для объединения частей выражения, имеющих самый высокий приоритет.

Давайте применим эти правила к описанию **int *oof[3] [4];**
***** и **[3]** примыкают к **oof** и имеют более высокий приоритет, чем **[4]** (правило 1). **[3]** имеет приоритет более высокий, чем ***** (правило 2). Следовательно, **oof** является 3-элементным массивом (перпый МОДИФИКАТОР) указателей (второй модификатор) на 4-элементный массив (третий модификатор) типа **int** (описание типа).

В описании

```
int (*uuf)[3][4];
```

скобки говорят, что модификатор ***** должен иметь первый приоритет, а это делает **uuf** указателем, как показано в предыдущем описании. Эти правила создают также следующие типы:

```
char *fump( ); /* функция, возвращающая указатель на тип char */
char (*frump) ( ); /* указатель на функцию, возвращающую тип char */
char *flump ( ) [3] /* функция, возвращающая указатель на 3-элементный
                                                    массив типа char */
char *flimp[3] ( ) /* 3-элементный массив указателей на функцию, которая
                                          возвращает тип char  */
```

Если вы примените структуры к этим примерам, то увидите, что возможности для описаний действительно растут причудливо. А применения ... так и быть, МЫ оставим их для более опытных программистов.

Язык Си со структурами, объединениями и **typedef** дает нам средства для эффективной и мобильной обработки данных.

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Что такое структурный шаблон, и как его определять
Что такое имя структуры и как оно используется
Как определить структурную переменную: **struct car honda**;
Как обратиться к элементу структуры: **honda.mpg**
Как обратиться к указателю на структуру: **struct car *ptcar;**
Как обратиться к элементу при помощи указателя: **ptcar->mpg**
Как передать в функцию элемент структуры: **eval(honda.mpg)**
Как сообщить функции о структуре: **rate(&honda)**
Как создать вложенную структуру
Как обратиться к элементу вложенной структуры: **honda.civic.cost**
Как создавать и использовать массивы структур: **struct car gm[5];**
Как создать объединение: подобно структуре
Как использовать typedef: **typedef struct car CRATE;**

Вопросы

1. Что неправильно в этом шаблоне?

```
structure {
char itible;
int num [20];
char *togs;
};
```

2. Вот фрагмент программы; что она напечатает?

```
struct house {
    float sqft;
    int rooms;
    int stories;
    char *address; };
main ( ) {
static struct house fruzt = { 1560.0, 6, 1, " 22 Spiffo Road";
struct house *sign;
sign = &fruzt;
printf(" %d %d\n" , fruzt.rooms, sign-> stories);
printf(" %s\n", frurt.address);
prinlf(" %c %c \n" sign- >address[3], fruzt.address[4]);
}
```

3. Придумайте структурный шаблон, который будет содержать название месяца, трехбуквенную аббревиатуру месяца, количество дней в месяце и номер месяца.

4. Определите массив, состоящий из двенадцати структур того же типа, что и в вопросе 3, и инициализируйте его для невисокосного года.

5. Напишите функцию, которая получает номер месяца, а возвращает общее число дней года вплоть до этого месяца. Считайте, что структурный шаблон и массив из вопросов 3 и 4 описаны как внешние.

6. Взяв за основу нижеследующую функцию **typedet**, опишите 10-элементный массив указанной структуры. Затем, используя присваивание отдельного элемента попытайтесь описать третьим элементом'массива линзу Ремаркатара с фокусным расстоянием 500 мм и апертурой f / 2.0.

```
typedef struct {          /* описатель линзы */
float foclen; /* фокусное расстояние, мм */
float fstop; /* апертура */
char *brand; /* фирменная марка */ } LENS;
```

Ответы:

1. Должно быть ключевое слово **struct**, а не **structure**. Шаблон требует либо имени структуры перед открывающей скобкой или имени переменной после закрывающей скобки. Кроме того, точка с запятой должна стоять после ***togs** и в конце шаблона.

2.

```
6 1
        22 Spiffo Road  S p
```

Элемент **fruzt.address** является символьной строкой, а **fruzt.address[4]** является пятым элементом этого массива.

3.

```
struct month {
char name[10]; /* или char *name; */
char abbrev[4]; /* или char *abbrev; */
int days;
int monumb;  };
```

4.

```
struct month months [12] = {
{" Январь" , " Янв" , 31, 1} , {" Февраль" , " Фев" , 28, 2} ,
и т. л. {"Декабрь", "Дек" , 31, 12}
```

5.

```
days(monlh);
inl month;
{
int index, tolal;
if(month < 1 ||  month > 12)
return (-1); /* признак ошибки */
else
for(index = 0, total = 0; index < month; index++)
total + = months [index].days;
return (total);}
```

Заметим, что **index** содержит номер месяца, уменьшенный на единицу, так как массивы начинаются с индекса 0; следовательно, мы используем выражение **index < month** вместо **index <= month**.

6.

```
ЛИНЗА tubby [10];
tubby [2].foclen = 300.0;
tubby [2].fstop = 2.0;
tubby [2].brand = "Рсмаркатар";
```

УПРАЖНЕНИЯ

1. Переделайте вопрос 5, используя в качестве аргумента написанное буквами название месяца вместо номера месяца. [Не забывайте о функции **strcmp( )**.]

2. Напишите программу, которая запрашивает у пользователя день, месяц и год. Месяц может обозначаться номером, названием месяца или его аббревиатурой. После работы программа выдает общее количество дней в году вплоть до данного дня.

3. Переделайте нашу программу инвентаризации книг таким образом, чтобы она печатала информацию о книгах, упорядоченную в алфавитном порядке по названиям книг, и затем печатала общую стоимость книг.

[Содержание] [Вверх]

15. Библиотека языка Си и файлы ввода-вывода

БИБЛИОТЕКА ЯЗЫКА СИ
ФАЙЛЫ В ЯЗЫКЕ СИ
ФУНКЦИИ РАБОТЫ С ФАЙЛОМ
МАКРООПРЕДЕЛЕНИЯ ДЛЯ ПРОВЕРКИ СИМВОЛОВ
ФУНКЦИИ РАСПРЕДЕЛЕНИЯ ПАМЯТИ

Всякий раз, когда нам нужно использовать такие функции, как **printf( )**, **getchar( )** и **strlen( )**, мы обращаемся в библиотеку языка Си. Она содержит множество функций и макроопределений. Библиотеки меняются от системы к системе, но есть ядро функций (называемое стандартной библиотекой), которое используется чаще всего. В этой главе мы рассмотрим пятнадцать наиболее общих из этих функций, уделяя больше внимания функциям ввода-вывода и использованию файлов.
Однако сначала давайте поговорим о том, как пользоваться библиотекой.

ДОСТУП В БИБЛИОТЕКУ ЯЗЫКА СИ

Далее Содержание

Получение доступа к библиотеке зависит от системы, поэтому вам нужно посмотреть в своей системе, как применять наиболее распространенные операторы. Во-первых, есть несколько различных мест расположения библиотечных функций. Например, **getchar( )** обычно задают как макроопределение в файле **stdio.h**, в то время как **strlen( )** обычно хранится в библиотечном файле. Во-вторых, различные системы имеют разные способы доступа к этим функциям. Вот три из них.

Автоматический доступ

Далее Содержание

Во многих больших системах UNIX вы только компилируете программы, а доступ к более общим библиотечным функциям выполняется автоматически.

Включение файла

Далее Содержание

Если функция задана как макроопределение, то можно директивой **#include** включить файл, содержащий ее определение. Часто подобные функции могут быть собраны в соответствующим образом названный заголовочный файл. Например, некоторые системы имеют файл **ctype.h**, содержащий макроопределения, задающие тип символа: прописная буква, цифра и т. д.

Включение библиотеки

Далее Содержание

На некотором этапе компиляции или загрузки программы вы можете выбрать библиотеку. В нашей системе, например, есть файл **lc.lib**, содержащий скомпилированную версию библиотечных функций, и мы предлагаем редактору связей IBM PC использовать эту библиотеку. Даже система, которая автоматически контролирует свою стандартную библиотеку, может иметь другие библиотеки редко применяемых функций, и эти библиотеки следует запрашивать явно, указывая соответствующий признак во время компиляции.

Очевидно, мы не сможем рассмотреть все особенности всех систем, но эти три примера должны показать, что вас ожидает. Теперь давайте рассмотрим некоторые функции.

БИБЛИОТЕЧНЫЕ ФУНКЦИИ, КОТОРЫЕ МЫ ИСПОЛЬЗОВАЛИ

Далее Содержание

Пока мы хотим только перечислить эти функции, чтобы напомнить о них. Сначала приведем функции ввода-вывода:

```
getchar( )          /* получение символа */
putchar( )          /* печать символа */
gefs( )             /* получение строки */
puts( )             /* печать строки */
scanf( )            /* получение форматированного ввода */
printf( )           /* печать форматированного вывода */
```

Затем приведем функции, работающие со строками:

```
strlen( )            /* нахождение длины строки */
strcmp( )           /* сравнение двух строк */
strcpy( )           /* копирование строки */
strcat( )            /* объединение двух строк */
```

К этому списку мы добавим функции открытия и закрытия файлов, связи с файлами, проверки и преобразования символов, преобразования строк, функцию выхода и функции распределения памяти.
Давайте сначала обратимся к проблеме связи между файлом и программой.

СВЯЗЬ С ФАЙЛАМИ

Далее Содержание

Часто нам бывает нужна программа получения информации от файла или размещения результатов в файле. Один способ организации связи программы с файлом заключается в использовании операций переключения **<** и **>**. Этот метод прост, но ограничен. Например, предположим, вы хотите написать диалоговую программу, которая спрашивает у вас названия книг (звучит фамильярно?), и вы намерены сохранить весь список в файле. Если вы используете переключение как, например, в

```
books > bklist
```

то ваши диалоговые приглашения также будут переключены на **bklist**. И тогда не только нежелательная чепуха запишется в **bklist**, но и пользователь будет избавлен от вопросов, на которые он, как предполагалось, должен отвечать.

К счастью, язык Си предоставляет и более мощные методы связи с файлами. Один подход заключается в использовании функции **fopen( )**, которая открывает файл, затем применяются специальные функции ввода-вывода для чтения файла или записи в этот файл и далее используется функция **fclose( )** для закрытия файла. Однако прежде чем исследовать эти функции, нам нужно хотя бы кратко познакомиться с сущностью файла.

ЧТО ТАКОЕ ФАЙЛ?

Далее Содержание

Для нас файл является частью памяти, обычно на диске, со своим именем. Мы считаем, например, **stdio.h** именем файла, содержащего некоторую полезную информацию. Для операционной системы файл более сложен, но это системные проблемы, а не наши. Однако мы должны знать, что означает файл для программы на языке Си. В предлагаемых для обсуждения функциях, работающих с файлами, язык Си "рассматривает" файл как структуру. Действительно, файл **stdio.h** содержит определение структуры файла. Вот типичный пример, взятый из IBM-версии компилятора Lattice C:

```
struct _ iobuf
{
char *_ptr;     /* текущий указатель буфера */
int_cnt;        /* текущий счетчик байтов */
char*_base;     /* базовый адрес буфера ввода-вывода*/
char_flag;      /* управляющий признак */
char _ file;    /* номер файла */
};
#define FILE struct_iobuf /* краткая запись */
```

Здесь мы не собираемся разбираться детально в этом определении. Главное состоит в том, что файл является структурой, и что краткое наименование шаблона файла - **FILE**. (Многие системы используют директиву **typedef** для установления этого соответствия.) Таким образом, программа, имеющая дело с файлами, будет использовать тип структуры **FILE**, чтобы делать так.
Имея это в виду, мы сможем лучше понять операции над файлами.

ПРОСТЫЕ ПРОГРАММЫ ЧТЕНИЯ ФАЙЛА: fopen( ), fclose( ), getc( ) и putc( )

Далее Содержание

Чтобы показать элементарные примеры использования файлов, мы составили небольшую программу, которая читает содержимое файла, названного test, и выводит его на экран. Вы найдете наши пояснения сразу после программы.

```
/* расскажите, что находится в файле "test" */
#include <stdio.h>
main( )
{
FILE *in; /* описываю указатель на файл */
int ch;
if((in = fopen("test", "r"))!=NULL)
/* открываю test для чтения, проверяя существует ли он */
/* указатель FILE ссылается теперь на test */
{
while((ch = getc(in) != EOF) /* получаю символ из in */
    putc(ch, stdout);  /* посылаю на стандартный вывод */
fclose(in);     /* закрываю файл */ }
else
        printf("я не смогла открыть файл \" test\" .\n");
}
```

Следует объяснить три основных момента: работу **fopen( )**, работу **fclose( )** и использование функций ввода-вывода файла. Займемся ими.

Открытие файла: fopen( )

Далее Содержание

Функцией **fopen( )** управляют три основных параметра. Первый - имя файла, который следует открыть. Он является и первым аргументом **fopen( );** в нашем случае это "test" .

Второй параметр [и второй аргумент **fopen( )**] описывает, как должен использоваться файл. Вот три основных применения файла:

"r": файл нужно считать

"w": файл нужно записать

"a": файл нужно дополнить

Некоторые системы предоставляют еще дополнительные возможности, но мы будем придерживаться этих. Заметим, что используемые нами коды являются строками, а не символьными константами; следовательно, они заключаются в двойные кавычки. При применении "**r**" открывается существующий файл. При двух других применениях тоже будет открываться существующий файл, но если такого файла нет, он будет создан.

Внимание: Если вы используете **"w"** для существующего файла, то старая версия его стирается, и ваша программа начинает на "чистом месте".

Третий параметр является указателем на файл; это значение возвращается функцией:

```
FILE *in;
in = fopen("test", "r");
```

Теперь **in** является указателем на файл **"test"**. С этого момента программа ссылается на файл при помощи указателя **in**, а не по имени **test**.

Если вы очень сообразительны, то теперь можете задать такой вопрос: "Если **fopen( )** возвращает указатель на **'FILE'** в качестве аргумента, то почему мы не должны объявить **fopen( )** как функцию указателя на **'FILE'** ?" Хороший вопрос. Ответ заключается в том, что это описание сделал для нас файл **stdio.h**, который содержит строку

```
FILE  *topen( );
```

Есть еще один важный момент относительно функции **fopen()**, которую мы использовали. Если **fopen()** не способна открыть требуемый файл, она возвращает значение **'NULL'** (определенное в **stdio.h** как **0**). Почему она не смогла открыть файл? Вы можете попросить ее считать файл, который не существует. Вот почему мы имеем в программе строку

```
if((in=fopen("test", "r"))!= NULL)
```

Заполнение диска, использование запрещенного имени или некоторые другие причины могут препятствовать открытию файла. Поэтому побеспокойтесь, чтобы их не было - маленькая ошибка может увести вас очень далеко.
Закрыть файл проще.

Закрытие файла: fclose( )

Далее Содержание

Наш пример показывает, как закрывать файл:

```
fclose(in);
```

Просто используйте функцию **fclose( )**. Заметим, что аргументом ее является **in**, указатель на файл, а не **test**, имя файла.

Для программы, более серьезной, чем эта, следовало бы посмотреть, успешно ли закрыт файл. Функция **fclose( )** возвращает значение **0**, если файл закрыт успешно, и **-1** в противном случае.

Текстовые файлы с буферизацией

Далее Содержание

Функции **fopen()** и **fclose()** работают с текстовыми файлами с "буферизацией". Под буферизацией мы понимаем, что вводимые и выводимые данные запоминаются во временной области памяти, называемой буфером. Если буфер заполнился, содержимое его передается в блок, и процесс буферизации начинается снова. Одна из основных задач **fclose( )** заключается в том, чтобы "освободить" любые частично заполненные буфера, если файл закрыт.

Текстовым считается файл, в котором информация запоминается в виде символов в коде ASCII (или аналогичном). Он отличается от двоичного файла, который обычно используется для запоминания кодов машинного языка. Функции ввода-вывода, о которых мы собираемся рассказать, предназначены только для работы с текстовыми файлами.

Ввод-вывод файла: getc( ) и putc( )

Далее Содержание

Две функции **getc( )** и **putc( )** работают аналогично функциям **getchar( )** и **putchar( )**. Разница заключается в том, что вы должны сообщить новичкам, какой файл следует использовать. Таким образом, наш "старый дружище"

```
ch = getchar( );
```

предназначен для получения символа от стандартного ввода, а

```
ch = getc(in);
```

- для получения символа от файла, на который указывает **in**. Аналогично функция

```
putc(ch, out);
```

предназначена для записи символа **ch** в файл, на который ссылается указатель **out** типа **FILE**. В списке аргументов функции **putc( )** этот символ стоит первым, а затем указатель файла. В нашем примере мы использовали

```
putc(ch, stdout);
```

где **stdout** является указателем на стандартный вывод. Таким образом, этот оператор эквивалентен

```
putchar(ch);
```

Действительно, оператор **putchar(ch)** определен директивой **#define** так же как функция **putc(ch, stdout)** определена в файле **stdio.h.** Этот ужасный файл к тому же определяет в директиве **#define** указатели **stdout** и **stdin** на стандартный вывод и стандартный ввод системы. Это выглядит довольно просто? Хорошо, добавим пару полезных новшеств.

ПРОСТАЯ ПРОГРАММА СЖАТИЯ ФАЙЛА

Далее Содержание

В нашем примере имя файла, который следовало открыть, было записано в программе. Мы нс обязаны считаться с этим ограничением. Используя аргументы командной строки, можно сообщить нашей программе имя файла, который хотим считать. В нашем следующем примере

(рис. 15.1) так и происходит. С помощью примитивного приема сжимается содержимое - остается только каждый третий символ. Наконец, сжатая версия размещается в новый файл, имя которого состоит из старого имени с добавкой **.red** (сокращение слова *reduced*). Обычно весьма важны первый и последний элементы (аргумент командной строки и добавка к имени файла). Само же сжатие имеет более ограниченное применение, но, как вы увидите, им можно пользоваться.

```
/* сожмите ваши файлы в 2-3 раза! */
#include <stdio.h>
main(argc, argv)
int argc;
char *argv[ ];
{
FILE *in, *out; /* описывает два указателя типа FILE */
int ch;
static char name[20]; /* память для имени выходного файла */
int count = 0;
if(argc < 2) /* проверяет, есть ли входной файл */
printf(" Извините, мне нужно имя файла в качестве аргумента.\n" );
else
{ if((in = fopen(argv[1], "r")) != NULL)
{
strcpy(name, argv[1]; /* копирует имя файла в массив */
strcat(name, ".red"); /* добавляет .red к имени */
out = fopen(name, "w"); /* открывает файл для записи */
while((ch = getc(in)) ! = EOF)
if( count ++  %3 ==0)
putc(ch, out); /* печатает каждый третий символ */
fclose(in);
fclose(out); }
else
    printf(" Я не смогла открыть файл\" %s\" .\n", argv[1]);
} }
```

РИС. 15.1. Программа сжатия файла.

Мы поместили программу в файл, названный **reduce** и применили эту программу к файлу, названному **eddy**, который содержал одну единственную строку

```
Даже Эдди нас опередил с детским хором.
```

Была выполнена команда

```
reduce eddy
```

и на выходе получен файл, названный **eddy.red**, который содержит

```
Дед спел тихо
```

Какая удача! Наш случайно выбранный файл сделал осмысленное сокращение.
Вот некоторые замечания по программе.

Вспомните, что **argc** содержит определенное количество аргументов, в число которых входит имя программного файла. Вспомните также, что с согласия операционной системы **argv[0]** представляет имя программы, т. е. в нашем случае **reduce**. Вспомните еще, что **argv[l]** представляет первый аргумент, в нашем случае **eddy**. Так как сам **argv[l]** является указателем на строку, он не должен заключаться в двойные кавычки в операторе вызова функции.

Мы используем **argc**, чтобы посмотреть, есть ли аргумент. Любые избыточные аргументы игнорируются. Помещая в программу еще один цикл, вы могли бы использовать дополнительные аргументы - имена файлов и пропускать в цикле каждый из этих файлов по очереди.

С целью создания нового имени выходного файла мы используем функцию **strcpy( )** для

копирования имени **eddy** в массив **name**. Затем применяем функцию **strcat( )** для объединения этого имени с **.red**.

Программа требует, чтобы два файла были открыты одновременно, поэтому мы описали два указателя типа **'FILE '**. Заметим, что каждый файл должен открываться и закрываться независимо от другого. Существует ограничение на количество файлов, которые вы можете держать открытыми одновременно. Оно зависит от типа системы, но чаще всего находится в пределах от 10 до 20. Можно использовать один и тот же указатель для различных файлов при условии, что они нс открываются в одно и то же время.

Мы не ограничиваемся использованием только функций **getc( )** и **putc( )** для файлов ввода-вывода. Далее мы рассмотрим некоторые другие возможности.

ВВОД-ВЫВОД ФАЙЛА: fprintf( ), fscanf( ), fgets( ) И fputs( )

Далее Содержание

Все функции ввода-вывода, которые мы использовали в предыдущих главах, имеют аналоги для ввода-вывода файла. Основное отличие состоит в том, что вам нужно использовать указатель типа **FILE**, чтобы сообщить новым функциям, с каким файлом им следует работать. Подобно **getc( )** и **putc( )** эти функции используются после функции **fopen( )**, открывающей файл, и перед **fclose( )**, закрывающей его.

Функции fprintf( ) и fscanf( )

Далее Содержание

Эти функции ввода-вывода работают почти как **printf( )** и **scanf( )**, но им нужен дополнительный аргумент для ссылки на сам файл. Он является первым в списке аргументов. Вот пример, иллюстрирующий обращение к этим функциям:

```
/* образец использования fprintf( ) и fscanf( ) */
#include <stdio.h>
main( )
{
FILE *fi;
int age;
fi = fopen("sam", "r"); /* считывание */
fscanf(fi, "%d", &age); /* fi указывает на sam */
fclose(fi);
fi = fopen("data", "a"); /* дополнение "/
fprintf(fi, "sam is  %d. \n", age); /* fi указывает на data */
fclose(fi);
}
```

Заметим, что можно было использовать **fi** для двух различных файлов, потому что мы закрыли первый файл, прежде чем открыть второй.

В отличие от **getc( )** и **putc( )** эти две функции получают указатель типа **FILE** в качестве первого аргумента. Две другие, описанные ниже, получают его в качестве последнего аргумента.

Функция fgets( )

Далее Содержание

Эта функция имеет три аргумента, в то время как **gets( )** имеет один. Вот пример ее использования:

```
/* считывает файл строка за строкой */
```

```
#include <stdio.h>
#define MAXLIN 80
main( )
{
FILE  *f1;
char *string [MAXLIN];
f1 = fopen("story", "r");
while(fgets(string, MAXLIN, f1) != NULL)
puts(string);
}
```

Первый из трех аргументов функции **fgets( )** является указателем на местоположение считываемой строки. Мы располагаем вводимую информацию в символьный массив **string**.

Второй аргумент содержит предельную длину считываемой строки. Функция прекращает работу после считывания символа новой строки или после считывания символов общим числом **MAXLIN - 1** (в зависимости от того, что произойдет раньше). В любом случае нуль-символ (**'\0'**) добавляется в самый конец строки.

Третий аргумент указывает, конечно, на файл, который будет читаться.

Разница между **gets( )** и **fgets( )** заключается в том, что **gets( )** заменяет символ новой строки на **'\0'**, в то время как **fgets( )** сохраняет символ новой строки.

Подобно **gets( )** функция **fgets( )** возвращает значение **NULL**, если встречает символ **EOF**. Это позволяет вам проверить, как мы и сделали, достигли ли вы конца файла.

Функция fputs( )

Далее Содержание

Эта функция очень похожа на **puts( )**. Оператор

```
fputs(" Вы были правы.", fileptr);
```

передает строку "Вы были правы." В файл, на который ссылается указатель **fileptr** типа **FILE**. Конечно, сначала нужно открыть файл при помощи функции **fopen( )**. В самом общем виде это выглядит так

```
status = fputs(указатель строки, указатель файла);
```

где **status** является целым числом, которое устанавливается в **EOF**, если **fputs( )** встречает **EOF** или ошибку.

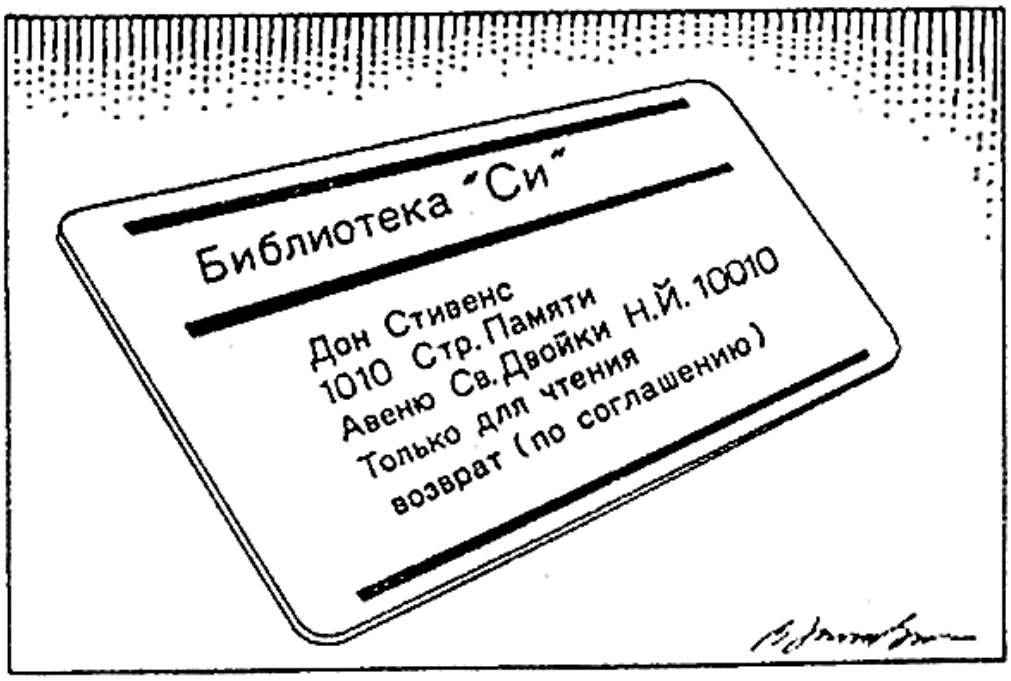
Подобно **puts( )** эта функция не ставит завершающий символ **'\0'** в конец копируемой строки. В отличие от **puts()** функция **fputs( )** не добавляет символ новой строки в ее вывод.

Шесть функций ввода-вывода которые мы только что обсудили, должны дать вам инструмент, для чтения и записи текстовых файлов. Есть еще одно средство, которое может оказаться полезным, и мы его сейчас обсудим.

ПРОИЗВОЛЬНЫЙ ДОСТУП: fseek( )

Далее Содержание

Функция **fseek( )** позволяет нам обрабатывать файл подобно массиву и непосредственно достигать любого определенного байта в файле, открытом функцией **fopen( )**. Вот простой пример, показывающий, как она работает. Как и в наших предыдущих примерах, функция использует аргумент командной строки для получения имени файла, с которым она работает. Заметим, что **fseek()** имеет три аргумента и возвращает значение типа **int**.




```
/*использование fscek( ) для печати содержимого файла */
#include <stdio.h&
main(number, names) /* не следует использовать argc и argv */
int number;
char  *names[ ];
{
FILE *fp;
long offset = OL;  /*обратите внимание, что это тип long */
if(number < 2)
puts("Мне нужно имя файла в качестве аргумента.");
else {
if((fp = fopen(names[1], "r")) ==  0)
printf(" Я не могу открыть %s.\n", names[1]);
else {
while(fseek(fp, offset++, 0) == 0)
putchar(getc(fp));
fclose(fp); }
    }
}
```

Перый из трех аргументов функции **fseek( )** является указателем типа **FILE** на файл, в котором ведется поиск. Файл следует открыть, используя функцию **fopen( )**.

Второй аргумент назван **"offset"** (вот почему мы выбрали данное имя для переменной). Этот аргумент сообщает, как далеко следует передвинуться от начальной точки (см. ниже); он должен иметь значение типа **long**, которое может быть положительным (движение вперед) или отрицательным (движение назад).

Третий аргумент является кодом, определяющим начальную точку:

Код Положение в файле

0 начало файла

1 текущая позиция

2 конец файла

Функция **fseek( )** возвращает **0**, если все хорошо, и **-1**, если есть ошибка, например попытка

перемещаться за границы файла. Теперь мы можем разъяснить наш маленький цикл:

```
while(fseek(fp, offset++, 0)==0)
        putchar(getc(fp));
```

Поскольку переменная **offset** инициализирована нулем, при первом прохождении через цикл мы имеем выражение

```
fseek(fp, 0L, 0)
```

означающее, что мы идем в файл, на который ссылается указатель **fp**, и находим байт, отстоящий на 0 байт от начала, т.е. первый байт. Затем функция **putchar( )** печатает содержимое этого байта. При следующем прохождении через цикл переменная **offset** увеличивается до **1L**, и печатается следующий байт. Посуществу, переменная **offset** действует подобно индексу для элементов файла. Процесс продолжается до тех пор, пока **offset** не попытается попасть в **fseek( )** после конца файла. В этом случае **fseek( )** возвращает значение - 1 и цикл прекращается.

Этот последний пример чисто учебный. Нам не нужно использовать **fseek( )**, потому что **getc( )** так или иначе проходит через файл байт за байтом; **fseek( )** приказала **getc( )** "посмотреть" туда, куда она сама уже собиралась посмотреть.

Вот пример (рис. 15.2), в котором выполняется что-то несколько более необычное (Мы благодарим Вильяма Шекспира за этот пример в пьесе "Двенадцатая ночь").

```
/* чередование печати в прямом и обратном направлениях */
#include <stdio.h>
main(number, names) /* вам не нужно применять argc и argv */
int number;
char *names[ ];
{
FILE *fp;
long offset = 0L;
if(number < 2)
puts(" Мне нужно имя файла в качестве аргумента.");
else {
if(fp = fopen(names[l], "r")) == 0)
printf(" Я не могу открыть %s.\n", names[l]);
else {
while(fseek(fp, offset++, 0) == 0)
{ putchar(getc(fp));
if(fseek(fp, -(offset + 3), 2) == 0)
putchar(getc(fp)); }
fclose(fp); }
} }
```

РИС. 15.2. Программа, чередующая печать в прямом и обратном направлениях.

Применение этой программы к файлу, содержащему имя "Мальволио", дает такой приятный результат:

```
Моаилльоввоьллиаом
```

Наша программа печатает первый символ файла, затем последний, затем второй, затем предшествующий последнему и т.д. Мы только добавили вот эти строки в последнюю программу:

```
if(fseek(fp, -(offset + 3), 2) == 0)
     putchar(getc(fp));
```

Код 2 в операторе предполагает, что мы будем считать позиции от конца файла. Знак минус означает счет в обратном направлении. **+3** стоит здесь потому, что мы начинаем с последнего регулярного символа файла и пропускаем несколько символов "новая строка" и **EOF** в самом

конце файла. (Точное значение этой корректировки зависит от типа системы. Наши файлы имеют в конце по два символа новой строки, за которыми следуют два **EOF**, поэтому мы как раз их и обходим.)

Таким образом, эта часть программы чередует печать в обратном направлении и печать в прямом направлении. Следует заметить, что в некоторых системах может не предусматриваться код 2 для **fseek( )**.
Теперь оставим на некоторое время файлы и перейдем к другому разделу библиотеки.

ПРОВЕРКА И ПРЕОБРАЗОВАНИЕ СИМВОЛОВ

Далее Содержание

Заголовочный файл **ctype.h** содержит несколько функций макроопределений, которые проверяют, к какому классу принадлежат символы. Функция **isalpha(c)**, например, возвращает ненулевое значение (истина), если с является символом буквы, и нуль (ложь), если символ не является буквой. Таким образом,

```
isalpha('S') != 0, но isalpha('#') ==0
```

Ниже перечислены функции, чаще всего находящиеся в этом файле. В каждом случае функция возвращает ненулевое значение, если с принадлежит к опрашиваемому классу, и нуль в противном случае.

ФУНКЦИЯ	ПРОВЕРЯЕТ, ЯВЛЯЕТСЯ ЛИ С
isalpha(c)	буквой
isdigit(c)	цифрой
islower(c)	строчной буквой
isspace(c)	пустым символом (пробел, табуляция или новая строка)
isupper(c)	прописной буквой

Ваша система может иметь дополнительные функции, такие как

ФУНКЦИЯ	ПРОВЕРЯЕТ, ЯВЛЯЕТСЯ ЛИ С
isalnum(c)	алфавитноцифровым (буква или цифра)
isascii(c)	кодом ASCII (0-127)
iscntrl(c)	управляющим символом
ispunct(c)	знаком пунктуации

Еще две функции выполняют преобразования:

toupper(c)	преобразует с в прописную букву
tolower(c)	преобразует с в строчную букву

В некоторых системах преобразование выполняется только в случае, если символ находится в регистре (прописных или строчных букв), противоположном тому, с которого следует начинать. Однако надежнее предварительно проверить регистр.

Ниже (рис. 15.2.) дана программа, использующая некоторые из этих функций для преобразования всего файла в прописные или строчные буквы, по вашему желанию. Для получения небольшого разнообразия используем диалоговый подход, вместо того чтобы применять аргументы командной строки для снабжения программы информацией.

```
/* преобразование строчных букв в прописные и обратно */
#include <stdio.h>
#include <ctype.h>  /* включает файл макроопределений */
#define  UPPER 1
#define  LOWER 0
main( )
{
int crit;  /* для установки регистра прописных или строчных букв */
char file1[14], file2[14]; /* имена входного и выходного файлов */
crit = choose( ); /* выбирает прописные или строчные буквы */
getfiles(file1, file2); /* получаст имена файлов */
conv(file1, file2, crit); /* выполняет преобразование */
} choose( )
{ int ch;
printf("Программа преобразует весь файл в прописные буквы или \n");
printf(" в строчные буквы. Вводит U, если нужны прописные буквы\n");
printf(" или вводит L, если нужны строчные буквы. \n");
while((ch=getchar( ))!='U' && ch!='u' && ch!='L'
         && ch!='l')
    printf(" Введите, пожайлуста, U или L.\n");
while(getchar( )!='\n')
;  /* сбрасывает последний символ новой строки */
if(ch =='U' ||  ch =='u')
{ printf(" Все в порядке, есть регистр прописных букв.");
return(UPPER);
else
{ printf(" Все в порядке, есть регистр строчных букв.");
return(LOWER); } }
getfiles(namel, name2);
char *namel, name2;
{ printf(" Какой файл вы хотите преобразовать?\n");
gets(name1);
printf(" Это\" %s\" .\n", name1);
printf("Какое имя вы хотите выбрать для преобразуемого файла?\n");
while(strcmp(gets(name2), name1) == NULL)
printf(" Выберите другое имя.\n" );
printf(" Ваш выходной файл\" %s \".\n", name2);
} conv(name1, name2, crit);
char *name1, name2;
int crit;
{ int ch;
FILE  *f1, *f2;
if((f1 = fopen(name1, "r" )) == NULL)
printf(Извините, я не могу открыть % s. До свидания.\n", name1);
else
{ puts(" Итак, начнем!");
f2 = fopen(name2, "w");
while((ch = getc(f1)) != EOF)
if(crit == UPPER)
ch = islower(ch) ? toupper(ch) : ch;
else
ch = isupper(ch) ? tolower(ch) : ch;
putc(ch, f2);
} fclosc(f2);
fclosc(f1);
puts("Сделано!");
} }
```

РИС. 15.3. Программа преобразования строчных букв в прописные и обратно.

Мы разделили программу на три части: получение от пользователя указания о виде преобразования, получение имени входного и выходного файлов и выполнение преобразования. Чтобы осуществить все это, мы создали разные функции для каждой части. Функция choose( ) довольно проста за исключением, может быть, цикла

```
while(getchar( ) != '\n');
```

Этот цикл включен для решения проблемы, с которой мы столкнулись в гл. 14. Когда пользователь отвечает на вопрос о виде преобразования, скажем, буквой **U**, он нажимает клавишу **U**, а затем клавишу [**ввод**], которая передает **'\n'**.

Первоначальная функция **getchar( )** извлекает **U**, но оставляет **'\n'** для следующего чтения строки. Функция **gets()**, входящая в **getnames()**, интерпретировала бы **'\n'** как пустую строку, поэтому мы использовали малый цикл **while**, чтобы избавиться от символа "новая строка". Действительно, простая **getchar( )**, сделала бы это, если бы пользователь непосредственно за **U** нажимал бы [**ввод**]. Но наша версия, кроме того, предусматривает возможность нажать на клавишу пробела несколько раз перед [**ввод**].

В функции **getnames( )** для вас не должно быть сюрпризов. Учтите, что мы запрещаем пользователю применять одинаковые имена для выходного и входного файлов. Стандартная версия функции **fopen( )** не позволяет вам и читать и записывать один и тот же файл, если вы открыли его один раз.

Функция **conv( )** является функцией копирования с выполнением преобразования. Значение **crit** используется для определения требуемого преобразования. Работа выполняется простым условным оператором, таким как

```
ch = islower(ch) ? toupper(ch) : ch;
```

Он проверяет, является ли **ch** строчной буквой. Если да, то символ преобразуется в прописную букву. Если нет, остается как есть.

Макрофункции файла **ctype.h** предоставляют удобные и полезные средства для программирования. Теперь давайте займемся некоторыми более сложными функциями преобразования.

ПРЕОБРАЗОВАНИЯ СИМВОЛЬНЫХ СТРОК: atoi( ), atof( )

Далее Содержание

Использование **scanf( )** для считывания цифровых значений не является самым надежным способом, поскольку **scanf( )** легко внести в заблуждение ошибками пользователей при вводе чисел с клавиатуры. Некоторые программисты предпочитают считывать даже числовые данные как символьные строки и преобразовывать строку в соответствующее числовое значение. Для этого используются функции **atoi( )** и **atof( )**. Первая преобразует строку в целое, вторая - в число с плавающей точкой. Вот (рис. 15.4) образец их использования:

```
/* включение atoi( ) */
#include <stdio.h>
#define issign(c) (((c) == '-' || (c) == '+') ? (1) : (0))
#define SIZE 10
#define YES 1
#define NO 0
main( )
{
char ch;
static char number[SIZE];
int value;
int digit = YES;
int count = 0;
puts(" Введите, пожалуйста, целое.");
gets(number);
```

```
if(number[SIZE - 1] != '\0')
{ puts("Слишком много цифр; вы уничтожили меня.");
exit(1);
} while((ch = number[count]) !='0' && digit == YES)
if(!issign(ch) && iisdigit(ch) && !isspace(ch))
    digit = NO;
if(digit == YES)
{ value = atoi(number);
printf(" Число было %d.\n" , value); }
else
printf(" Это не похоже  на целое.");
}
```

РИС. 15.4. Программа использования atoi( ).

Мы предусмотрели проверку некоторых ошибок. Во-первых, следует посмотреть, умещается ли входная строка в предназначенном для нее массиве. Поскольку **number** является статическим **символьным** массивом, он инициализируется нулями. Если последний элемент массива не является нулем, значит что-то неверно, и программа прекращает работу. Здесь мы использовали библиотечную функцию **exit( )**, которая выводит нас из программы. Немного позже мы расскажем кратко об этой функции.

Затем посмотрим, не содержит ли строка что-нибудь кроме пробелов, цифр и алгебраических знаков. Функция отвергает такие строки, как "дерево" или "1.2E2". Ее устраивает смесь, подобная "3 - 4 + 2", но **atoi( )** будет выполнять дальнейший отбор. Вспомним, что **!** является операцией отрицания, поэтому **!isdigit(c)** означает: "с не является цифрой". Строка

```
value = atoi(nuinbcr);
```

показывает, как используется функция **atoi( )**. Ее аргумент является указателем символьной строки; в этом случае мы применили имя массива **number**. Функция возвращает целое значение для такой строки. Таким образом, "1234" является строкой из четырех символов и переводится в 1234 - единое число типа **int**.

Функция **atoi( )** игнорирует ведущие пробелы, обрабатывает ведущий алгебраический знак, если он есть, и обрабатывает цифры вплоть до первого символа, нс являющегося цифрой. Поэтому наш пример "3 - 4 + 2" был бы превращен в значение 3. Посмотрите "Вопросы" в конце главы для возможного применения этой функции.

Функция **atof( )** выполняет подобные действия для чисел с плавающей точкой. Она возвращает тип **double**, поэтому должна быть описана как **double** в использующей ее программе.

Простые версии **atof( )** будут обрабатывать числа вида 10.2, 46 и - 124.26. Более мощные версии преобразуют также экспоненциальную запись, т. е. числа, подобные 1.25E - 13.

Ваша система может также иметь обратные функции, работающие в противоположном направлении. Функция **itoa( )** будет преобразовывать целое в символьную строку, а функция **ftoa( )** преобразовывать число с плавающей точкой в символьную строку.

ВЫХОД: exit( )

Далее Содержание

Функция **exit( )** даст вам удобный способ "покинуть" программу. Она часто используется для прекращения работы программы при появлении ошибки. Если к **exit( )** обратились из функции, вызванной главной программой, то прекращает работу вся программа, а не только эта функция. В приведенном выше примере с функцией **atoi( )** использование **exit( )** позволяет нам избежать включения дополнительного оператора **else** для обхода остатка программы.

Приятная способность **exit( )** заключается в том, что она закрывает любые файлы, открытые функцией **fopen( )**. Это делает наш выход из программы более корректным.

Аргументом **exit( )** является номер кода ошибки. В некоторых системах он может передаваться другой программе, если исходная прекратила работу. Существует соглашение, что **0** указывает на нормальное завершение, в то время как любое другое значение говорит об ошибке.
Есть еще одна тема, которую мы хотим обсудить.

РАСПРЕДЕЛЕНИЕ ПАМЯТИ: malloc( ) И ca11oc( )

Далее Содержание

Ваша программа должна предоставить достаточный объем памяти для запоминания используемых данных. Некоторые из этих ячеек памяти распределяются автоматически. Например, мы можем объявить

```
char place[ ] = "Залив Свиной печенки";
```

и будет выделена память, достаточная для запоминания этой строки.
Или мы можем быть более конкретны и запросить определенный объем памяти:

```
int plates[100];
```

Это описание выделяет 100 ячеек памяти, каждая из которых предназначена для запоминания целого значения.

Язык Си не останавливается на этом. Он позволяет вам распределять дополнительную память во время работы программы. Предположим, например, вы пишете диалоговую программу и не знаете заранее, сколько данных вам придется вводить. Можно выделить нужный вам (как вы считаете) объем памяти, а затем, если понадобится, потребовать еще. На рис. 15.5 дан пример, в котором используется функция **malloc( )**, чтобы сделать именно это. Кроме того, обратите внимание на то, как такая программа применяет указатели.

```
/* добавляет память, если необходимо */
#include <sldio.h>
#define STOP "  "   /* сигнал прекращения ввода */
#define BLOCK 100 /* байты памяти */
#define LIM 40     /* предельная длина вводимой строки */
#define MAX 50     /* максимальное число вводимых строк */
#define DRAMA 20000 /* большая задержка времени */
main( )
{
char store[BLOCK]; /* исходный блок памяти */
char symph[LIM]; /* приемник вводимых строк */
char *end;  /* указывает на конец памяти */
char *starts[MAX]; /* указывает на начала строк */
int index = 0; /* количество вводимых строк */
int count;  /* счетчик */
char *malloc( ); /* распределитель памяти */
starts[0] = store;
end = starts[0] + BLOCK - 1;
puts(" Назовите несколько симфонических оркестром.");
puts(" Вводите по одному: нажмите клавишу [ввод] в начале");
puts(" строки для завершения вашего списка. Хорошо, я готова." );
while(strcmp(fgets(symph, LIM, stdin), STOP) != 0 && index < MAX)
{ if(strlen(symph) > end - starts[index])
{ /* действия при недостатке памяти для запоминания вводимых данных*/
puts(" Подождите секунду. Я попробую найти дополнительную память.");
starts[index] = malloc(BLOCK);
end = starts[index] + BLOCK - 1;
```

```
for(count = 0; count < DRAMA; count++);
puts(" Нашла немного!" );  }
strcpy (starts [index], symph);
starts[index + 1] = starts[index] + strlen(symph) + 1;
if(++index < MAX)
    printf("Это %d. Продолжайте, если хотите.\n", index); }
puts(" Хорошо, вот что я получила:");
for(count = 0; count < index; count ++)
    puts(starts[count]);
}
```

РИС. 15.5. Программа, добавляющая память по требованию.
Вот образец работы программы:

```
Назовите несколько симфонических оркестров оркестров.
Вводите их по одному;  нажмите клавишу [ввод] в начале
строки для завершения нашего списка. Хорошо, я готова.
Сан-франциский симфонический.
Это 1. Продолжайте, если хотите.
Чикагский симфонический
Это 2. Продолжайте, если хотите.
Берлинский филармонический
Это 3. Продолжайте, если хотите.
Московский камерный
Это 4. Продолжайте, если хотите. Лондонский симфонический
Это 5. Продолжайте, если хотите. Венский филармонический
Подождите секунду.  Я попробую найти дополнительную память.
Нашла немного!
Это 6. Продолжайте, если хотите.
Питтсбургский симфонический
Это 7. Продолжайте, если хотите.

Хорошо, вот что я получила:
Сан-францизкий симфонический
Чикагский симфонический
Берлинский филармонический
Московский камерный
Лондонский симфонический
Венский филармонический
Питтсбургский симфонический
```

Сначала давайте посмотрим, что делает функция **malloc( )**. Она берет аргумент в виде целого без знака, которое представляет количество требуемых байтов памяти. Так, **malloc(BLOCK)** требует 100 байт. Функция возвращает указатель на тип **char** в начало нового блока памяти. Мы использовали описание

```
char *malloc( );
```

чтобы предупредить компилятор, что **malloc( )** возвращает указатель на тип **char**. Поэтому мы присвоили значение этого указателя элементу массива **starts[index]** при помощи оператора

```
starts[index] = malloc(BLOCK);
```

Хорошо, давайте теперь рассмотрим проект программы, заключающийся в том, чтобы запомнить все исходные строки подряд в большом массиве **store**. Мы хотим использовать **starts[0]** для ссылки на начало первой строки, **starts[l]** - второй строки и т. д. На промежуточном этапе программа вводит строку в массив **symph**. Мы использовали **fgets( )** вместо **gets( )**, чтобы ограничить входную строку длиной массива **symph**.

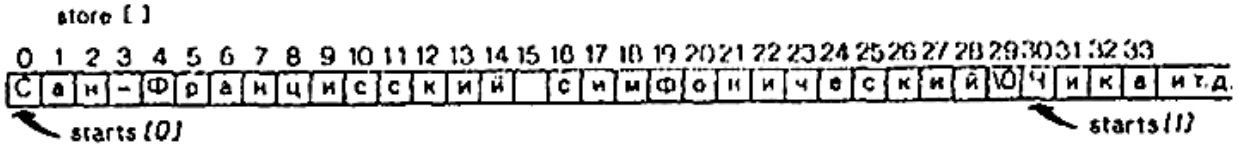


РИС. 15.6. Последовательные строки symph, записанные в массив store.

Прежде чем копировать **symph** в **store**, мы должны проверить, достаточно ли для нее оставшегося места. Указатель **end** ссылается на конец памяти, а текущее значение **starts[index]** ссылается на начало неиспользованной памяти. Таким образом, мы можем сравнить разницу между этими двумя указателями с длиной **symph** и определить, достаточно ли осталось памяти.

Если места недостаточно, вызываем **malloc( )**, чтобы подготовить дополнительную память. Мы устанавливаем **starts[index]** на начало нового блока памяти, а **end** - на конец нового блока. Заметим, что у нас нет имени этой новой памяти. Она не является, например, расширением **store**. У нас есть только обозначения указателей, ссылающихся на новую область памяти.

Когда программа работает, на каждую новую строку ссылается элемент массива указателей **starts**. Некоторые строки находятся в **store**, другие - в одной или нескольких новых областях памяти.
Но пока у нас есть указатели, мы можем работать со строками, как показывает нам часть программы, выполняющая вывод на печать.

Таким образом используется **ma11oc( )**. Но предположим, что вы хотите работать с памятью типа **int**, а не **char**. Можете и здесь использовать **ma11oc( )**. Вот как это делается:

```
char *malloc( ); /* по-прежнему описываем как указатель на char */
int *newmem;
newmem = (int *) malloc(l00); /* используем операцию приведения типа */
```

Снова требуется 100 байт. Операция приведения типа преобразует значение, возвращенное указателем на тип **char**, в указатель на тип **int**. Если, как в нашей системе, **int** занимает два байта памяти, это значит, что **newmem + 1** будет увеличивать указатель на два байта, т. е. передвигать его к следующему целому. Это также означает, что 100 байт можно использовать для запоминания 50 целых чисел.

Другую возможность распределения памяти дает нам применение функции **ca11oc( )**:

```
char *calloc( );
long *newmem;
newmem = (long *) calloc(100, sizeof(long));
```

Подобно **malloc( )** функция **ca11oc( )** возвращает указатель на **char**. Нужно использовать оператор приведения типа, если вы хотите запомнить другой тип. Эта новая функция имеет два аргумента, и оба они должны быть целыми без знака. Первый аргумент содержит количество требуемых ячеек памяти. Второй аргумент - размер каждой ячейки в байтах. В нашем случае **long** использует четыре байта, поэтому оператор выделит 100 четырехбайтных элементов, используя в целом 400 байтов памяти.

Применяя **sizeof (long)** вместо 4, мы сделали эту программу более мобильной. Она будет работать на системах, где **long** имеет размер, отличный от четырех.

Функция **ca11oc( )** имеет еще одну особенность; она обнуляет содержимое всего блока.
Ваша библиотека языка Си, вероятно, предоставляет несколько других функций управления памятью, и вы можете захотеть проверить их.

ДРУГИЕ БИБЛИОТЕЧНЫЕ ФУНКЦИИ

Далее Содержание

Большинство библиотек будут выполнять и ряд дополнительных функций в тех случаях, которые мы рассмотрели. Кроме функций, распределяющих память, есть функции, освобождающие память после работы с нею. Могут быть другие функции, работающие со строками, например такие, которые ищут в строке определенный символ или сочетание символов.

Некоторые функции, работающие с файлами, включают **open( )**, **close( )**, **create( )**, **fseek( )**, **read( )** и **write( )**. Они выполняют почти те же самые задачи, что и функции, которые мы обсудили, но на более фундаментальном уровне. Действительно, функции, подобные **fopen( )**, обычно пишутся с применением этих более общих функций. Они немного более трудны в использовании, но могут работать с двоичными файлами так же, как и с текстовыми.

Ваша система может иметь библиотеку математических функций. Обычно такая библиотека будет содержать функции квадратного корня, степенные, экспоненциальные, различные тригонометрические функции и функцию получения случайных чисел.

Вам нужно время, чтобы освоить то, что предлагает наша система. Если у нее нет того, что вам нужно, создайте свои собственные функции. Это часть языка Си. Если вы полагаете, что можете улучшить работу, скажем, функции ввода, сделайте это! А когда вы усовершенствуете и отшлифуете свои методы программирования, вы перейдете от обычного языка Си к блестящему языку Си.

ЗАКЛЮЧЕНИЕ

Далее Содержание

Мы прошли долгий путь от начала этого руководства. Теперь вы уже познакомились с большинством основных свойств языка Си. (Главное из того что, мы опустили,- операции с разрядами и расширения UNIX 7 - рассматриваются кратко в приложении Б). Вы узнали и использовали все изобилие его операторов, огромное разнообразие основных и производных типов данных, его "умные" управляющие конструкции и мощную систему указателей. Мы надеемся, что подготовили вас к использованию языка Си в ваших собственных целях. Поэтому начинайте программировать, и удачи вам!

ЧТО ВЫ ДОЛЖНЫ БЫЛИ УЗНАТЬ В ЭТОЙ ГЛАВЕ

Далее Содержание

Что такое библиотека языка Си и как ее использовать.
Как открывать и закрывать текстовые файлы: **fopen( )** и **fclose( )**
Что такое тип **FILE**
Как читать из файла и записывать в файл: **getc( )**, **putc( )**, **fgets( )**, **fputs( )**, **fscanf( )**, **fprintf( )**
Как проверять классы символов: **isdigit( )**, **isalpha( )** и т. д.
Как превращать строки в числа: **atoi( )** и **atof( )**
Как осуществлять быстрый выход: **exit( )**
Как распределять память: **malloc( )**, **ca11oc( )**

ВОПРОСЫ И ОТВЕТЫ

Далее Содержание

Вопросы

1. Что неправильно в этой программе?

```
main( )
{ int *fp;
int k;
fp = fopen("желе");
for(k = 0; k < 30; k++)
fputs(fp, "Нанетта ест желе.");
fclose("желе");
}
```

2. Что будет делать следующая программа?

```
#include <stdio.h>
#include <ctype.h>
main(argc, argv)
int argc;
char *argv[ ];
{ int ch;
FILE *fp;
if((fp=fopen(argv[1], "r")) == NULL)
exit(1);
while((ch=getc(fp)) != EOF)
  if(isdigit(ch))
    putchar(ch);
fclose (fp);
}
```

3. Все ли правильно в выражении **isalpha(c[i])**, где **c** является массивом типа **char**. Что можно сказать о **isalpha(c[i ++])**?

4.Используйте функции классификации символе" для подготовки выполненния **atoi( )**.

5. Как вы могли бы распределить, память для размещения массива структур?

Ответы

1. Должна быть директива **#include <stdio.h>** для определения ее файлов. Следует описать указатель **fp** файла: **FILE *fp;** функция **fopen( )** должна иметь вид **fopen("желе", "w")**, или, может быть, включать "**a**" . Порядок аргументов в **fputs( )** должен быть обратным. Функция **fclose( )** требует указателя файла, а не имени файла: **fclose(fp)**.

2. Она будет открывать файл, заданный как аргумент командной строки, и выво дить на печать все цифры в файле. Программа должна проверять (но не делает этого), не аргумент ли это командной строки.

3. Первое выражение правильно, так как **c[i]** имеет значение типа **char**. Второе выражение не выводит компьютер из строя, но может давать непредсказуемый результат. Причина в том, что **isalpha( )** является макроопределением, у которого, по всей вероятности, аргумент появляется дважды в определяющем выражении (проверка на принадлежность к регистру строчных букв, а зачем - прописных букв) и это дает в результате два увеличения **i**. Лучше всего избегать использования оператора увеличения в аргументе макрофункции.

4.

```
#include <stdio.h>
#include <ctype.h>
#define issign(c) (((c) == '-' || (c) == '+') ? (1) : (0)) atoi(s);
```

```
char *s;
{
int i = 0;
int n, sign;
while(isspace(s[i]))
i ++; /* пропуск пустого символа */
sign = 1;
if(issign(s[i]))     /* установка необязательного знака   */
sign = (s[i++] == '+') ? 1 : -1;
for(n = 0; isdigit(s[i]); i++)
    n = 10*n + s[i] - '0';
return(sign * n);
}
```

5. Предположим, что **wine** является именем структуры. Эти операторы, надлежащим образом расположенные в программе, будут выполнять данную работу.

```
struct wine *ptrwine;
char *calloc( );
ptrwine = (struc wine *) calloc(100, sizcof(struct wine));
```

УПРАЖНЕНИЯ

1. Напишите программу копирования файла, которая использует имена исходного файла файла и копируемого файла как аргументы командной строки.

2. Напишите программу, которая будет принимать все файлы, заданные радом аргументов командной строки, и печатать их один за другим. Используйте **argc** для создания цикла.

3. Модифицируйте вашу программу инвентаризации книг в гл. 14 так, чтобы информация, которую вы вводите, добавлялась в файл, названный **mybooks**.

4. Используйте **gets( )** и **atoi( )** для создания функции, эквивалентной нашей **getint( )** в гл. 10.

5. Перепишите нашу программу из гл. 7, считающую слова, используя макроопределения **ctype.h** и аргумент командной строки для обработки файла.

[Содержание] [Вверх]

Язык Си **Библиотека языка Си и файлы ввода-вывода**

М. Уэйт, С. Прата, Д. Мартин **Язык Си**

[Содержание] [Вниз]

Приложения

ПРИЛОЖЕНИЕ А

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

Если вы хотите больше узнать о языке Си и вообще о программировании, то найдете полезной следующую литературу:

Язык Си

Далее

Kernighan Brian W. and Ritchie Dennis M., *The C Programming Language*, Prentice-Hall, 1978. (Имеется перевод: КЕРНИГАН Б., Ритчи Д. *Язык програлширования Си*.- М.: Финансы и статисти ка, 1985.)
Первая и наиболее авторитетная книга по языку Си. (Заметим. что один из авторов этой книги Деннис Ритчи - создатель языка Си.) Практически она является официальным описанием языка и включает много интересных примеров. Однако авторы предполагают, что читатель знаком с системным программированием.

Feuer Alan R., *The C Puzzle Hook*, Prentice-Hall, 1982. (Имеется перевод: Фьюэр А. *Задачи по языку Си.*- М.: Финансы и статистика, 1985.)
Книга содержит большое количество программ, результат работы которых вы можете предсказать. Она дает хорошую возможность проверить и расширить ваши знания о языке Си. Книга включает ответы и пояснения.

Ritchie D. M., Johnson S. C., Lesk M. E., and Kernighan B. W., *The C Programming Language*, The Bell System Technical Journal, Vol. 57, No. 6, July-August 1978.
В статье обсуждается история создания языка Си и дается обзор особенностей программирования с использованием этого языка.

***BYTE*, Vol. 8, No. 8, August 1983.**
Этот выпуск журнала "Байт" посвящен языку Си. Он содержит статьи, где обсуждаются история его создания, концепции и применения. Проверяются и оцениваются двенадцать компиляторов языка Си для микропроцессоров. Включена также обширная современная библиография книг и статей по языку Си. Каждая книга и статья включает краткое содержание.

Программирование

Далее Содержание

Kernighan Brian W. and Plauger P. J., *The Elements of Programming Style (Second Edition)*, McGraw-Hill, 1978.
В этом стройном классическом произведении используются примеры, взятые из других книг, для иллюстрации того, что нужно и что не нужно делать для однозначного и результативного программирования.

Kernighan Brian V. and Plauger P. J., *Software Tools*, Addison-Wesley, 1976.
В книге описывается несколько полезных программ и программных систем, причем делается упор на эффективное проектирование структур программ. Представлены описания языка RATFOR (рационализированного Фортрана) и одной из версий Паскаля. Так как создание языка RATFOR - это попытка сделать сходной работу языков Фортран и Си, он лучше всего подходит пользователям языка как для знакомства с ним.

Операционная система UNIX

Далее Содержание

Waite Mitchell, Martin Don and Praia Stephen , *UNIX Primer Plus*, Howard W. Sams and Company,

Inc., 1983.
Эта книга - легко читаемое введение в операционную систему UNIX. В нее включены некоторые мощные расширения этой системы, реализованные в Калифорнийском университете (Беркли).

ПРИЛОЖЕНИЕ Б

Далее Содержание

КЛЮЧЕВЫЕ СЛОВА ЯЗЫКА СИ

Ключевые слова в языке являются словами, выражающими действия этого языка. Ключевые слова языка Си зарезервированы, т. е. вы не можете использовать их для других целей, таких как задание имени переменной.

Ключевые слова выполнения программы

Циклы
for while do

Принятие решения и выбор
if else switch case default

Переходы
break continue goto

Типы данных
char int short long unsigned float double struct union typedef

Классы памяти
auto extern register static

Разное
return sizeof

Еще не реализованное
entry

Применяемые только в некоторых системах
asm endasm fortran enum

ПРИЛОЖЕНИЕ В

Далее Содержание

ОПЕРАЦИИ ЯЗЫКА СИ

В языке Си предусмотрено множество операций. Затем мы приводим здесь таблицу операций, располагая их по приоритетам и показывая порядок выполнения. Мы рассказываем о всех операциях, за исключением поразрядных, которые будут рассмотрены в приложении Е.

I. Арифметические операции

+	Прибавляет величину, находящуюся справа, к величине, стоящей слева
-	Вычитает величину, стоящую справа, из величины, указанной слева
-	Будучи унарной операцией, изменяет знак величины, стоящей справа
*	Умножает величину справа на величину, находящуюся слева
/	Делит величину, стоящую слева, на величину справа. Результат усекается, если оба операнда целые числа
%	Дает остаток от деления величины слева на величину, стоящую справа (только для целых чисел)
++	Прибавляет 1 к значению переменной, стояшей слева (префикная форма), или к переменной, стояшей справа (постфиксная форма)
--	Аналогично ++, но вычитает 1

Операции (от высшего приоритета к низшему)

Порядок выполнения	
() {} -> .	Л-П
! ~ ++ -- - (тип) * & sizeof (все унарные)	П-Л
* / %	Л-П
+ -	Л-П
<< >>	Л-П
< <= > >=	Л-П
== !=	Л-П
&	Л-П
/\	Л-П
|	Л-П
&&	Л-П
||	Л-П
?:	Л-П
= += -= *= /* %=	П-Л
,	Л-П

Условные обозначения: Л-П - порядок выполнения слева направо, а П-Л - наоборот.

II. Операции присваивания

= Присваивает значение, указанное справа, переменной, стоящей слева

Каждая из приводимых ниже операции изменяет переменную, стоящую слева, на величину, находящуюся справа. Мы используем следующие обозначения: П для правой части и Л для левой.
+ = прибавляет величину П к переменной Л

-= вычитает величину П из переменной Л
*= умножает переменную Л на величину П
/= делит переменную Л на величину П
%= выдает остаток от деления переменной Л на величину П

Пример:

rabbits *= 1.6; то же самое, что **rabbits = rabbits * 1.6**;

III. Операции отношения

Каждая из этих операций сравнивает значение слева со значением справа. Оператор отношения, состоящий из операции и двух ее операндов, имеет значение 1, если выражение истинно, и значение 0, если выражение ложно.

< меньше
<= меньше или равно
== равно
>= больше или равно
> больше
!= не равно

IV. Логические операции

Обычно логические операции "считают" условные выражения операндами. Операция **!** имеет один операнд, расположенный справа. Остальные операции имеют два операнда: один слева и один справа.

&&	Логическое И: результат операции имеет значение "истина", если оба операнда истинны, и "ложь" в противном случае.
||	Логическое ИЛИ: результат операции имеет значение "истина", если один или оба операнда истинны, и "ложь" в противном случае.
!	Логическое НЕ: результат имеет значение "истина", если операнд имеет значение "ложь", и наоборот.

V. Операции над указателями

&	Операция получения адреса: выдаст адрес переменной, имя которой стоит за обозначением операции: **&nurse** является адресом переменной **nurse**
*	Косвенная адресация: выдает значение, записанное по адресу, на который ссылается указатель:

```
nurse = 22;
ptr = &nurse; /* указатель на nurse */
val = *ptr
```

В результате работы этих операторов переменной **val** присваивается значение 22.

VI. Операции над структурами и объединениями

. Операция принадлежности (точка) используется совместно с именем структуры (или объединения) для задания элемента этой структуры (иди объединения). Если name является именем структуры, а **member** - элементом, указанным в структурном шаблоне, то **name.member** определяет этот элемент структуры. Операцию принадлежности можно аналогичным образом применять и к объединениям.

Пример:

```
struct {
        int code;
        float cost;
                        } item;
item.code = 1265;
```

Выполняется присваивание значения элементу **code** структуры **item**.

-> Косвенная адресация (определение принадлежности) элемента используется совместно с указателем на структуру (или объединение) для идентификации того или иного элемента этих структур (или объединения). Предположим, что **ptrstr** является указателем на структуру, а **member** - элементом, указанным в структурном шаблоне. Тогда **ptrstr -> member** определяет, что это элемент структуры, на которую ссылается указатель. Операцию косвенной адресации элемента можно применять аналогичным образом и к объединениям.

Пример:

```
struct {
        int code;
        float cost;
} item, *ptrst;
ptrst = &item;
ptrst-> code = 3451;
```

Эти операторы присваивают значение элементу **code** структуры **item**. Следующие три выражения эквивалентны:

ptrst -> code item.code (*ptrst).code

VII. Дополнительные операции

sizeof Выдает размер (в байтах) операнда, стоящего справа.
Операндом может быть обозначение типа, заключенное в скобки, как, например, в **sizeof(float)**, либо имя конкретной переменной или определенного массива и т. д., как, например, в **sizeof foo**.

(type) Операция приведения типа: превращает следующее за ней значение в тип, определенный ключевым словом (или словами), заключенным в скобки. Например, **(float)9** превращает целое 9 в число с плавающей точкой 9.0. Операция "запятая" связывает два выражения в одно и гарантирует, что левое выражение будет вычислено первым. Типичным примером использования является включение дополнительной информации в управляющее выражение цикла **for**:

```
for(step = 2, fargo = 0; fargo < 1000; step *= 2)
    fargo + = step;
```

?: Операция условия имеет три операнда, каждый из которых является выражением. Они расположены следующим образом:
выражение1 ? *выражение2* : *выражение3*. Значение всего выражения равно значению *выражения2*, если *выражение1* истинно, и значению *выражения3* в противном случае.

Примеры:

```
(5 > 3) ? 1 : 2 имеет значение 1
(3 > 5) ? 1 : 2 имеет значение 2
(a > b) ? a : b имеет значение большего a и b
```

ПРИЛОЖЕНИЕ Г

Далее Содержание

ТИПЫ ДАННЫХ И КЛАССЫ ПАМЯТИ

Основные типы данных

Ключевые слова: Основные типы данных определяются с помощью следующих семи ключевых слов: **int, long, short, unsigned, char, float, double**

Целые со знаком: Могут иметь положительные и отрицательные значения.
int: основной тип целых чисел для конкретной системы.
long или **long int**: могут иметь целое как минимум такого же размера, как самое большое **int** и, возможно, больше.
short или **short int**: самое большое целое типа **short** не больше самого большого **int**, а возможно, и меньше. Обычно **long** должно быть больше **short**, а **int** должно быть таким же, как одно из них. Например, версия языка Си Lattice C для компьютера IBM PC имеет 16-разрядные целые типа **short** и **int** и 32-разрядные **long**. Все это зависит от используемой системы.

Целые без знака: имеют только нулевые и положительные значения. Они не могут быть больше самого большого возможного положительного числа.
Ключевое слово: **unsigned** используется перед обозначением типа: **unsigned int**, **unsigned long**, **unsigned short**.
Отдельно стоящее **unsigned** обозначает то же самое, что и **unsigned int**.

Символы: это типографские знаки, такие, как **A**, **&** и **+**.
Обычно каждый из них занимает в памяти только один байт.
char: ключевое слово для этого типа.

Числа с плавающей точкой: они могут иметь положительные и отрицательные значения.
float: основной (для используемой системы) размер чисел с плавающей точкой.
double или **long float**: больший (возможно) элемент для размещения чисел с плавающей точкой. С его помощью в принципе можно использовать больше значащих цифр и, возможно, больший порядок.

Как описать простую переменную:

Далее Содержание

1. Выберите необходимый тип.
2. Выберите имя для переменной.
3. Используйте следующий формат для оператора описания:

обизначение-типа имя-переменной;

Обозначение-типа состоит из одного или более ключевых слов типа. Вот несколько примеров:

```
int erest;
unsigned short cash;
```

4. Можно описать более чем одну переменную одного и того же типа, разделив имена переменных запятыми:

```
char ch, init, ans;
```

5. Можно инициализировать ту или иную переменную в операторе описания:

```
float mass = 6.0E24;
```

Классы памяти

Далее Содержание

I. Ключевые слова:
auto, external, static, register

II. Основные замечания

Класс памяти переменной определяет область ее действия и продолжительность использования. Класс памяти определяется местом задания переменной и соответствующим ключевым словом. Переменные, определенные вне функции, являются внешними и имеют глобальную область действия. Переменные, описанные внутри функции, являются автоматическими и локальными, если не используется какое-либо другое ключевое слово. Внешние переменные, определенные раньше функции, "известны" ей, даже если они не описаны внутри ее.

III. Свойства

Классы, перечисленные выше пунктирной линии, описываются внутри функции.
Классы, перечисленные ниже этой линии, определяются вне функции.

Класс памяти	*Ключевое слово*	*Продолжительность*	*Область деиствия*
Автоматический	auto	Временно	Локальная
Статический	static	Постоянно	Локальная
Внешняя	extern	Постоянно	Глобальная (все файлы)
Внешняя статическая	static	Постоянно	Глобальная (один файл)

ПРИЛОЖЕНИЕ Д

Далее Содержание

УПРАВЛЕНИЕ ХОДОМ ВЫПОЛНЕНИЯ ПРОГРАММЫ

Язык Си имеет несколько конструкций, предназначенных для управления выполнением программы. Здесь мы кратко описываем операторы циклов (**while, for** и **do while**), ветвлений (**if, if else** и

switch) и переходов (**goto**, **break** и **continue**).

Оператор while

Далее

Ключевое слово: while
Общие замечания:

Оператор **while** создает цикл, который повторяется до тех пор, пока проверяемое *выражение* не станет ложным, или нулем. Оператор **while** является циклом с *предусловием*, решение о прохождении цикла принимается *до* прохождения цикла. Поэтому возможно, что цикл никогда не будет пройден. Часть такой конструкции, относящаяся к *оператору*, может быть простым или составным оператором.

Форма записи:
while(*выражение*) *оператор*;

"*Оператор*" повторяется до тех пор, пока *выражение* не станет ложным, или нулем.

Примеры:

```
while(n++ < 100)
        printf(" %d %d\n", n, 2*n + 1);

while(fargo < 1000) {
  fargo = fargo + step;
  step = 2 * step;   }
```

Оператор for

Далее Содержание

Ключевое слово: for
Общие замечания:

Оператор **for** для управления циклическим процессом использует три выражения, разделенные символами "точка с запятой". *Инициализирующее* выражение выполняется один раз, до выполнения любого из операторов цикла. Если *проверяемое* выражение истинно (или не нуль), цикл должен быть пройден один раз. Затем выполняется *корректирующее* выражение, и нужно снова проанализировать *проверяемое* выражение. Оператор **for** является циклом с *предусловием*: *до* прохождения цикла выполняется проверка, проходить ли этот цикл еще один раз. Поэтому возможно, что цикл никогда не будет пройден. Часть такой конструкции, относящаяся к оператору, может быть простым или составным оператором.

Форма записи: for(*инициализация*; *проверка условия*; *коррекция*) *оператор*;

Цикл повторяется до тех пор, пока *проверяемое* выражение не станет ложным, или нулем.

Пример:

```
for(n = 0; n < 10; n++)
        printf(" %d %d\n", n, 2*n + 1);
```

Оператор do while

Далее Содержание

Ключевые слова: do, while

Общие замечания:

Оператор **do while** создает цикл, который повторяется до тех пор, пока *выражение*, проверяющее условие, не станет ложным, или нулем. Оператор **do while** является циклом с *постусловием*; *после* прохождения цикла принимается решение, проходить ли его еще раз. Поэтому цикл должен выполняться по крайней мере один раз. Часть конструкции, относящаяся к *оператору*, может быть простым или составным оператором.

Форма записи:
do оператор while(*выражение*)

Оператор повторяется до тех пор, пока *выражение* не станет ложным, или нулем.

Пример:

```
do
scanf(" %d", &number) while(number != 20);
```

Использование операторов if для выбора вариантов

Далее Содержание

Ключевые слова: if, else

Общие замечания:

В каждой из следующих форм оператор может быть либо простым, либо составным оператором. Вообще говоря, "истинное" выражение означает выражение с ненулевым значением.

Форма 1:
if(*выражение*) *оператор*
Оператор выполняется, если *выражение* истинно.

Форма 2:
if(*выражение*)
оператор1 else
оператор2
Если *выражение* истинно, выполняется *оператор1*. В противном случае выполняется *оператор2*.

Форма 3:
if(*выражение1*)
оператор1 else if(*выражение2*)
оператор2 else
оператор3
Если *выражение1* истинно, то выполняется *оператор2*.
Если *выражение1* ложно, но *выражение2* истинно, выполняется *оператор2*.
Если же оба выражения ложны, выполняется *оператор3*.

Пример:

```
if(legs == 4)
        printf(" Это, возможно, лошадь.\n");
else if(legs > 4)
        printf(" Это не лошадь.\n");
else   /* выполнить, если legs < 4 */
        { legs++;
          printf(" Теперь у нее еще одна нога.\n");
        }
```

Множественный выбор при помощи switch

Далее Содержание

Ключевое слово: switch

Общие замечания:

Управление программой переходит к оператору, имеющему значение *выражения* в качестве метки. Затем программа продолжает выполняться, проходя остальные операторы, если снова не произойдет переключения направления. И *выражение*, и метки должны иметь целые значения (включая тип **char**), а метки должны быть либо константами, либо выражениями, состоящими только из констант. Если ни одна метка не соответствует значению выражения, управление передается оператору, помеченному **default**, если он существует. В ином случае управление передается оператору, следующему за оператором **switch**.

Форма записи:
switch(*выражение*)
{
    case метка1 : *оператор1;*
    case метка2 : *оператор2;*
    default   : *оператор3;*
}
Можно записывать более двух помеченных операторов, а вариант **default** не обязателен.

Пример:

```
switch(буква) {
case 'a' :
case 'e' : printf(" %d гласная\n", буква);
case 'c' :
case 'n' : printf(" %d в \" наборе cane \" \n", буква);
default  : printf(" Прекрасный день. \n");
}
```

Если буква имеет значение '**a**' или '**e**', печатаются все *три* сообщения; '**c**' и '**n**' вызывают печать двух последних строк. Все остальные значения приводят к печати последнего сообщения.

Переходы в программе

Далее Содержание

Ключевые слова: break, continue, goto

Общие замечания:

Эти три команды вызывают переход от одного оператора программы к другому, расположенному в ином месте (в теле программы).

break

Команду **break** можно использовать с любой из трех форм цикла и с оператором **switch**. Она приводит к тому, что управление программой "игнорирует" остаток цикла или оператор **switch**, содержащий этот остаток, и возобновляет выполнение с оператора, следующего за циклом или **switch**.

Пример:

```
switch(number) {
        case 4: printf(" Хороший выбор!\n");
        break;
        case 5: printf("Это неплохой выбор. \n");
        break;
        default: printf("Это плохой выбор.\n");
```

continue

Команда **continue** может использоваться с любыми тремя формами цикла, но не со **switch**. Она приводит к тому, что управление программой игнорирует оставшиеся операторы цикла. Для цикла **while** или **for** начинается следующий шаг цикла. Для цикла **do while** проверяется условие выхода, а затем, если нужно, начинается следующий шаг цикла:

Пример:

```
while((ch = getchar( )) != EOF) {
        if(ch == ' ') continue;
        putchar(ch);
        chcount ++; }
```

Этот фрагмент программы выполняет эхо-копирование и подсчет символов, не являющихся пробелами.

goto

Оператор **goto** вызывает передачу управления в программе оператору, помеченному указанной меткой. Для отделения помеченного оператора от его метки используется двоеточие. Метке присваивается имя по правилам, принятым для имени переменной. Помеченный оператор может находиться до или после оператора **goto**.

Форма записи: goto label;
...
label : statement

Пример:

```
top : ch = getchar( );
if(ch != 'y') goto top;
```

ПРИЛОЖЕНИЕ Е

Далее Содержание

МАНИПУЛЯЦИИ РАЗРЯДАМИ: ОПЕРАЦИИ И ПОЛЯ

Для некоторых программ необходима (илн по крайней мере полезна) возможность манипулировать отдельными разрядами в байте или слове. Например, часто варианты режимов устройств ввода-вывода устанавливаются байтом, в котором каждый разряд действует как признак "включено-выключено". В языке Си есть ДВА средства, помогающие манипулировать разрядами. Во-первых, набор из шести "поразрядных" операций, выполняющихся над разрядами. Во-вторых, форма данных, называемая **field** (поле), дающая доступ к разрядам переменной типа **int**. Теперь мы кратко опишем эти характерные черты языка Си.

Операции

Далее Содержание

В языке Си предусматриваются поразрядные логические операции и операции сдвига. Далее мы будем записывать значения в двоичном коде, чтобы вы могли видеть, как выполняются операции. В реальных программах используются целые переменные или константы, записанные в обычной форме. Например, вместо (00011001) можно использовать 25 или 031, либо даже 0x19. В наших примерах будут применяться 8-разрядные числа, в которых разряды пронумерованы от 7 до 0 слева направо.

Поразрядные логические операции

Четыре операции производят действия над данными, относящимися к классу целых, включая **char**. Они называются "поразрядными", потому что выполняются отдельно над каждым разрядом независимо от разряда, находящегося слепа или справа.

~ : Дополнение до единицы, или поразрядное отрицание

Эта унарная операция изменяет каждую 1 на 0, а 0 на 1. Поэтому

~(10011010) == (01100101)

&: Поразрядное И

Эта бинарная операция сравнивает последовательно разряд за разрядом два операнда. Для каждого разряда результат равен 1, если только оба соответствующих разряда операндов равны 1. (В терминах "истинно-ложно" результат получается истинным, если только каждый из двух одноразрядных операндов является истинным.) Так,

(10010011) & (00111101) == (00010001)

потому что только четвертый и первый разряды обоих операндов содержат 1.

| : Поразрядное ИЛИ

Эта бинарная операция сравнивает последовательно разряд за разрядом два операнда. Для каждого разряда результат равен 1, если любой из соответствующих разрядов операндов равен 1. [В терминах "истинно-ложно" результат получается истинным, если один из двух (или оба) одноразрядных операндов является истинным.] Так,

(10010011) | (00111101) == (10111111)

потому что все разряды, кроме шестого, в одном из двух операндов имеют значение 1.

^: Поразрядное исключающее ИЛИ

Эта бинарная операция сравнивает последовательно разряд за разрядом два операнда. Для каждого разряда результат равен 1, если один нз двух (но не оба) соответствующих разрядов операндов равен 1. [В терминах "истинно-ложно" результат получается истинным, если один из двух (но не оба) одноразрядных операндов является истинным.] Поэтому

(10010011) ^ (00111101) == (10101110)

Заметим, что, поскольку нулевой разряд в обоих операндах имеет значение 1, нулевой разряд результата имеет значение 0.

Применение

Описанные выше операции часто используются для установки некоторых разрядов, причем другие разряды остаются неизменными. Например, предположим, что мы определили **MASK** в директиве **#define MASK**, равным 2, т. е. двоичному значению 00000010, имеющему ненулевое значение только в первом разряде. Тогда оператор

flags = flags & MASK;

установит все разряды **flags** (кроме первого) в 0, потому что любое значение разряда при выполнении операции. & дает 0, если разряд второго операнда равен 0. Однако первый разряд останется неизменным. (Если первый разряд операнда содержит 1, то результат операции 1 & 1 равен 1, а если первый разряд имеет значение 0, то 0 & 1 будет равно 0.) Аналогично оператор

flags = flags | MASK;

установит первый разряд в 1 и оставит все остальные разряды неизменными. Это происходит потому, что любое значение разряда при выполнении операции 1, если разряд второго операнда равен нулю, остается без изменения, а если разряд второго операнда равен 1, то первый разряд результата будет иметь значение 1.

Поразрядные операции сдвига

Эти операции сдвигают разряды влево или вправо. Мы снова запишем двоичные числа в явной форме, чтобы подробно показать механизм сдвига.

&& : Сдвиг влево

Эта операция сдвигает разряды левого операнда влево на число позиций, указанное правым операндом. Освобождающиеся позиции заполняются нулями, а разряды, сдвигаемые за левый предел левого операнда, теряются. Поэтому

(10001010) << 2 == 00101000

где каждый разряд сдвинулся на две позиции влево.

>> : Сдвиг вправо

Эта операция сдвигает разряды левого операнда вправо на число позиций, указанное правым операндом. Разряды, сдвигаемые за правый предел левого операнда, теряются. Для чисел типа unsigned позиции, освобождающиеся слева, заполняются нулями. Для чисел со знаком результат

зависит от типа ЭВМ. Освобождающиеся позиции могут заполняться нулями или значением знакового разряда (самого левого). Для значений без знака имеем

(10001010) >> 2 == (00100010)

где каждый разряд переместился на две позиции вправо.

Применение

Эти операции выполняют сдвиг, а также эффективное умножение и деление на степени 2:

number << n умножает **number** на 2 в n-й степени
number >> n делит **number** на 2 в n-й степени, если число неотрицательное.

Это аналогично соответствующему алгоритму для десятичной системы счисления, обеспечивающему сдвиг десятичной точки при умножении или делении на 10.

Поля

Далее Содержание

Второй способ манипуляции разрядами заключается в использовании поля. Полем считается последовательность соседних разрядов в числе типа **int** или **unsigned int**. Поле устанавливается при помощи определения структуры, в котором помечается каждое поле и определяется его размер. Следующее описание устанавливает четыре 1-разрядных поля:

```
struct {
unsigned autfd: 1;
unsigned bldfc: 1;
unsigned undln: 1;
unsigned itals: 1;
} prnt;
```

Переменная **prnt** содержит теперь четыре 1-разрядных ноля. Обычную операцию принадлежности элемента структуры можно использовать для присвоения значении отдельным полям:

```
prnt.itals = 0;
prnt.undln = 1;
```

Поскольку каждое поле состоит только из одного разряда, мы можем использовать для присваивания лишь значение 0 или 1.

Переменная **prnt** запоминается в ячейке памяти, имеющей размер, равный длине числа типа **int**, но для нашего примера используется только четыре разряда.

Размер поля не ограничивается одним разрядом. Мы можем делать, например, так:

```
struct {
unsigned code1 : 2;
unsigned code2 : 2;
unsigned code3 : 8;
} prcode;
```

Таким путем создаются два 2-разрядных поля и одно 8-разрядное. Мы можем выполнять присваивания, подобные следующим:

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 102;
```

Удостоверьтесь только, что значение не превышает размер поля.

Что произойдет, если общее число объявленных вами разрядов превысит размер переменной типа **int**? В этом случае используется следующая ячейка памяти типа **int**. Одиночное поле не может перекрывать границу между двумя **int**, компилятор автоматически сдвигает определение перекрывающего поля таким образом, чтобы данное поле было выравнено по границе **int**. Если это происходит, он оставляет в первом **int** безымянное "пустое место".

Вы можете заполнить структуру поля с безымянными пустыми местами, используя поле без имени. Применение поля без имени с размером 0 выравнивает очередное поле по границе следующего целого:

```
struct {
        field1 : 1;
               : 2;
        field2 : 1;
               : 0;
        field3 : 1; } stuff;
```

Здесь есть 2-разрядный промежуток между **stuff.field1** и **stuff.field2**, а **stuff.field3** запоминается в следующем **int**.

Порядок размещения полей в **int** зависит от типа ЭВМ. В одних машинах поля располагаются слева направо, в других - справа налево.

ПРИЛОЖЕНИЕ Ж

Далее Содержание

ДВОИЧНЫЕ И ДРУГИЕ ЧИСЛА

Двоичные числа

В основе способа, который мы обычно используем для записи чисел, лежит число 10. Может быть, вы когда-то слышали, что число 3652 имеет 3 в позиции тысяч, 6 в позиции сотен, 5 в позиции десятков и 2 в позиции единиц. Поэтому мы можем представить число 3652 в виде

3 ґ 1000 + 6 ґ 100 + 5 ґ 10 + 2 ґ 1

Однако 1000 - это 10 в кубе, 100 - десять в квадрате, 10 - десять в первой степени, а 1, как принято в математике, 10 (или любое положительное число) в нулевой степени. Следовательно, мы можем записать 3652 как

$3 ґ 10^3 + 6 ґ 10^2 + 5 ґ 10^1 + 2 ґ 10^0$

Так как наша система записи чисел основывается на степенях десяти, мы можем сказать, что 3652 записывается *по основанию* 10.

Вероятно, мы создали такую систему потому, что имеем 10 пальцев на руках. Компьютер же, в каком-то смысле, имеет только два "пальца", поэтому его можно установить только в состояние 0 или 1 (выключено или включено). Это делает систему с **основанием** 2 естественной для компьютера. Как она работает? Используются степени 2 вместо степеней 10. Например, такое двоичное число, как 1101, означало бы

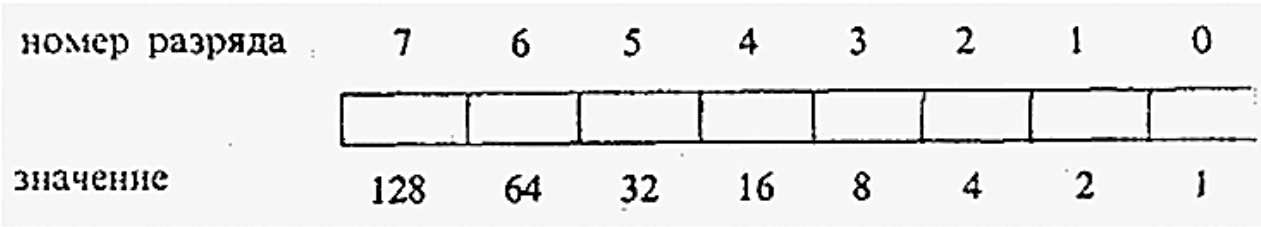
$1 ґ 2^3 + 1 ґ 2^2 + 0 ґ 2^1 + 1 ґ 2^0$

В десятичной записи оно становится равным

1 ґ 8 + 1 ґ 4 + 0 ґ 2 + 1 ґ 1 = 13

Система с основанием 2 (или "двоичная") позволяет выразите любое число (если у пас достаточно разрядов в двоичной системе, как комбинацию единиц и нулей. Это очень "приятно" для компьютера, особенно если учесть, что у него нет иного выбора. Посмотрим, как работает такой механизм для 1-байтного целого числа.

Можно считать его 8 разрядов пронумерованными слева направо от 7 до 0. Такие "номера разрядов" соответствуют степеням 2. Представьте себе, что байт выглядит примерно так:



Здесь 128 - это 2 в 7-и степени и т. д. Самое большое число, которое может содержать этот байт, имеет во всех разрядах 1 : 11111111. Значение такого двоичного числа

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

Самое маленькое двоичное число было бы равно 00000000, или просто 0. Байт может содержать числа от 0 до 255 для всех 256 возможных значений.

Двоичные числа с плавающей точкой

Далее Содержание

Числа с плавающей точкой хранятся в памяти в виде двух частей: двоичной дроби и двоичного порядка. Посмотрим, как это делается.

Двоичные дроби

Обычную дробь .324 можно представить в виде

3/10 + 2/100 + 4/1000,

где знаменатели - увеличивающиеся степени 10. В двоичной дроби мы используем в качестве знаменателей степени 2. Поэтому двоичную дробь .101 можно записать в виде

1/2 + 0/4 + 1/8,

что в десятичном виде даст

.50 + .00 + .125
или .625.

Многие дроби, такие как 1/3, нельзя точно предоставить десятичной форме, и аналогично многие дроби нельзя точно представить в двоичной форме. Действительно, только дроби, которые являются комбинациями чисел, кратных степеням 1/2, можно представить точно. Поэтому 3/4 и 7/8 можно точно представить как двоичные дроби, а 1/3 и 2/5 нельзя.

Представление чисел с плавающей точкой

Для представления в компьютере числа с плавающей точкой некоторое количество (в зависимости от системы) разрядов выделяется для хранения двоичной дроби и, кроме того, дополнительные разряды содержат показатель степени. В общем случае фактическое значение числа состоит из двоичной дроби, умноженной на 2 в указанной степени. Поэтому умножение числа с плавающей точкой, скажем, на 4 увеличивает показатель степени па 2 и оставляет двоичную дробь неизменной. Умножение на число, не являющееся степенью 2, изменяет двоичную дробь и, если необходимо, показатель степени.

Другие основания системы счисления

Далее Содержание

Пользователи компьютеров часто применяют системы счисления по основанию 8 или 16. Так как 8 и 16 являются степенями 2, эти системы более тесно связаны с двоичной системой счисления компьютеров, чем десятичная система.

Восьмеричные числа

"Восьмеричными" называются числа в системе счисления по основанию 8. В этой системе различные позиции в числе представляют степени числа 8. Мы используем для этого цифры от 0 до 7. Например, восьмеричное число 451 (записываемое как 0451 на языке Си) представляется в виде

$4 * 8^2 + 5 * 8^1 + 1 * 8^1 = 297$ (по основанию 10)

Шестнадцатеричные числа

"Шестнадцатеричными" (или hex) называются числа в системе по основанию 16. Поскольку у нас нет отдельных цифр для предстанления значении от 10 до 15, мы используем в этих целях буквы от A до F. Например, шестнадцатеричное число **A3F** (записанное как 0ꞌA3F на языке Си) представляется как

$10 * 16^2 + 3 * 16^1 + 15 * 16^0 = 2623$ (по основанию 10)

ПРИЛОЖЕНИЕ 3

Далее Содержание

"МУЗЫКА" В СИСТЕМЕ IBM PC

Громкоговорителем персонального компьютера IBM PC можно управлять, используя его порты ввода-вывода. В гл. 6 мы обсуждали, как применять порт 97 для возбуждения звукового сигнализатора компьютера IBM PC. Мы применяли специальные функции ввода-вывода **inp( )** и **outp( )**, которые предусмотрены в некоторых компиляторах с языка Си для систем IBM PC. Большинство компиляторов IBM PC позволяют также применять эквивалентные средства на языке ассемблера. Мы видели, как надо использовать циклы, реализующие временную задержку, для управления продолжительностью звучания; в этом приложении мы расширим наш подход, что позволит нам выбирать и частоту. Мы составим функцию, аргументами которой являются частота и продолжительность звучания. Затем покажем образец программы, использующей функцию **tone( )** для превращения части клавиатуры машины IBM PC в простую музыкальную клавиатуру.

Вот заголовок нашей функции:

```
tone(freq, time);
int freq, time;
```

Переменная **freq** описывает частоту тона, выражаемую в герцах (Гц), т. е. числом колебаний в секунду. Переменная **time** характеризует продолжительность звучания в десятых долях секунды, значение 10 для **time** означает продолжительность 10 десятых, или 1 секунда. Теперь мы должны разработать способы передачи этой информации на звуковоспроизводящее устройство. Сначала рассмотрим продолжительность звучания.

Продолжительность звучания

Мы можем регулировать продолжительность так, как было указано в гл. 6. Вспомним, что громкоговоритель управляется устройством, называемым "Программируемый параллельный интерфейсный контроллер 8255". Специальные каналы ввода-вывода, называемые портами, связывают этот и другие контроллеры с "мозгом" системы, микропроцессором 8088. Мы используем порт 97 для включения громкоговорителя, цикл, чтобы отмечать время, и затем порт 97 для отключения громкоговорителя. Вот фрагмент программы, которая будет выполнять эти действия:

```
#define TIMESCALE
1270         /* число отсчетов времени в 0,1 с */
#define BEEPPORT  97 /* порт управляет громкоговорителем */
#define ON 79         /* сигнал включения громкоговорителя */
count = TIMESCALE  *time;  /* преобразование времени
в единицы таймера */
port = inp(BEEPPORT);     /* запоминание состояния порта */
outp(BEEPPORT, ON);       /* включение громкоговорителя */
for(i = 0; i < count; i++)
;                          /* отметка времени */
outp(BEEPPORT, port);      /* выключение громкоговорителя, восстановление состояния */
```

Значение **count** (число отсчетов) дает время, в течение которого громкоговоритель включен. Коэффициент **TIMESCALE** преобразует десятые доли секунды в эквивалентное количество отсчетов времени. Конечно, мы должны установить требуемую частоту звука до того, как зазвучит громкоговоритель, поэтому рассмотрим этот параметр.

Частота звука

Частоту звука можно установить при помощи другого устройства, называемого "Программируемым интервальным таймером **8253**". Этот контроллер в числе прочего определяет, сколько импульсов в секунду следует послать на громкоговоритель. Устройство **8253** вырабатывает базовую частоту 1,190,000 Гц, которая значительно выше граничной частоты восприятия звука человеком. Однако мы можем послать на устройство **8253** число для деления этой базовой частоты. Например, если мы направляем туда 5000, то получаем частоту, следования импульсов

1,190,000/5000 = 238 Гц,

которая немного ниже среднего звука си (нота, а не версии более низкого класса рассматриваемого языка). Если мы знаем, какая частота **freq** нам нужна, можно вычислить требуемый делитель, скажем, так:

divisor = 1,190,000/freq;

Наша функция позволяет сделать это, в связи с чем нам нужно только знать, как подать значение переменной **divisor** на устройство **8253**. Теперь требуется использовать еще два порта.

Первый шаг заключается в установке таймера **8253** в правильный рабочий режим для приема делителя. Это достигается посылкой значения 182 (0ґB6 в шестнадцатеричном коде) через порт **67**. Как только такая посылка будет выполнена, можно использовать порт **66** для передачи делителя.

Посылка делителя представляет собой несложную задачу. Сам делитель является 16-разрядным числом, но его следует передавать двумя частями. Сначала мы посылаем младший байт, или последние 8 разрядов числа, а затем старший байт, т.е. начальные 8 разрядов числа. В следующей программе мы называем эти части **lobyt** и **hibyt** и вычисляем их значения через **divisor**:

```
lobyt = divisor % 256;
hibyt = divisor % 256;
```

Можно также использовать поразрядные операции:

```
lobyt = divisor & 255;
hibyt = divisor >> 8;
```

Первый оператор в каждой паре строк примеров устанавливает первые восемь разрядов в 0, оставляя в последних восьми разрядах первого байта число. Проверьте результаты операцией получения модуля и поразрядной операцией **И**, чтобы увидеть, как это делается. Второй оператор каждой пары берет исходное значение **divisor** и сдвигает его на 8 позиций вправо (что эквивалентно делению на 2^8, или на 256). Восемь левых разрядов устанавливаются в 0, сохраняя 8-разрядное число, содержащее исходные значения восьми левых разрядов.
Ниже показана такая функция целиком:

```
/* tone(freq, time) -- устанавливает звук заданной частоты и продолжительности */
#define TIMERMODE        182  /* код установки таймера в нужный режим */
#define FREQSCALE       119000L /* базовая частота в герцах */
#define TIMESCALE       1230L /* число отсчетов времени в 0,1 с */
#define T_MODEPORT      67 /* порт управляет режимом работы таймера */
#define FREQPORT         66 /* порт регулирует частоту звука*/
#define BEEPPORT         97 /* порт управляет громкоговорителем */
#define ON               97 /* сигнал включения громкоговорителя */
tone(freq, time)
int freq, time;
{
int hibyt, lobyt, port;
long i, count, divisor;
divisor = FREQSCALE/freq; /* масштабирование частоты в единицах таймера */
lobyt = divisor % 256; /* разбивает целое */
hibyt = divisor / 256; /* на два байта */
count = TIMESCALE * time; /* преобразует время в единицы таймера */
outp(T_MODEPORT, TIMERMODE); /* подготавливает таймер к вводу */
outp(FREQPORT, lobyt); /* устанавливает младший байт регистра таймера */
outp(FREQPORT, hibyt); /* устанавливает старший байт регистра таймера */
port = inp(BEEPPORT); /* запоминает состояние порта */
outp(BEEPPORT, ON) /* включает громкоговоритель */
for(i = 0, i < count; i++)
;          /* отметка задержки */
outp(BEEPPORT, port); /* выключает Громкоговоритель, восстанавливает состояние */
```

Мы определяем **TIMESCALE** в директиве **#define** как целое тип **long**, потому что вычисление **TIMESCALE * time** будет выполнять ся для типа **long**, а не **int**. Иначе результат, если он больше 32767 будет усекаться перед занесением в **count**.

Наша функция **tone( )** в значительной степени дублирует действие оператора **SOUND** языка Бейсик для компьютера IBM PC Здесь мы используем ее для создания довольно ограниченной (8 нот, одна октава) клавиатуры, в которой используются 8 клавишей, начиная с A, для воспроизведения нот. Ниже приведена соответствующая программа, а также некоторые пояснения к ней.

```
/* простая музыкальная клавиатура */
#include <conio.h> /* использует небуфсризованный ввод-вывод */
#include <ctype.h>
#define C 262 /* определяет частоты */
#define D 294
#define E 330
#define F 349
#define G 392
#define A 440
#define B 494
#define C2 524
main( )
{
int key, freq, tempo, time;
puts(" Введите, пожалуйста, основной темп: 10 = 1 с.");
scanf(" %d", &tempo);
printf(" %d \n \r", tempo);  /* эхо-ввод */
puts(" Спасибо. Используйте клавиши a - k для воспроизведения нот.\n\r");
puts(" Клавиша переключения регистра удваивает продолжительность звучания.
                        Символ ! прекращает работу.");
while((key = getchar( )) != '!')
{ time = isupper(key)? 2 * tempo : tempo;
  key = tolower(key);
  switch (key) {
   case 'a' :   tone(C, time);
                                break;
   case 's' :   tone(D, time);
                                break;
   case 'd' :   tone(E, time);
                                break;
   case 'f' :   tone(E, time);
                                break;
   case 'g' :   tone(G, time);
                                break;
   case 'h' :   tone(A, time);
                                break;
   case 'j' :   tone(B, time);
                                break;
   case 'k' :   tone(C2, time);
                                break;
   default :    break; }
 }
puts("До свидания!\n\r");
} }
```

Главной особенностью созданной программы является оператор **switch**, который присваивает разные звуки восьми клавишам от A до K. Кроме того, программа удваивает продолжительность звучания ноты, если вы используете верхний регистр. Эта продолжительность (**time**) устанавливается перед оператором **switch**, затем верхний регистр переключается на нижний, чтобы сократить число необходимых меток.

Вторая важная особенность заключается в том, что мы используем заголовочный файл **conio.h**. Этот файл содержит директивы **#define**, которые заменяют обычные функции ввода-вывода [такие, как **getchar( )**] на версии "пультового ввода-вывода", являющиеся небуферизованными. И в результате, если вы нажимаете, скажем, клавишу [**a**], немедленно звучит нота, и вам не нужно нажимать клавишу [**ввод**]. Между прочим, эти функции не только не выполняют эхо-печать, но и не

начинают автоматически новую строку. Поэтому мы вставили оператор **printf( )** для эхо-печати вводимой переменной **tempo** и использовали символы **\n** и **\r** для перемещения курсора на новую строку и возврата его к левой стороне экрана. Если вы хотите, чтобы символы, которые соответствуют нажимаемым клавишам, отображались одновременно на экране, вставьте

```
putchar(key);
```

в программу.

Хотя ввод не буферизован, клавиатура имеет свой собственный буфер. Это позволяет вам, если вы хотите, заранее набирать все требуемые символы. А ноты будут звучать в собственном устойчивом темпе. Вот, пример, начало мелодии "Радость мира"

```
KjhGfdsA
```

Предоставляем вам возможность закончить эту мелодию.

ПРИЛОЖЕНИЕ И

Далее Содержание

РАСШИРЕНИЕ ЯЗЫКА СИ

Версия 7 ОС UNIX предоставляет два важных расширения языка Си. Первое заключается в том, что можно использовать саму структуру (а не только адрес или элемент структуры) в качестве аргумента функции. Второе расширение позволяет использовать новую форму данных, называемую "перечислимый тип данных". Теперь рассмотрим эти расширения.

Структуры в качестве аргументов функции

Далее Содержание

В нерасширенном языке Си можно передавать функции *адрес* структуры. Например, если *montana* является структурной переменной структурного типа **player**, мы можем обратиться к функции следующим образом:

```
stats(&montana);
```

Функция **stats( )** будет иметь примерно такой заголовок:

```
stat(name) struct player * name;
```

После вызова функции указатель **name** будет ссылаться на структуру montana и функция будет использовать montana в своих манипуляциях.

В расширенном языке Си мы можем применять имя самой структуры в качестве аргумента, и это приведет к созданию *копии* исходной структуры в составе вызванной функции. Например, обращение к той или иной функции может выглядеть примерно так:

```
stats(montana);
```

Теперь функция **stats( )** должна иметь несколько иной заголовок:

```
stats(name) struct player name;
```

На этот раз после вызова функции создается новая структурная переменная типа **player**. Новая переменная получает название **name**, и каждый элемент name имеет такое же значение, как и соответствующий элемент структуры **montana**.

Это расширение позволяет функции иметь свою "личную" копию структуры точно так же, как она обычно имеет свои копии стандартных переменных. Преимущество здесь то же, что и раньше: структуры не изменяются необъяснимо из-за непредвиденного побочного воздействия функции.

Будьте осторожны: Некоторые компиляторы допускают обращение вида

```
stats(montana);
```

но на самом деле интерпретируют его как

```
stats(&montana);
```

В этом случае передается адрес, и функция работает с самой исходной структурной переменной, а не с ее копией.

Перечислимые типы

Далее Содержание

Ключевое слово **enum** позволяет создавать новый тип и определять значения, которые он может иметь. Приведем пример:

```
enum spectrum (red, orange, yellow, green, blue, violet);
enum spectrum color;
```

Первый оператор объявляет новый тип: **spectrum**. Он перечисляет также возможные значения переменных типа **spectrum: red, orange** и т. д. Они являются константами типа **spectrum** так же, как 4 является константой типа **int**, а '**g**' - константой типа **char**.

Второй оператор объявляет **color** переменной типа **spectrum**. Вы можете присвоить переменной **color** любую константу типа **spectrum;** например:

```
color = green;
```

На первый взгляд типы **enum** могут показаться похожими на определенные пользователем порядковые типы языка Паскаль. Действительно, сходство есть, но есть и существенные различия, поэтому, если вы знаете Паскаль, то нс придете к такому заключению.

Рассмотрим характер этих новых констант и операций, которые можно выполнять с использованием переменных типа

enum константы

Как компьютер запоминает что-нибудь подобное **red**? Он может рассматривать это как символьную строку, но у нее нет кавычек. И действительно, **red** и другие **enum** константы запоминаются как целые числа. Например, попробуйте выполнить

```
printf("red = %d, orange = %d\n", red, orange);
```

и с учетом вышеуказанных описании вы получите такой результат:

```
red = 0, orange = 1
```

По существу переменная **red** и ее "сестры" действуют как синонимы целых чисел от 0 до 5. Результат подобен использованию

```
#define red  0
```

за исключением того, что соответствие, установленное при помощи оператора **enum** более ограниченно. Например, если **index** является переменной типа **int**, то оба нижеследующих

оператора недопустимы:

```
index = blue; /* несоответствие типа */
color = 3;    /* несоответствие типа */
```

Позже мы рассмотрим другие ограничения при использовании констант и переменных типа **enum**. Сначала более внимательно рассмотрим значения констант типа **enum**.

Значения по умолчанию

Наш пример проиллюстрировал присваивание константам значений по умолчанию. Константам, появляющимся в описании enum, присваиваются целые числа 0, 1, 2 и т. д. в порядке их расположения. Так, описание

```
enum kids {nippy, slats, skip, nana, liz};
```

присваивает **nana** значение 3.

Присвоенные значения

Можно выбирать значения, которые вы хотите присвоить константам, но они должны быть целого типа (включая **char**). Для этого включите желаемые значения в описание:

```
enum levels {low = 100, medium = 500, high = 2000};
```

Если вы присваиваете какое-либо значение одной константе и не присваиваете ничего константам, следующим за ней, то им будут присвоены последовательные значения, идущие за явно присвоенным значением. Например,

```
enum feline {cat = 20, tiger, lion, puma};
```

присваивает переменной **tiger** значение 21, переменной **lion** - значение 22 и **puma** - значение 23.

Операции

Теперь рассмотрим, что можно и нельзя делать с величинами типа **enum**.

Вы можете присвоить константу типа **enum** переменной того же типа

```
enum feline pet;
pet = tiger;
```

Нельзя использовать другие операции присваивания:

```
pet += cat; /* недопустимо */
```

Можно провести сравнение с целью выявления равенства или неравенства:

```
if ( per = = cat) ...
if ( color ! = violet)
...
```

Нельзя использовать другие операции отношения:

```
if(color > yellow) /* недопустимо */
```

Можно применять арифметические операции к *константам* типа **enum**:

```
color = red + blue;
pet = puma * lion;
```

Имеют ли такие выражения какой-то смысл - это уже другой вопрос.

Нельзя использовать арифметические операции для *переменных* типа **enum**:

```
color = color + green; /* недопустимо */
```

Нельзя использовать операции увеличения и уменьшения:

```
color++ ; /* недопустимо */
```

Нельзя использовать константу типа enum для индекса массива:

```
marbles[red] = 23; /* недопустимо */
```

ПРИМЕНЕНИЕ

Основная причина использования типа **enum** заключается в улучшении читаемости программ. Если вы имеете дело с некоторым видом цветовых кодов, то использование **red** и **blue** что обычно типы enum предназначены для использования внутри программы, а не для ввода-вывода. Например, если вы хотите ввести значение для переменной **color** типа **spectrum**, то должны были бы ввести, скажем, целое число 1, а не слово **orange**. (Конечно, можно было бы создать функцию ввода, которая восприняла бы строку "orange" и затем преобразовала бы ее в целое число **orange**.)

Приложения

ПРИЛОЖЕНИЕ К

ТАБЛИЦА КОДОВ ASCII

Числовые преобразования

Числовые преобразования десятичное - шестнадцатеричное - восьмеричное - двоичное - ASCII

Десятеричное X_{10}	Шестнадцатеричное X_{16}	Восьмеричное X_8	Двоичное PX_x	ASCII	*Ключ**
0	00	00	00000000	NUL	CTRL/I
1	01	01	10000001	SOH	CTRL/A
2	02	02	10000010	STX	CTRL/B
3	03	03	00000011	ETX	CTRL/C
4	04	04	10000100	EOT	CTRL/D
5	05	05	00000101	ENQ	CTRL/E
6	06	06	00000110	ACK	CTRL/F
7	07	07	10000111	BEL	CTRL/G
8	08	10	10001000	BS	CTRL/H,возврат
9	09	11	00001001	HT	CTRL/I,табуляция

10	0A	12	00001010	LF	CTRL/J,новая строка
11	0B	13	10001011	VT	CTRL/K
12	0C	14	00001100	FF	CTRL/L
13	0D	15	10001101	CR	CTRL/M,возврат
14	0E	16	10001110	SO	CTRL/N
15	0F	17	00001111	SI	CTRL/O
16	10	20	10010000	DLE	CTRL/P
17	11	21	00010001	C1	CTRL/Q
18	12	22	00010010	DC2	CTRL/R
19	13	23	10010011	DC3	CTRL/S
20	14	24	00010100	DC4	CTRL/T
21	15	25	10010101	NAK	CTRL/U
22	16	26	10010110	SYN	CTRL/V
23	17	27	00010111	TB	CTRL/W
24	18	30	00011000	CAN	CTRL/X
25	19	31	10011001	EM	CTRL/Y
26	1A	32	10011010	SUB	CTRL/Z
27	1B	33	00011011	ESC	ESC,возврат
28	1C	34	10011100	FS	CTRL<
29	1D	35	00011101	GS	CTRL/
30	1E	36	00011110	RS	CTRL/=
31	1F	37	10011111	US	CTRL/-
32	20	40	10100000	SP	Пробел
33	21	41	00100001	!	!
34	22	42	00100010	"	"
35	23	43	10100011	#	#
36	24	44	00100100	$	$
37	25	45	10100101	½	½
38	26	46	10100110	&	&
39	27	47	00100111	'	'

40	28	50	00101000	(	(
41	29	51	10101001	)	)
42	2A	52	10101010	*	*
43	2B	53	00101011	+	+
44	2C	54	10101100	'	'
45	2D	55	00101101	-	-
46	2E	56	00101110	.	.
47	2F	57	10101111	/	/
48	30	60	00110000	0	0
49	31	61	10110001	1	1
50	32	62	10110010	2	2
51	33	63	00110011	3	3
52	34	64	10110100	4	4
53	35	65	00110101	5	5
54	36	66	00110110	6	6
55	37	67	10110111	7	7
56	38	70	10111000	8	8
57	39	71	00111001	9	9
58	3A	72	00111010	:	:
59	3B	73	10111011	;	;
60	3C	74	00111100	<	<
61	3D	75	10111101	=	=
62	3E	76	10111110	>	>
63	3F	77	00111111	?	?
64	40	100	11000000	@	@
65	41	101	01000001	A	A
66	42	102	01000010	в	в
67	43	103	11000011	c	c
68	44	104	01000100	D	D
69	45	105	11000101	E	E
70	46	106	11000110	F	F

71	47	107	01000111	G	G
72	48	110	01001000	н	H
73	49	111	11001001	I	I
74	4A	112	11001010	J	J
75	4B	113	01001011	к	K
76	4C	114	11001100	L	L
77	4D	115	01001101	M	M
78	4H	116	01001110	N	N
79	4F	117	11001111	0	0
80	50	120	01010000	P	P
81	51	121	11010001	Q	Q
82	52	122	11010010	R	R
83	53	123	01010011	S	S
84	53	124	11010100	T	T
85	55	125	01010101	U	U
86	56	126	01010110	V	V
87	57	127	11010111	W	W
88	58	130	11011000	X	X
89	59	131	01011001	Y	Y
90	5A	132	01011010	Z	Z
91	5B	133	11011011	[	[
92	5C	134	01011100	/	/
93	5D	135	11011101	]	]
94	5E	136	11011110	^	^
95	5F	137	01011111	-	-
96	60	140	01100000	.	.
97	61	141	11100001	a	a
98	62	142	11100010	b	b
99	63	143	01100011	c	c
100	64	144	11100100	d	d

101	65	145	01100101	e	e
102	66	146	01100110	f	f
103	67	147	11100111	g	g
104	68	150	11101000	h	h
105	69	151	01101001	i	I
106	6A	152	01101010	j	j
107	6B	153	11101011	k	k
108	6C	154	01101100	l	1
109	6D	155	11101101	m	m
110	6E	156	11101110	n	n
111	6F	157	01101111	o	o
112	70	160	11110000	p	p
113	71	161	01110001	q	q
114	72	162	01110010	r	r
115	73	163	11110011	s	s
116	74	164	01110100	t	t
117	75	165	11110101	u	u
118	76	166	11110110	v	v
119	77	167	01110111	w	w
120	78	170	01111000	x	x
121	79	171	11111001	y	y
122	7A	172	11111010	z	z
123	7B	173	01111011	R	R
124	7C	174	11111100	/	/
125	7D	175	01111101	T	T
126	7E	176	01111110	~	~
127	7F	177	11111111	DEL	DEL,отмена символа

[Содержание] [Вверх]