В.С. Макогон

Язык программирования C

ABC
FGHIK

для начинающих

ББК 22.183.49.Я7
М 164

УДК 519.682(078)

Язык программирования Си для начинающих: Учебное пособие / В.С. Макогон. —

Одесса: 1993. 96 с.

Учебное пособие представляет собой курс программирования на языке Си для начинающих. Практическая направленность книги делает ее общедоступной.

Автор не ставил перед собой цели создания полного и формального описания языка Си; основное внимание уделено ясности и доступности изложения. Приведено большое число детально разработанных примеров с решениями. Все программы отлажены на персональном компьютере IBM PC, но могут быть использованы практически на любом компьютере, снабженном соответствующим компилятором программ, написанных на языке Си (Turbo C,Microsoft C и др.)

Для широкого круга пользователей, не имеющих специальной подготовки в области программирования, а также студентов, аспирантов и учащихся школ, лицеев, гимназий.

Библиогр. 5 назв.

Рецензенты:

кандидат физико-математических наук *Т.И.Петрушина*

кандидат технических наук *П.Я.Гулинский.*

Редактор:

В.И.Костецкий

Подписано в печать 13.04.93. Формат 60x84 1/16.Офсетная печать. Усл.-печ.л.5,58. Тираж 10000. Заказ № 2717. Отпечатано в издательстве "Черноморье".Оригинал-макет подготовлен НПФ "Астропринт".Цена договорная.

ISBN 5-8404-0020-3

ВВЕДЕНИЕ

Си - сравнительно новый универсальный язык программирования для пользователей ЭВМ нашей страны, хотя создан он еще в 1972 г. Деннисом Ритчи (фирма Bell Laboratories, США). Первоначально язык был реализован в операционной системе UNIX на ЭВМ PDP-11 и тесно связан с этой системой. Как известно, на современном этапе развития вычислительной техники операционная система (ОС) является основой программного обеспечения любой ЭВМ. Именно она обеспечивает пользователя системой программирования (языки программирования, трансляторы, редакторы связей, библиотеки подпрограмм и обслуживающие программы), автоматизирует весь процесс обработки заданий пользователя на ЭВМ, распределяет ресурсы между отдельными заданиями, реализует систему управления вводом-выводом информации, задает режимы функционирования машины и т.д.

В условиях широкого распространения персональных и профессиональных компьютеров особо важное значение приобретают средства общения с ОС, такие, как языки управления заданиями, запросы на выполнение системных функций, директивы ввода-вывода и др. К сожалению, в каждой ОС эти средства абсолютно различны. Даже в тех случаях, когда несколько ОС предназначены для функционирования на ЭВМ одного и того же типа, они имеют существенные отличия в языке взаимодействия пользователя с системой. Таковы, например, ДОС и ОС для ЕС ЭВМ, ОС РВ и РАФОС для СМ ЭВМ. Эти отличия приводят к тому, что даже опытным программистам при переходе с одной системы на другую требуется серьезная переподготовка. Еще большие трудозатраты связаны с переносом прикладных программ.

Создание операционной системы UNIX положило начало важному этапу в развитии программного обеспечения ЭВМ. UNIX - операционная система, слабо зависящая от конкретной машинной архитектуры, обладающая развитыми инструментальными возможностями. В ней реализованы: единый язык взаимодействия пользователей с системой вне зависимости от применяемой ЭВМ; возможность применения в пользовательских программах единого набора системных вызовов; единообразное представление данных в виде унифицированных файловых структур; унифицированные языки программирования и программы - утилиты, максимально способствующие переносимости программ.

Особое значение такая универсальная операционная система приобретает в условиях интенсивного распространения в нашей стране персональных компьютеров различных типов. В ближайшие годы,

особенно в связи с развитием технической базы школьной информатики, резко возрастает число пользователей ЭВМ, не имеющих профессиональной подготовку в области вычислительной техники. Общий недостаток существующих ОС - невозможность переноса программ с одного компьютера на другой- значительно осложняет использование персональных ЭВМ.

Идеология, положенная в основу UNIX, возможность ее функционирования на компьютерах, имеющих различную архитектуру, - все это свидетельствует о том, что в самое ближайшее время операционная система UNIX станет действительно универсальной. В нашей стране реализована аналогичная UNIX инструментальная мобильная операционная система - ИНМОС [2]. Появилась реальная возможность оснастить единой операционной системой различные по архитектуре массовые ЭВМ. Благодаря этому создаются инструментальные средства, позволяющие сэкономить усилия системных программистов по разработке базового программного обеспечения и усилия пользователей, затрачиваемые на адаптацию к новой ОС.

Существенную роль при реализации ОС UNIX сыграл созданный ее разработчиками язык системного программирования Си. Все основные компоненты ОС UNIX, сам транслятор для Си и все обслуживающие программы системы были написаны на этом языке, хотя традиционно для этой цели используют обычно Ассемблеры. Но, как известно, программирование на Ассемблере чрезвычайно трудоемко и, что наиболее существенно, оно ориентировано на конкретную архитектуру ЭВМ, что препятствует переносу программ на другие типы машин без перепрограммирования. Си как язык системного программирования свободен от этих недостатков. В отличие от Ассемблера, он не связан с конкретной архитектурой машин. Технология подготовки и отладки программ на Си характерна для языков высокого уровня, таких, как ФОРТРАН, Паскаль, Ада. В то же время ему присущи многие черты языков низкого уровня (доступ к адресам объектов программы и работа с ними, работа с битовыми и символьными величинами и др.). Поэтому можно сказать, что Си занимает промежуточное положение между языками высокого уровня и машинно-ориентированными языками. По объему занимаемой памяти и времени выполнения программы, написанные на языке Си, приближаются к программам, написанным на языке Ассемблера. Си - это, пожалуй, наиболее удачная попытка поднять системное программирование на качественно новый уровень и дать в руки растущей армии программистов эффективное средство для реализации системных и прикладных программ. В настоящее время Си получает все более широкое распространение. Компиляторы языка функционируют на

разных по архитектуре ЭВМ и не только под управлением ОС UNIX, но и с другими операционными системами, такими, как RSX -11/M,RT - 11 (РАФОС), MS DOS, CP/M и др.

Отметим некоторые особенности языка Си. Он учитывает современные тенденции и технологию программирования. Язык располагает управляющими структурами, необходимыми для написания хорошо структурированных программ. В Си предусмотрены: блочная структура программы, оператор ветвления (if), операторы повторения с предусловием (while,for) и постусловием (do), оператор выбора одного из многих вариантов (switch). Язык использует стандартные типы данных: целый (int), символьный (char), вещественный (float, double). Из других данных, применяемых в Си, отметим массивы, структуры (записи), файлы; есть возможность конструирования более сложных структур: очередей, списков и др.

В языке Си отсутствуют многие средства, встроенные в другие языки: нет встроенной подсистемы ввода-вывода, нет операторов вроде READ и WRITE; нет встроенных средств для работы с символьными строками; отсутствуют операции, работающие с составными объектами типа структур (записей), массивов, списков, рассматриваемыми как нечто целое; нет динамического распределения памяти. При необходимости эти средства обеспечиваются за счет хорошо развитой техники внешних программ.

Программа, написанная на языке Си, состоит из одной или более программных единиц, которые могут быть скомпилированы вместе или отдельно. Все программные единицы в Си одного типа - функции. Аргументы передаются функциям посредством копирования значений аргументов и вызванная функция не может изменить фактический аргумент в вызывающей программе. Если желательно добиться "вызова по ссылке", можно явно передать указатель (в Си имеются указатели и предусмотрена адресная арифметика) и функция сможет изменить объект, на который ссылается этот указатель.

При изложении материала будем предполагать, что читатель знаком с основными понятиям программирования на языках высокого уровня. Однако и неискушенному в программировании читателю, мы надеемся, будет понятно наше изложение, которое основано на конструировании большого числа программ-примеров, иллюстрирующих описываемые возможности языка, а не просто перечисляющих таковые. Все программы подготовлены и проверены в среде операционных систем MSX DOS и MS DOS. Порядок подготовки Си - программ в других операционных системах остается примерно таким же, какой описан ниже.

1. ПОДГОТОВКА ПРОГРАММ НА ЯЗЫКЕ СИ

Единицей трансляции в языке является файл (поименованный набор данных на магнитном диске), содержащий в общем случае несколько программных единиц исходного текста. Подготовка исходного модуля осуществляется каким-нибудь текстовым редактором. Один из таких редакторов в операционной системе MS DOS называется NE.COM. Для его вызова при готовности системы (на экране дисплея имеется подсказка C >) достаточно набрать имя редактора: C > NE

После загрузки в память и ввода имени редактируемого файла редактор переходит в экранный режим, позволяющий приступить к набору текста. Редактор имеет обширные возможности по редактированию (вставка, удаление, замена не только отдельных символов, но и участков текста; контекстная замена, дублирование текста и др.). Чтобы записать подготовленный текст программы на диск, следует нажать клавиши F3 и E. Затем программист запускает процесс компиляции, вводя соответствующую команду:

C > tcc -IC:\tc\include - LC:\tc\LIB mak

Здесь ttc- вызов компилятора языка Си, mak - имя файла, содержащего исходный модуль (заметьте, что расширение имени C не указывается). Исходный текст может содержать некоторые препроцессорные утверждения языка, о чем речь пойдет позже. В этом случае вначале препроцессор языка Си расширяет некоторые сокращенные конструкции исходного текста, включает в программу требуемые стандартные подпрограммы из указанной библиотеки. С его выхода расширенный исходный код поступает на вход компилятора Си. Результатом работы компилятора является программа на языке Ассемблера, которая затем транслируется и получившийся объектный модуль компонуется с другими (библиотечными) модулями, образуя выполнимую программу. Эту последнюю работу выполняет редактор связей (компоновщик). Программист может выполнить эту программу, вводя с терминала имя файла, содержащего выполняемый модуль.

C > mak

В действительности этот файл записан на диск под именем MAK.EXE, в чем можно убедиться, просмотрев по окончании всей работы каталог файлов. Для этого достаточно ввести команду:

C > dir

Вывести текст исходного модуля на экран дисплея можно по команде:

C > type mak.c,

а распечатать на принтере можно, введя команду:

C > copy mak.c prn

2. ОСНОВНЫЕ ПОНЯТИЯ И ОСОБЕННОСТИ ЯЗЫКА

Прежде чем перейти к последовательному изложению основных конструкций языка, познакомимся с важнейшими понятиями и особенностями Си на нескольких примерах. При этом мы сконцентрируем наше внимание на таких основных понятиях, как константы и переменные, арифметические действия над ними, основные управляющие структуры, функции и простейший ввод-вывод.

2.1. Стандартные подпрограммы.

Как уже отмечалось, в языке Си отсутствуют многие встроенные в транслятор средства, присущие развитым языкам программирования, такие, как обработка символьных строк, работа с массивами и структурами (записями) как единым целым; в языке нет встроенных методов доступа к файлам, нет даже операций ввода-вывода в их обычном понимании. Выполнение этих функций обеспечивается стандартной библиотекой подпрограмм. Те, кто работал с языками высокого уровня типа Паскаль или ПЛ/1, будут удивлены необходимостью вызова в Си стандартной функции для сравнения двух символьных строк. В Паскале мы могли бы написать:

IF STR = 'ПРИВЕТ ' THEN ...

для сравнения значения переменной STR с символьной константой "ПРИВЕТ". В языке Си придется использовать этот оператор в таком виде:

if (strcmp(STR, "ПРИВЕТ")) ...

Другими словами, сравнение двух строк осуществляется с помощью стандартной функции strcmp. Стандартные функции ввода-вывода хранятся в общедоступном "библиотечном файле". Хотя эти функции не являются частью формального определения языка, независимо от используемой операционной системы. Чтобы связать программу пользователя со стандартной библиотекой ввода-вывода, следует в начале программы предусмотреть препроцессорное утверждение #include <stdio.h>.

К наиболее интересным и важным функциям языка относится printf. Она предназначена для форматированного вывода данных. Например, чтобы вывести некоторое сообщение на экран дисплея, достаточно использовать вызов функции:

printf ("Интересное сообщение\n");

Одним из механизмов взаимодействия между вызывающей и вызываемой программами являются параметры. Список параметров (аргументов) идет вслед за именем функции в круглых скобках. В данном случае аргументом служит строковая константа - любая последовательность символов, заключенная в двойные кавычки. Комбинаци "\n" в строках языка Си означает переход на новую строку. При попытке вывести ее на терминал курсор (светящийся на экране символ, указывающий позицию очередного вводимого или выводимого символа) перейдет в самую левую позицию следующей строки. Сама функция printf автоматически на новую строку курсор не переводит. Предыдущее сообщение можно вывести на терминал и таким образом:

printf("Интересное"); printf ("сообщение"); printf ("\n");

Результат был бы точно таким же, как и в первом случае. Обычно printf служит для вывода значений переменных. Первым аргументом служит строка форматов, а последующими, если они есть, - выводимые объекты. Строка форматов может включать обычные символы, просто копируемые при выводе и спецификации преобразования, которые начинаются со знака %, за ним следует символ преобразования. Например, если a, b, c - целые величины, то их значения могут быть выданы в десятичном виде следующей командой:

printf ("Значения a, b, c равны: %d %d %d\n", a, b, c);

Каждая спецификация преобразования соответствует одному из аргументов, которые следуют за форматной строкой; между ними устанавливается взаимно однозначное соответствие. Буква d в спецификации преобразования указывает, что значение аргумента должно быть напечатано как десятичное целое число. Из других символов преобразования отметим: c- для вывода отдельного символа; s-для печати символьной строки; x и o-для вывода шестнадцатеричных и восьмеричных чисел соответственно; f-для вывода чисел с плавающей точкой. В следующем примере

printf ("%c = %d\n",g, g);

значение переменной g выводится как символ алфавита, а после знака равенства-как числовое значение, соответствующее внутреннему (машинному) коду этого символа. Вот почему в качестве аргументов дважды использовано одно и то же имя. Перед символом преобразования может стоять числовой коэффициент, явно указывающий количество позиций в выводимой строке, отведенных для элемента вывода:

printf("%5d %c\n", i*i,s);

2.2. Пример простой программы на языке Си

Следующий пример простой, но вполне законченной программы поможет понять многие из рассмотренных ранее принципов построения программ на языке Си. Наша первая программа вводит два числа, вычисляет их сумму и печатает результат с поясняющим текстом "Сумма" *(рис. 2.1.)*.

```
#include <stdio.h>
main()
{
  int a,b,c;
  a=5; b=7;
  c=a+b;
  printf("СУММА=%d\n",c);
}
```

Рис. 2.1

Дадим некоторые пояснения. В языке Си любая программа, как уже упоминалось, состоит из одной или нескольких программных единиц и каждая из них - функция. Функции в Си подобны функциям или подпрограммам в Фортране или процедурам в Паскале, ПЛ/1 и других языках. Имена для функций выбираются произвольно, но одна из них (быть может единственная) непременно называется main. Это особое имя: именно с функции main начинается выполнение программы. Такая главная функция обычно обращается к другим функциям, которые находятся с одном файле с головной программой или извлекаются из библиотеки предварительно подготовленных функций. Функция main аргументов не имеет, поэтому ее список параметров выглядит так: ( ). Скобки { } обрамляют операторы, которые реализуют собственно алгоритм. Эти скобки аналогичны DO - END в ПЛ/1, begin-end в Паскале, Алголе и употребляются для группирования описаний данных и операторов в один **блок.**

Строка int a,b,c; объявляет a, b, c переменными целого типа. Все используемые в программе переменные должны быть объявлены. Далее, как легко догадаться, в операторах присваивания переменные a и b принимают соответствующие значения 5 и 7, а переменная c - значение их суммы. Каждый оператор заканчивается точкой с запятой. Точность представления переменных типа int (целый) зависит от конкретной ЭВМ

и для современных мини- и микрокомпьютеров обычно лежит в диапазоне от -32768 до 32767. Результаты работы программы функция printf выведет на экран в таком виде: СУММА=12.

Трудно быть всегда последовательным при описании нового языка, не введя его полных спецификаций. Стремление как можно скорее дать возможность составления вполне законченных программ на языке Си заставляет нас привести здесь несколько упрощенное описание функции scanf, которая обеспечивает форматированный ввод данных. Ни функция scanf, ни ранее рассмотренная функция printf не относятся к числу часто используемых в системном программировании средств. Но их наличие, наряду с другими возможностями, делает язык Си универсальным. Как и printf, функция scanf имеет переменное число параметров. Но при этом между ними есть существенное различие. Мы уже говорили, что аргументы передаются функции **по значению**, т.е. вызванная функция получает копии значений фактических параметров, а не их адреса. Это, кстати, отличается от Фортрана и ПЛ/1, где передача параметров идет по **ссылке**, т.е. вызванная функция обрабатывает **адреса** аргументов. Это приводит к тому, что в Си вызванная функция не может изменить фактические параметры в обратившейся к ней программе. Исключение составляют аргументы-массивы, которые фактически передаются по ссылке, т.е. функция получает адрес начала массива. Индексируя это значение, функция может выбирать и изменять любые элементы массива. Другое дело - простые переменные. Чтобы приспособить функцию для изменения переменной в вызываемой программе, обратившийся должен давать **адрес** переменной, которую надо менять. Но именно это нам и нужно, когда мы обращаемся к функции scanf: ведь введенные извне величины должны **изменять** значения переменных в вызывающей программе! Вот какое длинное отступление пришлось сделать, чтобы сообщить, наконец, что в качестве фактических параметров функция scanf использует адреса переменных, а не их значения. Для этого перед соответствующим параметром ставят знак & - символ взятия адреса переменной. Например, &XI означает "адрес переменной XI", а не значение, которое эта переменная имеет в данный момент.

Строка форматов функции scanf указывает, какие данные ожидаются на входе. Если функция встречает в форматной строке знак % , за которым следует символ преобразования, то она будет пропускать на входе символы до тех пор, пока не встретит какой-нибудь непустой символ.

Предыдущий пример страдает по меньшей мере одним недостатком: программа вычисления суммы двух чисел годится только для одного

конкретного случая, когда a=5, b=7. Улучшим эту программу, заменив соответствующие операторы присваивания вызовом функции scanf *(рис. 2.2.)*.

```
/* Ввод двух чисел, вычисление их суммы и
   печать результата                    */
#include <stdio.h>
main()
{
  int a, b, c;
  scanf("%d %d",&a,&b);
  c=a+b;
  printf("СУММА=%d\n", c);
}
```

```
c>mak1
5 7
СУММА=12
```

Рис. 2.2

Форматная строка предписывает функции scanf ввести десятичное целое число, которое надо поместить в переменную a, затем продвинуться к следующему непустому символу, на что указывает пробел после спецификации %d и с этой точки начать ввод нового десятичного целого числа, которое затем присвоить переменной b. Обратите внимание, что программа начинается со строки комментариев. Любые символы между /* и */ транслятор пропускает и их можно использовать для пояснений, облегчающих понимание программы. Комментарии могут появляться везде, где допускаются пробелы или переходы на новую строку. Сразу после закрывающей скобки } следует сокращенный протокол выполнения программы, которой при подготовке ее текста было присвоено имя mak1.c (C > - символ-подсказка о готовности системы).

2.3. Основные типы данных, операции и выражения

Основные объекты, с которыми работает программа на языке Си- переменные и константы. **Переменные** - это поименованные величины, значения которых, в отличие от констант, могут меняться в процессе выполнения программы. Все переменные должны быть описаны; в

описаниях указываются их типы и, возможно, начальные значения. Константам могут быть присвоены имена - синонимы констант в программе.

Имена переменных. Имена могут состоять из букв латинского алфавита, цифр и символа подчеркивания "_", который считается буквой. Первый символ имени должен быть буквой. Строчные и прописные буквы различаются, например Beta, beta и BETA - различные имена. Число символов в имени не ограничено, но различаются они по первым восьми символам. В качестве имен переменных нельзя использовать зарезервированные слова вроде if, else, for, char, int и т.д. Эти слова мы будем вводить по мере изложения материала. Все служебные слова должны быть набраны на нижнем регистре (малыми буквами).

Типы данных. В языке Си имеются два существенно различных типа данных: int-целый и float-вещественный (с плавающей точкой). Из них можно создавать еще два типа: char-**символьный** и double-**вещественный двойной точности.** Из этих четырех базовых типов может быть получено много других типов. При необходимости программист может изобрести любой желаемый тип данных. Мы будем использовать преимущественно величины целого и символьного типа. Данные с плавающей точкой не играют большой роли в тех областях применения, для которых предназначен язык Си.

Целые константы и константы **с плавающей точкой** записываются в общепринятой для языков программирования форме:

13, -941, 0, 76; 13.0, 13E+0, -1.76, 0.123e-2, 6.02E23

Плавающая константа состоит из десятичной целой части, десятичной точки, десятичной дробной части и степени, которая состоит из буквы E (или e), за которой следует десятичный порядок. Перед показателем может стоять знак (+ или -). Либо десятичная точка, либо показатель, но не оба одновременно, могут быть опущены. Могут быть опущены либо целая, либо дробная части.

В языке Си существуют правила записи восьмеричных и шестнадцатеричных чисел: если перед целым числом идет 0 (нуль), то это восьмеричная константа: 037,0776; начальные 0X или 0x указывают на шестнадцатеричное число: 0xf37, 0X1FA.

Символьная константа состоит из заключенного в одиночные кавычки символа, например'*' . Символы в языке Си фактически являются целочисленными значениями. Их числовое значение соответствует внутреннему (машинному) представлению символов в некотором

принятом стандартном коде. Например, в коде ASCII, широко применяемом в мини- и микрокомпьютерах, значение символа 'A' равно 65, 'b' - 98, '2' - 50. Использование целых для представления символов делает их взаимозаменяемыми во многих случаях.

Символы, не имеющие графического изображения, и некоторые специальные символы записывают следующим образом: \n - новая строка, \t - табуляция, \0 - нуль (пусто), \\ - обратная косая черта, \' - апостроф, \b - возврат на шаг, \r -возврат каретки, \f - перевод страницы. Такая комбинация выглядит как два символа, но фактически это один символ. Присвоение константам символических имен происходит с помощью препроцессорного утверждения #define. Например, запись #define MAX 100 перед текстом основной программы определяет имя MAX, являющееся синонимом константы 100. Попросту говоря, транслятор потом будет заменять вхождение указанного имени в программе на соответствующую константу.

Строковая константа - это последовательность нуля или более символов, заключенная в двойные кавычки, например "ИНТЕРЕСНОЕ СООБЩЕНИЕ" или " " (пустая строка). В конце каждой такой строки (в том числе и пустой) транслятор автоматически помещает нулевой символ \0, что сильно облегчает программисту поиск конца строки. Надо только не забывать, что число резервируемых для хранения строки ячеек (байтов) памяти должно быть на единицу больше числа символов в строке. Технически строка символов представляет собой одномерный массив, каждый элемент которого - один символ.

Описание переменных. Все переменные должны быть описаны до их использования. Описание задает тип, за которым следует список одной или более переменных этого типа. Например:

int a, b; int low; char c, строка; float x, dl;

Переменным в описаниях можно задавать начальные значения, объединяя таким образом описание и оператор присваивания:

int p = 1; float eps = 1.0e-5;

Арифметические операции. Существуют арифметические операции: +, -, *, / и вычисление остатка от деления целых чисел - %. Есть унарная операция - (унарный минус). При делении целых чисел дробная часть отбрасывается. Порядок выполнения операций совпадает с общепринятым: операции + и - имеют одинаковый приоритет, причем он ниже приоритета операций *, / и % . Самый высокий приоритет имеет унарный минус.

3. ОСНОВНЫЕ УПРАВЛЯЮЩИЕ СТРУКТУРЫ

Программисты, имеющие опыт работы с каким-нибудь языком программирования, приступая к изучению нового языка, ищут в нем знакомые конструкции и возможности. В первую очередь это относится к организации ввода-вывода и реализации **основных управляющих структур**. В работе [5] было показано, что любой корректный алгоритм может быть записан на языке программирования, использующем только три управляющие структуры: последовательное выполнение *(следование), ветвление и повторение.* В настоящее время это утверждение лежит в основе *структурного программирования*, которое предполагает ряд организационных мер и дисциплину программирования, способствующих созданию простых, легких для понимания и удобных для модификации программ.

Последовательное выполнение столь обычно, что мы редко о нем вспоминаем, как об управляющей структуре. Последовательность операторов выполняется в порядке их естественного расположения в программе, с возможным отклонением для вызова внешнего фрагмента (функции), но с обязательным возвратом в точку вызова. (Иногда вызов функции рассматривают как самостоятельную управляющую структуру).

Ветвление в простейшем случае описывается в языке Си с помощью условного оператора, имеющего вид:

```
if (выражение)
 оператор _1;
else
 оператор _2;
```

где часть **else** может и отсутствовать. Сначала вычисляется "выражение" в скобках; если оно истинно (отлично от нуля), то выполняется оператор_1. Если "выражение" ложно (равно нулю) и **else**-ветвь присутствует, то выполняется оператор _2, а оператор _1 пропускается. Если на месте условно выполняемых операторов должна располагаться группа из нескольких операторов языка, то они заключаются в фигурные скобки. Часто "выражение" в скобках представляет условие, заданное с помощью операций *отношений и логических операций.* Операции отношения обозначаются в языке Си следующим образом:

== равно; != не равно; < меньше; > больше;
<= меньше или равно; >= больше или равно.

Символ ! в языке Си обозначает логическое отрицание. Будем использовать еще две логические операции: || означает ИЛИ, а && - логическое И. Операции отношения имеют приоритет ниже арифметических операций, так что выражение вида k > n%i вычисляется как k > (n%i). Приоритет && выше, чем у ||, но обе логические операции выполняются после операций отношения и арифметических операций. В сомнительных случаях лучше расставлять скобки.

Для иллюстрации применения условного оператора рассмотрим программу определения большего из трех чисел *(Рис. 3.1).* Первый **if**-оператор представляет полную условную конструкцию, во втором часть **else** отсутствует. Обратите внимание, что точка с запятой, завершая оператор присваивания **max = x**, не нарушает единство **if**-оператора. Если **else**-ветвь пропускается во вложенных условных операторах, возможна неоднозначность их толкования. Во избежание двусмысленностей решают так: **else** соответствует ближайшему **if**, не имеющему своего **else**. В сомнительных случаях лучше использовать скобки.

```
#include <stdio.h>
main()
{
  int x,y,z,max;
  printf("Введите три числа:\n");
  scanf("%d %d %d",&x,&y,&z);
  if(x>y)
    max=x;
  else
    max=y;
  if(z>max)
    max=z;
  printf("Максимальное из(%d,%d,%d)=%d\n",x,y,z,max);
}
```

```
c>mak2
Введи три числа:
Максимальное из (33,-88,66)=66
```

Рис. 3.1

Рассмотрим пример программы, в которой применяются несколько вложенных друг в друга условных операторов *(Рис. 3.2)*. В этой программе строка float A, B, X объявляет три переменные A, B, X как величины вещественного типа. Форматная строка функции scanf предписывает ввести два вещественные числа, которые станут значениями переменных A и B соответственно.

```
/* РЕШЕНИЕ УРАВНЕНИЯ   AX = B   */
 #include <stdio.h>
 main()
 {
   float A,B,X;
   printf("ВВЕДИ A,B\n");
   scanf("%f %f",&A,&B);
   if(A !=0)
    printf("РЕШЕНИЕ:%f\n",B/A);
  else
    if(B == 0)
      printf("X-ЛЮБОЕ ЧИСЛО\n");
    else
      printf("РЕШЕНИЙ НЕТ\n");
 }
```

```
C>MAK1
ВВЕДИ A,B
2 5
РЕШЕНИЕ:2.500000
```

Рис. 3.2

Посмотрите, как выглядит ветвление, когда глубина вложенности условных операторов равна трем *(рис. 3.3)*. Если хоть одно из условий истинно, то все оставшиеся, разумеется, пропускаются. При глубине вложенности условных операторов свыше трех ветвление теряет наглядность и понятность.

Для реализации многозначного ветвления обычно прибегают к управляющей структуре **выбор** (переключатель) *(п. 9.4)*. Когда управляющая структура ветвления становится особенно запутанной, определенную ясность могут внести правильно расставленные фигурные

скобки. Они обязательны, когда в условном операторе некоторые альтернативные действия выражаются более, чем одним оператором; заключение такой группы операторов в фигурные скобки образует **составной оператор,** который можно применять там, где по определению должен быть только один оператор языка.

```
/*  Программа определяет поведение ракеты,
    стартующей на экваторе,в зависимости от
    ее начальной скорости V
*/
#include <stdio.h>
main()
{
 float V;
 printf("ВВЕДИ V\n");
 scanf("%f",&V);
 if (V < 7.8)
   printf("РАКЕТА УПАДЕТ НА ЗЕМЛЮ\n");
 else
  if (V < 11.2)
    printf("РАКЕТА СТАНЕТ СПУТНИКОМ ЗЕМЛИ\n");
  else
    if (V < 16.4)
      printf("РАКЕТА СТАНЕТ СПУТНИКОМ СОЛНЦА\n");
    else
      printf("РАКЕТА ПОКИНЕТ СОЛНЕЧНУЮ СИСТЕМУ\n");
}
```

```
C>MAK3
ВВЕДИ V
8.5
РАКЕТА СТАНЕТ СПУТНИКОМ ЗЕМЛИ
```

Рис. 3.3

Повторение. В языке Си основной структурой, управляющей повторением, служит цикл с предусловием w h i l e (пока). Он имеет следующий формат:

while (условие) оператор;

Условие обязательно заключено в скобки. Оно понимается в широком смысле и в общем случае может быть произвольным выражением (*см. оператор if*). Оператор while повторяет выполнение оператора, следующего после условия, до тех пор, пока это условие истинно. Если это условие не истинно с самого начала или становится не истинным, заданный оператор не выполняется и управление передается первому оператору, следующему за оператором цикла. Если повторяемая часть оператора (тело цикла) содержит не один, а несколько операторов, то вся повторяемая группа должны быть заключена в фигурные скобки:

```
while (условие)
{ оператор 1;
  оператор 2;
    ..........
  оператор
}
```

Для описания условия в операторе while используются те же операции отношений и логические операции, что и для if - оператора. Приведенная на *рис 3.4* программа подсчитывает сумму цифр введенного числа N. Цикл while последовательно выделяет и суммирует каждую цифру исходного числа, начиная с последней; для выделения применяется операция взятия остатка (%) от деления целых чисел. При делении целых чисел любая дробная часть отбрасывается, поэтому после оператора N=N/10; исходное число уменьшается в 10 раз при каждом "обороте" цикла, пока, наконец, не станет равным нулю, после чего цикл завершается и на экран дисплея выдается значение переменной S, в которой накоплена сумма цифр числа.

```
#include <stdio.h>
main()
{
  int N,S,Z;
  S=0;
  printf("ВВЕДИ N\n");
  scanf("%d",&N);
  while (N != 0)
  {
   Z=N%10;
   N=N/10;
```



```
   S=S+Z;
  }
  printf("СУММА ЦИФР=%d\n",S);
}
```

```
C>MAK1
ВВЕДИ N
32767
СУММА ЦИФР=25
```

Рис. 3.4

Рассмотрим еще один пример. На *рис. 3.5* приведена программа, которая реализует алгоритм разложения произвольного числа на простые множители. Для сокращения времени счета отдельно рассматривается случай множителя, равного двум, чтобы в последующем рассматривать только нечетные множители.

```
/* разложить число на множители    */
#include <stdio.h>
main()
{
   int M,i=3;
   printf("введи M:\n");
   scanf("%d",&M);
   printf("%d =1",M);
   while (M%2 == 0)
   {
      printf("*%d",2);
      M=M/2;
   }
   while (i <= M)
   {
      if (M%i == 0)
      {
         printf("*%d",i);
         M=M/i;
      }
      else
        i=i+2;
   }
}
```

```
C>mak6
введи M:
5784
5784=1*2*2*2*3*241
```

Рис 3.5

Иногда структуры со вложенными друг в друга операторами повторения называют сложными циклами. Следующая программа очень простая, хотя и содержит **вложенные** циклы. Она выводит на экран заполненный символом * прямоугольный треугольник, высота которого равна N *(Рис. 3.6).*

Во внешнем цикле устанавливается очередная строка вывода (параметр i), а во внутреннем (параметр j) в очередную строку выводится ровно i символов "* " . Вызов функции printf("\n") обеспечивает в нужный момент переход на новую строку. Обратите внимание, что для вывода одиночного символа в форматной строке функции printf используется спецификация преобразования %c.

Рассмотрим еще один пример. в котором используется сложный цикл. Приведенная на *рис. 3.7* программа позволяет найти в заданном интервале все совершенные числа. Напомним, что натуральное число называют совершенным, если оно равно сумме всех своих делителей, не считая его самого. Известно, что все совершенные числа - четные и что первое совершенное число из натурального ряда равно 6. Этим объясняется начальное значение параметра внешнего цикла. Так как все натуральные числа имеют своим делителем единицу, полагаем начальное значение суммы делителей числа S = 1. Во внутреннем цикле организуется перебор всех множителей текущего значения N. Из теории чисел известно, что такому испытанию имеет подвергать числа от 2 до N/2, либо даже до $\sqrt{N}$. Это очень несовершенный алгоритм и если вы захотите выполнить его на ЭВМ, имейте в виду, что программа работает слишком долго. Более эффективный алгоритм будет реализован чуть позже *(рис. 4.2).*


```
#include <stdio.h>
main()
{
  int i,j,N;
  printf("ВВЕДИ N\n");
  scanf("%d",&N);
  i=1;
  while (i <= N)
  {
    j=1;
    while (j <= i)
    {
      printf("%c",'*');
      j=j+1;
    }
    i=i+1;
    printf("\n");
  }
}
```

```
C>MAK1
ВВЕДИ N
7
*
**
***
****
*****
******
*******
```

Рис. 3.6

```
#include <stdio.h>
main()
{  int j,N,M,S;
   printf("ВВЕДИ M\n");
   scanf("%d",&M);
   N=4;
   while (N <= M)
   {  S=1;j=2;
      while (j <= N/2)
      {  if (N%j == 0)
            S=S+j;
            j=j+1;
      }
      if (N == S)
         printf("%d-СОВЕРШЕННОЕ\n",N);
      N=N+2;
   }
}
C>mak1
ВВЕДИ M
500
6-СОВЕРШЕННОЕ
28-СОВЕРШЕННОЕ
496-СОВЕРШЕННОЕ
```

Рис. 3.7

4. ОПРЕДЕЛЕНИЕ ФУНКЦИЙ

В библиотеке стандартных подпрограмм языка Си имеется много полезных библиотечных функций. С двумя из них -printf и scanf- мы уже познакомились, на ближайших страницах рассмотрим еще несколько. Но для нас особый интерес представляют функции, которые мы определяем сами. Не следует думать, что выделение функции как самостоятельной программной единицы целесообразно только тогда, когда к ней приходится обращаться многократно. Часто встречаются функции всего в несколько строк, вызываемые единожды, но оформленные как функции только для более **ясного** написания некоторого фрагмента программы.

Определение функции состоит из двух частей: **заголовка и тела.** Заголовок определяет имя функции, ее тип и формальные параметры, тело определяет действия над данными, выполняемые функцией. Возвращаемое функцией значение передается в вызывающую программу оператором return (выражение). Значение "выражения"и есть результат функции. Будем пока предполагать, что и вызывающая и вызываемая функции находятся в одном файле. Вообще говоря, там они могут располагаться в любом порядке. Но если в нашей операционной системе функция физически следует за вызывающей ее программой (main), то надо в последней объявить функцию **внешней** с помощью описателя extern : extern int fun(); либо проще int fun(). В противном случае при компиляции будет выдано сообщение об ошибке. Всякая функция имеет вид:

[тип] имя ([список формальных параметров])
описания формальных параметров;
{
описания;
операторы;
}

Здесь квадратные скобки, как обычно, указывают, что заключенная в них конструкция может отсутствовать. По умолчанию тип функции целый. Описание формальных параметров расположено между списком параметров и левой скобкой. Каждое описание заканчивается точкой с запятой. Формальные параметры функции полностью локализованы в ней и недоступны для любых других функций. Мы уже упоминали, что аргументы передаются функции **по значению**, т.е. вызываемая функция получает копии значений фактических параметров, а не их адреса. Позже мы расскажем и о других способах обмена данными между программами.

А пока рассмотрим такой пример. Так как в Си нет операции возведения в степень, составим полезную функцию power(t, n) для возведения целого числа t в целую положительную степень n. В нашей реализации *(рис. 4.1)* функция power предшествует головной программе, поэтому она там предварительно не объявлена. Можете непосредственно убедиться, что функция power выполняет поставленную задачу. Прежде всего обращаем внимание, что описание формальных параметров функции происходит в заголовке функции (int t, n;). Оператор int p=1; в теле функции объявляет p как переменную целого типа и, более того, присваивает ей начальное значение, равное единице. Здесь мы воспользовались возможностью инициализации переменных в операторах описания типа. Инициализация int p=1; в точности эквивалента последовательности операторов int p; p=1; Обращение к функции задает один из аргументов стандартной функции printf в программе main. Фактические и формальные параметры в точности соответствуют друг другу. Выражение power(a, n) предписывает вызов функции. Когда программа main достигает этой точки, управление от main передается функции power. Одновременно значения фактических аргументов f, n копируются в формальные параметры t и n. Операторы, содержащиеся в теле функции power, фактически оперируют данными, полученными от main. Когда достигается оператор return, осуществляется переход в ту точку программы main, из которой мы пришли в power. Значение, вычисляемое функцией power, передается в головную программу с помощью оператора return(p). В скобках в общем случае может быть любое выражение.

```
/*    Функция   y=t**n        */
int power(t,n)
int t,n;
{
  int p=1;
    while (n != 0)
    {
      if (n%2 !=0)
         p=p*t;
         n=n/2;
         t=t*t;
    }
    return(p);
}
```



```
/* Возведение в степень-головная программа*/
#include <stdio.h>
main()
{
  int n,a;
  printf("Введи a,n\n");
  scanf("%d %d",&a,&n);
  printf("%d в степени  %d = %d\n",a,n,
         power(a,n));
}
```

```
C>mak3
Введи a,n
2 14
2 в степени 14 = 16384
```

Рис. 4.1

Функция не обязательно возвращает какое-либо значение. Допускается "пустой" оператор return; либо он вообще может отсутствовать. В последнем случае нормальное завершение функции обеспечивается при достижении правой закрывающей фигурной скобки.

В заключение приведем реализацию алгоритма нахождения совершенных чисел, более эффективного по сравнению с приведенным ранее *(рис. 3.6)*. Этот алгоритм основан на известном из теории чисел утверждении, что если натуральное число $p=2^{k1} - 1$ - простое, то число $2^k(2^{k+1} -1)$ - совершенное (k=1,2,...).

```
/* Найти все совершенные числа,
   не превосходящие 32767        */
#include <stdio.h>
#include "mak3f.c"
main()
{
  extern proso();
  int K,P,N;
  K=1;
```

```
  while (K <= 7)
  {
    N=power(2,K);
    P=2*N-1;
    if (proso(P))
       printf("%d-СОВЕРШЕННОЕ\n",N*P);
       K=K+1;
  }
}
/*  Подпрограмма определяет, является
      ли число n простым                */
int proso(n)
int n;
{
   int i=3;
   while (i < n/2)
   {
     if (n%i == 0)
        return(0);
        i=i+2;
   }
   return(1);
}
```

```
C>mak5
6-СОВЕРШЕННОЕ
28-СОВЕРШЕННОЕ
496-СОВЕРШЕННОЕ
8128-СОВЕРШЕННОЕ
```

Рис. 4.2

Программа, реализующая алгоритм, состоит из трех функций: main, power и proso *(рис. 4.2)*. Функция power заимствована из предыдущего примера и на *рис. 4.2* не приведена. Она будет включена компилятором из файла mak3f.c благодаря препроцессорному утверждению #include "mak3f.c". Функция proso определяет, является ли число p простым и представляет самостоятельный интерес. Если число p -простое,

то функция возвращает в качестве результата единицу, иначе - ноль. Этот результат затем проверяется в операторе if (proso(p)) в программе main. Обратите внимание, что функция proso проверяет в качестве возможных делителей исходного числа только нечетные числа, так как само испытуемое число - нечетное. Отметим, что это очень эффективный алгоритм: приведенные на *рис. 4.2* результаты были получены программой приблизительно за 3 с на персональном компьютере YAMAXA. Программе на *рис. 3.6*. для получения тех же результатов требуется ... более 20 минут.

```
#include <stdio.h>
main()
{
  printf("проверка рекурсии\n");
  main();
}
```

```
C>mak5
проверка рекурсии
проверка рекурсии
проверка рекурсии
проверка рекурсии
^C
```

Рис. 4.3

Рекурсия. В языке Си функцию можно использовать рекурсивно, т.е. функция может вызывать сама себя. Простейший пример приведен *на рис. 4.3*: функция main обращается сама к себе, многократно выводя на экран одну и ту же фразу (пока не будут нажаты клавиши CTRL/C). При рекурсивном обращении функции к самой себе в каждом вызове создается новый экземпляр всех автоматических переменных, и он совершенно не зависит от предыдущего. Следующий пример *(рис.4.4)* использует рекурсию для вычисления k!. При первом вызове fact(i), если i не равно1, функция порождает выражение i*fact(i-1); в свою очередь, вызов fact (i-1) производит (i-1)*fact(i-2), что вместе с ранее полученным результатом дает i*(i-1)*fact(i-2). Рекурсия закончится, как только очередной вызов функции получит аргумент, равный единице. Нетрудно сообразить, что в конечном счете мы получим требуемое произведение.

```
/* Проверка рекурсии */
#include <stdio.h>
int fact(k)
int k;
{
 if (k == 1)
    return(1);
 else
    return(k*fact(k-1));
}
main()
{
   int i=1;
   printf("проверка рекурсии\n");
   while (i < 8)
   {
     printf("%d!= %d\n",i,fact(i));
     i=i+1;
   }
}
```

```
C>mak2
проверка рекурсии
1!=1
2!=2
3!=6
4!=24
5!=120
6!=720
7!=5040
```

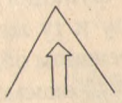


Рис. 4.4

5. КЛАССЫ ПАМЯТИ

Все переменные в программе характеризуются не только типом, но и классом памяти. Он определяет, главным образом, область действия имени, т.е. ту часть программы, в которой это имя имеет смысл. В языке Си существуют четыре класса памяти: автоматический (automatic), регистровый (register), статический (static) и внешний (external).

Автоматические переменные в программе можно описать так: auto int A; auto char c1; auto int x = 125. Если мы до сих пор этим не пользовались, то только потому, что опущенный описатель auto подразумевается по умолчанию. Область действия автоматической переменной ограничена той функцией или блоком, в которых она объявлена. Говорят, что такая переменная **локализована** в блоке. Она начинает существовать только после обращения к функции и исчезает после выхода из нее. Таким образом, автоматические переменные не занимают место в памяти, если они не нужны. И еще одно полезное свойство этих переменных: поскольку автоматическая переменная локальна в некоторой функции, ее значение не может быть случайно изменено действиями других функций, даже если автоматические переменные имеют в разных функциях (блоках) одинаковые имена. Проанализируем результаты работы следующей программы *(рис. 5.1)*.

```
#include <stdio.h>
main()
{
   int t;
   {
      int t=2;
      {
         int t=3;
         printf("%d\n",t);
      }
      printf("%d\n",t);
   }
   printf("%d\n",t);
}
```

```
C>mak0
3
2
746
```

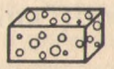


Рис. 5.1

Переменная t описана во внешнем блоке и ей не присваивается никакое значение; переменной t из среднего блока присвоено значение 2; переменная t, описанная в самом внутреннем блоке, получает значение 3. Когда выполняется внешний блок, значение t в нем не определено, но когда активизируется средний блок, переменная t из внешнего блока по-прежнему не определена, хотя в среднем блоке переменной с таким именем присваивается значение 2. Но это совсем другая переменная! Когда же управление переходит во внутренний блок, две первые величины с именем t становятся недоступными и некая третья переменная t принимает значение 3, и функция printf выдает это значение на экран. Как только управление возвращается в средний, а затем во внешний блоки, последовательно становятся доступными переменные, получившие в них значение 2 и 746. Они и выводятся на экран дисплея. Последнее значение - некоторое случайное число ("мусор"), как следствие того, что переменная t не была инициализирована во внешнем блоке.

```
int x=3;  /* описание внешней переменной  */
/*  увеличение  x        */
 int plus1()
 {
   x=x+1;
   printf("прибавляем единицу:x=%d\n",x);
 }
/*  уменьшение  x        */
 int minus1()
 {
   x=x-1;
   printf("вычитаем единицу:x=%d\n",x);
 }
main()
{
    printf("начальное значение x=%d\n",x);
    plus1();
    minus1();
    minus1();
    printf("конечное значение x=%d\n",x);
}
```



```
C>mak0
начальное значение x=3
прибавляем единицу:x=4
вычитаем единицу:  x=3
вычитаем единицу:  x=2
конечное значение  x=2
```

Рис. 5.2

Внешние переменные вводятся как нечто противоположное автоматическим. Это **глобальные** переменные и к ним можно обращаться по именам из любой функции. Такой механизм подобен общим блокам Фортрана или EXTERNAL -переменным языка ПЛ/1. Поскольку внешние переменные доступны везде, их можно использовать для связи между функциями, не прибегая к механизму формальных параметров.

```
#include <stdio.h>
int x=145;  /*описание внешнейпеременной x*/
main()
{
 extern int x,y;
 printf("x=%d  y=%d\n",x,y);
}
int y=541; /* описание внешней переменной y*/
C>mak0
x=145  y=541
```

Рис. 5.3

Внешние переменные могут определяться вне какой-либо функции; при этом для них выделяется фактическая память. В любой другой функции, обращающейся к этим переменным, они должны **описываться**; это делается явно с помощью описателя extern (внешний), либо неявно. Можно не описывать явно внешнюю переменную, если она объявлена вне этой функции и **выше** в том же файле. Обычно поступают так, как показано на *рис. 5.2*: все внешние переменные размещают в начале исходного модуля (вне всяких функций!), опуская дополнительные описания со словом extern внутри функций. Конечно, если внешняя переменная и функция, которая ее использует, размещены в разных файлах, описывать эту переменную в функции необходимо. Но самым надежным способом является описание каждой внешней переменной с помощью ключевого слова extern в любой функции, которая ее использует *(рис. 5.3)*. А еще лучше - вообще избегать применения внешних переменных, так как они часто служат источником труднообнаруживаемых ошибок.

Статические переменные, подобно автоматическим, локальны в той функции или блоке, где они описаны. Разница заключается в том, что статические переменные не исчезают, когда функция (блок) завершает работу, и их значения сохраняются для последующих вызовов функции. Описание статических переменных выглядит следующим образом: static char c1; static int a=1. Рассмотрим пример, в котором переменная x объявлена как статическая *(рис. 5.4)*.

```
/* статические переменные */
#include <stdio.h>
plus1()
{
  static int x=0;
  x=x+1;
  printf("x=%d\n",x);
}
main()
{
  plus1();
  plus1();
  plus1();
}
C>mak3
x=1
x=2
x=3
```

Рис. 5.4

Начальное значение, равное нулю, присваивается ей только один раз. При последующих вызовах функции инициализация не происходит. При первом вызове функции plus1 переменная x увеличивается на единицу, а при каждом последующем ее вызове к предыдущему значению x (оно сохраняется!) прибавляется еще единица. К уже сказанному о статических переменных добавим только, что для объединений данных, - таких, как массивы, структуры, - их начальная инициализация возможна, если они объявлены с атрибутами static или extern.

Регистровые переменные объявляются в программе с помощью ключевого слова register и по замыслу авторов языка Си должны храниться в сверхбыстрой памяти ЭВМ - регистрах. Используются они аналогично автоматическим переменным. Целесообразность их применения для увеличения быстродействия программы представляется в большинстве случаев сомнительной.

6. ОБРАБОТКА СИМВОЛЬНЫХ ДАННЫХ

Язык Си лучше всего подходит для системной работы: написания компиляторов, интерпретаторов, операционных систем, редакторов текстов и т.п. Эффективность этой работы во многом определяется наличием в языке развитых средств для обработки символьной информации. В стандартной библиотеке Си предусмотрены многие полезные функции, выполняющие простые действия с символьными данными. Рассмотрим две из них - getchar и putchar - выполняющие ввод и вывод символа соответственно и создадим на их основе ряд своих, тоже полезных функций. Прежде всего условимся, что в качестве **системного ввода**, с которым имеет дело функция getchar, мы будем понимать клавиатуру терминала пользователя, а в качестве **системного вывода** функции putchar - экран дисплея.

Функция getchar за одно обращение к ней выдает в качестве результата один символ, поступивший из системного ввода. Мы можем рассматривать getchar как функцию, имеющую заголовок:

int getchar()

Как видите, у нее совсем нет аргументов и она возвращает значение типа int -это значение символа во внутреннем представлении его для данной ЭВМ (например, в коде ASCII). Таким образом, после обращения

c = getchar()

переменная c содержит очередной символ, набранный нами на клавиатуре.

Функция putchar за одно обращение к ней выдает один символ в стандартный выходной поток. Вызов этой функции имеет вид:

putchar(c);

c - переменная символьного типа, которой предварительно было присвоено некоторое значение. Рассмотрим примеры. Программа на *рис. 6.1* выводит на экран все прописные латинские буквы. Мы уже упоминали, что типы char и int во многих случаях взаимозаменяемые. Последовательное прибавление единицы к очередному значению c *(рис. 6.1)* обеспечивает выбор очередной буквы ввиду их лексической упорядоченности (в коде ASCII: "A" - 65, "B" - 66, "C" - 67 и т.д.).Прежде, чем перейти к рассмотрению очередного примера, обсудим, как машина должна определять конец входного потока символов, вводимых с клавиатуры терминала.

```
#include <stdio.h>
main()
{
  char c;
  c='A';
  while (c <= "Z")
  {
    putchar(c);
    c=c+1;
  }
}
c>mak2
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Рис. 6.1

Можно, конечно, выбрать некоторый символ как признак конца потока (файла), но при этом должна быть полная гарантия, что этот символ-признак не появится среди обрабатываемых символов. Но обычно getchar распознает некоторый неграфический символ как конец файла. В используемой нами операционной системе для получения такого символа надо нажать одновременно служебную клавишу CTRL и клавишу Z. В программе значение этого символа мы будем использовать через символическое имя EOF (End Of File). Теперь мы можем написать программу копирования файла *(рис. 6.2)*. Пока не обнаружен конец входного потока (конец файла), ЭВМ получает с клавиатуры символ (это делает функция getchar) и сразу же выводит его на экран дисплея с помощью функции putchar. Для завершения программы достаточно нажать последовательно клавиши CTRL и Z. Программу копирования можно написать и более компактно. В языке Си любое присваивание, например, c = getchar(), можно использовать в любом выражении в качестве операнда; его значение - это просто значение, присваиваемое левой части. Учитывая сказанное, перепишем программу "Эхо" *(рис.6.3)*. Эта компактная и элегантная программа принимает символ с клавиатуры, присваивает его переменной c, а затем сравнивает его с признаком конца файла. Пока этот признак не обнаружен, выполняется тело цикла while и символ выдается на экран. В противном случае цикл, а вместе с ним и вся программа, завершаются.


```
/*   ЭХО-ПРОГРАММА  */
#include <stdio.h>
main()
{
  char c;
  c=getchar();
  while (c != EOF)
  {
    putchar(c);
    c=getchar();
  }
}
C>mak2
ААББРРААККААДДААББРРАА
```

Рис. 6.2

Отметим, что включение присваиваний в проверки - сильное средство языка, способствующее созданию компактных программ. Учтите, что скобки вокруг присваивания внутри условия необходимы: приоритет операции != выше приоритета операции присваивания.

```
/*   ЭХО-ПРОГРАММА 2 ВАР.   */
#include <stdio.h>
main()
{
  char c;
  while ((c =getchar()) != EOF)
    putchar(c);
}
C>mak2
ААББРРААККААДДААББРРАА
```

Рис. 6.3

Следующая программа *(рис. 6.4)* подсчитывает количество строк, слов и символов во введенном с клавиатуры тексте. Слово - это любая последовательность символов, не содержащая знаков табуляции ("\t"), пробелов и символов "\n". Будем предполагать также, что любая строка, в том числе и последняя, завершается символом перехода на новую строку "\n". Обратите внимание на строку line= word = kc = 0; в которой трем переменным присваивается значение нуля. Это следствие того факта, что присваивание можно использовать в выражениях:

```
line = ( word = (kc = 0 ));
```

Выражения, связанные логическими операциями &&(И) и || (ИЛИ), вычисляются слева направо. Как только истинность или ложность высказывания станет известной, дальнейшие вычисления прекращаются. Таким образом, если в строке

```
if (s == ' ' | s == '\n' | s == '\t')
```

символ s содержит пробел, то остальные две проверки уже не делаются. Отсюда следует важный для практики вывод: первыми надо проверять наиболее часто встречающиеся события.

```
/* подсчет строк,слов,символов    */
#include <stdio.h>
main()
{
  int c,line,word,kc,inword=0;
  line=word=kc=0;
  while ((c=getchar()) != EOF)
  {
    kc=kc+1;
    if (c == '\n')
     line=line+1;
    if (c == ' '||c == '\n'||c == '\t')
     inword=0;
    else
     if(!inword)
       {inword=1; word=word+1; }
  }
  printf("\n");
  printf("строк-%d\n",line);
  printf("слов-%d\n",word);
  printf("символов-%d\n",kc);
}
```

```
C>mak1
ОЙ У ПОЛІ КРИНИЧКА
ТАМ ХОЛОДНА ВОДИЧКА
строк-2
слов-7
символов-39
```

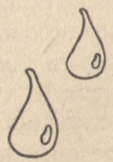


Рис. 6.4

7. ДОПОЛНИТЕЛЬНЫЕ ОПЕРАЦИИ ЯЗЫКА

Мы уже упоминали о компактности языка Си, столь ценимой в системном программировании. Этому способствует и ряд вспомогательных операций, присущих только языку Си, и делающих его непохожим на другие языки. Вот список основных операций языка:

```
+    &    \    ^    |
!:   --   /    ==   >
>=   ++   !=   <<   <
<=   &&   !    ||   %
~    >>   ->   -    =    *
```

С большинством из них мы уже познакомились, некоторые важные операции мы рассмотрим в ближайшее время, а некоторых только коснемся.

О п е р а ц и и увеличения и уменьшения. В языке широко используются две нетрадиционные операции для увеличения или уменьшения значения переменной, обозначаемые соответственно ++ и —. Операция ++ прибавляет 1 к операнду, а -- вычитает 1. Эти операции могут быть использованы или перед, или после своего операнда (префиксная или постфиксная операции соответственно). Они оказывают разные действия в выражениях: в записи ++n увеличение происходит до использования значения n, а в n++ увеличение идет уже после того, как используется значение n. Если считать, что значение переменной n равно 5, то выполнение оператора m = ++ n; присвоит m значение 6, а после выполнения m = n++; m станет равным 5. И в том, и в другом случае n получит значение 6. Первый пример в точности соответствует последовательности операторов n=n+1; m=n; а во втором случае - следующим двум операторам: m=n; n=n+1. Операция уменьшения — действует аналогично. Обе операции имеют самый низкий приоритет среди арифметических операций и выполняются после бинарных операций + и -. Приведенная на *рис. 7.1* программа иллюстрирует применение этих команд.

Поразрядные логические операции предназначены для работы с отдельными битами целого числа или символа:

&	поразрядное И
^	поразрядное исключающее ИЛИ
>>	сдвиг вправо
|	поразрядное ИЛИ
<<	сдвиг влево
~	инверсия

```
/* определить двузначные целые числа,
   которые делятся на сумму своих цифр */
#include <stdio.h>
main()
{
  int a,b,k,s,c;
  k=0;a=1;
  while (a <= 9)
  {
    b=0;
    while (b <= 9)
    {
      s=a+b;
      c=a*10+b;
      if (c%s == 0)
      {
     printf("%d ",c);
         k++;
      }
    b++;
    }
    a++;printf("\n");
  }
  printf("всего:%d\n",k);
}
```

```
C>mak4
 10 12 18
 20 21 24 27
 30 36
 40 42 45 48
 50 54
 60 63
 70 72
 80 81 84
 90
всего:23
```

Рис. 7.1

Поразрядная операция И (&) часто используется для выделения некоторой группы двоичных разрядов, например: n = n & 0177 устанавливает в нуль все двоичные разряды числа n, кроме семи младших. Операция ИЛИ (|) используется для установки отдельных разрядов в единицу. Например, m = m | OxFOF; "включает" 11, 10, 9, 8, 3, 2, 1 и 0 разряды числа m. Операции << и >> выполняют сдвиг операнда влево или вправо на заданное число разрядов. Например, m << 3 сдвигает значение m влево на три разряда, заполняя освобождающиеся младшие разряды нулями. Это, кстати, соответствует умножению числа на 8 (в общем случае на 2 в степени n, где n - число сдвигаемых разрядов). Аналогично выполняется и сдвиг вправо на n разрядов, что равносильно делению числа на 2 в n-й степени. При этом сдвиге высвобождающиеся разряды заполняются значением знакового бита. Унарная операция ~ выполняет инверсию двоичных разрядов числа (символа), т.е. преобразует каждый единичный бит в нулевой и наоборот.

Применение некоторых подразрядных логических операций покажем на примерах. В программе на *рис. 7.2* исходное целое число 511 задано в шестнадцатиричной форме: m = 0X1FF;. Напомним, что записанная перед константой цифра 0 указывает на восьмеричное число, а 0X - на шестнадцатиричное. Для вывода результатов предусмотрена функция PRINT. Она для удобства анализа результатов печатает числа в шестнадцатиричном, восьмеричном и десятичном форматах, используя для этого спецификации x, o, d. Остальные действия в программе комментируются и дополнительных разъяснений не требуют.

```
/* поразрядные логические операции */
 #include <stdio.h>
PRINT(n)
int n;
{
  printf("%5x %5o %5d\n",n,n,n);
}
main()
{
  int m,n;
  m=0X1F3;  PRINT(m); /* 16-ричное число */
  n=m & 0177;PRINT(n); /* выделение 7 мл.бит*/
  n=m | 013; PRINT(n); /* установка 4 мл.бит */
  n=m >> 4;  PRINT(n); /* сдвиг вправо  */
  m=n << 3;  PRINT(m); /* сдвиг влево   */
}
```

```
C>mak1
 1F3   763   499
  73   163   115
 1FB   773   507
  1F    37    31
  F8   370   248
```

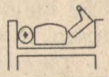


Рис. 7.2

Следующая программа *(рис. 7.3)*, используя команду сдвига числа вправо на один бит (m = m >> 1;) и выделения младшего разряда числа (m & 01), подсчитывает количество единичных битов исходного числа FOF и печатает результат.

```
/* подсчет единиц  */
#include <stdio.h>
main()
{
  int m,k;
  k=0;
  m=0xf0f;
  while (m !=0)
  {
    if (m & 01)
       k++;
    m=m >> 1;
  }
  printf("k=%d\n",k);
}
```

```
C>mak3
k=8
```

Рис. 7.3

Операции присваивания и выражения. Выражение вида i=i + b, где левая часть повторяется в правой части, могут быть записаны в сжатой форме: i+ = b. При этом используется операция присваивания вида +=, которая означает буквально "увеличить i на b". Для большинства бинарных операций допускается запись вида op=, где op - одна из операций: +-* / % | ^ & << >>. Если E1 и E2 - выражения, то E1 op = E2 эквивалентно E1 = (E1) op (E2). Обратите внимание на скобки вокруг E2: присваивание x*=y+1 фактически означает x=x*(y+1), а не x=x*y+1. В качестве иллюстрации, поясняющей сказанное, рассмотрим программу *(рис. 7.4)*,

в которой по заданному натуральному числу n строится число m, записанное теми же цифрами, что и n, но взятыми в обратном порядке. Правду говоря, запись m = m * 10 + z; более прозрачна, чем непривычная последовательность m * = 10; m + = z.

Условная операция. Фактически она представляет сокращенную форму оператора if - then - else, и в общем виде записывается так: выражение1? выражение2: выражение3 Если "выражение1" не равно нулю, то результатом операции будет значение "выражения2", в противном случае - значение "выражения3". Условная операция, называемая иногда **тернарной**, определяет обычное выражение, которое может, в частности, быть использовано в операторе присваивания.

```
main()
{
  int n,z,m=0;
  printf("введи n\n");
  scanf("%d",&n);
  while (n !=0)
  {
    z=n%10;
    n/=10;
    m*=10;
    m+=z;
  }
  printf("m=%d\n",m);
}
```

```
C>mak3
введи n
123
m=321
```

Рис. 7.4

Таким образом, вместо if (x > y) max=x; else max=y; достаточно написать: max = (x >y) ? x : y Скобки вокруг "выражения1" ставить не обязательно, так как приоритет операции ?: очень низкий, ниже он только у присваивания. Условная операция позволяет писать более короткие программы. Вот как выглядит в программе *(рис. 7.5)* цикл для печати квадратов натуральных чисел от 1 до m, по 6 чисел в строке; каждое число

занимает 5 позиций и колонка от колонки отделяется одним пробелом, а каждая строка, включая последнюю, заканчивается символом перехода на новую строку (\n). Переход на новую строку "печатается" после каждого шестого элемента и после m-го. За любыми другими элементами выводится один пробел.

```
/* квадраты натуральных чисел */
#include <stdio.h>
main()
{
  int m,i=1;scanf("%d",&m);
  while (i <= m)
  {
    printf("%5d%c",i*i,(i%6==0||i==m)? '\n':'');
    i++;
  }
}
```

```
A>mak3
15
    1    4    9   16   25   36
   49   64   81  100  121  144
  169  196  225
```

Рис. 7.5

8. МАССИВЫ

Как известно, **массив** - это конечная совокупность данных одного типа. Можно говорить о массивах целых чисел, массивах символов и т.д. Мы можем даже определить массив, элементы которого - массивы (массив массивов), определяя, таким образом, многомерные массивы. Любой массив в программе должен быть описан: после имени массива добавляют квадратные скобки [ ], внутри которых обычно стоит число, показывающее количество элементов массива. Например, запись int x[10]; определяет x как массив из 10 целых чисел. В случае многомерных массивов указывают столько пар скобок, какова размерность массива, а число внутри скобок показывает размер массива по данному измерению. Например, описание двумерного массива выглядит так : int a[2][5];. Такое описание можно трактовать как матрицу из 2 строк и 5 столбцов. Для обращения к некоторому элементу массива указывают его имя и индекс, заключенный в квадратные скобки (для многомерного массива - несколько индексов, заключенных в отдельные квадратные скобки): a[1][3], x[i], a[0][k+2]. Индексы массива в Си всегда начинаются с 0, а не с 1, как в ФОРТРАНе или ПЛ/1, т.е. описание int x[5]; порождает элементы x[0], x[1], x[2], x[3], x[4]. Индекс может быть не только целой константой или целой переменной, но и любым выражением целого типа. Переменная с индексами в программе используется наравне с простой переменной (например, в операторе присваивания, в функциях ввода-вывода). Начальные значения массивам в языке Си могут быть присвоены при компиляции **только** в том случае, если они объявлены с классом памяти extern или static, например:

static int a[6]={5,0,4,-17,49,1};

что обеспечивает присвоения a[0]=5; a[1]=0; a[2]=4; ...a[5]=1. Как видите, для начального присвоения значений некоторому массиву надо в описании поместить справа от знака = список инициирующих значений, заключенных в фигурные скобки и разделенных запятыми. Еще раз напомним, что автоматические массивы инициировать нельзя.

Двумерный массив можно инициировать, например, так:

static int matr[2][5] = {{ 3,4,0,1,2 }, { 6,5,1,4,6 }};

Матрица хранится в памяти построчно, т.е. самый правый индекс в наборе индексов массива меняется наиболее быстро.

Пусть, например, в заданном массиве из 10 целых чисел надо изменить порядок следования его элементов на обратный без привлечения

вспомогательного массива. Соответствующая программа приведена на *рис. 8.1*. Следующая программа *(рис.8.2)* позволяет в целочисленном массиве найти разность максимального и минимального элементов. Обратите внимание, что функция fmax при первом обращении к ней дает максимальный элемент массива, а при повторном вызове - минимальный, так как предварительно мы изменили знаки элементов на противоположные. Это изменение знака учитывается и при вызове функции printf.

```
/* обращение массива */
#include <stdio.h>
main()
{
  int p,i=0;
  static a[10]=
  {10,11,12,13,14,
   15,16,17,18,19};
  while (i < 10/2)
  {
    p=a[i];
    a[i]=a[9-i];
    a[9-i]=p;
    i++;
  }
  i=0;
  while (i<10)
       printf(" %d",a[i++]);
}
```

```
c>mak3
 19 18 17 16 15 14 13 12 11 10
```

Рис. 8.1

В языке Си отсутствует возможность динамически распределять память под массивы: надо при описании массива задать точно его размер. Но если тот же массив описывается еще раз в другой программе, размер можно не указывать; достаточно после имени сохранить пару квадратных скобок, например int x[ ]. Если при вызове функции в качестве аргумента ей передается имя массива, то, в отличие от простых переменных, берется фактически адрес начала этого массива. Поэтому записи fmax(a, 10) и fmax(&a[0], 10) равносильны.


```
/* в массиве найти разность
   мин. и макс. элементов   */
int fmax(x,n)
int x[],n;
{
  int max,i=0;max=x[0];
  while (i < n)
  {
    if (x[i] > max)
       max=x[i];
    i++;
  }
  return(max);
}
#include <stdio.h>
main()
{
  int max,min,i;
  static int a[10]=
  {1,-2,3,-4,5,-6,7,-8,9,-13};
  max=fmax(a,10);
  i=0;
  while (i < 10)
  {
    a[i]=-a[i];
    i++;
  }
  min=fmax(a,10);
  printf("макс-мин=%d\n",max+min);
}
```

```
c>mak2
макс-мин=22
```

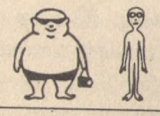


Рис. 8.2

В следующем примере *(рис. 8.3)* массив описан как внешний. Функция main подсчитывает наибольшее число одинаковых идущих подряд элементов массива, определенного вне функции main.

```
/* макс одинаковых подряд */
#include <stdio.h>
int a[]={5,6,6,6,4,3,3,3,3,8};
int n=10;
main()
{
  int i,k,max;
  i=k=max=1;
  while (i < n)
  {
    if (a[i] == a[i-1])
      k++;
    else
    {
      if (k > max)
        max=k;
      k=1;
    }
    i++;
  }
  printf("kmax=%d\n",(k>max)?k:max);
}
```

```
c>mak6
kmax=4
```

Рис. 8.3

Если, как в данном примере, размер массива пропущен, то транслятор определит его длину, считая присваиваемые значения во время начальной инициации. Условная операция (k>max)?k:max в операторе printf предусмотрена для того частного случая, когда весь массив состоит из одинаковых элементов.Приведем несколько примеров, в которых ведется обработка двумерных массивов. Но прежде рассмотрим одну полезную возможность языка Си. Речь идет о препроцессорном утверждении # define, позволяющем присваивать символические имена константам. В общем случае это утверждение записывают так: # define строка1 строка2 (точка с запятой в конце не ставится).

Прежде чем исходный текст программы будет передан компилятору, он обрабатывается препроцессором, который всюду в исходном тексте заменит вхождение "строка1" на "строка2". Например, строка # define max 80, записанная в начале программы, обеспечит всюду в ней замену указанного имени max на соответствующую константу. Замена имени связана не только с числами, но и с текстами. А теперь вернемся к примерам. В следующей программе *(рис. 8.4)* строится единичная матрица a[m][m], размер которой определяется с помощью конструкции #define m 5. Сам алгоритм вычисления элементов матрицы основан на том, что деление целых чисел в Си производится без остатка. Поэтому произведение (i/j)*(j/i) равно единице тогда и только тогда, когда i равно j. В остальных случаях оно равно нулю.

```
#define M 5
#include <stdio.h>
main()
{
  int a[M][M];
  int j,i=1;
  while (i<M)
  {
    j=1;
    while(j<M)
    {
      a[i][j]=(i/j)*(j/i);
      printf(" %d",a[i][j]);
      j++;
    }
    i++;printf("\n");
  }
}
```

```
A>mak1
 1 0 0 0
 0 1 0 0
 0 0 1 0
 0 0 0 1
```

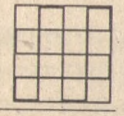


Рис. 8.4

В программе, приведенной на *рис. 8.5* определяется минимальный элемент каждой строки матрицы и выполняется обмен местами найден-

ного и диагонального элемента этой же строки.

Обращаем внимание на следующее обстоятельство. Если двумерный массив надо передавать функции, то описание параметра в ней должно обязательно включать в себя размер строки массива, а размер столбцов несущественен. Так, массив из трех строк и пяти столбцов в функции можно описать как int a[3][5]; либо int a[][5];

```
/* обмен мин с диагональю */
#include <stdio.h>
#define M 4
main()
{
  static a[M][M]={
       { 3,4,1,5},
       {-1,6,7,0},
       { 1,8,6,1},
       {4,9,7,-1}};
  int i,j,jmin,amin;
  i=0;
  while (i<M)
  {
    amin=a[i][0];
    jmin=0;j=1;
    while(j < M)
    {
      if (a[i][j] < amin)
      {
        amin=a[i][j];
        jmin=j;
      }
      j++;
    }
     a[i][jmin]=a[i][i];
    a[i][i]  =amin;
    i++;
  }
  i=0;
```



```
  while (i < M)
  {
    j=0;
    while (j < M)
      printf("%3d ",a[i][j++]);
    printf("\n");
    i++;
  }
}
```

```
c>mak8
  1   4   3   5
  6  -1   7   0
  6   8   1   1
  4   9   7  -1
```

Рис 8.5

8.1. Массивы символов

Любая символьная константа, например "ОЙ, У ПОЛІ КРИНИЧКА", представляет собой массив символов. Во внутреннем представлении транслятор завершает такой массив символом '\0', так что любая программа может по нему легко обнаружить конец строки. Поэтому строка занимает в памяти на один символ больше, чем записано между двойными кавычками. Нумерация элементов массива - отдельных символов строки - начинается, естественно, с нуля. Надо помнить, что, например, 'Т'-это символ (буква), а "Т" - это строка, состоящая из двух символов: 'Т' и '\0'. Отсюда следует, что пустых строк не бывает.

Строка в языке Си - это разновидность константы и ее можно присваивать некоторой переменной, представляющей массив символов:

char str[] = "ТЕКСТ";

Такая запись и короче и понятнее, чем общепринятая для начальной инициализации массивов:

char str[] = { 'Т', 'Е', 'К', 'С', 'Т', '\0' };

Если длина строки в квадратных скобках опущена, то она определяется автоматически, по количеству присваиваемых символов. В приведенном выше примере она равна шести.

Запишем программу *(рис. 8.6)*, в которой функция main формирует строку по указанному выше правилу и вызывает функцию length, которая

в качестве аргумента получает начальный адрес этой строки и вычисляет ее длину (без учета завершающего символа). Эта функция представляет самостоятельный интерес и будет использована нами в дальнейшем как библиотечная.

```
/* длина строки  */
length(s)
char s[];
{
   int l=0;
   while (s[l] != "\0")
      l++;
   return(l);
}
#include <stdio.h>
main()
{
   static char str[]="ОЙ У ПОЛІ КРИНИЧКА.";
   printf("%d\n",length(str));
}
```

```
C>mak4
19
```

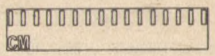


Рис. 8.6

Программа, приведенная на *рис. 8.7*, выполняет сцепление двух строк. Собственно, сцепление выполняет функция concat(s,t), которая присоединяет строку t в конец строки s. Объединяемые строки str1 и str2 объявлены как внешние, причем размер str1 достаточно большой, чтобы сохранить новую строку. Используя спецификацию формата %s, функция printf выводит всю строку сразу, причем она "знает", что печать надо остановить при достижении завершающего символа "\0".

```
/* сцепление строк */
concat(s,t)
char s[],t[];
{
   int i,j;
   i=j=0;
   /* поиск конца строки */
   while (s[i] != '\0')
      i++;
   while ((s[i++]=t[j++]) != '\0')
      ;   /* копия t    */
}
#include <stdio.h>
char str1[45]="ОЙ У ПОЛІ КРИНИЧКА,";
char str2[]= "ТАМ ХОЛОДНА ВОДИЧКА.";
main()
{
   concat(str1,str2);
   printf("%s",str1);
}
```

```
C>mak7
ОЙ У ПОЛІ КРИНИЧКА, ТАМ ХОЛОДНА ВОДИЧКА.
```

Рис. 8.7

На *рис. 8.8.* приведена программа с функцией revers(s), переставляющей символы строки s в обратном порядке. В качестве библиотечной используется ранее рассматривавшаяся функция length. Функция revers меняет местами символы строки, симметричные относительно ее середины; если в строке нечетное число символов, то средний остается на месте.

```
/* вращение строки */
#include <stdio.h>
#include "length.c"
revers(s)
char s[];
{
   int l,i=0;
   char c;
   l=length(s);
   while(i <= l/2)
   {
      c=s[i];
      s[i]=s[l-i-1];
      s[l-i-1]=c;
      i++;
   }
}
```

```
main()
{
  static char str[]=
  "ОЙ У ПОЛІ КРИНИЧКА.";
  revers(str);
  printf("%s\n",str);
}
```

```
c>mak5
.АКЧИНИРК ІЛОП У ЙО
```

Рис. 8.8.

9. ДОПОЛНИТЕЛЬНЫЕ УПРАВЛЯЮЩИЕ СТРУКТУРЫ ЯЗЫКА СИ

Кроме ранее рассмотренных управляющих структур - оператора цикла while и условного оператора if - в языке Си имеются еще три управляющие структуры: цикл do-while, цикл for и переключатель switch. Они, правда, не расширяют функциональных возможностей языка, так как все, что можно описать с их помощью, можно сделать и с помощью основных операторов while и if-else. Однако операторы do-while, for и switch часто позволяют сделать программу более простой и компактной.

9.1. Организация циклов с помощью оператора for

Оператора for является, пожалуй, самой популярной структурой для организации циклов. В общем случае его можно представить так:

for (выражение1; выражение2; выражение3) оператор;

В одной строке этот оператор определяет сразу три составляющие, отделяемые друг от друга точкой с запятой: а) начальное значение параметра цикла ("выражение1"); б) условие окончания цикла ("выражение2"); в) закон изменения параметра цикла ("выражение3"). Чаще всего "выражение1" и "выражение3" -присваивания или обращения к функциям, а "выражение2" - операция отношения. Формально этот оператор эквивалентен последовательности операторов:


```
выражение1;
while (выражение2)
{
  оператор;
  выражение3;
}
```

Использовать ли while или for - во многом дело вкуса. В одних случаях более естественным выглядит цикл while, в других - цикл for. В следующем примере *(рис. 9.1)* оператор for служит для организации и печати последовательных чисел Фибоначчи, не превосходящих m. Оператор k=1 в заголовке цикла for выполняется лишь однажды. Проверка k<=m происходит перед каждым (в том числе и перед первым) выполнением тела цикла. Тело цикла выполняется, если проверяемое условие истинно. Затем следует изменение параметра: k=k+j; этот оператор, кстати, формирует очередное число Фибоначчи, а оператор j=k-j хранит предыдущее.

```
/* числа Фибоначчи  */
#include <stdio.h>
main()
{
  int m,k,j=1;
  printf("введи m\n");
  scanf("%d",&m);
  for(k=1;k<=m;k=k+j)
  {
    printf(" %d",k);
    j=k-j;
  }
}
```

```
C>mak3
введи m
50
 1 1 2 3 5 8 13 21 34
```

Рис. 9.1

Как и в операторе while, тело цикла for может состоять из единственного оператора или из нескольких, заключенных в скобки. Следующая программа *(рис. 9.2)* предназначена для печати гистограммы длин читаемых слов (признак конца слова - пробел, запятая или "\n").

Оператор for управляет циклом, состоящим по сути из одного оператора if, так что скобки можно было бы и опустить. Для построения гистограммы выбран символ, внутреннее представление которого в коде ASCII численно равно 220.

В скобках после ключевого слова for можно размещать любые выражения. Однако "выражение1" и "выражение3" не обязаны присваивать начальное значение и изменять параметр цикла, а "выражение2" не обязано быть проверкой условия. Но компилятор всегда интерпретирует "выражение2" как истину или ложь (ложь - двоичный нуль, истина - все остальное). Любое из трех выражений может быть опущено, хотя точка с запятой должна остаться. Допускается даже такой вырожденный случай: for (;;);. Он означает бесконечный цикл, так как пропущенное "выражение2" всегда трактуется как истина.

```
#include <stdio.h>
main()
{
  char str[24];
  int i,n=0;
  printf("введи строку:\n");
   while((str[n++]=getchar())!='\n')
     ;
  printf("%d\n",n);
  for(i=0;i<n;i++)
  {
    if(str[i]!=' '&&str[i]!=','&&
      str[i]!='\n')
      printf("%c",220);
    else
      printf("\n");
  }
}
```

```
C>mak7
введи строку:
1 12 123 123456 1234 12
24
```

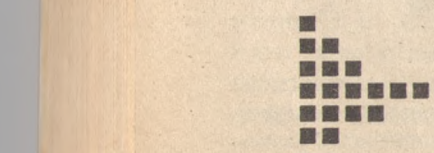


Рис. 9.2

В Си есть еще операция "," (запятая), которая чаще всего используется в операторе for. Пара выражений, разделенных запятой, вычисляются слева направо. В каждой из трех составляющих оператора for можно помещать несколько выражений, разделенных запятыми. Этот прием часто используют, например, для синхронного изменения значений двух индексов при обработке массивов. В качестве примера приведем функцию substr, которая позволяет из заданной строки s1 выделить подстроку s2 указанной длины L, начиная с k-го по порядку символа (нумерация символов начинается с нуля). Функция принимает также параметр m-общее количество символов в исходной строке, чтобы контролировать возможный выход за ее пределы *(рис. 9.3)*.

```
/* выделение подстроки */
substr(s1,s2,l,k,m)
char s1[],s2[];int m,l,k;
{
 int i,j;
 if(l+k >= m)
   k=m-l-1;
 for(i=1,j=0;i<l+k;i++,j++)
   s2[j]=s1[i];
}
#include <stdio.h>
main()
{
  char str1[80],str2[80];
  int k,l,m=0;
  printf("введи строку\n");
  while((str1[m++]=getchar()) != '\n')
    ;
  printf("\n");
 /* m сохраняет свое значение */
  printf("введи l и k\n");
  scanf("%d %d",&l,&k);
  substr(str1,str2,l,k,m);
  printf("%s\n",str2);
}
```

```
c>mak3
введи строку
ОЙ У ПОЛІ КРИНИЧКА.
введи l и k
7 6
ЛІ КРИ
```

Рис. 9.3

Для организации циклов при обработке массивов чаще всего используют именно оператор for. Например, заголовок for (i=0; i<M; i++) типичен для обработки первых M элементов массива. Это аналог DO-цикла в Фортране и ПЛ/1. Есть, правда,и существенные отличия: во-первых, пределы параметра цикла for в Си можно менять внутри самого цикла; во-вторых, как бы ни закончился цикл, параметр сохраняет свое последнее значение. Это справедливо для всех типов цикла языка Си и мы этим, кстати, уже воспользовались в предыдущем примере. И еще одна существенная деталь: поскольку компоненты оператора for - произвольные выражения, то этот тип цикла не ограничен только арифметическими прогрессиями.

Приведенная на *рис. 9.4* программа посвящена линейному поиску в неупорядоченном массиве. С помощью оператора for в программе main организуется ввод элементов исходного массива. Поиск заданного элемента в массиве обеспечивает функция lpoisk. Массив из n элементов дополняется (n+1)-м элементом - "сторожем", который равен ключу поиска g. "Пустой" цикл while (x[i] !=g) определяет значение i, соответствующее месту элемента g в исходном массиве. Если такой элемент с самого начала отсутствовал в исходном массиве, то i будет равно n+1. По этому признаку и определяется, есть элемент g в массиве или нет. Построим еще один пример, в котором естественным образом сочетается применение как оператора while, так и оператора for. Программа (*рис. 9.5*) находит в строке слово максимальной длины, а также индексы его начала и конца. Слова отделяются друг от друга пробелами или запятыми (других разделителей мы не рассматриваем). Программа последовательно анализирует символы строки на пробел или запятую (считается, что строка не может начинаться с пробела), подсчитывая количество букв в слове. При обнаружении конца слова его длина k сравнивается с длиной максимального слова lmax, найденной ранее. При необходимости (k > lmax) текущий максимум изменяет свое значение. Одновременно с поиском самого длинного слова фиксируется индекс конца этого слова (kon), по которому после просмотра всей строки определяется индекс его начала: nac = kon - lmax.


```
#include <stdio.h>
lpoisk(x,n,g)
int x[],g,n;
{
  int i=0;
  x[n]=g;
  while(x[i] != g)
    i++;
  if(i != n)
    printf("элемент %d на %d месте\n",g,i);
  else
    printf("элемент %d не обнаружен\n",g);
}
/*_____________________________*/
main()
{
  int n,g,i,a[13];
  printf("введи n\n");scanf("%d",&n);
  printf("введи массив:\n");
  for(i=0;i<n;i++)
  {
    printf("%s%d%s","a[",i,"]=");
    scanf("%d",&a[i]);
  }
  printf("введи ключ поиска\n");
  scanf("%d",&g);
  lpoisk(a,n,g);
}
>mak4
веди n
веди массив:
[0]=6
[1]=2
[2]=9
[3]=33
[4]=7
веди ключ поиска
3
лемент 33 на 3 месте
```

Рис. 9.4

```
/* слово максим.длины */
#include <stdio.h>
main()
{
  char str[48];
  int nac,kon,lmax,i,k,n;
  i=k=lmax=n=0;
  printf("введи строку:\n");
  while((str[n++]=getchar())!='\n')
    ;
  for(i=0;i<n;i++)
  {
    if(str[i] != ' ' &&
       str[i] != ',' &&
       str[i] != '\n')
      k++;
    else
      { if(lmax<k)
        {
          lmax=k;kon=i;
        }
        k=0;
      }
  }
  nac=kon-lmax;
  kon-=1;
  for(i=nac;i<=kon;i++)
    printf("%c",str[i]);
  printf("\n");
  printf("%d %d %d\n",nac,kon,lmax);
}
```

```
c>mak1
введи строку
ОЙ У ПОЛІ КРИНИЧКА
КРИНИЧКА
10 17 8
```

Рис. 9.5


```
/* СОРТИРОВКА                    */
#include "stdio.h"
main()
{
  extern int sor();
  int i,n;
  static a[10]={10,1,9,2,8,3,7,4,6,5};
  n=10;
  sor(a,n);
  for (i=0;i<n;i++)
     printf(" %d",a[i]);
}
/*_____________________________*/
int sor(x,n)
int x[];int n;
{
  int i,j,z;
  for(i=0;i<n-1;i++)
    for(j=i+1;j<n;j++)
      if(x[i]>x[j])
      {
        z=x[i];
        x[i]=x[j];
        x[j]=z;
      }
  return;
}
```

```
C>mak2
1 2 3 4 5 6 7 8 9 10
```

Рис. 9.6

Преимущество оператора for - компактность и объединение в одной строке трех операторов управления циклом - особенно заметно проявляется во вложенных циклах. Очередная программа *(рис. 9.6)* предназначена для упорядочения массива целых чисел по возрастанию. Функция sor реализует один из наиболее простых алгоритмов сортировки - метод

"пузырька". Еще один пример со вложенными циклами демонстрирует программа на *рис. 9.7*. Она использует одну полезную при обработке текстов функцию index (название substr, index заимствованы из языка ПЛ/1, где аналогичные функции применяются очень широко). Функция index определяет, входит ли в некоторую строку s1 заданная подстрока s2 и выдает положение (индекс) в строке s1, начиная с которого строка s2 содержится в s1. Если s2 не входит в s1, то функция возвращает - 1.

```
/* индекс строки  */
index(s1,s2)
char s1[],s2[];
{
  int i,j,k;
  for(i=0;s1[i] != '\0';i++)
  {
    for(j=i,k=0;s2[k] != '\0'
      &&s1[j] == s2[k];j++,k++)
      ;
    if(s2[k] == '\0')
      return(i);
  }
  return(-1);
}
char str1[]="информатика";
char str2[]="форма";
#include <stdio.h>
main()
{
  printf("%d\n",index(str1,str2));
}
```

```
A>mak9
2
```

Рис. 9.7

9.2. Организация циклов с помощью оператора do - while

В общем виде этот оператор можно записать следующим образом:

```
do
  оператор;
while (выражение);
```

Выполняется "оператор", а затем вычисляется "выражение". Если оно истинно, то снова выполняется "оператор" и т.д. Если "выражение" становится ложным, циклический процесс заканчивается. Это так называемый цикл с постусловием: условие завершения цикла проверяется не в его начале, как это имеет место в операторах while и for, а в конце, уже после прохода по телу цикла. Как следствие, тело цикла обязательно выполняется по крайней мере один раз. Этот тип цикла встречается нечасто, но иногда бывает полезен. Составим, например, функцию length(s), которая вычисляет длину строки s с учетом завершающего нуля. Известно, что число символов в строке не меньше единицы, если учитывать этот нулевой символ. Поэтому естественно предположить, что мы будем проходить по телу цикла по крайней мере один раз. Значит, можно использовать цикл do-while *(рис. 9.8)*.

9.3. Операторы break и continue

Часто при возникновении некоторого события удобно иметь возможность досрочно завершить цикл. Используемый для этой цели оператор break (разрыв) вызывает немедленный выход из циклов, организуемых с помощью операторов for, while, do-while ,а также прекращение оператора switch. Приведенная ниже программа обеспечивает поиск в заданном массиве элемента, равного g *(рис. 9.9)*. В случае обнаружения такого элемента оператор break прекращает дальнейшее выполнение цикла. Так как параметр i сохраняет значение после выхода из цикла, то дальнейший анализ его значения (if (i==n)) позволяет судить об удачном (i<=n) или неудачном (i==n) поиске. В случае вложенных циклов оператор break немедленно прекращает выполнение самого внутреннего из объемлющих его циклов.На *рис. 9.10* приведена программа, которая ведет подсчет числа различных элементов в массиве. Каждый очередной элемент a[i] сравнивается с последующими элементами массива. Если он не совпадает ни с одним из этих элементов, в счетчик k добавляется единица. В противном случае внутренний цикл прерывается оператором break и начинается новая итерация внешнего цикла.

```
/* длина строки   */
length(s)
char s[];
{
  int i,l;
  i=l=0;
  do
    l++;
  while(s[i++] != '\0');
  return(l);
}
#include <stdio.h>
main()
{
  printf("%d\n",length(""));
  printf("%d\n",length("мама"));
}
c>mak0
1
5
```

Рис. 9.8

```
/* линейный поиск  */
#include <stdio.h>
int a[]={1,2,3,33,5,6,0,8};
int n=8;
main()
{
  int i,g=33;
  for(i=0;i< n;i++)
    if(a[i] == g)
      break;
  if(i == n)
    printf("%d не найден\n",g);
  else
    printf("%d на %d месте\n",g,i);
}
c>mak3
33 на 3 месте
```

Рис. 9.9


```
/* число разных элементов */
#include <stdio.h>
main()
{
  static a[]={7,3,7,4,3,6};
  int i,j,m,k=1;m=6;
  for(i=0;i<m-1;i++)
  {
    for(j=i+1;j<m;j++)
      if(a[i] == a[j])
        break;
    if(j == m)
      k++;
  }
  printf("%d разных элем.\n",k);
}
c>mak1
4 разных элем.
```

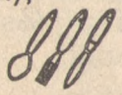


Рис.9.10

В следующем примере *(рис. 9.11)* решается такая задача. Даны две литерные величины A и B.Проверить,можно ли из букв, входящих в A,составить B. Буквы можно переставлять, но каждую из них нельзя использовать более одного раза. Очередная буква из B последовательно сравнивается со всеми буквами из A. При совпадении внутренний цикл немедленно прерывается первым оператором break, но предварительно найденная в строке A буква, нужная для построения B, затирается. Для лучшего понимания происходящего, при каждом проходе внутреннего цикла выводится состояние строки A. Второй оператор break завершает внешний цикл, как только обнаружится, что буквы, требуемой для создания строки B, нет в A. Обратите внимание на такую деталь: конец строки A определяется по символу "/0", который автоматически вставляет транслятор, а строка B завершается символом "\n", поступившим последним в строку при нажатии клавиши "ввод".

Оператор continue тоже предназначен для прерывания циклического процесса, организуемого операторами for, while, do-while. Но в отличие от оператора break, он не прекращает дальнейшее выполнение цикла, а только немедленно переходит к следующей итерации того цикла, в теле которого он оказался. Он как бы имитирует безусловный переход на конечный оператор цикла, но не за пределы самого цикла.

```
#include <stdio.h>
main()
{
  int j,i=0;char B[10];
  static char A[]="информатика";
  printf("введи строку B:\n");
  while((B[i++]=getchar())!='\n')
    ;
  for(i=0;B[i] != '\n';i++)
  {
    for(j=0;A[j] != '\0';j++)
    {
      if(A[j] == B[i])
      {
        A[j]='*';/*  затираем  */
        printf("%s\n",A);
        break;
      }
    }
    if (A[j] == '\0')
    {
      printf("нельзя\n");
      break;
    }
  }
  if(B[i] == '\n')
  printf("можно\n");
}
```

```
C>mak0
введи строку B:
мафия
инфор*атика
инфор**тика
ин*о**атика
*н*о**атика
нельзя
```

Рис.9.11

Программа на *рис. 9.12* использует оператор continue для пропуска отрицательных элементов массива, суммируя только положительные.

```
#include <stdio.h>
main()
{
  static int a[]={1,2,-3,4,-5,6};
  int i,n,s;
  n=6;s=0;
  for(i=0;i < n;i++)
  {
    if(a[i] <=0)
      continue;/* пропуск <0 */
    s+=a[i];
  }
  printf("сумма=%d\n",s);
}
```

```
C>mak4
сумма=13
```

Рис.9.12

9.4. Переключатель.

Мы уже упоминали о том, что использовать для многозначного ветвления в программе вложенные операторы if-then-else неудобно. Если глубина вложенности этих операторов свыше трех, то конструкция теряет ясность. Более наглядным и понятным в такой ситуации выглядит оператор switch (переключатель), специально предназначенный для принятия одного из многих решений. Чаще всего этот оператор выглядит следующим образом:

```
switch (целое выражение)
{
  case константа 1: оператор 1;
  case константа 2: оператор 2;
  . . .
  . . .
  . . .
  case константа n: оператор n;
  default         : оператор;
}
```

При выполнении этого оператора вычисляется выражение, стоящее в скобках после ключевого слова switch, которое должно быть целым. Оно, в частности, может быть и символьным значением (напомним, что в языке Си символьные значения автоматически расширяются до целых значений). Эта целая величина используется в качестве критерия для выбора одного из возможных вариантов. Ее значение сравнивается с константами операторов case. Вместо целой или литерной константы в операторе case может стоять даже некоторое константное выражение. Значения таких констант (выражений) должны быть различными в разных операторах case. При несовпадении выполняется переход к следующему case и сравнивается его константа. В случае совпадения "константы_i" выполняется "оператор_i", а также все последующие операторы case и default. Кстати, это несколько отличается от правил, принятых в других языках программирования, таких как Паскаль, АДА, Модула. Если не было ни одного совпадения и имеется оператор default, то выполняется стоящий за ним оператор. Если же оператора default не было, выполнение программы продолжится с оператора, следующего за структурой switch. Таким образом, при каждом выполнении оператора просматриваются все метки case. Рассмотрим следующую программу *(рис. 9.13)*.

```
/* проверка switch   */
#include <stdio.h>
main()
{
  int k=2;
  switch(k)
  {
    case 0: printf("выбор 0\n");
    case 1: printf("выбор 1\n");
    case 2: printf("выбор 2\n");
    case 3: printf("выбор 3\n");
    default:printf("default\n");
  }
}
```

```
C>mak11
выбор 2
выбор 3
default
```

Рис.9.13

Как видите, происходит то, о чем мы говорили: выполняется альтернатива, соответствующая k=2 и все последующие операторы case, а также вариант default! Чтобы обеспечить выбор одного из многих вариантов (что и требуется чаще всего), используют обычно оператор break, который вызывает немедленный выход из оператора switch *(рис. 9.14)*. Для этой же цели можно применять и оператор return, а continue можно применять лишь в случае, когда сам оператор switch находится внутри цикла. Тогда continue вызывает немедленный переход к следующей итерации, без рассмотрения оставшихся case.

```
/* проверка switch  */
#include <stdio.h>
main()
{
  int k=2;
  switch (k)
  {
    case 0: printf("выбор 0\n");
            break;
    case 1: printf("выбор 1\n");
            break;
    case 2: printf("выбор 2\n");
            break;
    case 3: printf("выбор 3\n");
            break;
    default:printf("default\n");
  }
}
```

```
C>mak11
выбор 2
```

Рис.9.14

В заключение этого раздела приведем программу для укрупненного начертания некоторого слова и использующую оператор - переключатель. Укрупненное начертание какого-нибудь символа связано с заданием этого символа в виде таблички, заполненные клетки которой образуют нужный контур. Содержимое такой таблички можно закодировать с

помощью 1 (клетка заполнена) и 0 (клетка не заполнена). Если такая табличка состоит из 7*5 элементов, то для полной идентификации символа будет достаточно пяти семиразрядных двоичных чисел. Каждое из них будет кодировать один из 5 вертикальных слоев. Например, начертание символа "M" *(рис. 9.15)* кодируется следующими двоичными числами: 1111111 (первый и пятый вертикальные слои таблички), 10 (второй и четвертый слои), 100 (третий слой), причем переход от младших разрядов к старшим соответствует просмотру исходной таблички сверху вниз. Для последовательной распечатки символа надо только установить, какие двоичные цифры находятся в соответствующих разрядах всех 5 чисел: если 0 - печатать пробел, если 1 - выбранный символ (например "*").

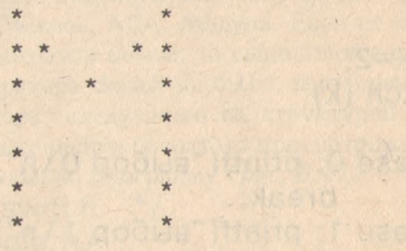


Рис.9.15

Для определения цифры k-го разряда каждого из 5 чисел, кодирующих букву, в программе *(рис. 9.16)* организовано выделение младшего разряда с помощью операции поразрядного логического умножения (&) с последующим сдвигом вправо на одну позицию всех разрядов числа (см. программу "подсчет единиц" на *рис. 7.3*).

Для кодирования вертикальных слоев всех букв исходного слова ("мама") используется оператор-переключатель. Просматривая в цикле все буквы исходного слова, оператор switch обеспечивает нужные присвоения элементам двумерного массива B соответствующих кодов, заданных восьмеричными константами. Буквы исходного слова могут быть набраны на любом из регистров клавиатуры терминала. Заметим только, что первый индекс двумерного массива соответствует порядковому номеру буквы в слове, а второй - вертикальному слою в выбранной букве. В следующих циклах (по k, i, j) организуется описанный выше процесс выделения всех k разрядов каждого j-го слоя всех i букв слова. Результаты работы программы приведены на *рис. 9.16*, вслед за текстом программы.


```
#include <stdio.h>
main(){
  int i,j,k,n;int B[6][5];
  static char s[]="Mama";
  char c;
  for (n=0;(c=s[n])!='\0';n++)
    switch(c) {
      case "m": case "M": case "м":
      case "M": B[n][0]=B[n][4]=0177;
                B[n][1]=B[n][3]=02;
                B[n][2]=04;
                break;
       case "a": case "A": case "а":
      case "A": B[n][0]=B[n][4]=0174;
                B[n][1]=B[n][3]=022;
                B[n][2]=021;
                break;
    }
  for(k=0;k<7;k++) {
    for(i=0;i<n;i++)
    { j=0;
     do
     {
      if(B[i][j] & 01)
        printf("%c",s[i]);
      else
        printf("%c",' ');
      B[i][j]=B[i][j] >> 1;
      j++;
     }
     while(j < 5);
     printf("%s","  ");
    }
    printf("\n");
  }
}
```

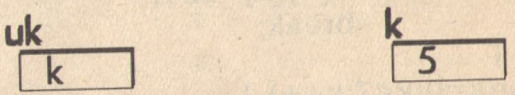
```
C>mak 2

M     M   a        m     m    a
MM   MM   a a      mm   mm   a   a
M  M  M  a   a     m  m  m  a   a
M     M  a   a     m        m  a   a
M     M  aaaaa     m        m  aaaaa
M     M  a   a     m        m  a   a
M     M  a   a     m        m  a   a
```

Рис.9.16

10. УКАЗАТЕЛИ

Указатель - это переменная, значением которой является адрес другой переменной. Пусть uk-указатель на переменную k, содержащую число 5. Схематически это можно изобразить так:



Над каждым из прямоугольников указано имя переменной, внутри - значение соответствующей переменной. Так как указатель может ссылаться на переменные разных типов, с указателем в языке Си связывается тип того объекта, на который он ссылается. Для описания указателей используется операция косвенной адресации *. Например, указатель uk описывается так: int *uk. Унарная операция &, примененная к некоторой переменной, показывает, что нам нужен адрес этой переменной, а не ее текущее значение. Если переменная uk объявлена как указатель, то оператор присваивания uk = & x означает: "взять адрес переменной x и присвоить его значение переменной-указателю uk".

Унарная операция *, примененная к указателю, обеспечивает доступ к содержимому ячейки памяти, на которую ссылается указатель. Например, *uk можно описать словами как "то, что содержится по адресу, на который указывает uk". Указатели могут использоваться в выражениях. Если, например, переменная uk указывает на целое x, то *uk может во всех случаях использоваться вместо x; так, *uk+1 увеличивает x на единицу, а *uk=0 равносильно x=0. Два оператора приваиания uk=&y; y=*uk; выполняет то же самое, что и один оператор y=x. Польза от применения указателей в таких ситуациях, мягко выражаясь, невелика.

Наиболее полно их преимущества проявляются при обработке массивов и, в частности,символьных строк. Указатели и массивы тесно связаны друг с другом. Прежде чем рассмотреть эту связь подробно, заметим, что если uk - некоторый указатель, то uk++ увеличивает его значение и он теперь указывает на следующий, соседний адресуемый объект. Значение uk используется в выражении, а затем увеличивается. Аналогично определяются операции uk—, ++uk, —uk. В общем случае указатель uk можно складывать с целым числом i. Оператор uk+=i передвигает ссылку на i элементов относительно текущего значения. Эти конструкции подчиняются правилам адресной арифметики.

А теперь вернемся к массивам. Пусть имеется описание int a[5]; Оно определяет массив размером в 5 элементов, т.е. пять последовательно расположенных ячеек памяти a[0], a[1], a[2], a[3], a[4]. Адрес i-го элемента массива равен сумме адреса начального элемента массива и смещения этого элемента на i единиц от начала массива. Это достигается индексированием: a[i] - i-й элемент массива. Но доступ к любому элементу массива может быть выполнен и с помощью указателей, причем, более эффективно. Если uk - указатель на целое, описанный как int *uk, то uk после выполнения операции uk=&a[0] содержит адрес a[0], а uk+i указывает на i-й элемент массива. Таким образом, uk+i является адресом a[i], а *(uk+1) - содержимым i-го элемента (операции * и & более приоритетны, чем арифметические операции). Так как имя массива в программе отождествляется с адресом его первого элемента, то выражение uk=&a[0] эквивалентно такому: uk=a. Поэтому значение a[i] можно записать как *(a+i). Применив к этим двум элементам операцию взятия адреса, получим, что &a[i] и a+i также идентичны.

Раньше, в связи с использованием функции scanf, мы говорили, что применение указателей в качестве аргументов функции дает способ обхода защиты значений вызывающей функции от действий вызванной функции. На *рис. 10.1* приведен текст программы с фукнцией obmen(x,y), которая меняет местами значения двух целых величин. Так как x,y - адреса переменных a и b, то *x и *y обеспечивают косвенный доступ к значениям a и b. К сказанному добавим, что использование указателей позволяет нам обходить ограничения языка Си, согласно которым функция может возвращать только одно значение.

Если в качестве аргумента функции передается имя массива, то оно фактически определяет адрес начала массива, т.е. является указателем. В качестве иллюстрации напишем очередную версию фукнции length, вычисляющей длину строки. Если имя массива передается функции, то в самой функции можно исходить из того, что она работает с массивом, можно исходить и из того, что она работает с указателем.

```
/* обмен A,B  */
obmen(x,y)
int *x,*y;
{
  int t;
  t=*x;
  *x=*y;
  *y=t;
}
#include <stdio.h>
main()
{
  int a,b;
  a=3;b=7;
  obmen(&a,&b);
  printf("a=%d b=%d",a,b);
}
```

```
C>mak0
a=7 b=3
```

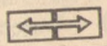


Рис. 10.1

В определении фукнции формальные параметры **char s[ ]** и **char *** совершенно идентичны. Операция **s++** *(рис. 10.2)* увеличивает н единицу текущее значение указателя, первоначально указывающее н первый символ строки, а операция ***s != '\0'** сравнивает очередной симво с признаком конца строки.

```
/* длина строки  */
length(s)
char *s;
{
  int i;
  for(i=0;*s!='\0';s++)
     i++;
  return(i);
}
```

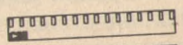


Рис. 10.2

Кроме ранее рассмотренных операций адресной арифметики, к указателям можно применять операции сравнения **==** и **!=**. Даже операции отношения **<,>=** и т.п. работают правильно, если указатели ссылаются на элементы одного и того же массива. В последнем случае возможно даже вычитание ссылок: если **u** и **s** ссылаются на элементы одного массива, то **u-s** есть число элементов между **u** и **s**. Используем этот факт для составления еще одной версии функции **length** *(рис. 10.3)*. Сначала **u** указывает на первый символ строки (**char *u = s**). Затем в цикле по очереди проверяется каждый символ, пока в конце концов не будет обнаружен "\0" . Разность **u-s** дает как раз длину строки.

```
/* длина строки */
length(s)
char *s;
{
  char *u=s;
  while(*u != '\0')
    u++;
  return(u-s);
}
```

Рис. 10.3

Для иллюстрации основных аспектов применения указателей в Си рассмотрим функцию копирования строки **s1** в строку **s2**. Сначала приведем версию, основанную на работе с массивами *(рис. 10.4)*. Для сравнения рядом помещена версия с использованием указателей *(рис. 10.5)*.

```
/* копия строки  */
copy(s1,s2)
char s1[],s2[];
{
  int i=0;
  while((s2[i]=s1[i])!='\0')
    i++;
}
```

Рис. 10.4

```
/* копия строки  */
copy(s1,s2)
char *s1,*s2;
{
  while((*s2=*s1)!='\0')
  {
   s2++;s1++;
  }
}
```

Рис. 10.5

Здесь операция копирования помещена непосредственно в условие, определяющее момент окончания цикла: **while ((*s2=*s1)!='\0')**. Продвижение вдоль массивов вплоть до тех пор, пока не встретиться "\0", обеспечивают операторы **s2++; s1++**. Их, однако, тоже можно поместить в проверку *(рис. 10.6)*

```
/* копия строки  */
copy(s1,s2)
char *s1,*s2;
{
  while((*s2++=*s1++)!='\0')
    ;
}
```

Рис. 10.6

```
/* копия строки  */
copy(s1,s2)
char *s1,*s2;
{
  while(*s2++=*s1++)
    ;
}
```

Рис. 10.7

Еще раз напомним, что унарные операции типа * и ++ выполняются справа налево. Значение ***s++** есть символ, на который указывает **s** до его увеличения. Так как значение "\0" есть нуль, а цикл **while** проверяет, не нуль ли выражение в скобках, то это позволяет опустить явное сравнение с нулем *(рис. 10.7)*. Так постепенно функция копирования становится все более компактной и ... все менее понятной. Но в системной программировании предпочтение чаще отдают именно компактным и, следовательно, более эффективным по быстродействию программам.

В языке Си принято, что некоторая литерная строка, выраженная как "строка", фактически рассматриватеся как указатель на нулевой элемент массива "строка". Допускается, например, такая интересная запись: **char *uk; uk = "ИНФОРМАТИКА";**. Последний оператор присваивает указателю адрес нулевого элемента строки, т.е. символа "И". Возникает вопрос, где находится массив, содержащий символы "ИНФОРМАТИКА"? Мы его нигде не описывали. Ответ такой: эта строка - константа; она является частью функции, в которой встречается, точно так же, как целая константа **4** или символьная константа **"A"** в операторах **i=4; c="A";**. Более детально пояснит сказанное программа на *рис. 10.8,* которая печатает строку символов в обратном порядке.

```
#include <stdio.h>
main()
{
 char *uk1,*uk2;
  uk1=uk2="ИНФОРМАТИКА";
  while(*uk2 != '\0')
    putchar(*uk2++);
  putchar('\n');
  while(--uk2  >= uk1)
    putchar(*uk2);
  putchar('\n');
}
```

```
C>mak0
ИНФОРМАТИКА
АКИТАМРОФНИ
```

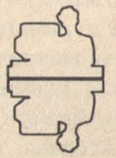


Рис. 10.8

В самом начале указателям uk1 и uk2 присваивается начальный адрес строки "ИНФОРМАТИКА". Затем строка посимвольно печатается и одновременно указатель uk2 смещается вдоль строки. В конце вывода uk2 указывает на последний символ исходной строки. Во втором цикле while все тот же указатель uk2 начинает изменяться в обратном направлении, уменьшаясь до тех пор, пока он не будет указывать на нулевой элемент массива, обеспечивая выдачу строки в обратном порядке. Приведенная на *рис. 10.9* программа использует фукнцию paly, которая определяет, является ли предъявленная ей строка палиндромом ("перевертышем"). Аргументом для нее служит строка, а это в соответствии с правилами языка означает, что мы передаем функции указатель на нулевой элемент строки. Функция использует и другой указатель u, который в первом цикле пробегает по строке, пока не найдет символ перехода на новую строку ("\n"). Именно этим символом мы завершает ввод строки в основной программе. Затем значение этого указателя уменьшается на единицу, чтобы он указывал на последнюю букву строки. Заметьте, что при этом количество символом в строке не фиксируется и нигде не используется. В следующем цикле while указатели сдвигаются навстречу друг другу (s увеличивается, а u уменьшается) и сравниваются равноудаленные от начала и конца символы, на которые ссылаются эти указатели соответственно. Когда указатели встретятся, мы объявим, что эта строка - *палиндром.*

В основной программе организован посимвольный ввод строки. Чтобы избавиться от проблемы верхнего и нижнего регистров, надо набирать все прописными буквами. Программа организована так, что выбрасывает из потока разделители: пробел, запятую, тире. Завершает строку символ "\n", который мы принудительно вставляем после того, как отфильтрованы все небуквенные символы.

В завершение этого раздела покажем, как с помощью указателей можно обрабатывать числовые массивы. На *рис. 10.10* показана работа функции dicho, определяющей, встречается ли заданный элемент g в упорядоченном массиве x из n элементов .Элементы массива расположены в возрастающем порядке.Функция выдает местоположение g в массиве(число между 0 и n-1), либо сообщает об отсутствии его. Функция рассматривает параметр x как указатель на начало массива. На каждом "обороте" цикла вычисляется средина массива mid, а x+mid указывает на этот "срединный" элемент. Всякий раз определяется, равен ли g среднему, меньше или больше его. В случае несовпадения элементов отбрасывается та половина массива, в которой искомого элемента заведомо быть не может (ввиду упорядоченности массива). К оставшейся части массива применяется та же процедура, что и к целому массиву. Так продолжается

пока не будет обнаружен искомый элемент, либо пока мы не удостоверимся, что такого элемента в массиве нет.

```
/* фразы-перевертыши */
play(s)
char *s;
{
  char *u=s;
  while(*u != '\0')
    ++u;
  --u;
  while(s < u)
    if(*s++ != *u--)
      return(0);
    return(1);
}
#include <stdio.h>
main()
{
  char c,s[40];int i;
  printf("введи строку\n");
  for(i=0;(c=getchar()) != "\n";)
    if(c!=' '&&c!=','&&c!='-')
      s[i++]=c;
  s[i]='\0';
  if (play(s))
    printf("палиндром\n");
  else
    printf("не палиндром\n");
}
```

```
A>c mak2
A>mak2
введи строку
ДЕМЕ ЛЕТОМ В МОТЕЛЕ-МЕД
палиндром
```

Рис 10.9

```
/* двоичный поиск */
#include <stdio.h>
dicho(x,n,g)
int n,g,*x;
{
  int low,high,mid,pr;
  low=0;high=n-1;pr=1;
  while(low <= high && pr)
  {
    mid=(low+high)/2;
    if(*(x+mid) == g)
      pr=0;
    else
      if(*(x+mid) < g)
        low=mid+1;
      else
        high=mid-1;
  }
  if(pr)
    printf("%d не найден\n",g);
  else
    printf("%d на %d месте\n",g,mid);
}
main()
{
  int i,g,n=9;
  static a[]={1,3,7,33,44,55,66,77,88};
  printf("введи ключ поиска\n");
  scanf("%d",&g);
  dicho(a,n,g);
}
```

```
C>mak1
введи ключ поиска
33
33 на 3 месте
```

Рис.10.10

11. СТРУКТУРЫ

Структура - это совокупность логически связанных переменных, возможно, различных типов, сгруппированных под одним именем для удобства дальнейшей обработки.

Традиционным примером структуры служит учетная карточка работающего: служащий предприятия описывается набором атрибутов, таких, как табельный номер, имя, дата рождения, пол, адрес, номер цеха (отдела), зарплата и т.д. В свою очередь, некоторые из этих атрибутов сами могут оказаться структурами. Таковы, например: имя, дата рождения, адрес, имеющие несколько компонент.

Элементы структуры обозначаются идентификаторами, с которыми можно связывать описатели типов. В результате структуру можно рассматривать как единое целое и осуществлять при необходимости выбор составляющих ее элементов.

Для образования структуры присваивают имена каждому из включаемых элементов и структуре в целом. Затем посредством специального описания задают иерархию, порядок следования и типы элементов, включаемых в структуру. Так, для рассмотренного выше примера с анкетой служащего можно выбрать имена:

tab_nom - табельный номер;
fio - фамилия, имя, отчество;
pol - пол;
adres - адрес;
summa - зарплата.

Все эти понятия можно объединить в такую, например, структуру:

```
struct anketa
{
  int tab_nom;
  char fio [30];
  char data [10];
  int pol;
  char adres [40];
  float summa;
};
```

Эта запись называется **описанием** структуры. Она начинается с ключевого слова struct и состоит из заключенного в фигурные скобки списка описаний. За словом struct может следовать необязательное имя,

которое называется **именем типа** структуры (иногда его называют тэгом или ярлыком структуры). Этот ярлык именует структуру и в дальнейшем может использоваться для сокращения подробного описания. Переменные, упоминающиеся в записи, называют **элементами.** Следом за правой фигурной скобкой, заканчивающей список элементов, может следовать список переменных, так же, как и в случае базисных типов. Вот почему в приведенном выше описании структуры после закрывающей фиругной скобки стоит точка с запятой: она завершает пустой список. Описание struct { ... } p1, p2, p3; синтаксически аналогично int p1, p2, p3; в том смысле, что каждый из операторов описывает p1, p2, p3 как переменные соответствующего типа и приводит к выделению для них памяти. Описание же структуры без последующего списка переменных не выделяет никакой памяти. Оно только определяет форму структуры и действует как шаблон. Если такое описание снабжено ярлыком (именем типа), то его можно позже использовать при определении фактических экземпляров структуры. Например, используя указанное выше описание anketa, можно с помощью строки

struct anketa a0, a1, a2;

описать структурные переменные a0, a1, a2, каждая из которых строится по шаблону, введенному структурой anketa. Любая из переменных a0, a1, a2 содержит в строго определенном порядке элементы tab_nom, fio, data, pol, adres, summa. Эти переменные, как и все остальные переменные языка, получают места в памяти.

Внешние и статические структуры можно инициировать, помещая следом за определением список начальных значений элементов:

```
struct anketa a0 = { 1024, "Макогон В.М.", "10.01.1943", 0,
                     "Одесса, Варненская, 23/99", 175.00 };
```

Каждой структурной переменной в нашем случае могут быть присвоены шесть значений, имеющих соответствующие базовые типы. Доступ к этим значениям осуществляется с помощью следующей конструкции:

имя_структуры . имя_элемента

Операция выделения элемента . (точка) связывает имя структуры и имя элемента. Например, мы можем с учетом введенных обозначений написать:

```
a0.data = "10.01.1943"; a1.summa = 0.0;
if (a2.pol == 1) man = man + 1 ;
```

Действия над структурами, в общем, ограничены. Все, что можно делать со структурой, - это взять ее адрес с помощью операции & и обращаться

к ее элементам, как показано выше. Записи нельзя копировать или присваивать как единое целое (что возможно, например, в языке ПЛ/1); их нельзя передавать в функцию или получать оттуда целиком. Однако к указателям на структуры это замечание не относится.

На практике структурные переменные обычно появляются в виде массива или списка. Нетрудно видеть, что наши три переменные a0, a1, a2 будет проще использовать, если их объединить в массив, состоящий из элементов типа struct anketa. Применив в программе описание

struct anketa a[3];

мы можем употреблять в ней, например, такие операторы:

```
a[0].fio = "Макогон В.М." ;
if (a[i].tab_nom > a[i+1].tab_nom)
{  p = a[i].tab_nom;
   a[i].tab_nom = a[i+1].tab_nom;
   a[i+1].tab_nom = p;
}
```

11.1. Структуры и указатели.

Мы уже упомянули, что нельзя передать функции структурную переменную целиком в качестве аргумента (хотя можно передавать отдельные элементы). Но существует возможность обойти это ограничение, используя в качестве аргумента указатель на структуру. Описание

struct anketa *uk ;

говорит, что uk - указатель на структуру типа anketa. Обозначение

uk -> элемент_структуры

относится к конкретному элементу структуры и означает выборку этого элемента, например: uk -> tab.nom. Поскольку uk есть указатель на структуру anketa, то к элементу tab_nom можно обращаться и так:

(*uk). tab_nom,

если учесть, что указатель установлен на начало массива структур. Имя массива, как обычно, эквивалентно адресу его начального элемента и при добавлении к указателю на структуру или вычитании из него целого числа размер структуры учитывается автоматически. Так, оператор uk = a; устанавливает указатель на первый экземпляр массива структур, а запись ++a обеспечивает автоматический переход к следующему экземпляру. В выражении (*uk).fio скобки обязательны, так как приоритет операции выделения элемента "." выше, чем у "*".

12. ПРЕПРОЦЕССОР ЯЗЫКА СИ

Мы уже использовали некоторые возможности препроцессора языка Си, сейчас поговорим о нем более подробно. **Препроцессор** (макропроцессор) - это составная часть стандартного пакета языка Си, которая обрабатывает исходный текст программы до того, как он пройдет через компилятор. Препроцессор читает строки текста и выполняет действия, определяемые командными строками. Если первый отличный от пробела символ в строке - управляющий (#), то такая строка рассматривается препроцессором как командная. Строки, не являющиеся командными, либо подвергаются преобразованиям, либо остаются без изменения.

Рассмотрим наиболее часто используемые возможности препроцессора: макрогенерация (замена лексических единиц), включение файлов, условная компиляция.

Замена лексических единиц. Командная строка вида **#define name text** вызывает в оставшейся части программы замену всех вхождений идентификатора name на строку **text**. Например, определение **#define P1 3.14159265** позволяет использовать в программе имя **P1** вместо константы **3.14159265**. Обратите внимание, что это определение не завершается точкой с запятой. Замещающий текст обычно представляет собою остаток строки. Длинное определение можно продолжить, если в конце продолжаемой строки поставить \. Внутри строк, заключенных в кавычки, подстановка не производится, так что, например, для определенного выше имени **P1** в **printf("P1")** подстановки не будет. Имена могут переопределяться и новые определения могут использовать ранее введенные определения.

Так как препроцессор не является частью компилятора языка, а представляет относительно простой макрогенератор, имеется возможность переопределять различные синтаксические единицы языка-лексемы (т.е. идентифиикаторы, ключевые слова, константы, цепочки литер, знаки операций и знаки пунктуации). В приведенной на *рис. 12.1* программе, предназначенной для выявления всех пар целых чисел из интервала **[-n,n]**, являющихся решениями уравнения **2y-x*x = 4**, используются привычные для таких языков, как Алгол, Паскаль, операторные скобки **begin - end** вместо пары { }, ключевое слово **then**. Это стало возможным благодаря предварительно определенным лексическим заменам. Приведенная выше технология замены лексем относится к **макросредствам** языка. Строка **#define name text** называется макроопределением, name называется макрошаблоном, а **text** - макрорасширением. Каждое появление имени name в теле программы называется **макровызовом**. Командрая строка

#define name (p1, p2, ... , pk) text

является макроопределением с аргументами. За именем name в круглых скобках (после **name** не должно быть пробела!) следуют разделенные запятыми формальные параметры **p1, p2, ... , pk**, также являющиеся идентификаторами. Каждый раз, когда в тексте программы встречается имя макроопределения с фактическими аргументами, они подставляются вместо формальных, так что заменяющий текст будет зависеть от вида макровызова. Определим в качестве примера такую макроподстановку:

#define MAX(X,Y) ((X) > (Y) ? (X) : (Y))

Как и в определении функции, переменные X и Y в макроопределении являются формальными параметрами. После этого строка в программе:

m = MAX (a + b, a - b);

будет заменена на строку:

m = ((a + b) > (a - b) ? (a + b) : (a - b));

Текст макрорасширения берут в скобки для обеспечения большей надежности программы. Пренебрежение скобками может привести к серьезным ошибкам, что иллюстрирует следующий пример. Программа на *рис. 12.2а* делит число **16** на квадрат числа **2** и дает правильный результат. Во второй программе *(рис. 12.2б)*, где скобки в макрорасширении опущены, результат ошибочный, так как макроподстановка породила текст:

printf ("%d\n", 16/2*2);

что, конечно, не равносильно задуманному.

Макроопределения иногда используются вместо определений функций, обычно из соображений эффективности. Но следует помнить, что препроцессор может лишь тупо и бездумно заменять одну строку на другую, не разбираясь, зачем это нужно. В отличие от параметра функции, параметр макроопределения вычисляется при каждом вхождении в макроопределение. Поэтому макровызов MAX(i++, j++) для приведенного выше макроопределениия приведет к увеличению i или j на 2.

```
#include <stdio.h>
#define then
#define begin {
#define end }
main()
begin
  int n,x,y,k=0;
  printf("введи n\n");scanf("%d",&n);
  for(x=-n;x<=n;x++)
    for(y=-n;y<=n;y++)
      if(2*y-x*x==4) then
      begin
        k=k+1;
        printf("x=%d,y=%d\n",x,y);
      end
    if(k==0) then
      printf("корней нет\n");
end
```

```
C>mak1
введи n
6
x=-2,y=4
x=0,y=2
x=2,y=4
```

Рис 12.1

```
#include <stdio.h>
#define SQR(n) (n*n)
main()
{
  printf("%d\n",16/SQR(2));
}
```

```
C>mak1
4
```

Рис 12.2а


```
#include <stdio.h>
#define SQR(n) n*n
main()
{
  printf("%d\n",16/SQR(2));
}
```

```
C>mak1
16
```

Рис 12.2б

Включение файлов. Командная строка для включения файлов выглядит следующим образом: **#include "filename"** и указывает препроцессору, что все содержимое файла с именем filename надо вставить в том месте программы, где использована командная строка. Эта возможность препроцессора позволяет следовать в Си идеям структурного программирования, согласно которым большая программа обычно расчленяется на логически завершенные части и каждая затем оформляется как самостоятельная функция. После отладки каждая из них оформляется в виде отдельного файла и при необходимости включается в отлаживаемую программу командой **#include**. Часто в таких файлах содержатся макроопределения и после включения их в исходный модуль утверждением **#include** они становятся доступными для всех функций. Например, приведенные ниже *(рис. 12.3)* полезные макроопределениия, используемые в программе на *рис. 12.4,* можно "замаскировать", поместив их в файл "**makro.h**", а в начале исходного файла с текстом программы поместить командную строку **#include "makro.h"**.

```
#include <stdio.h>
#define PR(int) printf("%d\n",int)
#define SKIP putchar('\n')
#define PRINT1(X1) PR(X1);SKIP
#define PRINT2(X1,X2) PR(X1);PRINT1(X2);SKIP
#define PRINT3(X1,X2,X3)PR(X1);PRINT2(X2,X3);SKIP
```

Рис 12.3

Команда включения может иметь другую форму: **#include <filename>**. В частности, все наши программы включали командную строку **#include <stdio.h>**, благодаря которой программы

пользователя могут обращаться к фукнциям, обеспечивающим стандартный ввод-вывод (getchar, putchar, printf, scanf и др.).

```
#include <makro.h>
main()
{
  int a,b,c;
  a=5;b=7;c=a+b;
  PRINT1(a);PRINT2(a,b);PRINT3(a,b,c);
}
```

```
C>mak1
5

5
7

5
7
12
```

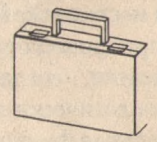


Рис 12.4

Условная компиляция. Условные конструкции препроцессора позволяют компилировать или пропускать часть программы в зависимости от выполнения некоторого условия. Условие может принимать одну из описываемых ниже форм.

#if константное_выражение

Проверяется значение выражения, составленного из констант и если оно не равно нулю, компилируется (включается) последующий текст.

#ifdef идентификатор

Последующий текст компилируется, если "идентификатор" уже был определен для препроцессора в команде **#define.**

#ifndef идентификатор

Последующий текст компилируется, если "идентификатор" в данный момент не определен. Конструкция

#undef индетификатор

исключает "идентификатор" из списка определенных для препроцессора имен. За любой из трех условных команд может следовать произвольное число строк текста, содержащих, возможно, команду вида **#else** и

заканчивающихся командой **#endif**. Если проверяемое условие справедливо, то строки между **#else** и **#endif** игнорируются. Если же проверяемое условие не выполняется, то игнорируются все строки между проверкой и командой **#else**, а если ее нет, то командой **#endif.**

Приведенная на *рис. 12.5* программа иллюстрирует применение некоторых из рассмотренных выше команд, обеспечивающих условную компиляцию.

```
#define SIZE 16
#include <stdio.h>
main()
{
  char c='A';
#ifndef SIZE
  int x=123;
  printf("x=%d\n",x);
#else
  static char x[SIZE]="информатика";
  printf("x=%s\n",x);
#endif
  printf("%c\n",c);
}
```

```
C>mak0
x=информатика
A
```

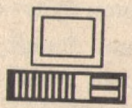


Рис 12.5

13. РАБОТА С ФАЙЛАМИ

В тех случаях, когда программа обрабатывает достаточно большой объем данных, последние обычно организуются в файлы и хранятся вне оперативной памяти ЭВМ. Наиболее эффективным устройством для организации внешнего хранения данных являются диски. Файл представляет организованную совокупность данных на внешнем устройстве компьютера - магнитном диске, ленте, принтере и т.д. Прежде чем читать или записывать информацию в файл, его нужно открыть с помощью стандартной библиотечной функции fopen. Программа, использующая эту функцию, должна включать во время компиляции системный файл stdio.h, в котором обычно с помощью оператора typedef определен новый тип данных - FILE.

В программе нужно описать ссылки на файлы и выглядит это, например, так: FILE *fopen( ); FILE *fu; (Первое описание требуется не во всех реализациях языка). Здесь fu означает указатель на FILE, а fopen выдает ссылку на этот файл. Функция fopen имеет следующий заголовок:

FILE *fopen (fname, type); char *fname, type;

Обращение к fopen в программе делается так: fu = fopen (fname, type); Строка символов fname содержит имя файла, который надо открыть; type - тоже строка символов, заключенная в кавычки и указывающая, как будет использоваться файл: "r" - чтение, "w" - запись, "a" -дозапись. Функция fopen возвращает указатель, с помощью которого мы в дальнейшем будет обращаться к этому файлу. Примеры:

```
FILE *uin, *uout;
uin=fopen("MAK1", "r");
uout=fopen(MAK2", "w");
```

Файл с именем MAK1 открывается для чтения и далее идентифицируется как uin; файл MAK2 открывается для записи и связывается с идентификатором uout. Если происходит открытие несуществующего файла, то он создается заново. Открытие существующего файла для записи приводит к уничтожению его старого содержимого. При обнаружении ошибки (скажем, при попытке чтения несуществующего файла) функция fopen выдает нулевой указатель со значением NULL.

Теперь нужно сказать, как читать уже открытый файл или же записывать в него информацию. Имеется несколько возможностей. Проще всего -использовать функции getc и putc. Эти функции, подобно

detchar и putchar, являются макрокомандами. Функция getc вводит (выбирает) из файла очередной символ; ей нужно только знать указатель на файл: s = getc(uin). Если при обработке достигается конец файла, то функция getc возвращает значение EOF.

Функция putc, в противоположность detc, заносит значение символа s в файл, на который указывает uout: putc(s,uout). Более того, она и возвращает значение этого символа.

После завершения обработки файла его следует закрыть с помощью функции fclose. Эта функция противоположна fopen. Она разрывает связь между указателем на файл и внешним набором данных. Освободившийся при этом указатель на файл можно использовать для обработки другого файла. При нормальном завершении программы в большинстве операционных систем все открытые файлы автоматически закрываются. Однако рекомендуется явно закрывать все файлы, дальнейшая обработка которых в программе не предполагается. В качестве иллюстрации приведем программу *(рис. 13.1)*, которая выбирает из входного потока, поступающего из клавиатуры дисплея (он завершается символом CTRL/Z), только буквы латинского алфавита и последовательно, символ за символом, выводит их в файл с именем "LITER", открытый на магнитном диске.

Имя устройства вывода предшествует имени файла, образуя единое целое: CAS: - магнитная лента, A: - диск. Если этот префикс опущен, то подразумевается диск.

```
/* создание литерного файла */
#include <stdio.h>
main()
{
  char c;
  FILE *out;
  out=fopen("a:LITER","w");
  while((c=getchar()) != EOF)
    if(c >= 'a'&&c <= 'z'||
      c >= 'A'&&c <= 'Z')
      putc(c,out);
  fclose(out);
}
```

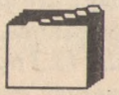


Рис 13.1

Выведя командой DIR на консоль (экран дисплея) оглавление диска, убеждаемся, что файл с именем "LITER" действительно создан. Его содержимое можно вывести на экран, введя команду: A > type LITER.

Следующая программа *(рис 13.2)* открывает файл "LITER" для ввода и последовательно, символ за символом, читает информацию из этого файла и выводит ее на печатающее устройство, которому соответствует стандартное имя "prn".

```
#include <stdio.h>
main()
{
  char c;int n=0;
  FILE *in,*fprn;
  in=fopen("a:liter","r");
  fprn=fopen("prn","w");
  while((c=getc(in)) != EOF)
  {
   putc(c,fprn);
   n++;
   if(n%10)
     putc(' ',fprn);
   else
     putc('\n',fprn);
  }
}
```

```
C>mak2

j c u k e n g Z H J
C U U K q s m A W Y
P R O L D V T X B A
```

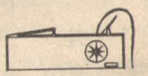


Рис 13.2

С началом работы любой программы автоматически открываются три стандартных файла: стандартный файл ввода, стандартный файл вывода и стандартный файл диагностики. Соответствующие указатели на эти файлы называются stdin, stdout, stderr и в программе их объявлять не надо. На *рис. 13.3* приведена модификация предыдущей программы, которая использует стандартный выводной файл stdout для выдачи последовательности литер из файла "LITER" на экран дисплея.


```
/* чтение литерного файла */
#include <stdio.h>
main()
{
  char c;int n=0;
  FILE *in;
  in=fopen("a:liter","r");
  while((c=getc(in)) != EOF)
  {
   putc(c,stdout);
   n++;
   if(n%10)
     putc(' ',stdout);
   else
     putc('\n',stdout);
  }
}
```

Рис 13.3

Для форматного ввода или вывода в файл можно использовать функции fscanf и fprintf. Они ведут себя точно так же, как и функции scanf и printf, но только первый параметр - это указатель на читаемый или записываемый файл, а форматная строка идет как второй параметр. На *рис. 13.4* приведена программа, которая формирует на магнитном диске файл "PROST", содержащий последовательность простых чисел, используя для форматного вывода в файл очередного числа n функцию fprintf.

Следующая программа *(рис. 13.5)* выводит часть файла "PROST" на принтер, комбинируя бесформатный ввод и форматный вывод. Обычно файл считают состоящим из отдельных записей, подобных карточкам в картотечном ящике. Запись, как правило, содержит несколько разнотипных **полей** (в предыдущих примерах запись содержала одно поле: символ, число). Наиболее просто обрабатывать записи **фиксированной** длины, в которых количество полей, порядок их расположения и длина не изменяются при переходе от одной записи к другой.

```
#include <stdio.h>
main()
{
  int i,n=3;
  FILE *out;
  out=fopen("a:prost","w");
  while(n < 500)
  {
   for(i=3;i < n/2;i+=2)
    if(n%i == 0)
      break;
   if(i >= n/2)
    fprintf(out,"%d ",n);
   n+=2;
  }
}
```

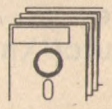


Рис 13.4

В языке Си наиболее просто обспечивается **последовательный** доступ к записям, при котором программа всегда начинает поиск с начала файла и проверяет по очереди каждую запись до тех пор, пока не будет найдена требуемая. Наиболее просто и естественно записи файла описываются с помощью структур языка Си.

Для уменьшения числа обращений к внешнему устройству пересылка данных производится блоками, а не по одному элементу за каждое обращение. С этой целью в оперативной памяти отводятся специальные участки, называемые **буферами** файлов. Управление буферами осуществляется автоматически. Однако возможен и небуферизованный ввод-вывод.

Среди стандартных функций, обслуживающих ввод-вывод, заслуживают внимания еще две функции: fgets и fputs, которые служат соответственно для ввода и вывода очередной записи файла. Обращениие к первой имеет вид:

fgets (string, size, in).

Эта функция читает из файла, на который указывает in, в символьный массив string очередную входную запись (до следующего символа новой строки, включая и символ "\n"). Может быть прочитано самое большое size-1 символов.


```
/* чтение файла  простых чисел  */
#include <stdio.h>
main()
{
  char c;int k=0;
  FILE *in,*out;
  out=fopen("prn","w");
  in=fopen("a:prost","r");
  while((c=getc(in)) != EOF)
  {
   fprintf(out,"%c",c);
   k++;
   if(k%24 == 0)
    fprintf(out,"\n");
  }
  fclose(in);fclose(out);
}
```

```
C>mak1
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
. . . . . .
```

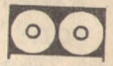


Рис 13.5

Получившаяся строка автоматически завершается символом "\0". При достижении конца файла функция выдает значение NULL. Обращение к функции вывода строки имеет вид: fputs (string, out). Она записывает в файл, на который указывает out, очередную порцию данных, выбираемых из строки string, включая символ новой строки.

Рассмотрим в качестве иллюстрации к сказанному программу формирования на магнитном диске файла - каталога книг некоторой библиотеки. Запись файла содержит поля: avtor, naswa, mesto, isdat, god и ее формирование происходит в режиме диалога с пользователем, во время которого программа на экран последовательно выдает вопросы: "автор?", "название?" и т.д. Сформированная запись функцией fprintf выводится в файл с именем KATALOG1. Сразу же после текста программы на *рис. 13.6* распечатано содержимое сформированного файла.

```
/* формирование каталога */
#define SIZE 48
#include <stdio.h>
main()
{
  int god,m,i=1;
  char avtor[SIZE];
  char nasva[SIZE];
  char mesto[SIZE];
  char isdat[SIZE];
  FILE *out;
  out=fopen("a:KATALOG1","w");
  printf("кол-во записей?\n");scanf("%d",&m);
  while(i<=m)
  {
    printf("введи %d запись:\n",i);
    printf("   автор?");scanf("%s",avtor);
    printf("   название?");scanf("%s",nasva);
    printf("   место?");scanf("%s",mesto);
    printf("   издательство?");scanf("%s",isdat);
    printf("   год?");scanf("%d",&god);
    fprintf(out,"%s %s %s %s%d\n",avtor,nasva,
                                  mesto, isdat,god);

    i++;
  }
  fclose(out);
}
```

```
C>copy KATALOG1 prn

Вирт Н. Алгоритмы+структуры=программыМосква Мир 1985
Брусенцов Н. Микрокомпьютеры Москва Наука 1985
Вирт Н. Систематическое программирование Москва Мир 1977
Баурн С. Операционная система UNIX Москва Мир 1986
Вирт Н. Паскаль Москва Финансы и статистика 1982
```

Рис 13.6

В следующей программе (*рис. 13.7*) файл KATALOG1 открывается как вводной и функция **fgets** последовательно, запись за записью, читает информацию этого файла, пока не будет достигнут его конец. В теле цикла осуществляется выборка и печать тех записей, поле **avtor** которых совпадает с ранее введенной строкой **a[size]**. Другими словами, программа производит выборку в файле записей с заранее заданным признаком.

```
/* выборка в каталоге */
#define SIZE 48
#define LONG 196
#include <stdio.h>
main()
{
  int god,i;char c;
  char str[LONG],a[SIZE];
  FILE *in,*out;
  in=fopen("A:KATALOG1","r");
  out=fopen("prn","w");
  printf("  автор?");fgets(a,SIZE,stdin);
  while((fgets(str,LONG,in)) != NULL)
  {
    i=0;
    while((c=a[i]) != "\n")
    {
      if(c != str[i++])
       break;
    }
    if(c == '\n')
      fputs(str,out);
  }
}
```

```
C>mak2
 автор?Вирт Н
Вирт Н. Алгоритмы+структуры=программы Москва Мир 1985
Вирт Н. Систематическое программирование Москва Мир 1977
Вирт Н. Паскаль Москва Финансы и статистика 1982
```

Рис 13.7

ЛИТЕРАТУРА

1. *Кристиан К.* Введение в операционную систему UNIX - М.: Финансы и статистика, 1985, - 318 с.
2. *Беляков М.И., Ливеровский А.Ю. и др.* Инструментальная мобильная операционная система ИНМОС. - М.: Финансы и статистика, 1985, - 229 с.
3. *Хэнкок Л., Кригер М.* Введение в программиирование на языке Си. - М.: Радио и связь, 1986. - 192 с.
4. *Керниган Б., Ритчи Д., Фьюэр А.* Язык программированиия Си. Задачи по языку Си. - М.: Финансы и статистика, 1985. - 279 с.
5. *Bohm C., Jacopini G. Flow-Diagrams,* Turing Machines and Languages with only Two Formation Pules // Commun. of ACM, 1966, V. 9, № 5, P.366.

Содержание

CodeBlocks