Министерство образования Республики Беларусь

Учреждение образования

Белорусский государственный университет
информатики и радиоэлектроники

Кафедра электронных вычислительных машин

Ю.А. Луцик, А.М. Ковальчук, Лукьянова И.А.

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ, ЯЗЫК Си

УЧЕБНОЕ ПОСОБИЕ
по курсу «Основы алгоритмизации и программирования»

для студентов специальности
I-40 02 01 «Вычислительные машины, системы и сети»
всех форм обучения

МИНСК 2007

УДК 681.322 (075.8)
ББК 32.97 я 73
Л 86

Рецензент:

Луцик Ю.А.
Л 86 Основы алгоритмизации и программирования, язык С.: Учеб. пособие по курсу «Основы алгоритмизации и программирования» для студ. спец. «Вычислительные машины, системы и сети» всех форм обучения/ Ю.А. Луцик, А.М. Ковальчук, Лукьянова И.А., -Мн.: БГУИР, 2007. - с.: ил. .
ISBN 985-444-985-8

В учебное пособие включены вопросы, связанные с разработкой алгоритмов решения задач, а также рассмотрены основные конструкции языка С.

Пособие будет полезно студентам всех специальностей, магистрантам и аспирантам.

УДК 681.322 (075.8)
ББК 32.97 я 73

© Луцик Ю.А., КовальчукА.М.,
Лукьянова И.В.

ISBN 985-444-985-8

© БГУИР, 2007

Введение

Язык С(С++) часто называют языком среднего уровня. Это означает, что С(С++) объединяет элементы языков высокого уровня с функциональностью Ассемблера.

Языки высокого уровня поддерживают концепцию типов данных. Тип данных определяет набор значений, которые переменная может хранить, и набор операций, которые могут выполняться над переменными. Наряду с тем, что в языке С(С++) представлены все основные типы данных, он не так жестко типизирован, как языки Паскаль или Ада. Язык С(С++) позволяет осуществлять большинство преобразований типов. Контроль за выполнением этих преобразований, а также проверка некоторых ошибок (например, выход за границы массива) возлагается на программиста.

Реализованная в С(С++) возможность напрямую манипулировать битами, байтами, словами и указателями необходима для программирования на системном уровне.

Язык С(С++) считается структурированным языком. Отличительной чертой структурированного языка является разделение кода и данных. Одним из способов решения этой проблемы является использование подпрограмм (функций), широко использующих локальные переменные. Необходимо отметить, что излишнее использование глобальных переменных может приводить к фатальным ошибкам.

Как и ряд других структурированных языков, С(С++) поддерживает ряд операторов циклов, условные операторы и операторы ветвления. Наряду с этим нежелательно использование оператора goto.

Язык С(С++) содержит стандартные библиотеки, предоставляющие функции, выполняющие наиболее типичные задачи. Эти библиотеки легко могут быть подключены, а также дополнены.

Язык С(С++) позволяет разбивать программу на части и выполнять их раздельную компиляцию. Откомпилированные таким образом файлы объединяются для создания полного объектного кода. Преимущество раздельной компиляции в том, что при изменении одного файла не требуется перекомпиляции всей программы.

В методическом пособии использованы материалы литературных источников [1,2,3,4,5,6,7,8,9,10,11].

Блок-схема алгоритма

Общие требования к блок-схеме алгоритма

Выбранный метод решения задачи необходимо привести к виду, удобному для записи программы. Такой промежуточной формой между формальной записью и записью в виде программы на алгоритмическом языке является блок- схема. При создании блок-схем необходимо учитывать требования ГОСТ 19.002-80 и ГОСТ 19.002-80.

Каждой программе (функции) должна соответствовать отдельная блок-

схема алгоритма.

Линейные и разветвляющиеся процессы

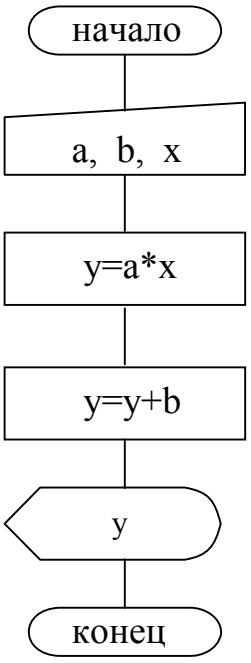
Вычислительный процесс называется линейным, если направление его продолжения на любом этапе вычислений является единственным. Алгоритм линейного вычислительного процесса описывает действия, последовательность выполнения которых не зависит от исходных данных и результатов промежуточных вычислений. ***Линейный*** процесс (как и любой другой вычислительный процесс) можно представить в виде следующих этапов:

задание исходных данных;

реализация вычислений

вывод результатов вычислений..

Пример: разработать блок-схему вывода значений функции y=ax+b. Значение аргументов a, b, x ввести с клавиатуры.



Разветвляющимся вычислительным процессом называется процесс, который в зависимости от исходных данных и результатов промежуточных вычислений осуществляется по одному из нескольких возможных вариантов направлений. Направления, по которым может выполняться вычислительный процесс, называется ветвями. Следовательно, отличие разветвляющегося вычислительного процесса от линейного состоит в том, что на этапе вычислений процесс может быть реализован по двум и более ветвям. При этом выбор направления (ветви) зависит от некоторого условия. Если условие выполняется, то выбирается одна ветвь иначе другая. Условие, определяющее выбор некоторой ветви алгоритма (вычислений), может быть сложным и состоять из нескольких простых условий. Выбор направления вычисления в этом случае осуществляется последовательной проверкой этих простых условий.

Пример: разработать блок-схему нахождения корней квадратного

уравнения $ax^2+bx+c=0$. Значение коэффициентов a, b, c ввести с клавиатуры.

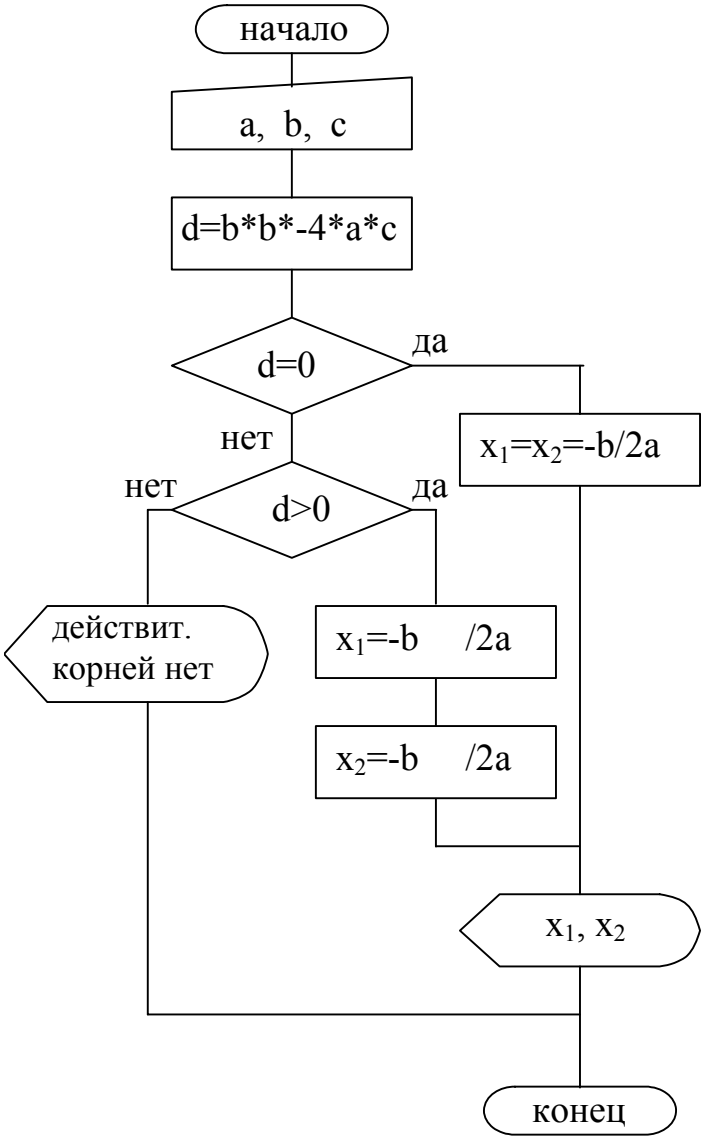


Рис 1. Блок-схема 1.

Пример . Разработать блок-схему разветвляющегося вычислительного процесса для решения системы функций.

$$y=\begin{cases}(a+2.3)/(b-1), & a-b\geq 0\\ (a-b)/a, & a-b<0\end{cases}$$

Значение коэффициентов a и b ввести с клавиатуры.

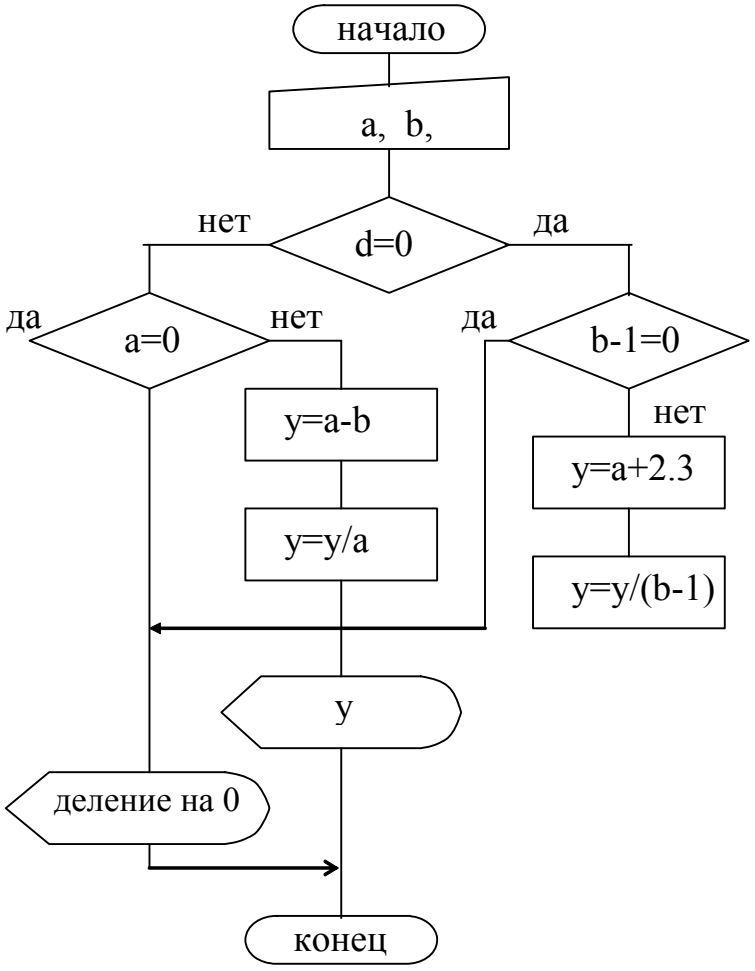


Рис 2. Блок-схема 2.

Циклические процессы

В большинстве задач, встречающихся на практике, вычисления по некоторой группе формул необходимо выполнять *многократно*. Например, вычислить значение функции в заданных точках отрезка. Этот многократно повторяющийся участок вычислительного процесса называют *циклом*. Таким образом, под *циклом* понимается оператор или группа операторов, повторяющихся некоторое количество раз. Каждый проход по телу цикла называется *итерацией*.

Цикл, не содержащий внутри себя других циклов, называют *простым*, если он содержит внутри себя другие циклы или разветвления, то цикл называют *сложным*. Обычно при каждом повторении цикла вычисления осуществляются с новыми значениями переменных. В любом циклическом процессе в ходе вычислений необходимо решать вопрос: повторять вычисления или нет? Ответ на этот вопрос получают в результате анализа значений одной или нескольких переменных, т.е. анализа некоторого условия. Анализируемую переменную называют *параметром* цикла. Из выше изложенного следует, что циклический процесс является разветвляющимся процессом с двумя ветвями, из которых одна возвращается на предыдущие блоки, т.е. реализует цикл..

Блок-схема циклического процесса должна содержать блоки:

задания начального значения параметру цикла;
блок реализации необходимых вычислений;
блок изменения параметра цикла;
блок проверки условия окончания цикла.

Во многих задачах встречаются так называемые циклы по простой переменной. В этом случае задается начальное значение переменной, закон (правило) ее изменения и конечное значение переменной. В процессе решения задачи текущее значение переменной каждый раз сравнивается с конечным, и если переменная достигла конечного значения, то осуществляется выход из цикла.

Пример: разработать блок-схему вывода значений функции y=x2+a, выбрать n значений аргумента x. Значение аргументов a, x, n ввести с клавиатуры, шаг аргумента x равен 0,25.

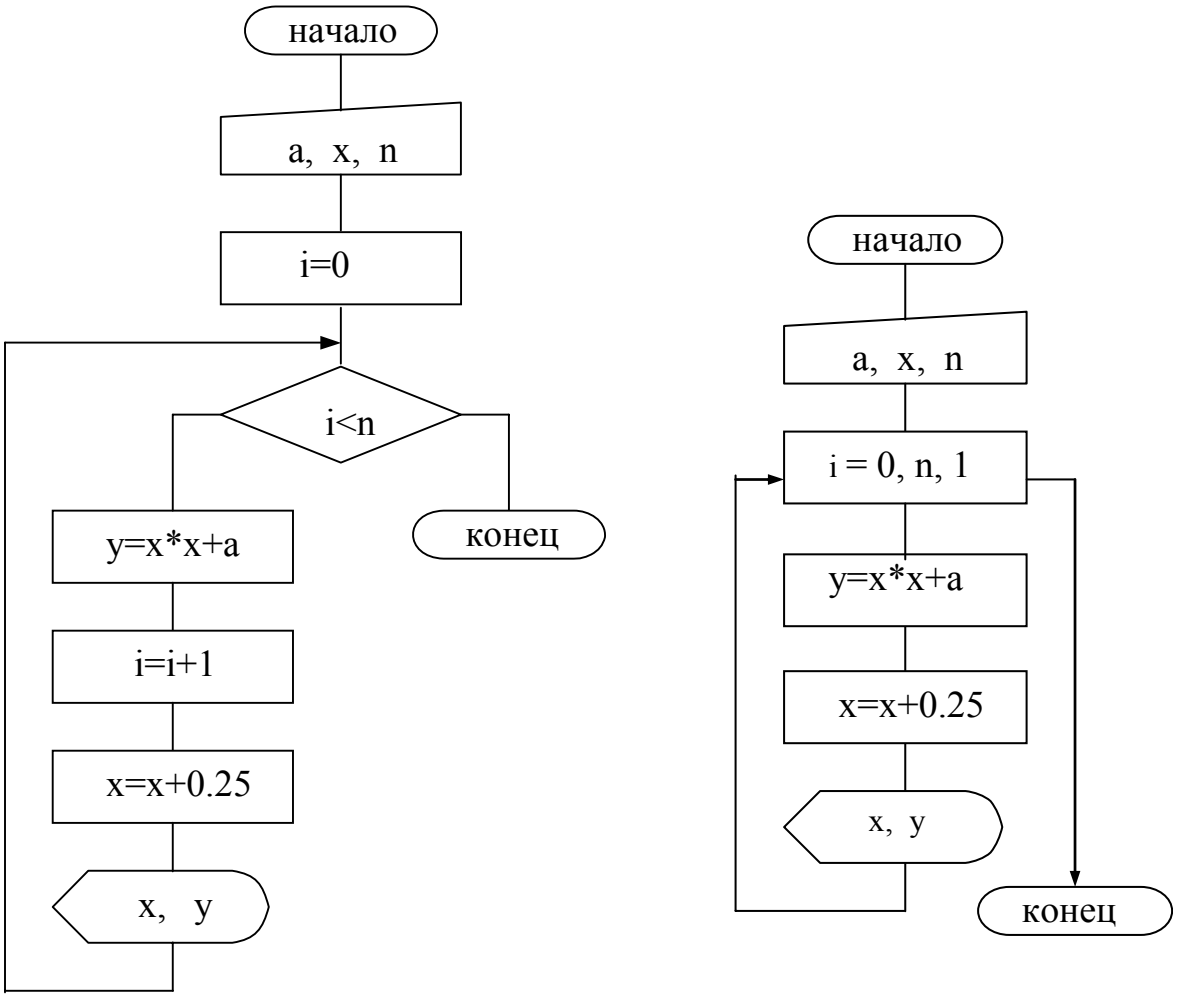


Рис 3. Блок-схема 3.

В некоторых задачах невозможно указать закон получения следующего значения переменной. В этом случае значение переменной цикла вводится из входного потока (клавиатуры).

Пример . Разработать блок-схему ввода с клавиатуры чисел в массив (ввод закончить при вводе числа 99), и вычисления max значения в массиве.

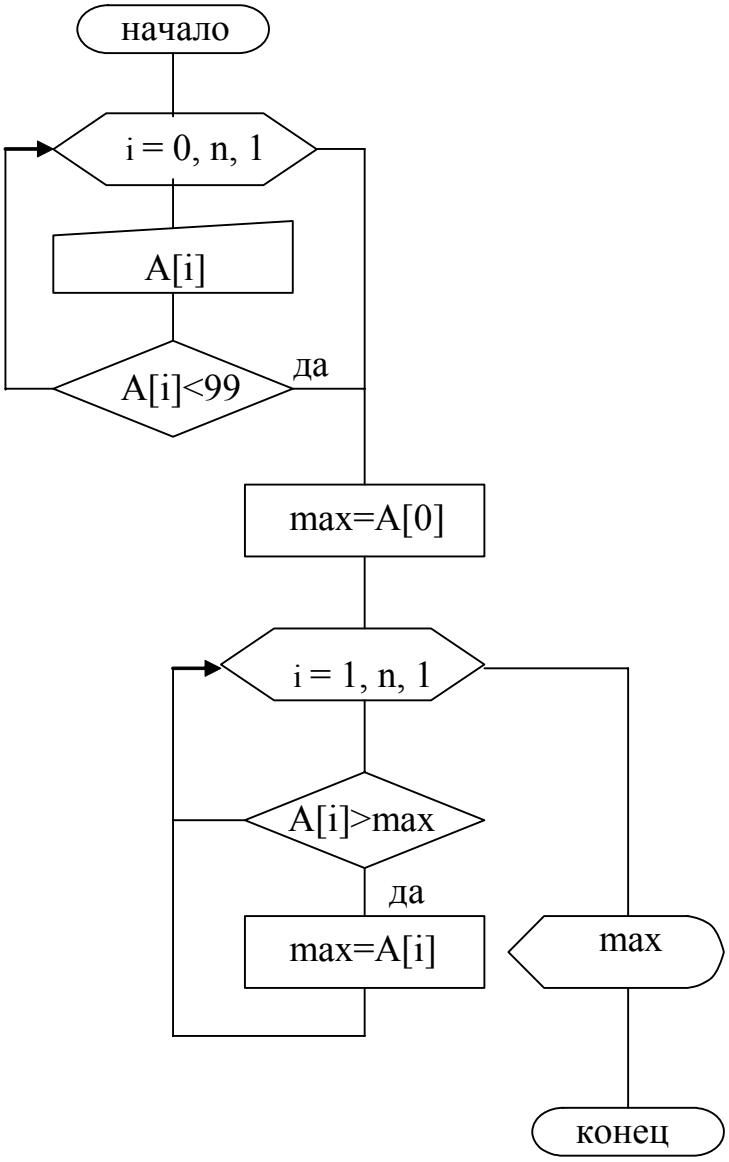


Рис 4. Блок-схема 4.

Если требуется вычислить например сумму некоторых чисел, то переменной используемой для накопления результата перед циклом необходимо присвоить нулевое значение или значение первого слагаемого. Иначе конечный результат может быть не определен (иметь любое значение).

Итерационные процессы

Итерационным процессом называется циклический процесс, который продолжается до тех пор, пока разность между соседними, уточняемыми на каждом шаге цикла (итерации) значениями, не окажется меньше или равной некоторой заданной величине. В виде итерационных вычислительных процессов реализуется большинство численных методов высшей математики: нахождение корней уравнения, вычисление значений интеграла, поиск экстремума некоторой целевой функции и др.

Пример . Разработать блок-схему нахождения квадратного корня числа n. Значение ищется на интервале [a,b] с точностью e. Значение аргументов a, b, e ввести с клавиатуры.

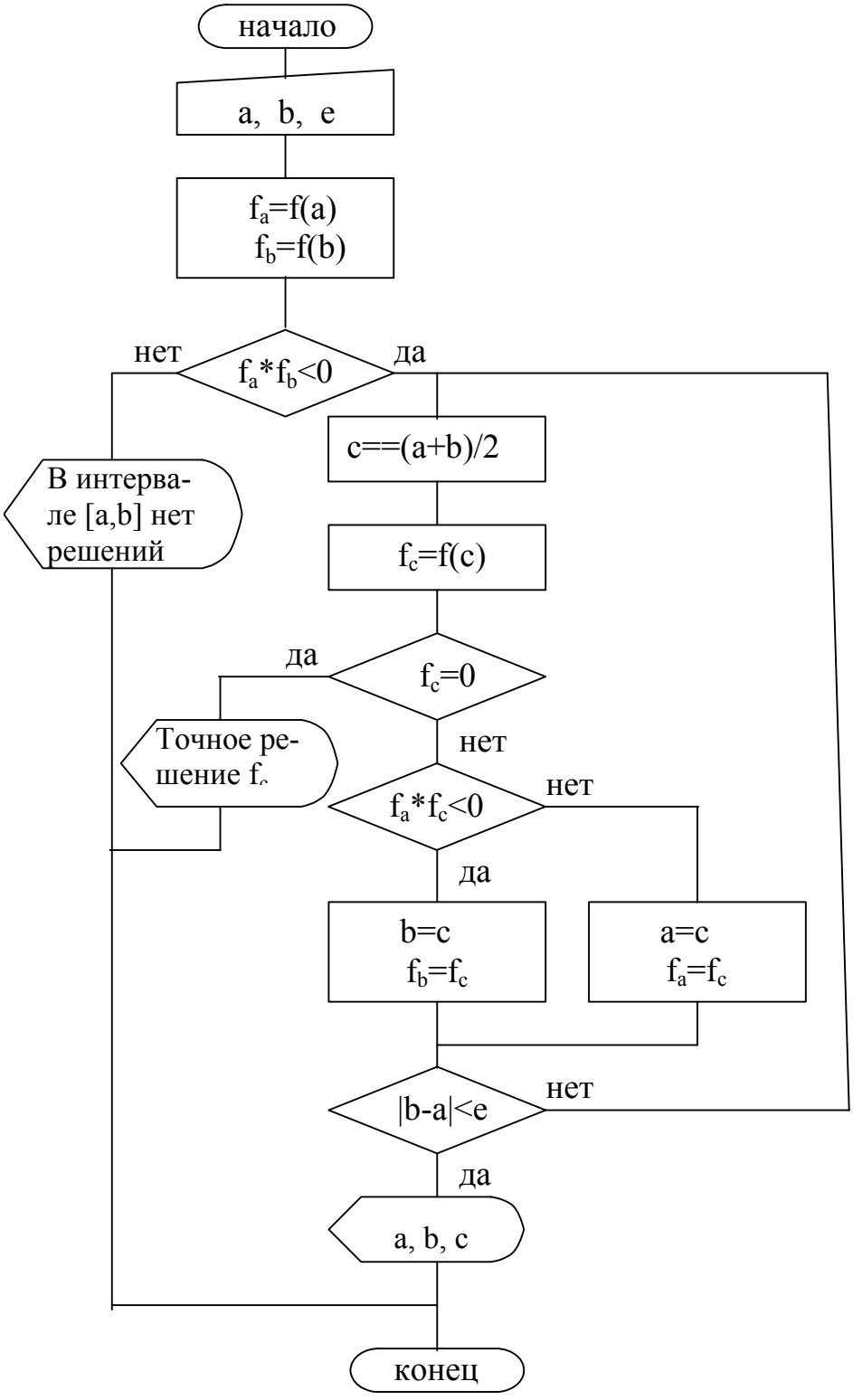


Рис 5. Блок-схема 5.

Характерной особенностью итерационного процесса является то, что в нем количество повторений вычислений (итераций, циклов) *заранее неизвестно* и становится определенным (известным) только после завершения вычислений. Решение об окончании вычислений принимается в том случае, когда результаты счета (значение функции, искомые величины) на очередной итерации отличаются от предыдущих или эталонных не более чем на некоторую, наперед заданную, величину, т.е. найдены с заданной точностью.

Второй особенностью итерационного процесса является то, что результаты вычислений очередного выполнения цикла используются как исходные данные при следующем выполнении цикла, т.е. решение находится *последовательными приближениями*, путем уточнения на каждом шаге цикла.

При решении задач вычислительный процесс, реализующий алгоритм, обычно является комбинированным, т.е. он содержит разветвления, является и циклическим и итерационным. Поэтому рекомендуется разделять весь процесс вычислений на отдельные участки, соответствующие типам рассмотренных процессов.

Основные понятия языка С(С++)

Программа на языке С(С++) состоит из последовательности *инструкций* (*операторов*), описывающих действия, выполняемые программой. Операторы используют *ключевые* слова, значение которых зафиксировано и не может использоваться для других целей (например, в качестве имени переменных). В С(С++) все операторы (за исключением команд препроцессора и описаний функций, расположенных в начале программной единицы) оканчиваются точкой с запятой ';'. Фигурные скобки {} в С(С++) применяются для обрамления начала и конца функции и некоторых других целей.

Алфавит языка (набор символов и цифр), используемый при составлении инструкций (операторов), включает латинские прописные и строчные буквы, цифры и специальные символы (. , ' : и др.).

Идентификатор – это имя, которым обозначается некоторый объект (данное) в программе. Данные в оперативной памяти размещаются по некоторым адресам, которые программисту заранее неизвестны. Для того чтобы в программе иметь возможность обращаться к этим данным и обрабатывать их, программист этим данным даёт условные имена, которые компилятор в программе заменит адресами данных в оперативной памяти.

Идентификатор должен удовлетворять следующим требованиям:

- начинаться с латинской буквы или символа _ (подчеркивание);
- содержать только символы латинского алфавита, цифры и символ подчеркивания. Использование других символов запрещено.

Нет ограничения на количество символов в идентификаторе, однако воспринимаются и анализируются только первые 32.

Примеры записи идентификаторов:

sum, result, n, m, c10, Beta, beta, _function, letter, array и т.д.

Ошибочные идентификаторы: a+b, -omega, 9c, if, int, long, &b, %f.

Так как строчные и прописные буквы различаются, то идентификаторы BETA, beta, Beta (main, Main, MAIN) будут различными. При выборе идентификатора необходимо учитывать следующее:

идентификатор не должен совпадать с кючевыми словами языка и именами функций из библиотеки языка С(С++) (С++);

не рекомендуется начинать идентификатор со знака подчеркивания, т.к. этот символ используется в именах некоторых библиотечных функций и при совпадении имен эти функции будут недоступны программе. Кроме того, к именам транслируемых функций, составляющих программу, компилятор автоматически добавляет первым символом знак подчеркивания.

Ключевые слова – это зарезервированные имена, используемые в языке С(С++) с некоторым заранее определённым смыслом. Все зарезервированные слова пишутся строчными буквами иначе эти имена будут восприняты как имена переменных. Эти слова нельзя использовать в качестве идентификаторов объектов (данных) пользователя. Ключевые слова сообщают компилятору о типе данных, способе их организации, о последовательности выполнения оперторов. Ниже приведены некоторые ключевые слова:

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

Ключевые слова **near, far, huge** определяют тип (размер) указателя на данные, а слова **_asm, cdecl, fortran, pascal** используются для организации связи с функциями, написанными на других языках программирования.

Переменные. Программы в процессе своего функционирования оперируют различными данными. Некоторые данные в процессе выполнения программы не могут изменять своего значения - *константы*, значение других может быть модифицировано - *переменные*.

Все переменные (и константы) перед их использованием должны быть объявлены. Для их объявления указывается тип данных и перечисляются через запятую список переменных имеющих данный тип. В модуле, в котором записано определение переменных. Каждой переменной в соответствии с типом выделяется необходимое количество байт памяти. Выделенному участку памяти присваивается имя переменной, которое в дальнейшем используется в программе.

Отличие объявления от определения заключается в том, что при объявлении переменной ей место в памяти соответствующей функции не выделяется, объявление лишь сообщает компилятору тип. Объявление переменных имеет следующий формат:

[спецификатор класса памяти] [модификатор] тип идентификатор_1
[=начальное значение] [,идентификатор_n [=начальное значение]];

В объявлении *идентификатора* могут присутствовать квадратные или круглые скобки, а также перед ним может быть один или несколько знаков *(звёздочка). Эти случаи объявления будут рассмотрены далее.

Спецификатор типа - одно или несколько ключевых слов, определяющих тип переменной. Язык Cu определяет стандартный набор основных типов данных(**int**, **char, double**), используя которые пользователь может объявлять свои производные (структурированные) типы (массив, структура, объединение и др.). При определении переменных им можно присвоить начальное значение.

Четыре ключевых слова: **auto, extern, register** и **static** определяют *класс памяти,* или каким образом для объявляемой переменной распределяется память и в каких фрагментах программы можно использовать её значение. Если ключевое слово, определяющее класс памяти, опущено, то класс памяти определяется по контексту.

Определения и объявления переменных рекомендуется размещать в начале программного модуля.

Комментарии

Комментарий – это сообщение, поясняющее что выполняется в данном месте программы. Комментарии позволяют повысить читабельность программы. При компиляции исходного текста компилятор игнорирует все комментарии.

Комментарий может занимать более чем одну строку. Если комментарий занимает только одну строку он может начинаться символами //, за которыми следует любой текст. Если комментарий занимает более одной строки, то либо каждую строку комментария нужно начинать с //, либо выделить начало его символами /* (начало) и */ (конец).

Типы данных

Данные в С(С++) могут быть простыми и структурированными. К данным простого типа относятся целые и вещественные числа, текстовая информация, указатели (адреса). В С имеются следующие 5 базовые *типов*: **char**(символьный), **int** (целый), **float** (вещественный), **double** (вещественный с двойной точностью), **void** (пустой тип). За исключением типа void перечисленные типы данных могут иметь *модификаторы*. К модификаторам относятся: **unsigned** (беззнаковый), **signed** (знаковый), **short** (короткий), **long** (длинный).

Тип данных и модификатор типа определяют:

- *формат* хранения данных в оперативной памяти (внутреннее представление данных);
- *диапазон* значений, в пределах которого может изменяться переменная;
- *операции,* которые могут выполняться над данными соответствующего типа.

Все типы данных можно разделить на две категории: скалярные и составные.

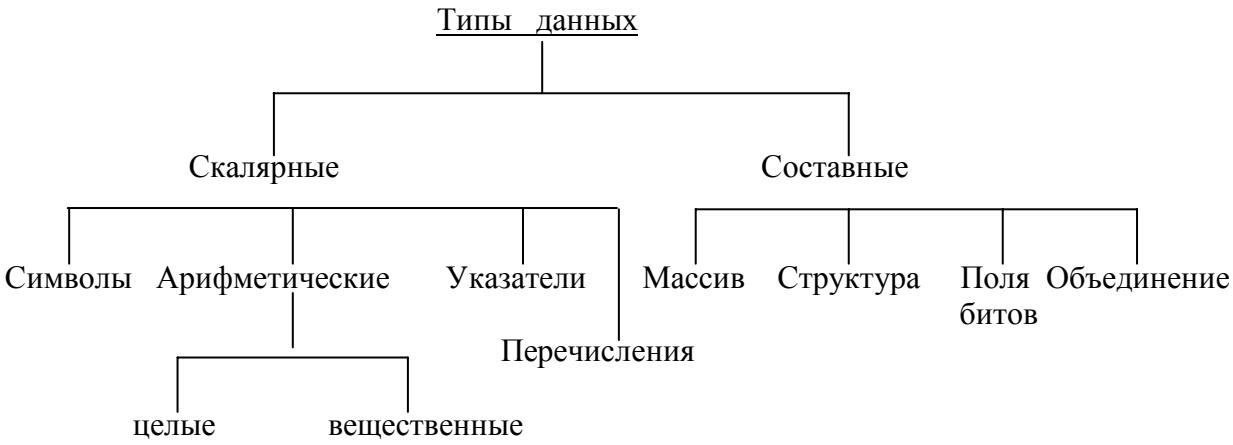


Рис 6. Типы данных .

В языке С(С++) различают понятия: описание и объявление. Описание устанавливает свойства объекта: тип, размер и так далее. Объявление приводит к выделению памяти для объекта некоторого типа. Для рассмотрения различия указанных типов данных (с точки зрения их размеров) надо познакомиться с терминами "биты", "байты", "слова".

Наименьшая единица памяти называется бит. Она может принимать одно из двух значений 0 или 1. Наименьшей адресуемой единицей памяти ПЭВМ является байт. Байт представляет собой комбинацию из 8 бит. Для удобства обмена информацией байты могут быть объединены (информационно) в слова (16,32,64 - разрядные).

Целые числа и числа с плавающей запятой в ЭВМ отличаются по размеру и способу хранения.

Данные целого типа

В таблице 1 приведены характеристики данных целого типа:

Таблица 1

название типа		байт		диапазон значений
char	символьный	1	1	-128...127
unsigned char		1	1	0...255
signed char		1	1	
int	целый	2	4	-32 768...32 767
signed int		2	4	
unsigned int		2	4	0...65 535
signed short int		2	2	-32 768...32 767
signed short int		2	2	
unsigned short int		2	2	0…65 536
long int		4	4	-147483648…2147483647
signed long int		4	4	
unsigned long int		4	4	0...4 294 967 295

Как видно из приведенной таблицы размер памяти для типов **int** и **unsigned int** не определены в языке С(С++). По умолчанию размер int соответствует размеру машинного слова. Например, на 16-разрядной машине

тип int всегда 16 бит, или 2 байта (слово). На 32-разрядной машине тип int всегда 32 бита, или 4 байта (слово). Таким образом, тип **int** эквивалентен типам **short int** или **long int** в зависимости от реализации. Аналогично тип **unsigned int** эквивалентен типам **unsigned short** или **unsigned long**. Поэтому программы, зависящие от спецификации размера **int** и **unsigned int,** могут быть непереносимы.

В процессе объявления переменные могут быть инициализированы, т.е. им присваиваются начальные значения, которые в ходе выполнения программы могут изменяться. При этом компилятор, как правило, не выполняет проверку на соответствие присваиваемого значения и типа переменной (например, int i=999999), а выдает лишь предупреждение.

Если в процессе работы требуется информация о размере некоторого объекта (или его типа), то может быть использована операция sizeof (имя_объекта), возвращающая число байт объекта. Например:

```
i=sizeof(int);          //  i равно числу байт отводимых для объектов типа int
j=sizeof(long int);  //  j аналогично, но для данных типа типа long int
k=sizeof(i);            // к размер памяти занимаемой переменной i
```

В заключение отметим, что от мест расположения объявлений данных в процедуре они различаются на **глобальные** и **локальные**. Первые объявляются вне любой из функций и, следовательно, доступны для многих из них. В отличие от них, локальные переменные являются внутренними по отношению к функциям. Они начинают существовать при входе в функцию и уничтожаются при выходе из нее.

Данные вещественного типа

Числа с плавающей точкой представляются в соответствии с IEEE стандартом, в виде двух частей – мантиссы M и порядка P числа в двоичной системе счисления:

$A=M*2^P$

Величины типа **float** занимают 4 байта. Из них 1 бит отводится для знака, 8 бит для избыточной экспоненты и 23 бита для мантиссы. Мантисса - это число между 1.0 и 2.0. Так как старший бит мантиссы всегда 1, он не запоминается в памяти. Такое представление даёт диапазон значений приблизительно от 1.17E-38 до 3.37E+38. Величины типа **double** занимают 8 байт памяти. Их формат аналогичен формату **float**. Биты памяти распределяются следующим образом: 1 бит для знака, 11 бит для экспоненты и 52 бита для мантиссы. С учётом опущенного старшего бита мантиссы, диапазон значений получается приблизительно от 2.23E-308 до 1.67E+308.

Таблица 2

Тип	Байты	Биты	Диапазон	Точность
float	4	3	1.17E-38…3.37E+38	7
double	8	64	2.23E-308…1.67E+308	15
long double	10	80	3.37E-4932…1.2E+4932	19

Признаком константы с плавающей точкой является наличие в её записи

точки, символа E или e.

Модификатор const

В C(C++) имеется модификатор const используемый для контроля за способом доступа или модификации переменных. Переменные, используемые с модификатором const, не могут изменять своего значения в процессе работы программы, например:

const int i=24;

создает целочисленную переменную i которой присваивается значение 24 и это значение не может быть модифицировано далее в программе. При объявлении переменных с модификатором const они должны быть обязательно инициализированы.

Число байт для хранения переменной с модификатором аналогично числу байт без него.

Обычно модификатор const используют при передаче адресов параметров структурного типа в функции для запрещения изменения значений находящихся по этим адресам.

Ниже приведен пример определения и инициализации данных стандартного типа.

```
#include<stdio.h>            //  подсоединение  библиотеки с функциями
                             //  ввода/вывода
void main()                  //  определение головного модуля
  {  int i1, i2, i3;         //  три переменные целого типа со знаком
     int ix= 5, iy= -7;      //  определение и инициализация переменных
                             // целого типа
     long l1, l2;            //  длинные целые знаковые переменные
     long unsigned  lu1= 125, lu2= 1234567;   //  длинные целые беззнаковые
                                              //  переменные
     float  f1= -1.57, f2= 3.14;          //  переменные вещественного типа
     char  let, symb= 'z', n_str=  '\n';   //  символьные переменные ,
     short  s1=  100,s2= 50;          //  короткие целые знаковые переменные
     const unsigned    u=  113;       //  беззнаковая целочисленная константа
     const  k=  17;                   //   целочисленная константа
    }
```

Переменные перечисляемого типа

Ключевое слово **enum** в C(C++) используется для объявления еще одного типа данных – перечисляемого. Он предназначен для описания целых констант (типа int) из некоторого заданного множества, например:

enum months {jan,feb,mar,apr,may,jun};

enum number {one,two,four,nine};

Здесь определены два типа months и days. Для определения переменной типа months запишем:

enum months ms;

Переменная ms может принимать любое значение из списка констант перечисленных в фигурных скобках. Каждому значению из списка соответствует целое десятичное число, начиная с нуля. Каждая следующая имеет значение на единицу больше, чем предыдущая:

jan=1, feb=2, mar=3 и так далее.

enum number i1,i2;

Каждая из переменных i1 и i2 может принимать одно из четырех значений: one, two, four или nine. Определение переменных можно выполнить и при объявлении типа, например:

enum number {one,two,four,nine} i1=one, i2=four;

Перечисление может быть описано и без задания имени типа. Имена в различных перечислениях должны отличаться друг от друга. Значения внутри одного перечисления могут совпадать:

enum number {one,two=one,four=4,six=4,nine} i1=one, i2=two;

В этом случае переменные i1 и i2 будут равны обе нулю и ассоциироваться с константой one. Константы four и six будут равны четырем .

В перечислении константам можно задавать значения не по порядку, при этом если не все значения констант явно специфицированы, то они продолжают прогрессию начиная от последнего специфицированного значения:

enum number{one= 2,two,four= two+one-1,six= two+3} i1=two, i2= four;

В этом случае значения именованных констант будут следующими:

one= 2, two=3, four= 4, six= 6.

Переменные типа enum могут использоваться в индексных выражениях, как операнды в арифметических выражениях и в операциях отношения. Имя константы из списка перечисление эквивалентно её числовому значению. Именованным константам можно устанавливать как положительные, так и отрицательные значения.

Константы

В отличие от переменных, константы не изменяют своего значения в процессе выполнения всей функции (группы взаимосвязанных функций). Аналогично переменным константы могут быть следующих основных типов:

- целые;
- беззнаковые (символ U);
- вещественные;
- символьные;
- константное выражение, состоящее из констант объединенных знаками операций.

Константа целого типа. Примером константы целого типа является, например, число 241. Если требуется ввести константу типа long, то для этого надо в конце числа указать признак L или l, например, 143L. Признак L гарантирует, что для константы 143 в памяти будет отведено соответствующее число (4) байт. Это может быть важным для достижения совместимости при использовании константы с другими переменными и константами типа long.

Кроме десятичной формы представления, константы целого типа могут

быть записаны в виде

- восьмеричного числа (если запись константы начинается с цифры 0), например 016, что соответствует десятичному числу 14.

- шестнадцатеричного числа (если число начинается с символов 0x или 0X), например 0x16, что соответствует десятичному числу 22.

Ниже приведены примеры целочисленных констант:

4356; - десятичная константа,

431L; - десятичная константа типа long,

0427; - восьмеричная константа,

0x136; - шестнадцатеричная константа.

Символьная константа. Символьные константы представляют собой одиночные символы, заключенные в апострофы, например:

simv='S';

Если в апострофы заключено более одного символа, то компилятор трактует это как ошибку:

simv='SS';

При описании символьной константы вместо символа может быть использован его ASCII код, например:

simv='\123'; /* 123 - ASCII код символа S*/

В качестве символьных переменных могут использоваться управляющие символы (табл. 3).

Таблица 3.

Управляющий знак	Наименование	Код
\n	Переход на новую строку	\x0A
\t	Горизонтальная табуляция	\x09
\v	Вертикальная табуляция	\x0B
\b	Возврат на одну позицию	\x08
\r	Перевод курсора в начало строки	\x0C
\f	Новая страница	\x0D
\a	Звонок (сигнал)	\x07
\'	Одиночная кавычка	\x27
\"	Двойная кавычка	\x22
\\	Наклонная черта влево(обратный слэш)	\x5C
\ddd	ASCII символ в восьмеричном представлении	
\xdd	ASCII символ в шестнадцатеричном представлении	

При присваивании символьной переменной эти символы должны быть заключены также в апострофы:

simv='\n'; simb='\f';

Вещественные константы. В языке C(C++) допустимо несколько способов описания вещественных чисел. Наиболее общим является способ, при котором последовательность цифр включает в себя десятичную точку и символ e(E): 3.142e-2, -1.732E+4. Знак + не обязателен в записи числа. Можно пускать либо десятичную точку, либо экспоненциальную часть, но не обе

одновременно: 2.5348, .34e-2, 34e-2,, .34, 34. Использование пробелов в записи константы недопустимо.

В процессе работы константы с плавающей точкой представляются в формате double, то есть им отводится по 8 байт памяти.

```
main()
{ float sm;
  sm=3.45*3.5;
}
```

Результат операции умножения 3.45*3.5 имеет двойную точность. При выполнении присваивания происходит усечение результата до размера float. Это позволит достичь максимальной точности.

В заключение отметим, что использование модификатора const запрещает любые переопределения константы. Например:

```
const char s[]=”БГУИР”;
const float ff=23.527;
const i=341;
```

Применение модификатора без указания типа константы подразумевает по умолчанию тип int.

Использование препроцессора (директивы препроцессора #define, будет рассмотрено далее) является еще одним механизмом, позволяющим существенно упростить процедуру определения и изменения значения констант.

Структура программы на языке С(С++)

Программа на языке С(С+) представляет собой набор инструкций (операторов), объединенных в одну или более функций, которые должны быть выполнены. Среди этих функций есть одна, которая всегда должна иметь имя main, с которой и начинается выполнение всей программы. В процессе выполнения функция main обращается к другим функциям, находящимся в той же программе, либо в библиотеках, содержащих ранее написанные функции. Структура программы на С имеет вид:

```
#include<stdio.h>                    //
#include "file.h"                    //
void   fun(void);                    //   прототип функции  fun
main()                               //  описание функции main
{   int i, j, k;                     //   раздел описания данных
    scanf("%d%d", &i,  &j);          //   функция ввода нанных  i и j
    printf(" a+b= %d  a*b=  %d\n", c, a*b );  //   вывод результатов
    fun();                           //   обращение к функции fun
    return 0;                        //
}
void funk()                                    //   тело функции fun
 {    набор инструкций   }
```

Операции и выражения

В программах широко используются выражения, состоящие из одного или более операндов, объединенных знаками операций. Операции – это специальные комбинации символов, специфицирующие действия по преобразованию различных величин. Компилятор интерпретирует каждую из этих комбинаций как самостоятельную единицу, называемую лексемой. В качестве операндов могут быть использованы переменные, константы или другие выражения. Имеются три основных класса операций: арифметические, отношения (логические), побитовые. Различают операции унарные (с не более чем одним операндом) и бинарные (с двумя и более операндами). Рассмотрим основные операции, используемые в С(С++). Операции должны использоваться точно так, как они представлены в таблице: без пробельных символов между символами в тех операциях, которые представлены несколькими символами.

Таблица 4.

Знак Операции	Наименование операции	Пример
Арифметические операции		
+	Сложение	a= b + c; если b=6, c=4, то a=10
-	Вычитание	a= b - c; если b=3, c=8, то a= -5
-	Арифметическое отрицание	a= - b; если b=132, то a= -132
*	Умножение	a= b * c; если b=2, c=4, то a= 8
/	Деление	a= b / c; если b=7.0, c=2.0, то a= 3.5
%	Остаток от деления	a= b % c; если b=10, c=3, то a= 1
Логические операции		
&	Логическое “И” (конъюнкция)	a= b && c; если b и c не равны нулю, то a=1, в противном случае a=0
||	Логическое “ИЛИ” (дизъюнкция)	a= b || c; если b и c равны нулю, то a=0, в противном случае a=1
!	Логическое отрицание (инверсия)	a= ! b; если b равно нулю, то a=1, если b не равно нулю, то a=1
Побитовые операции		
&	Поразрядная конъюнкция	a= b & c; если оба сравниваемых бита единицы, то бит результата равен 1, в противном случае – 0. Если b=1010, c=0110, то a=0010
|	Поразрядная дизъюнкция	a= b | c; если любой (или оба) из сравниваемых битов равен 1, то бит результата устанавливается в 1, в противном случае - в нуль. Если b=1010, c=0110, то a=1110

^	Поразрядное “исключающее ИЛИ”	a= b ^ c; если один из сравниваемых битов равен 0, а второй бит равен 1, то бит результата равен 1, в противном случае (оба бита равны 1 или 0) – 0. Если b=1010, c=0110, то a=1100
~	Поразрядная инверсия	a= ~ b; если b=1010, то a=0101
<<	Сдвиг влево	a= b << c; осуществляется сдвиг значения b влево на c разрядов; в освободившееся разряды заносятся нули. Если b=1011, c=2, то a=1100
>>	Сдвиг вправо	a= b >> c; осуществляется сдвиг значения b вправо на c разрядов; в освободившееся разряды заносятся нули, если b имеет тип unsigned и копии знакового бита в противном случае. Если b=1011, c=2, то a=0010
Операции отношения		
==	Сравнение на равенство	a == b; вырабатывается 1, если a равно b, и 0 – в противном случае
>	Больше	a > b; вырабатывается 1, если a больше b, и 0 – в противном случае
>=	Больше или равно	a >= b; вырабатывается 1, если a больше или равно b, и 0 – в противном случае
<	Меньше	a < b; вырабатывается 1, если a меньше b, и 0 – в противном случае
<=	Меньше или равно	a <= b; вырабатывается 1, если a меньше или равно b, и 0 – в противном случае
!=	Не равно	a != b; вырабатывается 1, если a не равно b, и 0 – в противном случае
Операции присваивания		
=	Простое присваивание	a = b; a присваивается значение b
++	Унарный инкремент	a++ (++a) значение a увеличивается на единицу
--	Унарный декремент	a-- (--a) значение a уменьшается на единицу
znak=	Составная операция присваивания; znak∈{*, /, %, +, -, <<, >>, &, |, ^}	a znak= b; понимается как a = a znak b; Например, a += b, если a=3, b=4, то a=7
Операции адресации и разадресации		
&	Адресация	ptr= & b; ptr присваивается адрес b
*	Разадресация	a= * ptr; a присваивается значение по адресу ptr

Операция последовательного вычисления		
,	Запятая	a = (c-- , ++b); “,” вычисляет два своих операнда слева направо. Результат операции имеет значение и тип второго операнда. Если c=2, b=3, то a=4,c=1,b=4
Операция условного выражения		
?:	Условная операция	a= (b<0) ? (-b) : (b); a присваивается абсолютное значение b
Size-операция		
sizeof	Определение размера в байтах	a= sizeof(int); определяет размер памяти, которому соответствует идентификатору или типу. a присваивается размер типа int

Операция присваивания

Операция присваивания (=). В результате выполнения этой операции переменная, стоящая слева от знака "=", принимает значение выражения, расположенного справа. Отличительной чертой операции присваивания в языке С является то, что она может быть использована в одном выражении более одного раза. Например:

```
 void main(void)
{  int i,j,k;
   i=j=k=23;
}
```

В этом примере присваивания выполняются справа налево: сначала переменная k принимает значение 23, затем j и, наконец, i принимают то же значение.

Арифметические операции

Операция сложения (+). Выполнение операции ”+" приводит к сложению двух величин, стоящих справа и слева от этого знака. Операнды, используемые в операции, могут быть как переменными, так и константами. Операция "+" является бинарной:

```
i=j+k;
printf("%d",j+k);
```

Операция вычитания (-). В результате выполнения этой операции переменная получает значение разности чисел, стоящих соответственно слева и справа от знака "-":

```
i=j-k;
printf("%d",j-k);
```

Операция изменения знака (-). Знак "-" может использоваться для задания или изменения алгебраического знака некоторой величины (выражения):

```
i=-10;
i=-(j-k);
n=-j;
```

Операция умножения (*). В результате выполнения этой операции вычисляется произведение двух или более операндов.

Операция деления (/). Операция деления над целыми числами выполняется не так, как над числами с плавающей точкой. Результат деления целых чисел есть число целое. При этом дробная часть у результата отбрасывается - "усечение". Результат деления целых чисел округляется не до ближайшего целого, а всегда до меньшего целого. Например:

```
void main(void)
{  int i;
   float k;
   i=6/4;                    //  (результат = 1)
   k=7./4.;                  //  (результат = 1.75)
   k=6/4.;                   //  (результат = 1.25)
}
```

Операция деления по модулю (%). Операция используется в целочисленной арифметике. В результате получается остаток от деления. Например, в результате 14%4 получаем 2.

Большинство операций С(С++) выполняют преобразование типов, чтобы привести операнды выражений к общему типу, или чтобы расширить короткие величины до размера целых величин, используемых в машинных операциях. Преобразования, выполняемые операциями С(С++), зависят от специфики операций и от типа операнда или операндов. Тем не менее, многие операции выполняют похожие преобразования целых и плавающих типов. Эти преобразования известны как арифметические преобразования.

Обычные арифметические преобразования осуществляются следующим образом:

- операнды типа float преобразуются к типу double;
- если один операнд типа double, то второй операнд преобразуется к типу double;
- любые операнды типов char или short преобразуются к int;
- любые операнды типов unsigned char или unsigned short преобразуются к типу unsigned int;
- если один операнд типа unsigned long, то второй операнд преобразуется к типу unsigned long;
- если один операнд типа long, то второй операнд преобразуется к типу long;
- если один операнд типа unsigned int, то второй операнд преобразуется к unsigned int.

Операндами операции % должны быть целые числа. Остальные операции выполняются над целыми и плавающими операндами. Типы первого и второго операндов могут отличаться. Арифметические операции выполняют обычные арифметические преобразования операндов. Типом результата является тип операндов после преобразования.

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

```
void main()
{   int i, j, k, l, m, n;
    float f,f1;
    char c,c1;
    clrscr();
    i=12;
    k=32767;      // максимальное значение для int 32767, при увеличении
    k=k+1;        //  значения переменной k происходит переполнение
    c='a';        //  c присваивается значение ASCII кода символа   a
    c=c+1;        //  переход к следующей букве   (б)
    printf(" i=%d,  j = %d    k=%d   c=%c   c=%d  i-5=%d   i % 3%d",
            i,j=i+1,k,c,c,i-5,i%3);
}
```

Результатом выполнения программы будет:

Операции поразрядной арифметики

Поразрядные операции используются для преобразования информации, содержащейся в памяти ПЭВМ, путем изменения некоторых отдельных, либо всех разрядов (битов) памяти. К числу поразрядных операций относятся сдвиговые операции (вправо (>>) и влево (<<)) и операции конъюнкции (&), дизъюнкции (|), исключающее ИЛИ (^) и поразрядная инверсия (~). Операция ~i приводит к тому, что все разряды (биты), соответствующие переменной i, изменяют свое значение на противоположное, например:

```
i=j>>k;              i=j<<k;
i=j&k;               i=j|k;
i=j^k;               j=~k;
```

Операнды побитовых операций должны быть целого типа, но их типы могут быть отличными (выполняются обычные арифметические преобразования). Тип результата определяется типом операндов после преобразования. Результат операции сдвига не определен, если второй операнд отрицательный.

```
#include <stdio.h>
void main()                 //   демонстрация побитовых операций и сдвигов
{  int i, j, k, l, m, n;
   i=0xab00;  j=0xabcd;     // i и j  в шестнадцатиричной с/с
   k=i&j;
   l=i | j;
   m=i ^ j;
   n= ~j;
   printf("\ni=%x, j=%x, k=i&j=%x, l=i|j=%x,  m=i^j=%x, n=~j=%x\n",
            i,j,k,l,m,n);
          //  выводится: i=ab00, j=abcd, k=i&j=ab00, l=i|j=abcd,  m=i^j=cd,
          //  n=~j=5432; заметим, что abcd+5432=ffff,
          //  поэтому ~ ещё   называют   поразрядным дополнением
   i=10; j=16; k=i<<2; l=j>>3;
```

```
        //   i<<n равносильно i * (2 в степени n); j>>n = j / (2 в степени n)
    printf("\ni=%d, j=%d, k=i<<2=%d, l=j>>3=%d\n",i,j,k,l);
        //   выводится: i=10, j=16, k=i<<2=40, l=j>>3=2
}
```

Язык С(С++) дополнительно позволяет расширить возможности оперирования некоторыми стандартными операциями, рассмотренными выше. Так, выражение i=i+3 может быть записано i+=3. Сказанное выше справедливо и для некоторых других операций, т.е. вместо знака + в выражении может быть записаны символы (-, *, /, %, <<, >>, & ,| ,^).

Логические операции

Логические операции выполняют логическое отрицание (!), логическое И (&&) и логическое ИЛИ (||). Операнды логических операций могут быть целого, вещественного или адресного типа. Типы первого и второго операндов могут быть различными. Операнды логических выражений вычисляются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции (значение первого операнда равно нулю для логического И и единице для логического ИЛИ), то второй операнд не вычисляется.

Логические операции не выполняют стандартные арифметические преобразования. Вместо этого они вычисляют каждый операнд с точки зрения его эквивалентности нулю. Указатель имеет значение 0, если это значение явно установлено путем присваивания или инициализации. Результатом логической операции является 0 или 1. Тип результата есть int.

```
#include <stdio.h>
void main()
{  int i, j, k, l, m, n;
   i=10; j=0; k= j && (i++);        // т.к. j=0, то i не инкрементируется
   m=!j; l= i || (j++);             // т.к. i ≠ 0, то j не инкрементируется
   printf("\n i=%d, j=%d, k=j && (i++)=%d, m= !j=%d ", i, j, k, m);
   printf("\n l= i||(j++)=%d, i=%d, j=%d\n", l, i, j);
}
```

Результатом выполнения программы будет:

```
i=10, j=0, k=j && (i++)=0, m=!j=1
l=i||(j++)=1, i=10, j=0
```

Операции отношения

Бинарные *операции отношений* используются для вычисления соотношения между операндами. Операция вырабатывает значение 1(true) или 0 (false). Типом результата является int. Операнды могут быть целого, плавающего или адресного типов. Типы первого и второго операндов могут различаться.

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

```
void main()
{  int i, j, k, l, m, n;
   i=10; j=15; k=10;
   printf("\ni=%d, j=%d, k=%d, (i>j)=%d, (i<j)=%d ",i, j, k, i>j, i<j);
   printf("\n(i>=k)=%d, (j<=k)=%d, (j==k)=%d, (j!=k)=%d\n",i>=k,j<=k,
            j==k,j!=k);
}
```

Результатом выполнения программы будет:
i=10, j=15, k=10, (i>j)=0, (i<j)=1
(i>=k)=1, (j<=k)=0, (j==k)=0, (j!=k)=1.

Инкрементные и декрементные операции

Операция инкрементирования (*увеличения ++)* и декрементирования (*уменьшения --)* значения операнда. Операнд должен быть целого, плавающего или адресного типа. Операнды целого или плавающего типа преобразуются путем сложения (++) или вычитания (--) единицы. Тип результата соответствует типу операнда. Операнд адресного типа инкрементируется или декрементируется размером объекта, который он адресует. Инкрементированный указатель адресует следующий объект, а декрементированный указатель – предыдущий.

Существуют две возможности использования операции: когда знак ++ (--) находится слева от переменной - "префиксная" форма, и справа - "постфиксная" форма. По внешнему виду операция аналогична простой операции сложения (вычитания). Каковы же преимущества операции ++ (--) по сравнению с +(-).

Во-первых, более компактная форма записи. Например:

```
i=2; j=3;                       i=2; j=3;
while(i>12)                     while(i++>12)
{ j=j*i;                        { j=j*i;
  printf(”j= %d”,j)               printf(”j= %d”,j);
  i=i+1;                        }
}
```

Во-вторых, при использовании операции ++ (--) в результате компиляции получается более эффективный объектный код, так как она идентична соответствующим машинной команде.

Если операция является префиксом своего операнда, то операнд инкрементируется или декрементируется перед его использованием в выражении, а если постфиксом – после его использования

```
while(++i>12)      // в начале увеличение значения i, затем сравнение
while(i++>12)      // в начале сравнение, затем увеличение значения i
```

Операция sizeof

Операция sizeof - унарная операция, возвращающая длину в байтпх переменной или типа, помещенного в скобках. Выражение sizeof имеет форму:

```
sizeof(name);
```

где name - или идентификатор или имя типа. Имя типа не может быть void. Значением выражения sizeof является размер памяти в байтах, соответствующий поименованному идентификатору или типу.

Когда операция sizeof применяется к идентификатору массива, то результатом является размер всего массива в байтах, а не размер указателя, соответствующего идентификатору массива.

```
#include <stdio.h>
#include <conio.h>
void main()
{ int i,j;
  double d;
  char c;
  i=sizeof(i);    j=sizeof(d);
  printf("\nразмер i(int)=%d, c(char)=%d, float=%d, d(double)=%d “,
          i,sizeof(c),sizeof(float),j);
}
```

Результатом выполнения программы будет:

размер i(int)=2, c(char)=1, float=4, d(double)=8,

Порядок выполнения операций

Изменение порядка выполнения операций в выражении может привести к различным результатам, следовательно, требуется ввести набор непротиворечивых правил, определяющих порядок действий. В С(С++) это осуществляется путем задания приоритета той или иной операции. Приоритет - это уровень старшинства (первоочередности выполнения) операции. Если несколько операций имеют один и тот же уровень приоритета, то они выполняются в том порядке, в котором записаны в операторе (выражении). Операции деления и умножения имеют более высокий приоритет по сравнению со сложением и вычитанием. Это означает, что в операторе i=35*j-k вначале выполняется 35*j, затем из полученного значения вычитается k. Если требуется, чтобы сложение или (и) вычитание в операторе выполнялось ранее деления и умножения, необходимо использовать механизм скобок. Так, в операторе i=35*(j+k) вначале выполняется (j+k), затем умножение.

Приоритет операций

В приведенной ниже табл.5 показан приоритет (порядок вычисления) операций языка С(С++). Операции упорядочены по приоритету сверху вниз.

Необходимо отметить, что если в вычисляемом выражении (или его некоторой части) скобки отсутствуют, то порядок вычисления составляющих его

Таблица 5.

Операция	Приоритет
() [] -> .	слева направо
! ~ ++ — + - * & sizeof	справа налево
* / %	слева направо

+ -	слева направо
<< >>	слева направо
< <= > >=	слева направо
== !=	слева направо
&	слева направо
^	слева направо
|	слева направо
&&	слева направо
||	слева направо
?:	справа налево
= += -= *= /= %= &= ^= |= <<= >>=	слева направо

операндов в С(С++) (как и в ряде других языков) не фиксирован. Это означает, что если в выражении x=операнд1 + операнд2 значение операнда1 влияет на величину операнда2, то не обязательно значение операнда1 будет вычислено ранее операнда2.

Порядок вычисления аргументов функции также не определен и на разных компиляторах может давать несовпадающие результаты.

printf("%d %d",n++,n+i);

Во избежание этого эффекта до входа в функцию все операнды, влияющие друг на друга, должны быть вычислены.

n++;

printf("%d %d",n,n+i);

Преобразование типов

Для получения верного результата выражения желательно, чтобы операнды, входящие в выражение, и результирующая переменная были одного типа. В отличие от компиляторов с других языков, например Паскаль, это не приводит к фатальной ошибке на этапе компиляции. Вместо этого компилятор использует правила автоматического преобразования типов к некоторому общему типу:

1) char и short преобразуется в int, float - в double (в С(С++) все действия с вещественными числами производятся с двойной точностью). В С(С++) наряду с модификатором unsigned существует модификатор signed. Если он используется сам по

себе, то это равносильно signed int. Имеют место следующие преобразования:

char - int (со знаком),

unsigned char - int (старший байт всегда нулевой),

signed char - int (в знаковый разряд int передается знак из char),

short - int (знаковый или беззнаковый),

float - double;

2) enum преобразуется в int;

3) если один из операндов имеет тип double (long, unsigned), то другие преобразуются к double (long, unsigned) соответственно, результат будет иметь

тип double (long, unsigned);

4) последовательность имен типов, упорядоченных от "высшего" типа к "низшему", имеет следующий вид: double, float, long, int, short, и char. Применение модификатора unsigned повышает ранг соответствующего типа данных со знаком.

"Повышение" типа обычно заканчивается успешно, в то время как "понижение" может закончиться не всегда успешно. Это связано с тем, что все число может не поместиться в элементе данных низшего типа.

```
#include<stdio.h>
void main(void)
{ char c;      /* объявление символьной переменной */
  int i;       /* объявление переменной целого типа */
  float f;     /* объявление переменной вещественного типа */
  f=i=c='C';
  printf("c= %c, i= %d, f= %2.3f \n",c,i,f);
  c=c+1;       /*преобразования выполнятся успешно, т. к. размер-*/
  i=f+c;       /*ность памяти под переменными c, i, f  достаточна*/
  f=i/c;       /*для размещ. в них значений вычисленных выраж. */
  printf("c= %c, i= %d, f= %f \n",c,i,f);
  c=3.67e17; /*для числа 3.67e17 требуется > 1 байта (ошибка)*/
  printf("c= %c \n",c);
}
```

В С(С++) при вычислении все величины типа float преобразуются в тип double, что уменьшает вероятность погрешности при округлении. Конечный результат преобразуется к типу float (если это требуется).

Операция приведения

Рассмотренные выше преобразования типов выполняются автоматически. Существует возможность точно указать тип данных, к которому необходимо привести (преобразовать) некоторое выражение (переменную). Это заключается в том, что перед приводимым к требуемому типу выражением ставится тип в круглых скобках: (тип) выражение. Например:

```
int i:
i=(int)3.4e-2+(int)'A';
```

Вообще говоря, при разработке программ желательно не смешивать типы, во многих языках это запрещено. Однако в некоторых случаях этого избежать нельзя или это упрощает программу. В С(С++) вся ответственность за подобные преобразования ложится на программиста.

Операция запятая

В С(С++) любое выражение с оператором присваивания, заключенное в скобки, имеет значение, равное присваиваемому. Например, выражение (i=j+k) будет иметь значение суммы j+k. Использование скобок позволяет расширить смысл указанного выражения далее простого суммирования и присваивания полученного значения переменной i. Значение суммы присваивается также

всему выражению, над которым здесь же может быть выполнена другая операция, например, операция сравнения ((i=j+k)<=n). Выражение в целом будет принимать значение FALSE или TRUE в зависимости от выполнения условия. Выражение в скобках может представлять собой сложное выражение, состоящее из более чем одной формулы. Для этого и используется операция запятая. Вычисления в таком выражении выполняются слева направо, и все выражение примет значение последней вычисленной формулы. Например (i=j+1,j+4), значение всего выражения будет равно 5.

Ввод и вывод информации

Ввод/вывод (в/в) информации является одной из частей языка С(С++), которая встречается практически во всех программах. Качество организации в/в во многом определяет интерфейс пользователя с ПЭВМ. Рассмотрим основные функции в/в: printf(), scanf(), putchar(), getchar(), puts() и gets(). Первые две предназначены для организации форматного в/в данных. Вначале рассмотрим работу функции вывода данных.

Общий формат записи функции printf имеет следующий вид:

int printf(”управляющая строка”,аргумент1,аргумент2,...);

Управляющая строка представляет собой символьную строку, определяющую порядок и формат печати выводимой информации. Управляющая информация представляет собой символьную строку, заключенную в кавычки (”xxxx”). Символьная строка может содержать информацию трех видов:

1) обычная текстовая информация;

2) спецификация преобразования, согласно которой осуществляется вывод одного из аргументов, содержащихся в списке аргументов. Формат спецификации:

% [знак] [размер поля вывода][.точность] символ_преобразования.

Знак, расположенный после %, указывает на необходимость выравнивания выводимого параметра в поле вывода и имеет следующий смысл (табл. 6).

Таблица 6.

Знак	Действие
-	Выравнивание влево выводимого числа в пределах выделенного поля. Правая сторона выделенного поля дополняется пробелами. Если этот знак не указан, то по умолчанию производится выравнивание вправо, т.е. пробелы ставятся перед числом
+	Выводится знак числа. Знак "минус" при отрицательных значениях выводится всегда и не зависит от наличия данного флага
пробел	Выводится знак пробела перед положительным числом
0	Заполняет поле нулями
#	Действие зависит от установленного для аргумента типа формата. Для целых чисел выводится идентификатор системы счисления: 0 - перед восьмеричным числом; 0x или 0X – перед шестнадцатеричным. При указании типа формата e, E или f

	происходит вывод десятичной точки. Действие данного символа при использовании формата g и G идентично действию при e и E

Размер поля вывода задает количество позиций для вывода символов. Если число подлежащих выводу символов меньше, чем указано в этом поле, то слева или справа добавляются пробелы для достижения указанного значения.

Точность задает число подлежащих выводу десятичных знаков и должно начинаться точкой. Действие поля зависит от типа данных (табл. 7).

Таблица 7.

Символ	Действие
d, i, u, o, x, X	Указывает минимальное число выводимых цифр
e, E, f	Указывает число цифр, которые выводятся после десятичной точки. Последняя цифра округляется
g, G	Выводится указанное число значащих цифр
c	Не действует. Выводится соответствующий символ
s	Указывает максимальное число выводимых символов

Далее следует один из символов преобразования (табл. 8):

Таблица 8.

Тип формата	Представление данных при выводе
c	Отдельный символ
s	Символьная строка
d, i	Целое десятичное число
u	Целое беззнаковое десятичное число
O	Целое беззнаковое восьмеричное число
x	Целое беззнаковое шестнадцатеричное число (для вывода используются символы 0-f)
X	Целое беззнаковое шестнадцатеричное число (для вывода используются символы 0-F)
f	Число с плавающей запятой в записи с фиксированной десятичной точкой
e	Значение со знаком в форме [-]d.dddde[+|-]ddd
E	Значение со знаком в форме [-]d.ddddE[+|-]ddd
g	Значение со знаком в формате 'e' или 'f', в зависимости от значения и указанной точности
G	Значение со знаком в формате 'E' или 'f' в зависимости от значения и указанной точности
p	Указатель
n	Число записанных на данный момент символов
[...]	Соответствует самой длинной строке, которая состоит из перечисленных в скобках символов
[^...]	Соответствует самой длинной строке, которая не содержит перечисленных в скобках символов
%	Сам знак %, преобразование не производится

Перед типом формата могут стоять следующие модификаторы типа (табл. 9):

Таблица 9.

Модификтор	Значение
h	Если тип формата – d, i, o, x или X, то тип параметра - short int. При типе формата u тип параметра – unsigned short int
L	При типе формата – d, i, o, x или X тип параметра – long int, при u - unsigned long. При типе формата – e, E, f, g или G тип параметра – double вместо float
L	При типе формата – e, E, f, g или G тип параметра – long double

Аргументами функции printf могут быть переменные, константы, выражения, вызовы функции. Основным требованием правильной работы функции является соответствие выводимой информации и спецификации, описывающей эту информацию. В противном случае результат будет неверным.

Модификаторы * и #. Помещение # перед g, G, f, E или e приводит к выводу десятичной точки при отсутствии дробной части. Использование # перед спецификаторами x или X приводит к печати числа с префиксом 0x, перед o - к печати с префиксом 0. Применение # с другими спецификаторами запрещено.

Функция возвращает число выведенных символов

Наряду с функцией printf в C имеется несколько родственных функций форматного вывода информации. Синтаксис этих функций имеет вид:

```
int printf(const char *format, ...);
int cprintf(const char *format, ...);
int sprintf(char *buffer, const char *format, ...);
```

Рассмотрим пример использования функции printf.

```
#include <stdio.h>
void main(void)          // Пример использования функции printf
{  int j,i=32;           // переменная целого типа
   float f=132.45;       // переменная вещественного типа
   double d=-132e3;      // переменная удвоенной точности
   enum time {sec,min=3,how} t;  // объявление переменной
                                 //  перечисляемого типа
   printf("\n d= %d \n f= %f d= %lf",i,f,d);
   printf("\n d= %d \n f= %1.3f d= %.4e",i,f,d);
   printf("\n d= %#d",how);
}
```

Далее рассмотрим описание функции ввода данных scanf(), имеющей следующий формат записи:

int scanf(”управляющая строка”,аргумент1,аргумент2,...);

По виду функции printf и scanf отличаются только названием. Обе имеют управляющую строку и список аргументов. Основное отличие этих функций состоит в особенностях этого списка. Функция printf использует имена переменных, константы и выражения, в то время как функция scanf только указатели на переменные. При этом:

- если требуется ввести некоторое значение и присвоить его переменной одного из рассмотренных выше типов, то перед именем переменной требуется поставить символ & (указатель);

- если необходимо ввести значение строковой переменной, то перед ее именем ставить символ & не нужно.

Управляющая строка содержит спецификацию преобразования, в записи которой могут быть использованы также пробелы, символы табуляции и символ перехода на новую строку. Формат спецификации преобразования аналогичен функции printf. В спецификации преобразования допустимы следующие символы:

d или i - ожидается ввод десятичного числа;

o - ожидается ввод восьмеричного целого числа;

x - ожидается ввод шестнадцатеричного целого числа;

u - ожидается ввод беззнакового числа;

c - ожидается ввод одиночного символа;

s - ожидается ввод символьной строки;

e, f или g - ожидается ввод вещественного числа;

p - ожидается ввод указателя (адреса) в виде шестнадцатеричного числа;

n – получается значение, равное числу прочитанных символов из входного потока на момент обнаружения %n;

[] – сканируется множество символов.

Перед символами d, o, x, f может стоять буква l. В первых трех случаях переменные, которым присваивается вводимое значение, должны иметь тип long, в последнем - тип double. В отличие от функции printf, в спецификации преобразования функции scanf:

- отсутствует спецификация %g;

- спецификации %f и %e эквивалентны;

- для ввода целых чисел типа short применяется спецификация %p.

Ниже приведен пример использования функции scanf().

```
#include <stdio.h>
  void main(void)
  {  long l;
     float j;
     unsigned int k;
     char st[10], ss;
     scanf(”%ld %x%5.1f \n %u %c\n%s”,&i,&f,&k,&ss,st);
   }
```

Входной поток будет иметь вид

```
2365  -56  -1.467e+2   <ввод>
31256  k                <ввод>
```

строка символов <ввод>

Множество сканирования определяет набор символов, то есть символы считываются в соответствующий аргументу %[ ] массив до тех пор, пока они являются частью множества сканирования.

```
#include <stdio.h>
  void main(void)       // использование множества сканирования
 { int i;               // при вводе информации в символьную строку
   char s1[20], s2[20];
   scanf("%d%[abd ef]%s",&i,s1,s2);
   printf("%d %s %s",i,s1,s2);
 }
```

Входной поток имеет вид 24acf bhacn <ввод>.

В результате работы программы получим 24 acf b hacn. Поскольку h не является частью множества сканирования, то ввод в s1 прекращается на этом символе. Пробелы в описании спецификации преобразования игнорируются.

Для ввода/вывода символов могут быть использованы также функции getchar() и putchar(). За одно обращение к функции getchar() (putchar()) осуществляется ввод(вывод) одного символа из(в) текстового потока. Текстовый поток - это последовательность символов, разбитая на строки. Пример записи функции getchar (putchar) имеет следующий вид:

```
char c;
. . .
c=getchar();
 putchar(c);
```

Рассмотрим простейший пример программы, осуществляющей копирование по одному символу с входного потока в выходной:

```
#include <stdio.h>
void main(void)       // пример ввода / вывода символа
{   int c;
    c=getchar();          // ввод символа
    while(c!=EOF)         // пока не нажата комбинация Ctrl+z
    {  putchar(c);        // вывод символа
       c=getchar();       // ввод символа
    }
 }
```

В рассматриваемом примере переменная для вводимого символа имеет тип int, хотя для символа достаточно переменной типа char. Это делается для того, чтобы иметь возможность окончить ввод символов при прочтении из входного потока символа EOF (end of file). Для прочтения символа EOF недостаточно переменной типа char (один байт). Символ EOF выбран таким образом, чтобы он был отличен от любых других символов. Рассмотренная выше программа может быть написана короче:

```
void main(void)
{  int c;
   while((c=getchar())!=EOF)    // ввод символа и проверка его на Ctrl+z
```

```
    putchar(c);                    //  вывод введенного символа
}
```

В рассматриваемом примере приоритет операции != выше чем =, таким образом, выражение c=getchar() взято в скобки, чтобы сначала ввести символ, а затем сравнить его с EOF.

Директивы препроцессора

Директива #include

Директива препроцессора #include вставляет текст внешнего файла в текст программы, содержащей директиву. Директива имеет один из форматов:

#include <имя файла>

#include ”имя файла”

Файл, указываемый в директиве #include должен быть в формате ASCII. Угловые скобки указывают препроцессору искать включаемый файл в каталогах, определенных по умолчанию в IDE. Двойные кавычки указывают вначале производить поиск в текущем каталоге, а затем если файл не найден, в тех же каталогах, что и с угловыми скобками. Директива #include чаще всего используются для *включения заголовочных файлов*. Заголовочные файлы представляют собой библиотечные функции и имеют расширение h.

Директива #define

Директива предназначена для выполнения функции найти и заменить и имеет следующий формат:

#define argument1 argument 2

где argument1 – идентификатор не содержащий пробелов, argument2 может содержать любые символы. Директива заменяет все вхождения argument1 в тексте программы на argument2.

Так как директива препроцессора позволяет определять и изменять константы, заменяемые в ней аргументы *называют предопределенными константами*. Определить можно константу любого типа, включая строковый.

Операторы языка С(С++)

Понятие пустого и составного операторов

Составной оператор – это несколько операторов объединенных в блок с помощью фигурных скобок “{}”, или разделенных запятой. Такой блок можно рассматривать как один оператор, например:

```
#include<stdio.h>
void    main ()
{
  {     //   первый вариант составного оператора
        int  i=1, j=2, k;
        k=i+j;   printf(“\n  k=i+j =  %d”, k);
        k*=2;   printf(“\n  k=  %d”,  k);
```

```
  }     //  второй вариант составного оператора
  int  i=0, j, k;
  k=i+=j, j=4;
}
```

Пустой оператор – состоит только из знака “;”, например:

```
#include<stdio.h>
void    main ()
{  int i= 2;
   i*=3; ;          //  обычный и пустой операторы
   ;                //  пустой оператор
   ; ;              //  двойной пустой оператор
};                  //  конец составного оператора и пустой оператор
```

Оператор условного перехода

В языке С(С++) существует группа операторов позволяющая организовать ветвление в программе. В С(С++) имеются три оператора, которые могут нарушить простой линейный характер выполнения программы. К ним относятся: **if** (если) **else** (иначе) (*ветвления*), **switch** (*переключатель*) и **goto** (*безусловного перехода*).

Оператор ветвления предназначен для выбора в программе из нескольких возможных вариантов единственного варианта продолжения вычислительного процесса. Выбор выполняется исходя из результатов анализа значения некоторого выражения.

Оператор if имеет следующую общую форму записи:

if (проверка условия) {группа операторов 1 }

[else {группа операторов 2 }]

При выполнении оператора if сначала выполняется проверка условия. Если результат - истина (любое отличное от нуля значение), то выполняется группа операторов 1.

Если результат анализа условия - ложь (равен 0), то выполняется группа операторов 2. Если слово else отсутствует, то управление передается на следующий после if оператор.

В качестве условия может использоваться арифметическое, логическое выражение, выражение сравнения, целое число, переменная целого типа, вызов функции с соответствующим типом возвращаемого значения.

Ниже приведен пример функции, использующей оператор if . . . else:

```
void main(void)
#include <stdio.h>
{   short i;
    printf("введите число i \n i= ");
    scanf("%d",&i);
    printf("введенное число ");
    if (i>10) printf("больше 10 ");
    else printf("меньше 10 ");
 }
```

Если необходимо осуществить выбор из более чем двух условий, то можно конструкцию if ... else расширить конструкцией else if. Это позволяет ввести в анализ дополнительное условие, что приведет к увеличению ветвления в программе.

Одним из условий правильности работы программы, использующей конструкции вида if ...; else ...; if ...; else ...; ..., является соблюдение правил:

- каждый оператор else соответствует ближайшему оператору if, если только ближайший if не входит в группу операторов, выполняющихся при условии верхнего (предыдущего if).

- условия проверяются в том порядке, в котором они перечислены в программе;

- если одно из условий истинно, то выполняется оператор соответствующий этому условию, а проверка оставшихся условий не производится;

- если ни одно из проверенных условий не дало истинного результата, то выполняются операторы, относящиеся к последнему else;

- последний else является необязательным, следовательно, он и относящийся к нему оператор может отсутствовать.

```
        . . .                                   . . .
if (i==j){ n++;                       if (i==j){  n++;
  k*=j;}                                 k*=j;
if (n>10 && k<n)                        if (n>10 && k<n)
  printf(”n=%d”,n);                     printf(”n=%d”,n);
else n%=2;                            }  else n%=2;
```

Если анализируется более одного логического условия, то они заключаются в () и разделяются логическими операциями || (или) и && (и).

Понятия: истина и ложь. В С(С++) каждое выражение, в том числе и условное, имеет значение. При этом каждое условное значение принимает либо истинное, либо ложное значение.

```
void main(void)
{ int true,false;
  true=(4>2);       /* истина */
  false=(3>5);      /* ложь*/
  printf(”true=%d  false=%d”,true,false);
}
```

Результатом работы функции является:

true = 1 false = 0

Следовательно, значение ”истина” - это 1, а "ложь" - 0. Более того, как следует из приводимого ниже примера, только значение 0 соответствует ложному высказыванию, при остальных значениях выражение в () принимает истинное значение.

```
#include<stdio.h>
main()
{ if (1) printf("истина1");
  else printf("ложь1");
```

```
  if (0) printf("истина 0");
  else printf("ложь 0");
  if (-2) printf("истина 2");
  else printf("ложь 2");
}
```

Это свойство может быть использовано для сокращения логических условий. Например, выражения if(k!=0) и if(k) эквивалентны, так как оба принимают значение "ложь" только в случае k=0.

Необходимо различать следующие высказывания:

if(k=3) и if(k==3)

Первое из них имеет всегда истинное значение, так как в результате его выполнения переменная k принимает значение 3, и, следовательно, выражение в скобках не равно 0 (истинно) всегда. Во втором случае выражение истинно только в случае, когда k примет значение 3.

Переключатель switch. Использование оператора if позволяет выбрать один из двух путей. В отличие от него оператор switch используется для выбора одного из многих путей. Общая структура оператора switch имеет следующий вид:

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

В операторе switch производится проверка на совпадение вычисленного значения выражения с одним из значений, входящих во множество констант. Константа представляет собой целочисленную константу или константное выражение (символьные константы автоматически преобразуются в целочисленные). В операторе switch не должно быть одинаковых констант. В процессе выполнения оператора switch выражение сравнивается с каждой из констант. При совпадении выполняется соответствующий константе вариант (один или более операторов), иначе выполняется вариант, помеченный ключевым словом default, а при его отсутствии операторы, следующие за ”}”. Оператор break, приводит к немедленному выходу из switch (или далее из ниже рассматриваемых операторов цикла). Кроме break, для выхода из switch может быть использован также return.

```
#include <stdio.h>
void main(void)
{ int i;
  char c;
  while((i=getchar())!=EOF)    /* EOF  -->  Ctrl + Z */
  { switch (i)
    {  case 97: c='a'; /* из-за отсутствия оператора break выполняется */
                       /* следующая ветвь и  c  принимает значение 'b'*/
       case 98: c='b'; break;
```

```
        case 99: c='c'; break;
        default: c=NULL; return;   /*выход из функции*/
      } printf("c= %c\n",c);
    }
  }
```

Ниже приведены несколько примеров программ, в которых используется переключатель switch.

Пример простейшего калькулятора:

```
#include<stdio.h>
void main( )
{    float a, b, res;   int  prizn= 1;
     char opr;
     do{ puts("\nВведите выражение  (число_1  знак  число_2 ):");
          scanf("%f%c%f", &a, &opr, &b);
       } while( ( b == 0 ) && (( opr ==’/’) || (opr == ‘:’));
      switch (opr)
       { case '+':              res=a+b;      break;
         case '-':              res=a-b;      break;
         case '*':              res=a*b;      break;
         case ':': case '/':    res=a/b;      break;
         default: printf("\операция не определена"); pr= 0;  break;
        }
    if(pr)  printf("\n Результат: %f",   res);
  }
```

Операция условия ?:. В C(C++) имеется короткий способ записи оператора if ... else. Он называется "условным выражением" и записывается с помощью операции ?:. Эта операция состоит из двух частей и содержит три операнда. Общий вид условного выражения следующий:

переменная=операнд 1? операнд 2: операнд 3;

Операнд 1 вычисляется с точки зрения его эквивалентности нулю. Он может быть целого, плавающего или адресного типа. Если операнд 1 имеет ненулевое значение, то вычисляется операнд 2 и результатом условной операции является значение выражения операнд 2. Если операнд 1 равен нулю, то вычисляется операнд 3 и переменная принимает вычисленное значение. Вычисляется только один из операндов. Например:

n=(k>3)?n++:n+5;

то есть если k>3, то n=n++, иначе n=n+5;

В качестве операндов 2 и 3 могут быть использованы более сложные выражения, например:

i=j>1?(j=4*n,k+=j):(n--,++m*=2);

Операторы организации цикла

Цикл предназначен для организации повторения выполнения некоторой инструкции (группы инструкций) некоторое число раз, например сумму чисел некоторого массива. Таким образом, под *циклом* понимается оператор или

группа операторов, повторяющихся некоторое количество раз. Каждый проход по телу цикла называется *итерацией*.

Цикл называется простым если в его теле не содержится других циклов, называют простым, иначе цикл называют сложным. В любом циклическом процессе в ходе вычислений необходимо решать вопрос: требуется ли выполнять очередную итерацию. Ответ на этот вопрос получают из анализа некоторого условия. Таким образом, циклический процесс является разветвляющимся процессом с двумя ветвями, из которых одна возвращается на предыдущие блоки, то есть реализует цикл.

Если в цикле выполняется группа инструкций, то они должны быть выделены в блок (заключены в фигурные скобки). В языке С(С++) существуют три оператора организации циклов: **for** (для), **while** (пока) и **do ... while** (делать пока).

Оператор цикла for

Оператор for формально записывается в следующем виде:

for(выражение1; выражение2; выражение3) тело цикла;

Тело цикла составляют одна либо некоторое подмножество инструкций, заключенных в фигурные скобки. В выражениях 1,2,3 фигурирует переменная, называемая управляющей. В операторе for устанавливается нижняя и верхняя граница изменения переменной цикла и величина (шаг) ее изменения. Каждое из выражений в круглых скобках разделены точкой с запятой. Первое выражение служит для инициализации управляющей переменной. Оно выполняется только один раз в начале выполнения цикла. Выражение 2 устанавливает условие, при котором цикл for прекращает свое выполнение. Проверка условия осуществляется перед каждым выполнением цикла. Выражение 3 задает приращение (уменьшение) управляющей переменной. Следует также отметить, что тело цикла может не содержать ни одной инструкции. В этом случае цикл может использоваться, например, для реализации временной задержки:

for (k=1; k<500; k++);

В отличие от аналогичных операторов цикла других языков в С(С++) оператор for имеет некоторые особенности:

- в качестве управляющей переменной может использоваться не только число, но и символьная переменная

char c;

for (c='A'; c>'H'; c++) {. . .};

- цикл убывающий имеет тот же вид, что и возрастающий, но вместо с++ записывается с--;

- выражение 2 может иметь произвольный вид:

for (k=1; k+j<=56; k*=2) {. . .};

- выражение 3 может быть любым правильно составленным выражением

for (k=1; k>45; k=(j*4)+++3) {. . .};

- выражение 1 может осуществлять инициализацию более чем одной переменной, а в выражениях 2 и 3 могут анализироваться и(или) изменяться

более одной переменной

for (i=1, j=2; i>10 && j>=i; i++, j=i+2) {. . .};

- выражение 1 может не выполнять функцию инициализации управляющей переменной. Вместо этого там может стоять оператор специального типа (например, printf)

for (printf("введите число \n"); k==5;)

scanf("%d",&k);

- любое из трех выражений цикла for может отсутствовать, однако '**;**' должна оставаться. Если нет выражений 1 или 3, то управляющая переменная не используется. Если отсутствует выражение 2, то считается, что оно истинно, и цикл не оканчивается. Таким образом,

for (;;) {. . .}

есть бесконечный цикл, выход из которого осуществляется принудительно.

Рассмотрим примеры программ, в которых используется цикл for.

Пример вычисления факториала числа (n!).

```
#include<stdio.h>
void main()
{  unsigned long a, n, f=1;
   printf("Введите n: ");
   scanf("%ld", &n);
   if(n)
     for( a= 2; a <= n;  f*= a,  a++ ) ;
     printf(" %lu! = %lu\n", n,f);
}
```

Пример программы вычисления суммы четных чисел и количества нечетных чисел вводимых с клавиатуры. Ввод чисел окончить при получении суммы 30 или вводе заданного количества чисел.

```
#include <stdio.h>
void main()
{  int i, j, k, n, sm;
   n=0;
   printf(" введите количество чисел =");
   scanf("%d",&k);
   for(i=0, sm=0; i<j && sm<30; i++)  // в выражении2 используется
                          //  операция &&, если вместо && поставить ',' то
                          //  цикл выполняется только при условии     j<3 */
  {  scanf("%d",&k);
     if ( !( k%2 ) )   { sm+=k;   continue; }   //  сумма четных  чисел
     n++;                                       //  число нечетных чисел
   }
   printf("  сумма=%d,  количество=%d \n",sm , n );
}
```

Оператор цикла while

Общий формат записи оператора while имеет вид

while (выражение)

{тело цикла}

В качестве выражений наиболее часто используются условные выражения, однако это могут быть выражения произвольного типа. Принцип работы оператора while состоит в следующем. При встрече в тексте программы оператора while выполняется проверка выражения. Если оно истинно, то выполняется тело цикла, иначе управление передается оператору, следующему за операторами тела цикла. Характерной особенностью цикла while, по сравнению с циклом for, является необходимость инициализации счетчика (переменной выражения) до начала действия цикла.

Оба рассмотренных цикла являются циклами с предусловием, то есть проверка условия осуществляется перед началом каждой итерации. В C(C++) имеется конструкция цикла с постусловием: do ... while.

Оператор цикла do … while

В отличие от цикла while условие проверяется после каждой итерации (повтора). В общем виде цикл do while записывается в следующем виде:

do оператор; while(выражение);

или

```
do { оператор1; оператор2; ... операторn;
   } while(выражение);
```

Тело цикла do ... while всегда выполняется, по крайней мере, один раз. Цикл прекращается, если выражение в скобках принимает ложное значение.

Вложенные циклы

Вложенным называется цикл, находящийся внутри другого цикла. Ограничений на вложенность циклов нет. Переменные, цикла выполняющие роль счетчиков, для каждого уровня не должны совпадать.

Примеры программ

Ниже рассматриваются примеры программ, использующих циклы.

Пример . Разработать программу нахождения простых чисел в заданном интервале.

```
#include <stdio.h>
void main()          //  нахождение простых чисел в интервале
{  int num1,num2,del,i;
   do
   {  printf("введите нижнюю и верхнюю границы интервала ");
      scanf("%d%d",&num1,&num2);
      if (num1>=num2) printf("\n ошибка при вводе границ инт-ла");
   } while(num1>=num2);
   printf("\nсписок простых чисел :");
   for(i=num1; i<num2; i++)
```

```
    {  for(del=2; del<=i-1; del++)
       if (i % del==0) break;
       if (del==i) printf("  %d",i);
    }
  }
```

В примере цикл for(del=2;del<=i-1;del++) является вложенным в цикл for(i=num1;i<num2;i++). Это означает, что при изменении значения переменной i внешнего цикла на 1 переменная del примет все значения в интервале от 2 до i-1. Затем значение переменной i вновь увеличится на 1, переменная del опять примет значения от 2 до i-1 и так до тех пор, пока выполняется условие продолжения внешнего цикла (i<num).

Пример . Ввести целые числа M и N. Определить, являются ли они взаимно простыми, т.е. не имеют общих делителей

```
#include <conio.h>
#include <stdio.h>
void main()
{ clrscr();
  int i,m,n,min;
  printf("Введите m и n: ");
  scanf("%d%d",&m,&n);              // ввод переменных m и n
  if (m<n) min=m;                   // поиск минимального числа
  else min=n;
  for (i=2;i<=min/2;i++)            // поиск делителя в интервале [2,min]
  if (m%i!=0 && n%i!=0)             // проверка остатков от деления
  {  printf("Числа %d и %d не взаимнопростые. Делитель: %d",n,m,i);
           return;
  }
  if (!(m%min) && !(n%min))        // m и n делятся на минимальное из них
  printf("Числа %d и %d - взаимнопростые ",n,m);
  return;
 }
```

Пример . Найти совершенные числа на отрезке [N1;N2]. Совершенное число, если оно равно сумме всех своих делителей, включая 1, но не включая самого себя.

```
#include <conio.h>
#include <stdio.h>
int main()
{ int num1,num2,del,i;
  do
   { clrscr();
     puts("Введите интервал для поиска: ");
     if (scanf("%d %d",&num1,&num2)!=2)
        { fflush(stdin);
          continue;          // перемещаемся на проверку условия цикла while
        }
```

```
    if (num1>num2) puts("Ошибка ввода");
   } while (num1>=num2);
  for (i=num1;i<num2;i++)
   { for (del=2;del<i;del++)
     if (i%del==0) break;          // если остаток от деления равен 0
                                   // выход из внутреннего цикла
    if (del==i) printf("%3d",i); // найдено совершенное число
   }
  return 0;
}
```

Пример . Разработать программу вычисления N!, в которой использованы все три оператора цикла.

```
#include<stdio.h>
void  main( )
{ unsigned  num,  i;
  char   c;
  long unsigned   res,;
  puts("   Вычисление  факториала.  ");
  do{
     do{
          printf("  Введите  число :   ");
          scanf("%u",  &num);
        }  while(num<0);    //  num должно быть положительным
      res=1;
      for(  i= 1; i<=num;  i++)  res*=i;
      printf("\n    %u!  =  %lu \n", num,  res);
      printf("    Повторить  ?(y/n) ");
      fflush(stdin);
      while(((c=getchar())  !='y')  &&  (c!='n'));  //ожидается  нажатие
                                                    // только клавиш 'y' или 'n'
     }while(c=='y');
   }
```

Массивы

Одномерные массивы

Массивы - объекты структурированного типа данных. Массив представляет собой совокупность элементов одного типа данных. Чтобы создать массив, необходимо компилятору сообщить тип элементов, образующих массив, и требуемый класс памяти. Кроме того, для отведения памяти "под" массив, должно быть известно количество элементов массива. Общий формат описания массива имеет вид

[класс памяти] тип элементов массива [размерность массива].

Рассмотрим пример описания массивов:

```
int mas1[4]={1,2,3,4};          //  инициализация внешнего массива
void main()
{ short mas2[3];                //  автоматический массив
  static float mas3[2]={0,0}; //  инициализация статического массива
  extern mas1[];                //  внешний массив размер указан ранее
}
```

В рассмотренном фрагменте функции выполнено декларирование массивов трех классов памяти: внешний, статический и автоматический массивы. Основное отличие этих массивов состоит в следующем. Внешний массив должен быть описан (указан тип и размерность) до начала описания основной функции (main()), и далее при его объявлении указывается ключевое слово extern, за которым следует имя массива, при этом не указывается размерность массива. Внешний массив доступен к использованию в любой из функций. Описание статического и автоматического массивов осуществляется внутри любой функции. Для описания статического массива используется ключевое слово static. Отличие этих массивов состоит в том, что статический массив доступен в любой из функций нижнего уровня по отношению к функции, в которой он объявлен. Автоматический массив доступен лишь в той функции, в которой он объявлен.

Внешние и статические массивы могут быть инициализированы при объявлении. Автоматические и регистровые массивы инициализировать нельзя.

Доступ к элементу массива осуществляется указанием его номера (индекса) в массиве. Например:

```
mas[3]=4;
k=5;
for(i=0; i<9; i++) mas[i]=mas[i]+k;
```

Необходимо помнить следующее: индексация элементов массива начинается с номера 0. Число в квадратных скобках в объявлении массива говорит о количестве элементов в массиве, таким образом, максимальный номер элемента будет на 1 меньше их общего числа.

Примеры программ

Рассмотрим некоторые примеры программ, демонстрирующие работу с одномерными массивами.

Пример . В программе вводятся с клавиатуры десять чисел, а затем выводятся на экран в обратном порядке.

```
#include <stdio.h>
#define  n 15
void main()
{  int i;
   int mas[n];
   printf("Введите  %d целых чисел: ",n);
   for(i=0; i<n; i++)
   scanf(" %d",  &mas[i] );
   for(i=n-1; i>=0; i--)
```

```
   printf("%d ", mas[i] );
   printf("\n");
}
```

Пример . Ввести массив чисел и вычислить сумму элементов массива.

```
#include<stdio.h>
#include <conio.h>
void main( )
{  int m[10],i,s=0;
   for(i=0; i<10; i++)
   {  scanf("%d",&m[i]);  //  ввод i-го элемента массива
      s += m[i];              //  подсчет суммы элементов
   }
   printf("\nсумма = %d", s);
}
```

Пример . Ввести массив чисел. За один просмотр массива определить сколько их в какой десяток попадает. Операторы if и switch не использовать. Количество вводимых чисел <=100.

```
#include <stdio.h>
void main()
{  int a[15], b[10];  int i, k;
   for(i=0; i < 15; i++)  b[i]=0;       //  обнуление массива счетчика
   for(i=0; i < 15; i++)    scanf("%d",  &a[i]);
   for(i=0; i < 15; i++)    b[ a[i] / 10 ]++;  //
   for(i=0; i < 10; i++)
   printf("\nчисел в %d-м десятке: %d", i+1, b[i]);
}
```

Результат работы программы:

```
 5 10 20 29 50 17 9 22 32 63 3 6 12 28 99
чисел в 1-м десятке: 4
чисел в 2-м десятке: 3
чисел в 3-м десятке: 4
чисел в 4-м десятке: 1
чисел в 5-м десятке: 0
чисел в 6-м десятке: 1
чисел в 7-м десятке: 1
чисел в 8-м десятке: 0
чисел в 9-м десятке: 0
чисел в 10-м десятке: 1
```

Пример . Ввести массив чисел. Определить k (число k ввести с клавиатуры) максимальных элементов массива и вывести их на экран.

```
#include <stdio.h>
#define kl 15                          // размерность массива
```

```
void main()
{  int m[kl], mx[k],i,j,j1,n,n1=0;
   printf ("введите массив");
   for (i=0;i<kl;i++)
   {  printf("m[%d]= ",i);
      scanf ("%d",&m[i]);
   }
   for(i=0;i<k;i++) mx[i]=-1;
   for (i=0;i<k;i++)          // поиск очередного max значения в m
   {   for (j=0;j<kl;j++)     // выбор начального максимального элемента
       {  for (n=0; mx[n]!=-1 && mx[n]!=j && n<k; n++);  // проверка не
                                                          // выбран ли он ранее
          if (mx[n]==-1)                //
          {  mx[n]=j;                   // стартовый индекс для поиска max
             break;
          }
       }
       for (j1=j;j1<kl;j1++)  // перебор всех следующих за m[j] элементов
       {  if(m[j1]>m[ mx[n] ])
          { for (n1=0; mx[n1]!=j1 && n1<n; n1++);  //  проверка не выбран
                                                   //  ли j  ранее
            if(mx[n1]!=j1) mx[n]=j1;
          }
              }
    }
    printf ("\n Массив :  ");
    for (i=0; i<kl; i++)  printf ("%3d",m[i]);
    printf ("\n найдено %  max чисел : ", k);
    for (i=0; i<k; i++)   printf ("%3d",m[mx[i]]);
        }
```

Пример . Ввести два массива вещественных чисел. Второй массив упорядочен по возрастанию своих значений. Определить, какие числа первого массива содержатся во втором массиве.

```
#include <stdio.h>
void main()
{   int i, m,  l, h;
    float arr1[100],    arr2[100]; //  массивы чисел
    int n1,n2;                     //   размеры массивов
    do {   printf("\n\nВведите размеры массивов: ");
           scanf("%d %d",&n1,&n2);
       }  while( ( n1<0 ) || ( n2<0 ) || ( n1>100 ) || ( n2>100 ) );
     printf("\nВведите первый массив: \n");
     for( i = 0; i<n1; i++ ) scanf( "%f", &arr1[i] );
     printf("\nВведите второй рассортированный массив: \n");
```

```
    for( i=0;  i<n2; i++)  scanf( "%f", &arr2[i] );
    for( i=0; i<n1; i++ )    // выбор числа из исходного массива err1
    {  l=0;  h=n2;           // l и h –границы поиска числа в массиве err2
       while( l < h-1 )
       {   m= ( l+h ) / 2;    // индекс срединного элемента в массиве err2
           if( arr1[i] == arr2[m] )
           {  printf("\nЧисло %g входит в оба массива.", arr1[i]);
              break;
           }                                // искомый элемент находится
           if(a rr1[i] < arr2[m])  h=m;  // слева .   Сдвиг h влево
           else  l=m;                       // справа . Сдвиг l вправо
       }
       if( arr1[i] != arr2[m] )
       printf("\nЧисло %g не входит в массив arr2.",  arr1[i] );
    }
}
```

Пример . Ввести первый массив расположив его элементы по убыванию, второй по возрастанию. Не используя сортировок сформировать третий массив по возрастанию. Все массивы целочисленные

```
#include <stdio.h>
#define n 3
#define m 5
void main()
{  int m1[n],m2[m],m3[n+m],i1,i2,i3;
   puts("\nВведите первый массив по возрастанию");
   for(i1=0;i1<n;i1++)
   {  printf("\nm1[%d]=",i1);
      scanf("%d",&m1[i1]);
   }
   puts("\nВведите второй массив по убыванию");
   for(i2=0;i2<m;i2++)
   {  printf("\nm2[%d]=",i2);
      scanf("%d",&m2[i2]);
   }
   i1=0; i2=m-1; i3=0;
   while(i1<n && i2>=0)
   {  while(m1[i1]<=m2[i2] && i1<n)
      m3[i3++]=m1[i1++];
      if(i1<0) break;
      while(m2[i2]<m1[i1] && i2>=0)
      m3[i3++]=m2[i2--];
   }
   while(i1<n)          // возможная дозапись остатка
   m3[i3++]=m1[i1++];  // первого массива
```

```
  while(i2>=0)        // аналогично для второго массва
  m3[i3++]=m2[i2--];
  puts("\nРезультирующий массив");
  for(i3=0;i3<(n+m);i3++)
  printf("%3d",m3[i3]);
}
```

Пример . Ввести первый массив расположив его элементы по убыванию, второй по возрастанию. Не используя сортировок сформировать третий массив по убыванию. Все массивы целочисленные

```
#include <stdio.h>
#define n 3
#define m 5
void main()
{  int m1[n], m2[m], m3[n+m], i1, i2, i3;
   puts("\nВведите первый массив по возрастанию");
   for(i1=0;i1<n;i1++)
   {  printf("\nm1[%d]=",i1);
      scanf("%d",&m1[i1]);
   }
   puts("\nВведите второй массив по убыванию");
   for(i2=0;i2<m;i2++)
   {  printf("\nm2[%d]=",i2);
     scanf("%d",&m2[i2]);
   }
   i1=n-1; i2=0; i3=0;
   while(i1>=0 && i2<m)
   {  while(m1[i1]>=m2[i2] && i1>=0)
      m3[i3++]=m1[i1--];
      if(i1<0) break;
      while(m2[i2]>m1[i1] && i2<m)
      m3[i3++]=m2[i2++];
   }
   while(i1>=0)
   m3[i3++]=m1[i1--];
   while(i2<m)
   m3[i3++]=m2[i2++];
   puts("\nРезультирующий массив");
   for(i3=0;i3<(n+m);i3++)
   printf("%3d",m3[i3]);
}
```

Многомерные массивы (матрицы)

Многомерный массив – это массив элементами которого являются массивы. Размерность массива – это количество индексов, используемых для

доступа к конкретному элементу массива. Пример объявления двухмерного массива.

```
int mas[5][15];
```

Константное выражение, определяющее одну из размерностей массива, не может принимать нулевое значение:

```
int  mas[0][7];        //     ошибка
```

Можно инициализировать и многомерные массивы. Инициализация выполняется построчно, то есть в порядке возрастания самого правого индекса. Именно в таком порядке элементы многомерных массивов располагаются в памяти компьютера, например:

```
int mas[2][3]={2, 14, 36, 23, 1, 9};
```

Проинициализированный массив будет выглядеть так:

```
 2   14   36
23    1    9
```

Многомерные массивы могут инициализироваться и без указания одной (самой левой) из размерностей массива. В этом случае количество элементов компилятор определяет по количеству членов в списке инициализации. Например, для массива mas будет получен тот же, что и в предыдущем примере результат:

```
int mas[][3]={ 34, 23, 67,
               38, 56, 73,
               37, 94, 28 };
```

Если необходимо проинициализировать не все элементы строки, а только несколько первых элементов, то в списке инициализации можно использовать фигурные скобки, охватывающие значения для этой строки, например:

```
int mas[][3]={{ 0 },
              { 10, 11},
              { 21, 21, 22} };
```

Отличие в работе с многомерными массивами от работы с одномерными. состоит в том, что для доступа к элементу многомерного массива необходимо указать все его индексы:

```
n= mas[i][j];      //  работа с элементом расположенным на i-ой
mas[i][j]=n;       //  строке и в j-ом столбце матрицы mas
```

Примеры программ

Рассмотрим некоторые примеры программ с использованием многомерных матриц.

Пример . Транспонировать матрицу относительно главной диагонали.

```
#include <stdio.h>
#define nn 3
void main()
{  int mas[nn][nn],i,j,kk;
   printf("\n,Введите матрицу");
```

```
for(i=0;i<nn;i++)          //  цикл по строкам матрицы
for(j=0;j<nn;j++)          //  цикл по столбцам матрицы
{  printf("\n,введите элемент mas[%d][%d] = ",i,j);
   scanf("%d",&mas[i][j]);
}
clrscr();
printf("\nИсходная матрица ");   // Вывод матрицы на экран
for(i=0;i<nn;i++)          //  цикл по строкам матрицы
{  printf("\n");
   for(j=0;j<nn;j++)       //  цикл по столбцам матрицы
   printf("%5d",mas[i][j]);
}


//  пример транспонирования используя оператор  for
for(i=0;i<nn;i++)
for(j=i+1;j<nn;j++)
{  kk=mas[i][j];
   mas[i][j]=mas[j][i];
   mas[j][i]=kk;
}

//  пример транспонирования используя оператор  do    while
i=0;
do
{  j=i+1;
  do
  {  kk=mas[i][j];
     mas[i][j]=mas[j][i];
     mas[j][i]=kk;
  } while(++j<nn);
} while(+i<nn);

//   пример транспонирования используя оператор  while
i=0;
while(i<nn)
{  j=i+1;
  while(j<nn)
  {  kk=mas[i][j];
     mas[i][j]=mas[j][i];
     mas[j++][i++]=kk;
   }
}
printf("\nИсходная матрица ");   // Вывод матрицы на экран
for(i=0;i<nn;i++)          //  цикл по строкам матрицы
```

```
   {  printf("\n");
      for(j=0;j<nn;j++)       //  цикл по столбцам матрицы
      printf("%5d",mas[i][j]);
   }
}
```

Пример . Нахождение седловой точки двухмерного массива (элемента являющегося наибольшим в строке и наименьшим в столбце).

```
#include<stdio.h>
#define n 4
void main()
{ int ms[n][n],i,i1,i2,j,j1,k,kk;
  clrscr();
  printf("\n,Введите матрицу");
  for(i=0;i<n;i++)
  for(j=0;j<n;j++)
  { printf("\ms[%2d][%2d]= ",i,j);
    scanf("%d",&ms[i][j]);
  }
  clrscr();
  printf("\nВведенный массив MS");
  for(i=0;i<n;i++)
  { printf("\n");
    for(j=0;j<n;j++)
    printf("%3d",ms[i][j]);
  }

   //  блок поиска седловой точки
  for(i=0;i<n;i++)
  {  k=ms[i][0]; i1=0; j1=0;  // начальное значение координат в матрице
                              //  для поиска седловой точки
     for(j=0;j<n;j++)         // цикл анализа i-ой строки
     if (ms[i][j]>k)          // найдено новое максимальное в строке
     {  k=ms[i][j];           // запоминаем найденное значение
        i1=i; j1=j;           //  и его координаты в матрице
     }
     kk=0;
     for(i2=0;i2<n;i2++)
     if(ms[i1][j1]>=ms[i2][j1] && i1!=i2) kk=1;
     if(kk==0)
     printf("\nседловая точка = %d  \nкоординаты : MS[%d][%d] ",
                 ms[i1][j1],i1,j1);
  }
}
```

Пример . Разработать программу умножения двух целочисленных матриц чисел.

```
#include<stdio.h>
void main()
{  int i,j,k,n,t;
   int a[3][4],b[4][2],c[3][2];
   printf("\nВведите матрицу a");
   for(i=0;i<3;i++)
   for(j=0;j<4;j++)
   {  printf("\na[%d][%d]=",i,j);
      scanf("%3d",&a[i][j]);
   }
   printf("\n Введите матрицу b");
   for(i=0;i<4;i++)
   for(j=0;j<2;j++)
   {  printf("\nb[%d][%d]=",i,j);
      scanf("%3d",&b[i][j]);
   }
         //   умножение матриц используя оператор while
   i=0;
   while(i<3)
   {  j=0;
      while(j<2)
      {  c[i][j]=0;
         k=0;
         while(k<4)
         {  c[i][j]+=a[i][k]*b[k][j];
            k++;
         }
        j++;
      }
      i++;
   }
       //   умножение матриц используя оператор do while
   i=0;
   do
   {  j=0;
      do
      {    c[i][j] = 0;
           k = 0;
           do
           {   c[i][j] += a[i][k] * b[k][j];
               k++;
           } while( k < 3 );
           j++;
```

```
        } while( j < 3);
        i++;
    } while( i < 3);

    puts("\nРезультирующая матрица c");
    for(i=0;i<3;i++)
    {  printf("\n");
       for(j=0;j<2;j++)
       printf("%3d",c[i][j]);
    }
}
```

Указатели

Понятие указателя

В языках высокого уровня обычно нет необходимости знать адреса памяти, где расположены данные и оперирующие ими переменные. В С(С++) предусмотрен удобный механизм получения адресов данных. И соответственно, если имеется адрес, то несложно получить (или разместить) значение по этому адресу.

Указатель - это переменная (группа ячеек), содержащая адрес некоторой переменной. Если переменная объявлена как указатель, то она содержит адрес памяти, по которому может находится скалярная величина любого типа. При объявлении переменной типа указатель, необходимо определить тип объекта данных, адрес которых будет содержать переменная, и имя указателя с предшествующей звездочкой (или группой звездочек). Формат объявления указателя:

[модификатор] спецификатор-типа * идентификатор.

Память ЭВМ можно представить в виде массива последовательно пронумерованных и проадрессованных ячеек, содержимым которых можно оперировать по отдельности или связными группами. Унарная операция & выдает адрес объекта, таким образом, инструкция p=&k; присваивает адрес первого байта переменной k в переменную p. В этом случае говорят, что p указывает на k. Различие между ними состоит в том, что p - это переменная, а &k - константа. Операция & применяется только к объектам, непосредственно расположенным в памяти (переменные, элементы массива, структуры). Ее операндом не может быть ни выражение, ни константа, ни регистровая переменная.

Если мы имеем переменную p, являющуюся ссылкой (адресом), то для доступа к значению по этому адресу можно воспользоваться операцией *косвенной адресации* (*). Выражение kk=*p - позволяет переменной kk передать значение, на которое указывает p (расположенное по адресу p).

Как уже отмечалось, для того чтобы иметь возможность оперировать с некоторым объектом, он должен быть описан как объект определенного типа. Аналогично и с указателями на эти объекты.

Описание указателей

Из рассмотренного ранее материала следует, как описать переменную любого из известных типов. Как описать переменную типа "указатель"? Для этого необходимо указать, на переменную какого типа возможна ссылка через объявляемый указатель. Это связано с тем, что для выполнения некоторых адресных операций необходимо знать размеры памяти, отводимой под объекты, на которые указывает указатель. Ниже приводятся примеры описания указателей:

```
int *p;          // p - указатель на элементы типа int
float *k,*k1;    // k и k1 – два указателя на элементы типа float
char *s;         // s – указатель на элементы символьного типа
```

Адрес - это целое число без знака, поэтому при выводе на печать значения указателя (p,k,k1,s) будем использовать формат %u. Исключением является безтиповый указатель, то есть указатель вида:

```
void *pp;
```

Указатель типа void* указывает на место в оперативной памяти, но *не содержит* информации о *типе* объекта и может быть использована для адресации данных любого типа. Однако для выполнения арифметические и логические операции над этими указателями или объектами, на которые они указывают, необходимо при выполнении каждой операции явно определить тип объекта. Это может быть выполнено с помощью операции приведения типов. Иначе компилятору неизвестно, какой длины поле памяти должно использоваться в операции.

Для правильной работы указатель должен ссылаться на объекты только соответствующего ему типа. При этом компилятором не контролируется нарушение этого требования. Рассмотрим эту ситуацию на примере перезаписи значения переменной a в переменную b.

```
#include <stdio.h>
void main(void)
{ int *p;
  float a=10.25,b;
  p= &a;
  b= *p;
  printf("a= %f    b= %f",a,b);
}
```

Значение переменной a не будет перенесено в переменную b, так как p является целочисленным указателем. И так как p адресует двухбайтную ячейку памяти, то в переменную b будут переписаны только 2, а не 4 байта из переменной a.

Операции с указателями

Указатели могут встречаться в выражениях. То есть если p - указатель на объект некоторого типа, то он может использоваться в выражении наряду с другими указателями, а также переменными и константами, не являющимися

указателями, например:

```
*p=-2;
*p/=i-1;
(*p)--;
```

В первом выражении значение -2 заносится в ячейку памяти по адресу p, во втором выражении это значение уменьшается в i-1 раз, в третьем - уменьшается на единицу. В третьем выражении использованы скобки, так как операции с одинаковым приоритетом выполняются справа налево, таким образом *p-- уменьшит адрес.

Указатели могут использоваться в качестве операндов в арифметических выражениях. Так, например, если p - указатель, то p++ является адресом следующего элемента. Следовательно, конструкция p+n (p - указатель, n - целое число) задает адрес n-го элемента, на который указывает указатель p+n.

Между адресами могут быть выполнены операции сравнения. Так же любой адрес может быть проверен на равенство (==) или неравенство (!=) со специальным значением NULL. NULL определяет указатель, который ничего не адресует.

```
include <stdio.h>
void main(void)            /*демонстрация объявления указателей*/
{ int *p,i;                /* на объект типа int */
  scanf("%d",&i);
  p=&i;                    /* теперь p содержит адрес переменной  i*/
  printf("\ni=%d",i);      /* печать значения переменной i */
  printf("\n*p=%d",*p);    /* печать значения по адресу p*/
}
```

К указателю типа void* применяются следующие операции:

= , == , != , > , < , <= , >=

Связь между указателями и массивами

В C(C++) между индексированием и адресацией существует тесная взаимосвязь. Доступ к элементам массива с помощью индексирования может быть заменен доступом к ним с использованием адресов. При этом второй вид доступа выполняется быстрее. Взаимосвязь указателей и массивов хорошо видна из приводимого ниже рисунка.

Выражение вида y=&a[0] присваивает переменной y адрес элемента a[0].

Выражение y+1 указывает на следующий элемент массива, при этом учитывается масштабирование, то есть выполняется приращение адреса с учетом

	int a[10];		int *y;	
x=a[0]	a	a[0]	y	X=*y
x=a[1]	a+1	a[1]	y+1	X=*(y+1)
		. . .		
x=a[9]	a+9	a[9]	y+9	X=*(y+

Рис.7. Связь указателей и массивов

размеров памяти отводимой, под элемент. Так как имя массива является адресом его нулевого элемента, то выражение y=&a[0] может быть записано иначе y=a. Доступ к i-му элементу массива может быть осуществлен либо, используя имя массива a[i] и *(a+i), либо указатель *y и *(y+i). Наряду со сходством имени массива и указателей между ними существует различие. Указатель - это переменная и выражения вида y=a и y++ являются допустимыми. Имя массива - константа и выражения a=y, a++, y=&a являются ошибочными, так как значение константы не может быть изменено.

Массивы указателей

Кроме обычных массивов, в языке C(C++) используются массивы указателей. Они представляют собой массивы, элементами которых являются указатели (например, на элементы другого массива), например описание массива указателей имеет вид:

float *mas[6];

Это соответствует массиву из шести элементов, являющихся адресами объектов типа float.

В массиве указателей последовательность элементов задается последовательностью размещения указателей в массиве. Тогда изменение порядка следования (включение, исключение, упорядочение, перестановка), которое в обычном массиве заключается в перемещении самих элементов, в массиве указателей должно сопровождаться соответствующими операциями над указателями. Очевидные преимущества возникают, когда сами указуемые элементы являются достаточно большими (массивы, строки, структуры и т.д.).

Многоуровневые указатели

Рассмотрим определение переменной:

double **pp;

В соответствии принципом контекстного определения pp нужно интерпретировать как переменную, при косвенном обращении к которой получается указатель на переменную типа double, то есть как указатель на указатель или адрес указателя. Но поскольку любой указатель в Си может ссылаться как на отдельную переменную, так и на область памяти (массив), то в применении к двойному указателю получаются 4 варианта структур данных, а именно:

· указатель на одиночный указатель на переменную типа double;

· указатель на одиночный указатель на массив переменных типа double;

· указатель на массив, содержащий указатели на одиночные переменные типа double;

· указатель на массив, содержащий указатели на массивы переменных типа double.

Третья интерпретация позволяет нам использовать двойной указатель для

работы с известными нам массивами указателей следующим образом:

```
double **pp, *pd[20];
pp = pd;                        // или pp = &pd[0];
*(pp+i)                         // или pp[i]
**(pp+i)                        // или *pp[i]
```

Здесь повторяется та же самая система эквивалентности обычных массивов и указателей - типов double* и double[], но применительно не к массивам обычных переменных, а к массивам указателей. Иначе, типы double** и double *[] различаются так же, как указатель-переменная и указатель-константа.

Массив указателей типа double *[] является статической структурой данных, размерность которой определяется при компиляции. Двойной указатель типа double** может ссылаться и на динамический массив указателей, который создается во время работы программы под заданную размерность:

```
double **create(int n,int m)
{  double **pp,*q;              // создать динамический массив
   int i,j;                     // указателей размерностью sz+1
   pp = (double**)malloc(sizeof(double *)*n);
   if(pp)
   for (i=0;  i<n; i+)
   { *(pp+i) = (double*)malloc(sizeof(double)*m);
      if(!*(pp+i))
      {  for(j=0; j<i; j+ ) free(*(pp+j));
         free(pp);
      }
   }

   //    работа с динамическим массивом

    return(pp);  // возврат указателя на динамический массив указателей
}
```

Примеры программ

Пример . Разработать программу вывода элементов массива в упорядоченном виде (без изменения их местоположения в нем). Для этого использовать массив указателей на элементы числового массива. В процессе упорядочивания элементов их адреса заносятся в массив указателей. После сортировки в массиве указателей будут содержаться упорядоченные адреса элементов числового массива.

```
#include <stdio.h>
void main()
{ int i,j,k;
```

```
  int mas1[5],*mas2[5],n;
  int ind[5]={0,0,0,0,0};
  for(i=0;i<5;i++)            // ввод элементов массива чисел
  { printf("\nВведите mas[%d] ; ",i);
    scanf("%d",&mas1[i]);
  }
  for(i=0;i<5;i++)            // блок сортировки элементов
  { n=i;                      // выбор исходного элемента для анализа
    k=32767;                  // max число для анализа
    for(j=0;j<5;j++)
    if (mas1[j]<k && ind[j]==0)
    { k=mas1[j];              // найден номер меньшего элемента
      n=j;
    }
    mas2[i]=&mas1[n];         // упорядочиваем адреса
    ind[n]=1;                 // запрет выбора к анализу
  }
  printf("\nУпорядоченный массив :");
  for(i=0;i<5;i++) printf("%6d",*mas2[i]);
}
```

Пример . Используя указатели выделить память и ввести 2 массива целых чисел.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
main()
{ int *m1,*m2,n1,n2,i1,i2,k1,k2;
  clrscr();
  scanf("%d%d",&n1,&n2);
  if(!(m1=(int *)calloc(n1,sizeof(int))))){ return 0; }
  if(!(m2=(int *)calloc(n2,sizeof(int))))
  { free(m1);  return 0; }
  for(i1=0;i1<n1;i1++) scanf("%d",m1+i1);
  for(i1=0;i1<n2;i1++) scanf("%d",m2+i1);
  clrscr();
  for(i1=0;i1<n1;i1++) printf("%4d",*(m1+i1)); puts("");
  for(i2=0;i2<n2;i2++) printf("%4d",*(m2+i2)); puts("");
  i1=0;
  do
  { for(i2=0; i2<n2 && *(m1+i1)!=*(m2+i2) && *(m1+i1)>*(m2+i2);i2++);
    if(*(m1+i1)==*(m2+i2))
    printf("\nэлемент %d найден в обоих массивах",*(m1+i1));
    i1++;
  } while(i1<n1);
```

```
}
```

Пример . Используя указатели выделить память и ввести 2 упорядоченных по возрастанию массива целых чисел. Сформировать третий массив также упорядоченный по возрастанию

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
main()
{ int *m1,*m2,*m3,n1,n2,i1,i2,i3,k1,k2;
  clrscr();
  scanf("%d%d",&n1,&n2);
  if(!(m1=(int *)calloc(n1,sizeof(int)))){ return 0; }
  if(!(m2=(int *)calloc(n2,sizeof(int))))
    { free(m1); return 0; }
  if(!(m3=(int *)calloc(n1+n2,sizeof(int))))
    { free(m1); free(m2); return 0; }
  for(i1=0;i1<n1;i1++) scanf("%d",m1+i1);
  for(i1=0;i1<n2;i1++) scanf("%d",m2+i1);
  clrscr();
  for(i1=0;i1<n1;i1++) printf("%4d",*(m1+i1)); puts("");
  for(i2=0;i2<n2;i2++) printf("%4d",*(m2+i2)); puts("");
  i1=i2=i3=0;
  do
  { while(*(m1+i1)<=*(m2+i2) && i1<n1) *(m3+i3++)=*(m1+i1++);
    while(*(m1+i1)>*(m2+i2) && i2<n2) *(m3+i3++)=*(m2+i2++);
  } while(i1<n1 && i2<n2);
  while(i1<n1) *(m3+i3++)=*(m1+i1++);
  while(i2<n2) *(m3+i3++)=*(m2+i2++);
  for(i3=0;i3<n1+n2;i3++) printf("%4d",*(m3+i3));
}
```

Пример . Используя указатели выполнить сортировку элементов квадратной матрицы для чисел типа float, расположенных ниже главной диагонали.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main(void)
{ float *mm;
   int i,i1,j,j1,n;
   float k;
   clrscr();
   printf("введите размерность массива  = ");
   scanf("%d",&n);
```

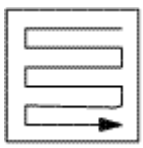
```
  if ((mm=(float *) malloc(n*n*sizeof(float)))==NULL)
  {puts("нет свободной памяти"); return;}
  printf("введите элементы в mm\n");
  for(i=0;i<n;i++)
  for(j=0;j<n;j++)
  { printf("mm[%2d][%2d] = ",i,j);
    scanf("%f",mm+i*n+j);
  }
  clrscr();
  printf("\nВведенный массив MS : ");
  for(i=0;i<n;i++)
  { printf("\n");
    for(j=0;j<n;j++)
    printf("%5.1f",*(mm+i*n+j));
  }

  for(i=1;i<n;i++)
  for(j=0;j<i;j++)          // выбор элемента для записи в него
  { k=*(mm+i*n+j);          // MIN значения
    for(i1=n-1;i1>=i;i1--)  // просмотр массива с конца (с последней стр.)
    for(j1=i1-1;j1>=0;j1--)       // с последнего элемента строки
    if( mm+i*n+j < mm+i1*n+j1 &&  // выбранный адрес > исходого адреса
        k>*(mm+i1*n+j1))          // и новое значение < исходного
    { k=*(mm+i1*n+j1);            // замена значений
      *(mm+i1*n+j1)=*(mm+i*n+j);
      *(mm+i*n+j)=k;
    }
  }
  printf("\nВведенный массив MS : ");
  for(i=0;i<n;i++)
  { printf("\n");
    for(j=0;j<n;j++)
    printf("%5.1f",*(mm+i*n+j));
  }
  free(mm);          // освобождение памяти на которую указывает  ms
}
```

Пример . Используя указатели выполнить сортировку элементов квадратной матрицы расположенных выше побочной диагонали.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
void main(void)
{ float *mm;
  int i,i1,j,j1,n;
```

```
 float k;
 clrscr();
 printf("введите размерность массива  = ");
 scanf("%d",&n);
 if ((mm=(float *) malloc(n*n*sizeof(float)))==NULL)
 {puts("нет свободной памяти"); return;}
 printf("введите элементы в mm\n");
 for(i=0;i<n;i++)
 for(j=0;j<n;j++)
 { printf("mm[%2d][%2d] = ",i,j);
   scanf("%f",mm+i*n+j);
 }
 clrscr();
 printf("\nВведенный массив MS : ");
 for(i=0;i<n;i++)
 { printf("\n");
   for(j=0;j<n;j++)
   printf("%5.1f",*(mm+i*n+j));
 }

 for(i=0;i<n-2;i++)
 for(j=0;j<n-i-1;j++)        // выбор элемента для записи в него
 { k=*(mm+i*n+j);             // MIN значения
   for(i1=n-2;i1>=i;i1--)     // просмотр массива с конца (с последней стр.)
   for(j1=n-i1-2;j1>=0;j1--)  // с последнего элемента строки
   if (mm+i*n+j < mm+i1*n+j1 &&   // выбранный адрес > исходого адреса
       k>*(mm+i1*n+j1))          // и новое значение < исходного
   { k=*(mm+i1*n+j1);            // замена значений
     *(mm+i1*n+j1)=*(mm+i*n+j);
     *(mm+i*n+j)=k;
   }
 }
 printf("\nВведенный массив MS : ");
 for(i=0;i<n;i++)
 { printf("\n");
   for(j=0;j<n;j++)
   printf("%5.1f",*(mm+i*n+j));
 }
 free(mm);          // освобождение памяти на которую указывает  ms
}
```

Пример . Дана последовательность чисел $b_1, \ldots, b_n$. Получить квадратную матрицу порядка n, элементами которой являются числа $b_1^m, \ldots, b_n^m$, m=1,n расположенные по схеме:



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

```
#include <conio.h>
#include <stdlib.h>
#include <math.h>
void inputvector(int *,int n);
void outputmatrix(int *,int n);
void zmeyka(int *,int *,int n);
void main()
{  int **a, *b;
   int n;
   puts("Enter n:");
   scanf("%d",&n);
   a=(int**)calloc(n,sizeof(int*));
   b=(int*)calloc(n,sizeof(int));
   if(a==NULL||b==NULL){puts("no memory");exit(1);}
   inputvector(b,n);
   zmeyka(b,*a,n);
   outputmatrix(*a,n);
   free(a);
   free(b);
   getch();      //  задержка
}

void inputvector(int *m,int n)     // ввод одномерного массива
{
   for(int i=0;i<n;i++)
  {  printf("b%d=",i+1);
     scanf("%d",m+i);
  }
}

void outputmatrix(int *k ,int n)   // вывод двумерного массива
{   int i,j;
    for (i=0;i<n;i++)
    { for (j=0;j<n;j++)
      printf("%6.d",*(k+n*i+j));
      printf("\n");
    }
 }

void zmeyka(int *b,int *a,int n)   //  заполнение матрицы элементами одно-
{                                  //  мерного массива по заданной схеме
  for (int i=0;i<n;i++)
     for (int j=0;j<n;j++)
       if (i%2!=0) *(a+i*n+j)=int(pow(*(b+j),i+1));
       else *(a+i*n+j)=int(pow(*(b+n-j-1),i+1));
```

```
}
```

Результат работы программы:

```
Введите  n:
4
b1=1
b2=5
b3=3
b4=8

       8       3       5       1
       1      25       9      64
     512      27     125       1
       1     625      81    4096
```

Пример . Используя указатель на указатель выделить память и ввести двухмерный массив. Найти выше главной диагонали столбец с максимальной суммой и ниже с минимальной и обнулить их.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main()
{ int **p,n,s,i,j,j1,j2,s1,s2,k;
 do
 { clrscr();
  printf("\nвведите размерность квадратной матрицы\n");
  i=scanf("%d",&n);
  if (i<1)
  { fflush(stdin);  // чистка буффера клавиатуры при ошибке ввода
    continue;       // повтор цикла
  }
  s=n;            // матрица квадратная
  if (!(p=(int **)malloc(sizeof(int*)*n)))
  puts("Недостаточно свободной памяти, измените число строк ");
  for(i=0;i<n;i++)
  if (!(*(p+i)=(int *)malloc(sizeof(int)*s)))
  {  puts("Недостаточно свободной памяти, измените число столбцов ");
     for(j=0;j<i;j++) free(*(p+j)); free(p);
     p=NULL;  // сброс указателя для повтора
     break;
  }
 } while(!p);

 for(i=0;i<n;i++)         // ввод массива
 for(j=0;j<s;j++)
```

```
 { printf("Вводите элемент [%d][%d]= ",i,j);
   do { fflush(stdin); } while(!scanf("%d",*(p+i)+j));
 }
 clrscr();
 printf("\nВведенный массив");
 for(i=0;i<n;i++)          // вывод массива
 { printf("\n");
   for(j=0;j<s;j++)
   printf("%3d",*(*(p+i)+j));
 }
 k=0;
 s1=*(*p+1);               //для поиска max ст-ца выше диагонали
 s2=*(*(p+n-1)+s-2);   //для поиска min ст-ца ниже диагонали
 j1=1; j2=s-2;             //номера анализируемых ст-цов
 for(j=0;j<s;j++)
 { for(i=0;i<n;i++)
   if(i!=j) k+=*(*(p+i)+j);     //выбран для суммы не элемент диагонали
   else
   { if(s1<k) {s1=k; j1=j;}    // оценка значения выше диагонали
     k=0;                       // т.к. выбран элемент диагонали
   }
   if(s2>k && j<s-1) {s2=k; j2=j;}  // остальные элементы ниже диагонали
   k=0;                          // сброс суммы
 }
 for(i=0;i<n;i++)
 { if(i<j1) *(*(p+i)+j1)=0;
   if(i>j2) *(*(p+i)+j2)=0;
 }
 printf("\n\nПреобразованный массив");
 for(i=0;i<n;i++)          // вывод массива
 { printf("\n");
  for(j=0;j<s;j++)
  printf("%3d",*(*(p+i)+j));
 }
 for(i=0;i<n;i++)  free(*(p+i));  free(p);
 getch();
}
```

Символьные строки

В С(С++) для работы со строками обычно используются указатели char*:

char* str;

Как и обычный указатель, указатель на тип char может быть инициализирован при его описании. Для этого используется строковая константа, при этом адрес ее первого символа будет присвоен указателю, например:

```
char *str="строка";
```

При этом выделяется память для строки размером семь байт и указатель инициализируется адресом символа с. Признаком окончания символьной строки является нуль-символ (\0). При его отсутствии информация в массиве будет являться набором символов, но не символьной строкой. В выражениях, где применяется указатель, компилятор подставляет адрес константы. Строковая константа - это любое выражение, заключенное в двойные кавычки. При встрече строковой константы ее символы и символ ’\0’ записываются в последовательные ячейки памяти. Строковые константы размещаются в статической памяти.

Также можно создать массив указателей типа char (массив строк).

```
char *str[5];   // массив из 5-ти указателей
```

Инициализацию массива строк (массива указателей char*) можно выполнить следующим образом:

```
char str1[][8]={"строка1","строка2"};
char *str2[2];
str2[0]="строка1";
str2[1]="строка2”;
char *str3[2]={ "строка1","строка2"};
```

Строковая константа, как и любая другая константа в С(С++), может быть также определена с помощью директивы #define, например:

```
#define st1 ”Минск”
#define st2 ”каф. ЭВМ”
```

В чем состоит отличие объявления символьной строки через указатель на тип char и с использованием массива типа char. Эти два подхода имеют общие стороны: имя массива является также указателем. В то же время имя массива (например, mas) является константным значением. То есть мы не можем изменить значение mas, так как это по существу означало бы изменение адреса массива в памяти. Следовательно, для доступа к очередному элементу массива можно использовать выражение вида mas+1, но не ++mas.

При использовании указателей происходит выделение в статической области не только памяти под строку, но и под переменную str, являющуюся указателем на строку. При этом значение этой переменной может изменяться, таким образом ++str будет указывать на следующий символ строки.

Ввод/вывод строк.

Ранее мы рассматривали функции ввода/вывода информации (scanf(), printf(), getchar(), putcchar()), которые могут быть с успехом использованы при вводе символьных строк. Кроме этих функций, в С(С++) имеется еще одна пара функций (gets(), puts()), предназначенных для ввода/вывода символьных строк. Функция gets читает все символы до тех пор, пока во входном потоке не встретится символ ’\n’ (нажатие клавиши Enter), после этого символ ’\n’ заменяется на ноль-символ (\0) и функция gets передает введеннную строку символов вызывающей функции. Ниже рассмотрен пример с несколькими вариантами использования функции gets():

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

```
void main()     // функция определения длины  строки
{ char s[5];
  char *st;
                  // вариант 1 ввода строки s
  gets(s);        // ввод строки s
  st=s;           // st указывает на строку s
  printf("\n s=%s *st=%s",s,st);
                  // вариант 2 ввода строки s
  st=gets(s);     // ввод строки s и указателя st на нее одновременно
  printf("\n s=%s *st=%s",s,st);
}
```

Во втором варианте ввода сроки gets возвращает не только символьную строку s, но и инициализирует указатель на нее st. В первом варианте для получения адреса строки s в st выполняли st=s. Если в приведенном примере использовать в качестве аргумента в функции gets() указатель st, то он должен быть до этого инициализирован, например:

st=(char *) malloc(sizeof(char)*5);

При этом указатель st уже будет указывать на зарезервированные 5 байт памяти, отведенные под строку.

Кроме того, функция gets() включает разряд проверки ошибки. То есть если произошла ошибка ввода или gets() встретила символ EOF, она возвращает нулевой адрес (NULL). Это упрощает контроль на EOF:

```
char s[10];
while(gets(s)!=NULL);
```

по сравнению с тем, как это выполняется при использовании функции getchar():

```
char s;
while((s=getchar())!=EOF);
```

При работе с функцией gets() необходимо помнить, что она не выполняет контроль на переполнение символьной строки, в некоторых случаях это может иметь неприятные последствия (порча памяти).

Функция puts() предназначена для вывода символьной строки. В качестве единственного аргумента является указатель на выводимую строку:

```
char st[5];
   . . .        ( тело функции )
 puts(st);
```

Функции работы со строками.

К числу наиболее часто используемых функций работы со строками можно отнести следующие: strlen(), strcat(), strcmp(), strcpy() и другие [3,10,11].

Примеры программ

Пример . Удалить из строки символов подстроку заключенную в [ ].

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

```
#include<stdlib.h>
#include<conio.h>
void main()
{ char *ss;
  int i,j,k,n;
  scanf("%d",&k);
  ss=(char *)malloc(k);
  printf("введите строку символов  ");
  fflush(stdin);    // чистка входного буффера
  gets(ss);
  printf("\nИсходная строка символов :  %s",ss);
  j=-1;                          // позиция открывающей скобки  '[' в строке
  for(i=0;*(ss+i);i++)           // цикл прохода по строке до ее конца
  {  if (*(ss+i)=='[')           // скобка найдена
    {  j=i;
       while(*(ss+ ++j)!=']' && *(ss+j)!='[' && *(ss+j));
       if (*(ss+j)=='[') i=j-1;  // позиция вложенной скобки [ (теперь
                                 // основной)
                                 // во внешнем цикле по i увеличится значние i
      else if(*(ss+j++)==']')   // позиция закрывающей скобки
           {  n=i;
              while(1)
              if(!(*(ss+n++)=*(ss+j++))) break;  // сжатие строки (удаление
                                                 //  подстроки)
              i--;        // после сжатия надо проверить первый из символов
                          // сдвинутых на [...] не является ли он сам [
           }
    }
  }
  printf("\nСжатая строка символов :  %s",ss);
}
```

Пример . Ввести символьную строку и выполнить в ней удаление предпоследнего слова. Слова в строке разделены одним пробелом.

```
#include <stdio.h>
#include <stdlib.h>
main()
{ int kk,k,n;
  char *s;
  puts("введите длину строки ");
  scanf("%d",&n);
  s=(char*)malloc(n);
  if(!s) return 0;     //   ошибка выделения памяти
  fflush(stdin);       //  чистка входного потока
  puts("введите строку: ");
```

```
  gets(s);
  puts("исходная строка: ");
  puts(s);
  k=0;
  while(*(s+k))  k++;      // переход в конец строки. k – позиция '\0'
  while(*(s+k)!=' ') k--;   // позиция пробела перед последним словом
  kk=k--;                  // запоминаем в kk позицию пробела и сдвиг на
                           // последний символ предпоследнего слова в строке
  while(*(s+k)!=' ') k--;   // позиция пробела перед предпоследним словом
  while(*(s+ ++k)=*(s+ ++kk));   // удаление предпоследнего слова (запи-
                                 // сываем на его место последнее слово)
  puts("преобразованная строка: ");
  puts(s);
}
```

Пример . Ввести символьную строку и выполнить в ней замену местами последнего и предпоследнего слов.

```
#include <stdio.h>
#include <stdlib.h>
main()
{ int kk,k,n;
  char *s,*s1,t;
  scanf("%d",&n);
  s=(char*)malloc(n);              // выделение памяти без инициализации
  s1=(char*)calloc(n,1);           // выделение памяти с инициализацией
  if(!s||!s1) return 0;
  fflush(stdin);
  gets(s);
  k=0;
  while(*(s+k)) k++;       // переход в конец строки. k – позиция '\0'
  while(*(s+k)!=' ') k--;   // позиция пробела перед последним словом
  k--;
  while(*(s+k)!=' ') k--;   // позиция пробела перед предпоследним словом

  kk=++k;                  // запоминаем позицию первого символа этого слова
  n=0;                     // перезапись в строку s1 предпоследнего слова
  while(*(s+k)!=' ') *(s1+n++)=*(s+k++);
  k++;                     // запись последнего слова на предпоследнее
  while(*(s+k)) *(s+kk++)=*(s+k++);
  *(s+kk++)=' ';       // вставка пробела после слова
  n=0;                     // дозапись слова из строки s1
  while(*(s+kk++)=*(s1+n++));
  printf("%s",s);
  free(s); free(s1);     // освобождение памяти
}
```

Пример . Создать массив указателей на слова для каждой вводимой строки. Строка может содержать до 10 слов. Используя массив указателей исключить из каждой строки слово с максимальной длинной.

```
#include <stdio.h>
#include <stdlib.h>
#define size 2
void fff(char *,char *);
void main()
{ char *st[size][10],*mst,*pst;
  int i,j,n=0,k=0,kk=0;
  n=0;
  do
  { st[n][0]=(char *)malloc(20); // выделение памяти под строку
    gets(st[n][0]);                    // ввод строки
  } while(*st[n++][0]);
  n--;
  for(i=0;i<n;i++)
  { k=0;
    for(j=0; *(st[i][0]+j); j++)
    if(*(st[i][0]+j)==' ')          // найден пробел - конец слова
      st[i][++k]=st[i][0]+j+1; // адрес начала слова
    st[i][++k]=st[i][0]+j+1;   // адрес '\0' (для последнего слова)
    for(; k<9; )
    st[i][++k]=NULL;           // остальные адреса для слов в строке
  }
  for(i=0; i<n; i++)              // перебор строк
  for(j=0; st[i][j]; j++)         // перебор слов в строке
  fff(st[i][j],st[i][j+1]);       // сравнение 2 слов для поиска max

  for(i=0; i<n; i++)
  puts(st[i][0]);
}

void fff(char *st1,char *st2)
{  int i;
   static j;
   static char *k;
   j=(st2-st1>j) ? k=st1,j=st2-st1 : j;
                    // j - длина максимального слова
                    // k - указатель на начало этого слова
   if (!st2)        // передано не одно слово
   { for(; j>0; j--)
     { for(i=0; *(k+i+1); i++)
```

```
            *(k+i)=*(k+i+1);     // сдвиг строки на позицию max слова
            *(k+i)='\0';
          }
        }
      }
```

Пример . Ввести символьную строку и найти в ней слово максимальной длины.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
main()
{ char *st;
        int i,k,kk=0,i1,i2,i3,size;
  printf("Введите длину строки");
  scanf("%d",&size);
  st=(char *)malloc(size);
  fflush(stdin);
  gets(st);
  i=i1=i2=0;
  while(*(st+i))
  {  i3=i;                                  // индекс начала слова в втроке
     for(k=0; *(st+i)!=' ' && *(st+i); i++,k++);
     if(kk<k) { kk=k; i1=i3; i2=i; }  // координаты max слова
     if(*(st+i)==' ') while(*(st+(++i))==' '); //удаление повторных пробелов
  }
  puts("\nслово max длины ");
  for(i=i1; i<i2; printf("%c",*(st+i++)));
 }
```

Пример . Ввести строку и выполнить в ней замену слова с минимальной и максимальной длинной.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
main()
{ char c,*s,*s1,*s2;
  int i,n,nn,min,max,i1,i2,l,i3,i4;
  clrscr();
  puts("");
  scanf("%d",&nn);
  s=(char*)malloc(nn);
  s1=(char*)calloc(nn,1);
  s2=(char*)calloc(nn,1);
  if(!s||!s1||!s2) return 0;
  fflush(stdin);
```

```
 gets(s);
 i=i1=i2=0;
 min=nn; max=0;
 while(*(s+i))
 { for(n=0;*(s+i)!=' ' && *(s+i); i++,n++);
  if(max<n) { max=n; i1=i-n; }
  if(min>n) { min=n; i3=i-n; }

  if(*(s+i)==' ')
  while(*(s+(++i))==' ');
 }
 l=0; i=i1;
 while(*(s+i)!=' ' && *(s+i))  *(s1+l++)=*(s+i++);

 l=0; i=i3;
 while(*(s+i)!=' ' && *(s+i))  *(s2+l++)=*(s+i++);

 l=0;
 if(i1<i3)
 {  i2=i1+max;
    while(*(s2+l)) *(s+i1++)=*(s2+l++);
    while(i2<i3)   *(s+i1++)=*(s+i2++);
    l=0;
    while(*(s1+l)) *(s+i1++)=*(s1+l++);
 }
 else
 { i=i1+max-min;
  while(*(s2+l))  *(s+i++)=*(s2+l++);
  i2=i-min-1; i1--;
  while(i1>=i3+min)
  *(s+i2--)=*(s+i1--);
  l=0;
  while(*(s1+l))  *(s+i3++)=*(s1+l++);
 }
 puts(s);
 free(s);
 free(s1);
 free(s2);
}
```

Пример . Ввести строки. Определить, какие строки из введенных читаются из начала в конец и наоборот

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

```
void main()
{ char mas[20][20];
  int i,n,length,j;
  clrscr();
  printf("Сколько строк вы хотите ввести: ");
  scanf("%d",&n);
  for (i=0;i<n;i++) { fflush(stdin); gets(mas[i]); } // ввод n строк
  printf("\n\n");
  for (i=0;i<n;i++)
  {  length=0; j=0;
     while (mas[i][j])
     {  length++; j++; }    // подсчитываем длину i строки
     for (j=0;j<length/2;j++)    // проверяем необходимое условие
     if (mas[i][j]!=mas[i][length-1-j]) break;
     if (mas[i][j]==mas[i][length-1-j]) puts(mas[i]); // найдена
                                                      // искомая строка
   }
}
```

Функции

Принцип программирования на языке С(С++), как и в ряде других языков программирования, основан на понятии *функции*. В рассмотренных ранее программах мы уже пользовались функциями. К их числу относятся функции ввода/вывода, работы с символьными строками и ряд других. Эти функции являются системными, в то же время мы разработали несколько функций, использующих вышеназванные и озаглавленных одним общим именем main(). Несколько подробнее остановимся на вопросе, как создавать свои собственные функции и делать их доступными для функции main(), а также друг для друга.

Что такое функция? Функция - самостоятельная единица программы, разработанная для выполнения некоторого функционально законченного действия. Связь между функциями осуществляется через аргументы, возвращаемые значения и внешние переменные. Если внутри функции используются уже определенные ранее имена, то это соответствует неявной передаче информации в функцию. В исходном файле функции разрешается располагать в любом порядке, исходную программу можно подразделять на произвольное число файлов. Следует различать понятия «*прототип функции*» и «*определение функции*».

Прототип функции.

Прототип функции – это ее имя с указанием типа возвращаемого результата и перечислением в круглых скобках (через запятую) типов передаваемых параметров. Прототип функции завершается точкой с запятой. Например:

double step(float x, int n);
int min(int , int);
char * prod(char *, int k);

Прототип является предварительным объявлением данных функции: имени функции, количества и типов параметров, типа результата. Прототип необходим компилятору для того, чтобы проконтролировать корректность вызова функции и в необходимых случаях, выполнить преобразование аргументов к типу принимаемых параметров, а также сгенерировать корректный возвращаемый результат.

Имена параметров в прототипе можно опустить, однако рекомендуется их указывать. Компилятор будет использовать имена параметров при выдаче диагностических сообщений о несоответствии типов аргументов и параметров. В том случае, когда в функцию не передаются аргументы, в прототипе в круглых скобках вместо списка параметров записывается слово void либо ничего не указывается.

Определение функции.

Определение функции – это ее полная реализация. Определение функции может располагаться в любом месте программы, но определение одной функции не может вмещать в себя определение другой функции, но может содержать ее прототип, если она вызывает эту функцию.

Если определение некоторой функции располагается ниже точки ее вызова или в другом файле, то для такой функции выше точки ее вызова обязательно должен быть помещен прототип. В соответствии с синтаксисом языка СИ определение функции имеет следующую форму:

[спецификатор_класса_памяти] [спецификатор_типа] имя_функции
([список_формальных_параметров])
{ тело_функции }

Необязательный спецификатор_класса_памяти задает класс памяти функции, который может быть static или extern. Спецификатор_типа функции задает тип возвращаемого результата. Результат может быть любого стандартного типа, а также типа, определенного пользователем (структура, объединение, класс и др.). Функция возвращает явно только одно значение указанных типов. Допускается реализация функций, не имеющих параметров и/или не возвращающих никаких значений. Если спецификатор_типа не задан, то предполагается, что функция возвращает значение типа int.

Передача (возврат) значения из вызванной функции в вызывающую осуществляется посредством использования оператора возврата return, имеющего следующий вид:

return выражение; или return (выражение);

Выражение может являться как некоторой константой, так и вычисляемым выражением:

return -12; или return (-12);
return i+5/j; или return (i+5/j);

Оператор return в теле функции может встречаться более одного раза.

При его встрече происходит прекращение выполнения функции и осуществляется передача управления очередной инструкции (команде) в вызывающей функции. Если в операторе return отсутствует выражение, а вызваемая функция должна вернуть некоторое значение то компилятор генерирует ошибку. При отсутствии оператора return функция будет выполнена до конца. В этом случае функция также явно не возвращает результат, перед именем такой функции в ее прототипе и в ее определении должно быть записано слово void.

Если тип выражения в операторе return не соответствует типу значения, возвращаемому функцией, то будут выполнены автоматически соответствующие преобразования типа выражения к типу возвращаемого функцией значения. В случае невозможности таких преобразований выводится сообщение об ошибке.

Функция *не может* в качестве результата возвращать *массив* любого типа, но может передать *указатель* на такой массив. Это связано с тем, что в C(C++) нет операции присваивания массивов.

Спецификатор типа возвращаемого функцией результата должен быть указан перед именем функции в операторе прототипа и в определении функции. Отдельные части спецификации могут отсутствовать. При отсутствии типа возвращаемого результата предполагается, что функция возвращает значение типа int.

Ниже приводится пример программы, в которой из функции main() производится вызов поочередно двух функций (fun1, fun2):

```
#include <stdio.h>
int fun1(int , int );       //  прототип функции fun1
int fun2(int *, int *);     //  прототип функции fun2
void main()
{  int a,b,c;
   scanf("%d%d",&a,&b);
   c=fun1(a,b);          //  вызов функции fun1
   printf("\n 1: результат %d - %d = %d",a,b,c);
   c=fun2(&a,&b);     //  вызов функции fun2
   printf("\n 2: результат %d - %d = %d",a,b,c);
}

fun1(int a,int b)
{  a--;                    // измененное локальное значение переменной a
   return  a-b;           // в main передается значение разности a-b
}

fun2(int *a,int *b)
{ (*a)++;                // изменяется значение по адресу  a
  return *a-*b;
}
```

Функция fun1 принимает в качестве аргументов значения переменных a и

b, копируя их в локальные переменные a и b. Изменение переменной a в функции не приводит к изменению значения переменной a описанной в main.

Функция fun2 принимает адреса переменных a и b, копируя их в указатели. Изменение значения по адресу переменной a приводит к изменению значения переменной a в функции main.

Любая функция, может завершаться при вызове системной функции abort() или exit(n) , где n – целое число, являющееся кодом завершения. Код завершения обычно используется программой, которая породила текущий процесс. В этом случае прекращается выполнение всей программы, автоматически закрываются все открытые файлы и освобождаются все области динамической памяти.

Параметры функции

Параметры в списке аргументов различаются на формальные и фактические. Фактические параметры - это параметры, формируемые в вызывающей функции и передаваемые в списке при вызове функции. Формальные - это параметры, описываемые в списке параметров вызываемой функции. Порядок и типы формальных параметров должны быть одинаковыми в определении функции и во всех ее объявлениях. Типы фактических параметров при вызове функции должны быть совместимы с типами соответствующих формальных параметров.

```
void fun(int,char,double);      // прототип функции
imt k;                          // глобальная переменная
main()
{  int n=1;
   char c='a';
  fun(n,c,1.5);                 //  n, c, 1.5   фактические параметры
}
void fun(int nn, char cc, double dd) // nn, cc, dd  формальные параметры
{
        . . .
}
```

В функцию параметры могут передаваться следующими способами:

- используя глобальные переменные, что соответствует неявной передаче параметров;
- используя в качестве формального параметра ссылки;
- по значению, посредством копирования некоторого значения фактического параметра в формальный;
- используя в качестве параметров (фактического и формального) адреса.

В первом и втором случаях любое изменение формального параметра приводит к изменению фактического, а в третьем изменение формального параметра не влияют на значение фактического.

Аргументы функции, передаваемые в функцию по значению, являются локальными переменными. Это значит, что при передаче аргументов в

функцию создаются их локальные копии, с которыми работает функция, то есть если в функции изменяется значение аргумента, то изменяется не переменная, которую передали функции в качестве аргумента, а её локальная копия:

```
#include <stdio.h>
void func(int x,int y);
main()
{  int x=1,y=2;
   printf("\nmain() : x=  %d   y= %d",x,y);
   func(x,y);
   printf("\nmain() : x=  %d   y= %d",x,y);
   return 0;
}
void func(int x,int y)
{ printf("\nfunc() : x=  %d   y= %d",x,y);
  x=3;
  y=4;
 printf("\nfunc() : x=  %d   y= %d",x,y);
}
```

результат выполнения:

main(): x=1 y=2
func(): x=1 y=2
func(): x=3 y=4
main(): x=1 y=2

Если значение формального параметра передается в функцию по адресу, то, как и в первых двух случаях любое изменение значения по переданному в функцию адресу влияет на значение переменной, адрес которой передан в функцию в качестве фактического параметра.

Областью действия имени считается часть программы, в которой это имя можно использовать. Для автоматических переменных область действия - функция, в которой они описаны. Локальные (автоматические) переменные разных функций, имеющие одинаковые имена, не являются связанными друг с другом.

Область действия внешней переменной или функции распространяется от места ее декларирования до конца файла, в котором они расположены. Если же на внешнюю переменную требуется сослаться до того, как она определена, или если она определена в другом файле, то при ее декларировании используется слово extern.

Например, если декларации вида

```
int i;
float mas[5];
```

расположены вне всех функций, то они определяют внешние переменные i и mas, то есть декларирование и выделение памяти под эти объекты выполнено для остальной части файла. В то же время инструкции вида

```
extern int i;
extern float mas[];
```

декларируют, что объекты i (переменная) и mas (массив), имеющие тип int и float, не создаются, а память им выделяется при их объявлении в другой функции.

Для всего множества функций, образующих программу, должно быть не более одного определения каждой из внешних переменных. В объявлении массивов необходимо указывать их размерность, однако при декларировании массива с extern это не требуется.

Передача массива в функцию

Если в качестве передаваемого аргумента используется массив, то в функцию передается *указатель на массив*. При вызове функции в списке аргументов записывается имя массива являющееся адресом первого.

Можно использовать *три варианта описания массива*, в качестве формального параметра функции.

1. Параметр в функции может быть объявлен как массив соответствующего типа с указанием его размера.

```
#include <stdio.h>
int max(int ms[10])                    //  прототип функции fun
{   int k=ms[0];
    for(int i=1; i<10; i++)
    if(k<ms[i]) k=ms[i];
    return k;
}
main()
{  int mas[3][4], i, j;
   for(i=0; i<3; i++)
   { for(j=0; j<4; j++) scanf( ”%d”&mas[i][j]);
     printf(“\nмаксимальное значение в строке %d =%d”, i,fun(mas[i]));
   }
}
```

В этом случае выполняется преобразование mas[i] к указателю на целый тип, и в функцию передается адрес mas[i][0]. В функции находится максимальное значение каждой строки и выводится на экран.

2. Массив в качестве параметра в функции может быть объявлен без указания его размера. Так как сам массив в стек не копируется, то размер массив в общем случае компилятору не требуется.

```
int max(int ms[])                  //  прототип функции fun
{ // тело функции аналогично опмсанному выше
}
```

3. Наиболее распространенный способ – объявление параметра-массива указателем на соответствующий тип данных. Функцию можно описать следующим образом:

```
int max(int *ms)
{   // тело функции аналогично опмсанному выше
}
```

Во всех случаях в функцию передается адрес первого элемента массива, то есть указатель соответствующего типа.

В С каждая функция должна иметь уникальное имя. В С++ функции можно переопределять, то есть одно и то же имя можно использовать для вызова различных функций. В этом случае вызов функции определяется либо *по количеству* параметров, либо *по их типу*, либо *по типу возвращаемого результата*, или и тем и другим.

inline функции

При каждом вызове функции в памяти создается её копия, то есть вызов функции расходует память. Если функция, имеющая небольшой размер, часто вызывается, то её можно объявить как inline (подставляемую). Объявленная таким образом функция не вызывается каждый раз, вместо этого текст функции подставляется в то место программы где она вызвана. При этом экономится память, но увеличивается размер кода. Объявление inline функции:

inline тип_возвращаемого_данного имя_функции(аргументы);

Класс памяти

В С(С+) каждая переменная и функция принадлежит некоторому классу памяти. Существуют следующие классы памяти переменных: автоматический (auto), статический (static), регистровый (register) и внешний (extern). Все переменные, объявленные в теле функции без указания класса памяти, имеют класс памяти auto, т.е. они являются локальными.

Автоматические переменные

Переменные этого класса являются локальными, то есть они доступны (существуют) в пределах блока, в котором они объявлены.

Память для нее автоматически выделяется каждый раз вначале блока и освобождается в конце. Доступ к такой переменной возможен только в блоке в котором она определена. Автоматические переменные инициализируют при их определении или присваивания.

Статические переменные

В программе могут находиться некоторые переменные, используемые только в функциях, входящих в состав программы. Указание static, примененное к внешней переменной или функции, ограничивает область действия соответствующего объекта концом файла.

```
static int i;          //  внешняя статическая переменная
main()
{   . . .             //   тело функции
}
```

Декларацию static можно использовать и для внутренних переменных. Аналогично автоматическим внутренние статические переменные локальны в функциях, но в отличие от автоматических они не возникают только на период выполнения функции, а существуют в памяти постоянно.

```
fun()
{   static int i;      // статическая переменная
      . . .            //   тело функции
}
```

Регистровые переменные

Спецификация register при объявлении переменных сообщает компилятору, что декларируемая переменная будет использоваться интенсивно. Такую переменную желательно размещать на одном из регистров машины. Это приведет к тому, что программа станет короче и работа ускорится. Пример декларации:

```
register int i,j;
register char k;
```

При этом компилятор имеет право проигнорировать указание разместить данные переменные в регистре машины. Спецификация register может применяться только к арифметическим переменным и к формальным параметрам функции. Объявление для формальных параметров имеет вид

```
fun(register unsigned int n, register char c)
{ register int k;
  . . .
}
```

Ограниченное число регистров накладывает ограничение на число register–переменных: достаточно небольшое их число и при этом только определенных типов может быть размещено на регистрах. Избыточные register объявления игнорируются, а для переменной независимо от того выделен ей регистр или нет, не определено понятие адреса. Ограничение на количество и тип register–переменных зависят от типа процессора.

Блочная структура

Блочная структура программы принятая в других языках (Паскаль и др.) не допустима в С(С++) в том смысле, что функции не могут быть расположены одна внутри другой. Однако переменные внутри функций можно определять в блочно-структурной манере.

Декларации переменных (а также их инициализация) может быть размещена не только в начале функции, но и после любой левой фигурной скобки ({). Переменная, описанная таким способом, ”маскирует” одноименные переменные, расположенные вне блока, и действует до соответствующей правой фигурной скобки. Например:

```
{  float n;
       . . .
   for (i=0; i<max; i++)
   {  int n=mas[0];
      for(j=i+1; j<max; j++)
      n+=mas[j];
   }
```

```
    {  char n;
         . . .
    }
}
```

Автоматические переменные, декларируемые и инициализируемые в блоке, инициализируются каждый раз при входе в блок. Переменные static инициализируются только один раз при первом входе в блок.

Автоматические переменные и формальные параметры ”затеняют” внешние переменные с теми же именами. Например:

```
int x, y;          //  x,y - внешние переменные
fun(float x)    //  x,y - автоматические переменные заменяют на
{ char y;         //  на время работы fun  внешние переменные x и y
        . . .
}
```

Примеры программ

Вначале рассмотрим несколько программ использующих функции для работы с символьными строками.

Пример . Разработать функции: ввода символьной строки, вычисляющей длину строки, сравнивающую две строки, копирующую одну строку в другую.

```
#include<stdio.h>
#include<stdlib.h>
char *getstr(char *);         // прототипы функций используемых в main
get_str(char *, int );
int str_len(char *);
int str_cmp(char *,char *);
char * str_cat(char *,char *,int);
double atof(char *);
int atoi(char []);
void itoa(int ,char *);
main()
{  char *s1,*s2;
   int n,n1,n2;
   double d;
   do
   {  fflush(stdin);
      printf("введите размерность ПЕРВОЙ строки = ");
   } while(!scanf("%d",&n1) || n1<=0);
   if (!(s1=(char *)calloc(n1,sizeof(char))))
   {  printf("\nНедостаточно свободной памяти \n");
      return 0;
   }
   do
   {  clrscr();
      fflush(stdin);
```

```
      printf("введите размерность ВТОРОЙ строки = ");
   } while(!scanf("%d",&n2) || n2<=0);
   if (!(s2=(char *)calloc(n2,sizeof(char))))
   {  printf("\nНедостаточно свободной памяти \n");
      free(s1);
      return 0;
   }
   printf("Вводите первую строку  ");
   fflush(stdin);
   printf("\n строка %s",getstr(s1));    // ввод и вывод первой строки
   printf("\nДлина первой строки == %d байт",str_len(s1));
   d=atof(s1);                                  // преобразование строки s1 в double
   printf("\nЗначение числа (double) в строке %s == %lf \n",s1,d);
   printf("\nДлина второй строки == %d байт",get_str(s2,n2));
   printf("\nЗначение числа (int) в строке %s == %d ",n,atoi(s2));
   printf("\nВводите число для перевода в строку  ");
   scanf("%d",&n);
   itoa(n,s2);                                  // преобразование int в строку s2
   printf("\n строка %s",s2);
   if (str_cmp(s1,s2)>0) printf("\n строка 1 > строки 2");
   else if (str_cmp(s1,s2)<0) printf("\n строка 1 < строки 2");
        else printf("\n строка 1 = строке 2");
   s1=str_cat(s1,s2,3);                         // добавление строки s2 в строку s1
   printf("\nстрока (s1+s2) == %s",s1);
}
```

Ниже приводится описание функций использованных в программе.

```
char *getstr(char *s)               //  функция ввода строки
{ int i=0;
  while((*(s+i++)=(char)getchar())!='\n');  // посимвольный ввод в строку
  *(s+ --i)='\0';                    //  до нажатия клавиши Enter
  return s;                          // возврат указателя на строку
}

get_str(char *s, int k)           // ввод строки и подсчет ее длины
{ int c,i=0;
    // символы заносятся в буфер и после нажатия клавиши Enter
    //  k-1 символ из буфера переносятся в строку s
  while(--k>0 && (c=getchar())!=EOF && c!='\n')
  *(s+i++)=c;          // ввод строки до заданного кол-ва символов или
                       // пока не нажата клавиша Enter или Ctrl + Z
  *(s+i)='\0';         // признак конца строки
  return i;            // возврат длинны строки
}
```

```
int str_len(char *s)    // функция определения длины строки
{  for(int n=0; *(s+n); n++);
   return t;
}

int str_cmp(char *s1,char *s2)  // функция сравнения двух строк
{ while(*s1 && *s2)             // пока не достигнут конец одной из строк
  if (*s1-*s2) return *s1-*s2;  // найдены различающиеся символы
  else                          //
  { s1++;                       // переход к новому символу  строк s1 и s2
    s2++;
  }
  return 0;                     //  строки равны.
}
// Еще один вариант функции сравнения двух строк.
int str_cmp(char *s1,char *s2)  // функция сравнения двух строк
{  for (;*s==*t; s++,t++)
   if(!*s && !*t) return 0;
   return *s-*t;
}

// функция вставки (добавления) в строку s1 строки s2 с позиции k
char * str_cat(char *s1,char *s2,int k)
{  char *s;
   int i;
   if (k>str_len(s1)) k=str_len(s1);
   if (!(ss=(char *)malloc(sizeof(char)*(str_len(s1)+str_len(s2))))
   {  printf("\nНедостаточно свободной памяти ");
      return s1;                    // возврат строки без изменения
   }
   for (i=0; i<k; i++)              // перезапись s1 в ss k символов
   *(s+i)=*(s1+i);
   for (i=0; *(s2+i); i++)         // дозапись s2 в ss srt_len символов
   *(s+k+i)=*(s2+i);
   for (i=0; *(s1+k+i); i++)     // дозапись остатка s1 в ss
   *(s+k+str_len(s2)+i)=*(s1+k+i);
   *(s+k+str_len(s2)+i)='\0';
   free(s1);
   return s                         // строки полностью совпали
}

//   функция перевода цифровой символьной строки в signed double
double atof(char *ss)
{  double n, ii=0.0;
```

```
   int i=0, znak;
   while(*(ss+i)==' ') i++;
   if (!*(ss+i)) return 0.;                    // в строке одни пробелы
   znak=(*(ss+i)=='-')? -1: 1;
   if (*(ss+i)=='+' || *(ss+i)=='-')   i++;      // пропуск позиции знака
   for(n=0.0; (*(ss+i)>'0' && *(ss+i)<='9') || *(ss+i)=='.'; i++)
   {  if (ii) ii*=10;                    // число цифр после десятичной точки
      if (*(ss+i)!='.')  n=10.0*n+(*(ss+i)-'0');   // выбор цифр числа из строки
      else ii=1;                         // выбрана десятичная точка
   }
   if(!ii) ii=1;                          //  если введено целое число
   return znak*n/ii;                      // возврат signed double

}

// функция перевода цифровой символьной строки в signed int
int atoi(char ss[])                 // в функцию передается указатель
{  int n,i=0,znak;
   while(*(ss+i)==' ') i++;
   if (!*(ss+i)) return 0;                          // в строке одни пробелы
   znak=(*(ss+i)=='-')? -1: 1;
   if (*(ss+i)=='+' || *(ss+i)=='-') i++;           // пропуск позиции знака
   for(n=0; *(ss+i)>'0' && *(ss+i)<='9'; i++)  // выбор цифр числа из строки
   n=10*n+(*(ss+i)-'0');
   return znak*n;                                   // возврат signed int
}

// функция перевода signed int в символьную строку
void itoa(int n,char *ss)
{  int i=0, znak;
   if ((znak=n)<0) n=-n;        //
   do                           //
   {  ss[i++]=n%10+'0';         //
   } while((n/=10)>0);          //
   if (znak<0) ss[i++]='-';
   ss[i]='\0';
   for(n=0;n<i/2;n++)
   {  ss[n]+=ss[i-1-n];
      ss[i-1-n]=ss[n]-ss[i-1-n];
      ss[n]-=ss[i-1-n];
   }
}
```

Указатели на функции

В языке С (С++) функция не может быть значением переменной, в тоже

время, аналогично переменным функция физически расположена в памяти и соответственно имеет адрес. Таким образом, ее адрес, присвоенный указателю, является точкой входа в функцию. Указатель на функцию может быть использован для вызова функции вместо ее имени, а также позволяет передавать функцию как обычный параметр в другую функцию. Следовательно с указателем на функцию можно обращаться, как с переменной (передавать его другим функциям, помещать в массивы и т.д).

Указатель на функцию – это такой тип переменной, которой можно присваивать *адрес точки входа* в функцию, то есть адрес первой исполняемой команды. Эта переменная в дальнейшем может использоваться для вызова функции вместо ее имени. *Определение* указателя на функцию имеет следующий общий вид:

тип_результата (*имя_указателя_на функцию)(список_типов_параметров);

В объявлении

int (*fun)(int, int *);

переменная fun является указателем на функцию с двумя параметрами: типа int и указателем на int. Сама функция должна возвращать значение типа int. Круглые скобки, содержащие имя указателя fun и признак указателя *, обязательны, иначе запись

int *fun (int, int *);

будет интерпретироваться как объявление функции fun возвращающей указатель на int. Это следует из того, что приоритет префиксного оператора * ниже, чем приоритет ().

Вызов функции возможен только после инициализации значения указателя fun и имеет вид:

(*fun)(i,&j);

В этом выражении для получения адреса функции, на которую ссылается указатель fun используется операция разадресации * . Вызов функции через указатель возможен так же в обычным способом:

fun(i,&j);

Ниже приведен пример использования указателя на функцию инициализированного адресом системной функции strcmp.

```
# include <stdio.h>
# include <string.h>
# define n 80
void check(char *, char , int ( cmp)(const char *,const char *)));
void main(void)
{  char s1[n],s2[n];
   int (*p)(const char *,const char *);
   p=strcmp;     // p – адрес  strcmp()
   gets(s1);  gets(s2);
   check(s1,s2,p);
}
void check(char *a, char b, int ( cmp)(const char *,const char *)))
```

```
{  printf("\n строки ");
   if (!(*cmp)(a,b)) printf("равны");
   else printf("не равны");
}
```

При вызове check ей передается два указателя на символьные строки и один на функцию. Аналогично можно обращаться к check, используя и указатели на другие функции, если возвращаемый тип и передаваемые параметры, совпадают с рассмотренными выше. Выражение if(!(*cmp)(s1,s2)) можно заменить на if(!cmp(s1,s2)). Можно вызвать функцию check также следующим образом: check(s1,s2,strcmp). При этом не требуется использовать дополнительно указатель p.

Ниже приводятся примеры некоторых более сложных деклараций:

int (*ukz)[10]; - объявление указателя ukz на массив из 10 элементов int;

char (*(*fun))[]() - функция, возвращающая указатель на массив из функций, возвращающих значение типа char;

char (*(*m_ukz[5])())[10] - m_ukz[5] массив из указателей на функции, возвращающие указатели на массив из [10] элементов char.

Примеры программ

Пример . Использование указателя на функцию в качестве параметра функции вычисляющей производную от функции cos(x).

```
double proiz(double x, double dx, double (*f)(double x) );
double fun(double z);
int main()
{  double x;            //  точка вычисления производной
   double dx;           //  приращение
   double z;            //  значение производной
   scanf("%f,%f",&x,&dx);  //  ввод значений x и dx
   z=proiz(x,dx,fun);      //  вызов функции
   printf("%f",z);         //  печать значения производной
   return 0;
}

double proiz(double x,double dx, double (*f)(double z) )
{       // функция вычисляющая производную
    double xk,xk1,pr;
    xk=fun(x);
    xk1=fun(x+dx);
    pr=(xk1/xk-1e0)*xk/dx;
    return pr;
}

double fun( double z)
{    // функция от которой вычисляется производная
    return (cos(z));
```

```
}
```

Для вычисления производной от какой-либо другой функции можно изменить тело функции fun или использовать при вызове функции proiz имя другой функции. В частности, для вычисления производной от функции cos(x) можно вызвать функцию proiz в форме

```
z=proiz(x,dx,cos);
```

а для вычисления производной от функции sin(x) в форме

```
z=proiz(x,dx,sin);
```

Пример . При разработке диалогового меню в приложениях, требующих высокой производительности, используется массив указателей на вызываемые функции.

```
# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# include <conio.h>
void enter();   void del();   void rew();   void quit();
int menu(void);
void (*options[])(void)={enter, del, rew, quit};
int main(void)
{  int i;
   i=menu();          // выбор режима работы
   (* options[i])();  // вызов функции
   return 0;
}
int menu(void)
{  char ch;
   do
   {  printf("1 - ввод");
         . . .
      printf("4 - выход");
      printf("Выберите режим работы");
      ch=getch();
      printf("\n");
   } while(!strchr("1234",ch));
   return ch-49;         // преобразуем ch в целое число
}
void enter()         . . .          void quit()
{  . . .                            {  . . .
}                                   }
```

Рекурсия

Функция является рекурсивной, если некоторая инструкция этой функции содержит вызов самой этой функции. Компилятор допускает любое число рекурсивных вызовов. Рекурсивный вызов не создает новую копию функции.

При каждом вызове для формальных параметров и переменных с классом памяти auto и register выделяется новая область памяти, так что их значения из предыдущих вызовов не теряются, но в каждый момент времени доступны только значения текущего вызова.

Переменные, объявленные с классом памяти static, не требуют выделения новой области памяти при каждом рекурсивном вызове функции и их значения доступны в течение всего времени выполнения программы.

Хотя компилятор языка СИ не ограничивает число рекурсивных вызовов функций, это число ограничивается ресурсом памяти компьютера. Большое число рекурсивных вызовов функции может привести к переполнению стека, что в свою очередь приведет к ошибочному окончанию работы программы. При разработке рекурсивной функции следует, используя операторы if и return, предусмотреть возможность завершения ее работы. В противном случае возможно "зацикливание" программы.

Примеры программ

Пример . Программа нахождения наибольшего общего делителя двух чисел.

```
#include <stdio.h>
int nod(int, int );
void main()
{ int a,b;
  fflush(stdin);
  scanf("%d%d",&a,&b);
  printf("НОД чисел %d и %d  =  %d",a,b,nod(a,b));
}
nod(int a,int b)                    // рекурсивная функция, вычисляющая
{ int c;                            // НОД чисел  a  и  b
  if (b>a) c=nod(b,a);
  else if (b<=0) c=a;
       else c=nod(b,a%b);
  return(c);
}
```

Пример . Вычисление факториала n! .

```
#include<stdio.h>
#include<conio.h>
int fact1(int);       // прототип первого варианта функции вычисления n!
int fact2(int);       // прототип второго варианта функции вычисления n!
int fact3(int);       // прототип третьего варианта функции вычисления n!
void main(void)
{ int n,i;
  char c;
  do
```

```
  { clrscr();
    fflush(stdin);
    puts("\nВводите число >=0 для вычисления его факт-ла :");
    i=scanf("%d",&n);
  } while(i<1 || n<0);
  printf("\nФакториал числа %d (%d!) = %d   вариант 1",n,n,fact1(n));
  printf("\nФакториал числа %d (%d!) = %d   вариант 2",n,n,fact2(n));
  printf("\nФакториал числа %d (%d!) = %d   вариант 3",n,n,fact3(n));
  getch();
}
int fact1(int n)      // первый вариант функции вычисления n!
{  int i;
   if (n==1 || n==0) return (1);    // выход из рекурсии
   return fact1(n-1)*n;
}
int fact2(int n)      // второй вариант функции вычисления n!
{  static long i;              // т.к. i-статическая, то для нее не созда-
                               // ется копии и при возврате через return
   if(!n) return(1);           // для вычисления 0!
   if (++i<n)
   return --i*fact2(n);
   return n;                   // i сохраняет const максимальное значение
}

int fact3(int n)      // третий вариант функции вычисления n!
{return n ?n*fact3(n-1):1;}
```

Пример . Программа вывода последовательности Фибоначчи состоящей из 15 членов.

```
#include<stdio.h>
int fibon(int);
void main(void)
{ int n,i;
  clrscr();
  printf("\nпоследовательность Фибоначчи -   ");
  for(n=0;n<15;n++) printf("%4d",fibon(n));
}

int fibon(int n)
{return ((n==0 || n==1)) ?1:fibon(n-1)+fibon(n-2);}
```

Пример . Программа перевода чисел из десятичной системы счисления в систему счисления с основанием до 32.

```
#include <stdio.h>
void fc(int);
```

```
void fd(double,int);

int ss;
char znak='+';
void main()
{  int t;
   double num;
   printf("\nВведите исходное число : ");
   scanf("%lf",&num);
   num<0?znak='-',num*=-1:num;  // выделение знака числа
   printf("\nВведите основание новой с/с");
   scanf("%d",&ss);
   printf("\nВведите точность для дробной части числа с/с");
   scanf("%d",&t);
   fc((int)num);
   putchar('.');
   fd(num-(int)num,t);
}

void fc(int num1)      // функция перевода целой части числа
{  int i;
   if(num1>=ss)
   {  i=num1%ss;      // получаем остаток от деления на основание
      num1/=ss;       // получаем целую часть от деления
      fc(num1);       // рекурсивный вызов функции fc
      printf("%c",i>9? i-10+'a': i+'0');
   }
  else printf("%c %c",znak,num1>9? num1-10+'A': num1+'0');
}

void fd(double num,int t)  // функция перевода дробной части числа
{  static n;
   int i;
   if(num!=0 && n++<t)
   { i=num*=ss;      // получаем произведение и выделяем целую частьk
     printf("%c",i>9? i-10+'a': i+'0');
     fd(num-(int)num,t); // рекурсивный вызов fd
   }
   else printf("%c",num>9? (int)num-10+'A': (int)num+'0');
}
```

Пример . Программа использующая рекурсивную функцию сортировки массива чисел методом Хоора.

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

```
#define k 7
void hoor(int *,int,int);
void swap(int *,int,int);

void main(void)
{ int a[k],i,n;
  do
  { clrscr();
    fflush(stdin);
    puts("\nВводите кол-во чисел");
    i=scanf("%d",&n);
  } while(i<1 || n>k);
  for(i=n-1;i<k;i++) a[i]=0;
  puts("\nВводите элементы массива ");
  for(i=0;i<n;i++)
  {  fflush(stdin);
     if (!scanf("%d",&a[i])) i--;
  }
  clrscr();
  printf("\nИсходный массив        :");
  for(i=0;i<n;i++) printf("%4d",a[i]);
  hoor(a,0,n-1);
  printf("\n\nПреобразованный массив :");
  for(i=0;i<n;i++) printf("%4d",a[i]);
  getch();
}

void hoor(int *a,int l,int r)
{ int i,las;
  if(l>=r)  return;
  swap(a,l,(l+r)/2);       // делящий эл-т переносится в a[l] (a[l]<->a[(l+r)/2])
  las=l;                   // позиция посл-го эл-та большего чем делящий
  for(i=l+1;i<=r;i++)   // деление [l,r] на [l,las-1] и [las+1,r]
    if(a[i]<a[l])   swap(a,++las,i);
  swap(a,l,las);
  hoor(a,l,las-1);       // сортировка для [l,las-1]
  hoor(a,las+1,r);       // сортировка для [las+1,r]
}

void swap(int *a,int i,int j)
{ int tmp;             // функция замены i и j эл-тов в массиве a
  tmp=a[i];
  a[i]=a[j];
  a[j]=tmp;
}
```

Аргументы в командной строке

В операционной среде имеется возможность передавать аргументы запускаемой программе посредством командной строки. *Аргументы командной строки* – это информация, следующая за именем программы в командной строке операционной системы. Borland C++ поддерживает три аргумента функции main(). Первые два из них обычно обозначаются **argc** и **argv**. В параметре argc содержится количество передаваемых в командной строке аргументов. Значение argc не может быть меньше 1, так как имя программы рассматривается как первый аргумент. Второй параметр - argv[], является указателем на массив символьных строк (указателей). Каждый элемент данного массива указывает на аргумент командной строки. Аргументы командной строки должны отделяться пробелами или табуляцией. Запятые, точки и прочие символы не рассматриваются как разделители. Например:

C:\> fun.exe 3 aaa bbbb ccc – argc=3 и argv[] – указывает на три строки;

C:\> fun.exe 3 aaa,bbbb,ccc – argc=3 а argv[] – указывает на одну строку;

Если необходимо в качестве аргумента передать строку, содержащую пробелы и табуляции, то ее следует заключить в двойные кавычки. Например:

”aaa bbbb ccccc”

Объявление char *argv[] указывает на то, что массив argv не имеет фиксированной длины. По соглашению argv[0] есть имя вызываемой таким образом программы. Если argc=1, то в командной строке после имени программы нет никаких аргументов. Если в командной строке не указаны аргументы, то это приводит к ошибке. Использование второго индекса в argv позволяет осуществить доступ к отдельным символам командной строки. Например:

```
# include<stdio.h>
void main(int argc, char *argv[])
{ int i,j;
  for(i=0;i<argc;i++)
  { j=0;
    while(argv[i][j]) printf(“%c”,argv[i][j++]);
    printf(”  ”);
  }
}
```

Эта программа выводит по одному символу все аргументы, с которыми она вызывалась.

Помимо argc и argv, в Borland C++ имеется третий аргумент – **env**. Этот параметр позволяет программе получить доступ к информации о среде операционной системы и следует за argc и argv. Как и argv, env является указателем на массив строк, где каждая строка – это строка среды, определенная операционной системой. Параметр env не имеет аналога параметра argc сообщающего о количестве строк среды. Для этого используется нулевая последняя строка среды. Например:

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

```
void main(int argc, char *argv[], char *env[])
{ int i;
  for(i=0;env[i];i++) printf("%s\n",env[i]);
}
```

Отметим, что хотя параметры argc и argv в программе и не используются, но должны присутствовать в списке параметров.

Вместо традиционных имен аргументов argc, argv и env можно использовать любые другие имена.

Функции с переменным числом параметров

В C(C++) наряду с использованием функций с фиксированным числом параметров можно использовать и функции с переменным числом параметров, то есть функции в которые можно передавать данные, не описывая их в прототипе и заголовке функции. Описания этих данных заменяются ***тремя точками***. В таких функциях может находиться также один или более постоянный параметр (признак), с помощью которого могут считываться данные. Если в функции имеется несколько постоянных параметров, то сначала перечисляются эти параметры, а затем ставятся три точки.

В случае использования функции с переменным числом аргументов программист сам контролирует реальное количество аргументов, находящихся в стеке, и отвечает за выбор из стека дополнительных аргументов. В стек информация заносится словами. Целочисленные данные преобразуются к типу int, а данные с плавающей точкой к типу double.

В C++ должен быть хотя бы один фиксированный параметр.

Возможность передачи переменного списка параметров определяется способом (порядком) записи аргументов функции в стек программы и способом считывания параметров из стека.

В C(C++) аргументы, заданные в списке фактических параметров при вызове функции, размещаются в стеке с *конца к началу списка* параметров (рис. 8). При этом в вершине стека размещается первый аргумент, например, при вызове **sum(a,b,c)** аргументы **a,b** и **c** в стеке размещаются следующим образом:

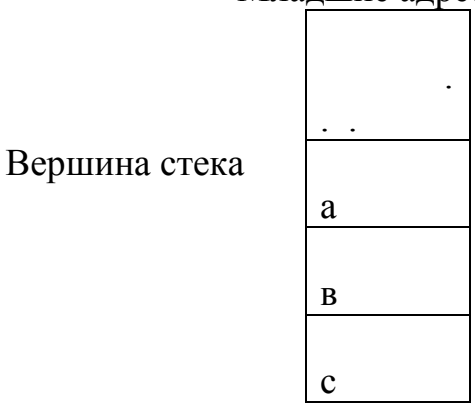


Рис.8. Структура стек

Возможны ***два способа задания длины*** переменного списка параметров:

- указание числа аргументов передаваемых в функцию;

- задание признака конца списка аргументов.

Реализовать функции с переменным числом параметров можно ***тремя способами***:

- используя *указатель без типа*, например: void *;

- используя *указатель, соответствующий типу* переменных списка параметров, например: int * , double *;

- используя указатель, определенный самой системой программирования (с помощью *стандартных макросов* va_list, va_start, va_arg, va_end.).

Существует несколько способов описания данных, передаваемых в функцию. Рассмотрим их на примерах:

```
#include <stdio.h>
int sum(int,…);
void main(void)
{ int a,b,c;
  scanf("%d%d%d",&a,&b,&c);
  printf("сумма 2 чисел равна %d",sum(3,a,b,c));
  printf("сумма 3 чисел равна %d",sum(2,a,b));
}
int sum(int k,…)          // пример простой функции, вычисляющей сумму
{ int s=0,*p;             // произвольного числа аргументов
   p=&k;                  // указатель на количество аргументов
   p++;                   // указатель на первый суммируемый аргумент
   for(i=0;i<k;i++)
   s+=*p++;
}
```

В этом варианте первый аргумент k является количеством чисел, которые будут считаны из стека. Во втором варианте функции sum в списке аргументов нет ни одного параметра:

```
int sum(…)                // второй вариант функции, вычисляющей сумму
{ int s=0,n;              // произвольного числа аргументов
   void *p;
   p=(…);                 // указатель на первый суммируемый аргумент
   do
   { n=*(int *)p;         // извлекаем из стека значение, расположенное по
                          // адресу (int *)p
     (int *)p++;          // наращиваем адрес в стеке
      s+=n;
   } while(n);
}
```

В функции использован бестиповой указатель void *p, далее приводимый к указателю int *. Функция прекращает работу, если из стека будет считано число 0. Приводимый ниже пример, иллюстрирует применение бестипового указателя для обработки списка переменной длинны, в котором в качестве параметров передаются данные различных типов: int, float, char. При вызове

функции задается длина списка аргументов и тип данных.

```
#include<stdio.h>
#include<conio.h>
enum Type{Char,Int,Float};
void main(void)
{ void out(int,enum Type,...);
  int k1,k2,k3,k4,k5;
  float v1,v2,v3,v4,v5;
  out(5,Char,'m','i','n','s','k');
  out(4,Int,9,8,7,6);
  out(6,Float,1.2,2.3 ,3.4,4.5,5.6,6.7);
  puts("\nвведите 5 значений типа INT= ");
  scanf("%d%d%d%d%d",&k1,&k2,&k3,&k4,&k5);
  puts("\nвведите 5 значений типа FLOAT= ");
  scanf("%f%f%f%f%f",&v1,&v2,&v3,&v4,&v5);
  out(5,Int,k1,k2,k3,k4,k5);
  out(5,Float,v1,v2,v3,v4,v5);
}
void out(int k,enum Type t,...)
{  void *pv; pv=(…);
   printf("\n");
   while(k--)                                    //  k- число аргументов
   {  switch(t)
      { case Char:  printf("%c",*(int *)pv);   // вывод аргумента
                    ((int *)pv)++; break;       // переход к следующему
        case Int:   printf(" %d ",",*(int *)pv);
                    ((int *)pv)++; break;
        case Float: printf("  %f  ",*(double *)pv);
                    ((double *)pv)++; break;
        default: puts("ошибка");
      }
   }
}
```

В C(C++) имеются четыре стандартные функции (макрокоманды) для работы со списком переменной длинны, описанные в библиотеке stdarg.h: va_list, va_start, va_arg, va_end. Изменим рассмотренную выше функцию out, чтобы в ней были использованы эти функции:

```
void out(int k,enum Type t,...)
{  va_list pr;
   va_start(pr,t);     // указатель pr на начало списка параметров
   printf("\n");
   while(k--)
   {  switch(t)        //  считывание параметров из стека
      { case Char: printf("%c",va_arg(pr,int)); break;
        case Int: printf(" %d ",va_arg(pr,int)); break;
```

```
            case Float: printf("  %f  ",va_arg(pr,double)); break;
            default: puts("ошибка");
          }
       }
       va_end(pr);    //
}
```

Макрокоманда va_list позволяет определить безтиповой указатель pr, далее используемый для работы со списком переменной длинны. Указатель используется макрокомандами va_start, va_arg и va_end.

Макрокоманда va_start(pr) устанавливает переменную pr (имеющую тип va_list) в качестве указателя на первый необязательный аргумент в списке параметров, передаваемых функции. Макрокоманда va_start должна вызываться перед первым вызовом макрокоманд va_arg или va_end.

Макрокоманда va_arg(pr,t), при каждом очередном вызове, позволяет извлечь из стека очередной элемент списка переменных параметров и присвоить переменной pr адрес очередного элемента из стека. Второй аргумент t функции va_start – является (задает) именем типа данного считываемого из стека. В макрокоманде va_arg можно использовать типы данных определенные языком C(C++).

Макрокоманда va_end(pr) устанавливает значение указателя pr равным NULL и должна быть вызвана только после считывания последнего аргумента. В противном случае ее вызов может повлечь непредсказуемое поведение программы. После выполнения va_end нарушается связь указателя с элементами стека, но информация в стеке сохраняется до окончания работы этой функции.

Примеры программ

Пример . Реализация функций с произвольным числом параметров

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

void fun1(int,...);    // прототип ф-ции имеющей 1 постоянный параметр
void fun2(int,...);    // прототип ф-ции имеющей 1 постоянный параметр
void fun3(...);        // прототип ф-ции не имеющей постоянных параметров

void main(void)
{ int a1,a2,a3;
  float f1,f2;
  clrscr();
  puts("введите 3 значения типа int и 2 типа float");
  scanf("%d%d%d",&a1,&a2,&a3);
  scanf("%f%f",&f1,&f2);
  fflush(stdin);
  fun1(3,a1,a2,a3);
```

```
  fun2(2,f1,f2);
  fun1(2,'1','2');
  fun3(a1,a3,a1,0);
  getch();              // задержка
}

void fun1(int k,...)
{ int s=0;
  void *p;
  p=&k;                 // p указатель на элемент стека k
  ((int *)p)++;         // указатель на следующий элемент стека
  while(k--) s+=*((int *)p)++;   // считывание k чисел int из стека
  printf("\n 1 сумма  элементов = %d",s);
}

void fun2(int k,...)
{ float s=0;
  void *p;
  p=(...);      // указатель на первый элемент перем. списка парам.
  while(k--) s+=*((double *)p)++;   // считывание k чисел float из стека
  printf("\n 2 сумма  элементов = %5.2f",s);
}

void fun3(...)
{ int s=0,k;
  void *p;
  p=(...);      // указатель на первый элемент перем. списка парам.
  while(k=*((int *)p)++) s+=k;
  printf("\n 3 сумма  элементов = %d",s);
}
```

Пример . Реализация функции с произвольным числом параметров одного типа. Доступ к данным (параметрам) реализован с помощью безтиповых указателей (void *p). При чтении данных из стека данные приводятся к требуемому типу.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
int * fun(...);    // прототип ф-ции не имеющей постоянных параметров
void main(void)
{ int *ms;         // для возврата из fun адресов полученных сумм
  clrscr();
  ms=fun(1,7,2,7,8,6,8,4,0);
  for(;*ms;)       // вывод пока *ms!=0
  printf("%6d",*ms++);
```

```
  getch();        // задержка
}
     // функция выделения k MAX элементов из списка параметров
     // найденные элементы возвратить в main
int * fun(...)
{ int i,j,k,n=0,*ii,**jj;
  void *p;
  printf("\n сколько MAX переменных надо выделить из списка =  ");
  scanf("%d",&k);     // ввести ограничение на k (не более чем параметров)
  ii=(int *)calloc(k+1,sizeof(int));  // k max значений из списка (k+1 эл-т
                                      // всегда ==0 - признак конца для main)
  jj=(int **)calloc(k,sizeof(int *)); // адреса k max значений в стеке
  while(n<=k-1)        // пока не найдено k max элементов
  { p=(...);           // элипсис, p устанавливается на начало списка
                       // параметров переменной длинны
    do
    { i=0;
      while(i<n && jj[i])       // поиск первого еще не занесенного
      if (jj[i++]==(int *)p)    // адрес уже есть
      { ((int *)p)++;           // выбор следующего элемента из стека
        break;
      }
    } while(jj[i]);
    ii[n]=*(int *)p;          // предварительное max значение для сравнения
    jj[n]=((int *)p)++;     // его адрес в стеке
    while(j=*((int *)p))   // пока считанный j!= 0 (признак конца списка)
    { for(i=0;i<n && jj[i]!=(int *)p && jj[i++];); // поиск адреса эл-та в jj
      if (j>ii[n] && jj[i]!=(int *)p)    // эл-т max и его адрес не найден
      { ii[n]=*((int *)p);    // новое max значение для сравнения
        jj[n]=((int *)p);      // его адрес в стеке
      }
      ((int *)p)++;           // указатель на следующий элемент стека
    }
    n++;
  }
  free(jj);
  return ii;
}
```

Сортировка

Сортировка применяется во многих программах оперирующих информацией. Цель сортировки – упорядочить информацию и облегчить поиск требуемых данных. Существует много методов сортировки данных. Причём для разных типов данных иногда целесообразно применять разные методы сортировки.

Практически каждый алгоритм сортировки можно разбить на три части:

- сравнение двух элементов, определяющее упорядоченность этой пары;
- перестановку, меняющую местами неупорядоченную пару элементов;
- сортирующий алгоритм, определяющий выбор элементов для сравнения и отслеживающий общую упорядоченность массива данных.

Сортировка данных в оперативной памяти называется внутренней, а сортировка данных в файлах называется внешней. Ниже будут рассматриваться методы сортировки данных в оперативной памяти.

Пузырьковая сортировка.

Название метода отражает его суть. ”Легкие”элементы массива ”всплывают” вверх(в начало), а ”тяжелые” опускаются вниз(в конец). Пузырьковая сортировка проходит по массиву снизу вверх. Каждый элемент массива сравнивается с элементом, который находится непосредственно над ним. И если при их сравнении оказывается, что они удовлетворяют условию их перестановки, то она выполняется. Процесс сравнений и перестановок продолжается до тех пор, пока ”легкий” элемент не “всплывет” вверх. В конце каждого прохода по массиву, верхняя граница сдвигается на один элемент вниз(вправо). Сортировка продолжается анализируя все меньшие массивы. В примере выполняется сортировка элементов массива размерностью k.

```
void sort(int *ms, int k)
{ int i,j,m;
  for(i=0;i<k-1;++i)       // выбор верхней границы массива
  for(j=k-1;j>i;--j)       // просмотр массива ”снизу”  ”вверх”
  {if(ms[j-1]>ms[j])       // условие замены выполнено
    { m=ms[j-1];           // замена j-1  и j  элементов
      ms[j-1]=ms[j];
      ms[j]=m;
    }
  }
}
```

Внутренний цикл for выполняет проход по подмассиву в направлении обратном внешнему циклу. Если изменить направление этих циклов, то не наименьший будет “всплывать”, а наибольший “тонуть”.

Шейкер сортировка

В этом методе учитывается тот факт, что от последней перестановки до конца массива будут находиться уже упорядоченные данные. Тогда просмотр имеет смысл делать не до конца массива, а до последней перестановки на предыдущем просмотре. Если же просмотр делать попеременно в двух направлениях и фиксировать нижнюю и верхнюю границы неупорядоченной части, то получим шейкер-сортировку. Ниже приведен пример программы, использующей шейкер-сортировку элементов массива размерностью k.

```
void shaker(int *ms, int k)
{ register int i,a,b,c,d;
  c=1;
  b=k-1;   //номер элемента на котором остановка
  d=k-1;   //номер стартового элемента для сортировки справа налево
  do
  { for(i=d;i>=c;--i)
    { if (ms[i-1]>ms[i])
      { a=ms[i-1];
        ms[i-1]=ms[i];
        ms[i]=a;
        b=i;            // крайний слева упорядоченный элемент
      }
    }
    c=b+1;  //номер стартового элемента для сортировки слева направо
    for(i=c;i<=d;++i)
    { if (ms[i-1]>ms[i])
      { a=ms[i-1];
        ms[i-1]=ms[i];
        ms[i]=a;
        b=i;            // крайний слева упорядоченный элемент
      }
    }
    d=b-1;
  } while(c<=d);
}
```

Сортировка вставкой

В методе вставки на первом шаге выполняется сортировка первых двух элементов массива. Далее алгоритм ставит третий элемент в порядковую позицию, соответствующую его положению относительно первых двух элементов. Затем в этот список вставляется четвертый элемент и т.д. Процесс продолжается до тех пор, пока все элементы не будут отсортированы. Например, если бы строка "dcab" сортировалась методом вставки, то каждый из проходов давал бы результат:

В этом методе определяется место в отсортированной части массива, куда должен помещаться элемент. Элементы от данной позиции сдвигаются вверх и на освободившееся место помещается элемент. Трудоемкость алгоритма при линейном поиске n*n/4. Если используется двоичный поиск для определения места элемента, то трудоемкость алгоритма составляет n*log(n). Пример программы, использующей сортировку вставкой элементов массива размерностью k :

```
void srt2(int *ms, int k)
{  register int i,j,n;
```

```
   for(i=1;i<k;++i)          // индекс элемента для упорядочивания
   {  j=i-1;                 // индекс предыдущего элемента
      n=ms[i];               // значение предыдущего элемента
      while (j>=0 && n<ms[j])
      ms[j--+1]=ms[j];   // сдвиг всех элементов направо
      ms[j+1]=n;           // запись в освободившийся или в тот же элемент
   }
}
```

Сортировка выбором

Выбирается очередной исходный элемент массива. Последовательно проходя весь массив, сравниваем этот элемент со всеми, находящимися после него, и, при нахождении элемента удовлетворяющего условию замены запоминаем его индекс. После просмотра всего массива, переставляем выбранный элемент с исходным. После перебора всех элементов, получим отсортированный массив. Пример программы, реализующей сортировку выбором, приведен ниже:

```
void sort(int *ms, int k)
{  int i,i1,j,m;
   for(i=0;  i<k-1; i++)      // выбор исходного элемента к сравнению
   {  i1=i;                    //
      for(j=i+1; j>i; j--)       // просмотр массива ”снизу”  ”вверх”
      if(ms[k]>ms[j]) i1=j;  // фиксируем координату элемента в массиве
      m=ms[i];                 // замена i1  и  i  элементов
      ms[i]=ms[i1];
      ms[i1]=m;
   }
}
```

Отмеченные методы сортировки не являются эффективными. В программах, где требуется высокая производительность работы сортировок, применяются следующие методы.

Метод Шелла

Метод был предложен автором (Donald Lewis Shell) в 1959г. Основная идея алгоритма состоит в том, что на начальном этапе реализуется сравнивание и, если требуется, перемещение далеко отстоящих друг от друга элементов. Интервал между сравниваемыми элементами (gap) постепенно уменьшается до единицы, что приводит к перестановке соседних элементов на последних стадиях сортировки (если это необходимо).

Реализуем метод Шелла следующим образом. Начальный шаг сортировки примем равным n/2, т.е. 1/2 от общей длины массива, и после каждого прохода будем уменьшать его в два раза. Каждый этап сортировки включает в себя проход всего массива и сравнение отстоящих на gap элементов. Проход с тем же шагом повторяется, если элементы переставлялись. Заметим, что после

каждого этапа отстоящие на gap элементы отсортированы.

Рассмотрим пример. Рассортировать массив чисел: 41,53,11,37,79,19,7,61.

В строке после массива в круглых скобках указаны индексы сравниваемых элементов и указан номер внешнего цикла.

```
41  53  11  37  79  19   7   61 – исходный массив.
41  19  11  37  79  53   7   61 – (0,4), (1,5) ┐
41  19   7  37  79  53  11  61 – (2,6), (3,7) ┘ 1-ый цикл
 7  19  41  37  11  53  79  61 – (0,2),(1,3),(2,4),(3,5),(4,6),(5,7) ┐ 2-ой цикл
 7  19  11  37  41  53  79  61 – (0,2),(1,3),(2,4),(3,5),(4,6),(5,7) ┘
 7  11  19  37  41  53   61 79 – сравнивались соседние.(3-ий цикл).
```

```
void sort(int *ms, int k)
{ int  i, j, n;
    int  gap;              // шаг сортировки
    int  flg;              // флаг окончания этапа сортировки
    for(gap = k/2; gap > 0; gap /= 2)
    do
    {  flg = 0;
       for(i = 0, j = gap; j < k; i++, j++)
       if(*(ms+i) > *(ms+j))    // сравниваем отстоящие на gap элементы
       {   n = *(ms+j);
           *(ms+j) = *(ms+i);
           *(ms+i) = n;
                flg = 1;              // есть еще не рассортированные данные
       }
    } while (flg);                // окончание этапа сортировки
}
```

Метод Хора

Алгоритм быстрой сортировки построен на основе идеи разбиения массива на разделы. Общая процедура заключается в выборе пограничного значения, называемого *компарандом*, которое разбивает сортируемый массив на две части. Все элементы, значения которых больше пограничного значения, переносятся в один раздел, а все элементы с меньшими значениями - в другой. Затем это процесс повторяется для каждой из частей и так до тех пор, пока массив не будет отсортирован. Например, допустим, что необходимо отсортировать следующий массив: "fedacb". Если в качестве компарадера использовать "d", то после первого прохода алгоритм быстрой сортировки упорядочит массив следующим образом

Исходный массив	Fedacb
Проход 1	Bcadef

Затем этот подход повторяется для каждого раздела, а именно "bca" И "def".

Проход 2	Acb	Def
Проход 3	Abc	Def

Процесс рекурсивен по своей природе и наилучшие решения получаются при рекурсивном подходе. Пограничное значение можно выбирать двумя путями. Во-первых, его можно делать случайным образом, или путем осреднения небольшого набора значений, принадлежащих к разделу. Для того, чтобы сортировка была оптимальной, надо выбирать значение, расположенное точно в середине диапазона значений. Поскольку на практике это не всегда осуществимо, можно выбирать срединный элемент каждого из разделов.

Метод Хоора (Charles Antony Richard Hoare, 1962 г.), называемый также методом быстрой сортировки (Quicksort) основывается на следующем: находится такой элемент, который разбивает множество на два подмножества так, что в одном все элементы больше, а в другом - меньше делящего. Каждое из подмножеств также разделяется на два, по такому же признаку. Конечным итогом такого разделения станет рассортированное множество. Рассмотрим один из вариантов реализации сортировки Хоора.

В процедуре сортировки сначала выберем срединный элемент. Далее, в слева от срединного элемента выбираются элементы большие, а в правой – меньшие срединного. Найденные элементы меняются местами. это выполняется пока ленная и правая границы не равны. В результате будут получены два подмножества. Если эти подмножества существуют, то выполняем по отдельности их сортировку.

Например, необходимо рассортировать массив: 13,3,23,19,7,53,29,17. Переставляемые элементы будем подчёркивать, средний - выделим жирным шрифтом, а элемент попавший на своё место и не участвующий в последующих сравнениях - наклонным шрифтом. Индекс i будет задавать номер элемента слева от среднего, а j справа от среднего.

13 3 <u>23</u> **19** 7 53 29 <u>17</u>
13 3 17 **<u>19</u>** <u>7</u> 53 29 23
13 3 17 7 19 53 29 23
делим на два подмножества <u>13</u> **<u>3</u>** 17 7 19 **<u>53</u>** 29 <u>23</u>
Выделяем подмножество
3 13 17 7 19 23 29 53
13 **<u>17</u>** <u>7</u>
Выделяем подмножество <u>13</u> **<u>7</u>** 17
Результат: 3 7 13 17 19 23 29 53

```
//  l – левая r – правая границы сортируемого массива
void hoar(int  *ms, int  l, int  r) // передаем  левую/правую границы
{   int  i, j, k;
```

```
    int sr = ms[(l+r)/2];          // срединный элемент
            i = l;    j = r;       // начальные значения границ массива
    do
    {   while(ms[i] < sr) i++;  // ищем слева элемент больше среднего
              while(ms[j] > sr)  j--;  // ищем справа элемент меньше среднего
         if(i <= j)                // если левая граница не прошла за правую
               {  k = ms[i];       // перестановка элементов
            ms[i] = ms[j];
            ms[j] = k;
            i++;  j--;             // переходим к следующим элементам
         }
    } while(i <= j);               // пока границы не совпали
                                   //  массив разбит на два подмножества
    if(i < r)                      // если есть что-нибудь справа
       hoar(ms,i,r);               // сортируем  правый подмассив
    if(j>l)                        // если есть что-нибудь слева
       hoar(ms,l,j);               // сортируем  левый подмассив
}
```

Еще один вариант программы сортировки Хоора:

```
void hoor(int *a,int l,int r)
{ int i,las;
  if(l>=r)  return;
  swap(a,l,(l+r)/2);      //   делящий эл-т переносится в a[l] (a[l]<->a[(l+r)/2])
  las=l;                   //  позиция последнего элемента большего
                           //  чем делящий
  for(i=l+1;i<=r;i++)   //  деление [l,r] на [l,las-1] и [las+1,r]
  if(a[i]<a[l])  swap(a,++las,i);
  swap(a,l,las);
  hoor(a,l,las-1);        //  сортировка для   [l, las-1]
  hoor(a,las+1,r);        //  сортировка для   [las+1, r]
}

void swap(int *a,int i,int j)
{ int tmp;                // функция замены i и j элементов в массиве a
  tmp=a[i];
  a[i]=a[j];
  a[j]=tmp;
}
```

Структуры

Как отмечалось ранее, массив предназначен для работы с данными, имеющими один тип. Рассматриваемые ниже объекты используются для снятия этого ограничения.

Структуры - это составной объект, в который входят элементы любых типов, за исключением функций. В отличие от массива, который является однородным объектом, структура может быть неоднородной. Для удобства работы с этими данными они сгруппированы под одним именем. При разработке программ структуры помогают в организации хранения и обработке сложных данных, не разобщая их по различным объектам, а группируя в одном. Кроме того, объекты, входящие в структуру сами могут быть структурой, т.е. структуры могут быть вложенными. Структуры позволяют группу связанных между собой переменных использовать как множество отдельных элементов, а также как единое целое. Как и массив, структура представляет собой совокупность данных, но отличается от него тем, что к ее элементам (компонентам) необходимо обращаться по имени и ее элементы могут быть различного типа. Структуры целесообразно использовать там, где необходимо объединить разнообразные данные, относящиеся к одному объекту.

Примером использования структуры может служить, например, строка платежной ведомости. В ней должна содержаться информация о служащем: ФИО, адрес, табельный номер, зарплата и т.д.

Как и другие объекты в С(С++) (переменные, массивы и т.д.) структуры должны быть определены. Для этого создается (объявляется) некоторый тип являющийся структурой, а затем по мере необходимости определяются переменные этого типа (типа структура).

Объявление структуры осуществляется с помощью ключевого слова **struct,** за которым следует имя, называемое **тегом (именем) структуры** и списка элементов, заключенных в фигурные скобки. Тег дает название структуре данного вида и в дальнейшем служит кратким обозначением той части описания структуры, которая заключена в фигурные скобки, то есть является спецификатором. В общем виде определение структуры может быть представлена в следующем виде:

```
struct [имя типа структуры]
{   тип элемента 1 имя элемента 1;
                . . . . . .
    тип элемента n имя элемента n;
}[имя переменной стр-ры];
```

Имена структур должны быть уникальными в пределах их области видимости (действия), для того, чтобы компилятор мог различать различные типы шаблонов.

Именем элемента структуры может быть любой идентификатор. Как и ранее, при описании нескольких идентификаторов одного типа в структуре они могут быть описаны через запятую. Имена элементов структуры (полей) могут совпадать с именами обычных переменных (не являющихся элементами структуры), так как они всегда различимы по контексту. Более того, одни и те же имена элементов могут встречаться в объявлениях различных структур.

Задание только типа не влечет выделения “под него” памяти компилятором. Описание типа структуры предоставляет компилятору

необходимую информацию об элементах структурной переменной для резервирования места в оперативной памяти и организации доступа к ней при определении структурной переменной и использовании отдельных элементов структурной переменной.

Определение переменной имеющей тип структура аналогично определению переменных рассмотренных ранее типов, например:

```
struct inform
{  char fam[30];     // фалилия
   int god;          // год рождения
   float stag;       // стаж работы
};
inform rab;     // определение переменной rab типа struct inform
```

С синтаксической точки зрения эта запись аналогична записи вида:

```
char fam[30];        // фалилия
 int god;            // год рождения
 float stag;         // стаж работы
```

В обоих случаях компилятор выделит под каждую переменную количество байтов памяти, требуемое согласно определения этой переменной (элемента структуры).

Описание шаблона и определение структурной переменной могут быть совмещены в одной записи, например:

```
struct book
{    char  title [20];
     char autor [30];
     int   page;
}  bk1, bk2, *ptr_bk=&bk1;
```

В этом случае если имя структуры (тег) book более ни где не используется, то оно может быть опущено.

Ранее мы уже рассматривали процедуру инициализации переменных и массивов. Как и обычные переменные, элементы, входящие в состав структуры (определяющие структурную переменную), могут быть *инициализированы*. При этом можно инициализировать, как только все элементы, определяющие структурную переменную при ее декларировании:

```
struct st{char c; int i1,i2};
struct st a={‘a’,105,25};
```

так и некоторые из них в процессе работы с ними:

```
a.c=‘a’;
a.i2=25;
```

При выполнении инициализации структурных переменных необходимо следовать некоторым правилам:

- присваиваемые значения должны совпадать по типу с соответствующими полями структуры;
- можно объявлять меньшее количество присваиваемых значений, чем количество полей. Компилятор присвоит нули остальным полям структуры;
- список инициализации последовательно присваивает значения полям

структуры вложенных структур и массивов.

Например:

```
struct comp
{   char nazv[20];
    float speed;
    int cena;
} cp[3]={  {“Celeron”,0.6,525},
           {“Duron”,1.0,547},
           {“Atlon”,1.4,610}
        };
```

Доступ к элементам структуры. Доступ к полям структуры осуществляется с помощью оператора «точка» - при непосредственной работе со структурой и "->" - при использовании указателей на структуру. Возможны три вида спецификации доступа к элементам структуры:

имя_переменной_структуры . имя_поля;

имя_указателя -> имя_поля;

(*имя_указателя) . имя_поля;

например:

```
struct str
{   int i;
    float j;
}  st1, st2[3],*st3,*st4[5];
```

а) *Прямой* доступ к элементу (оператор “ **.**”)

```
st1.i=12;
st2[1].j= .52;
```

б) Доступ по *указателю* (оператор “->”).

```
      st3->i=12;
(st4+1)->j=1.23;
*(st3).j=23;
*(st4[2]).i=123;
```

Шаблон структуры в качестве одного из полей не может содержать переменные своего типа, т.е. неправильным будет следующее объявление шаблона:

```
struct  str
{ char pole_1;
  double pole_2;
  str pole_3;          // ошибка
};
```

Однако структура может *включать* в себя элементы, являющиеся *указателями* на шаблон этой же структуры:

```
struct  str
{ char pole_1;
  double pole_2;
  str *pole_3;         // верно
```

};

Так же как и массив, структура может быть введена поэлементно. Например, для описанной выше структуры st

gets(a.c);

scanf(”% d % f”, &a.i1, &a.i2);

Как отмечалось выше, структуры могут быть вложены дуг в друга:

struct FIO {char F[15], I[10], O[12]} f;

struct rab { struct FIO fio; int tab; float zarpl} rb;

Структура rab содержит структуру FIO. Доступ к элементам структуры FIO и rab осуществляется следующим образом:

f. F="Петров";

f.I =" ";

rb.fio.O= "Иванович";

Единственно возможные операции над структурами - это их копирование, присваивание, взятие адреса с помощью & и осуществление доступа к ее элементам (полям). Следует отметить, что оператор присваивания выполняет то, что называется *поверхностной* копией в применении к структурам-переменным. Поверхностная копия представляет собой копирование бит за битом значений полей переменной-источника в соответствующие поля переменной-приемника. При этом может возникнуть проблема с такими полями, как *указатели*, поэтому использовать поверхностное копирование структур надо осторожно.

Копирование всех полей одной структуры в другую может быть выполнено как поэлементно, так и целиком, как это показано ниже:

```
#include <stdio.h>
struct book
{  char avt[10];
   int izd;
} st1,st2;
void main(void)
{ gets(st1.avt);       // ввод структуры st1
  scanf("%d",&st1.izd);
  st2=st1;             // копия всех полей структуры st1 в st2
}
```

Необходимо отметить, что копирование структур (st2=st1) допустимо, только если st1 и st2 являются структурными переменными соответствующими одному структурному типу.

Структуры нельзя сравнивать, то есть выражение if (st1==st2) неверно. Сравнение структур требуется выполнять поэлементно.

Если данные, объединенные в структуры, должны содержать информацию не об одном объекте, а о некотором их множестве, например, каталог библиотеки, телефонный справочник и т.д., то удобно использовать массивы структур.

Указатели на структуры.

Как и для других структурированных типов данных, указатели могут быть использованы и для структур. Это имеет следующие положительные черты:

- указателями на структуры легче пользоваться, чем самими структурами (например, в задаче сортировки);

- ряд данных удобно представлять в виде структур, полями которых являются указатели на другие структуры.

Рассмотрим пример, показывающий, как определять указатель на структуру и использовать его для работы с элементами структуры.

```
#include <stdio.h>
#define DL 30    // max число символов в строке char
struct name{char fio[DL];};       // ФИО объекта
struct inform{struct name nm;   // ссылка на стр-ру name
              char prof[DL];           // профессия
              int god;                 // год рождения
              float okl;};             // оклад
void main(void)
{ static struct inform mas[2]=
            { "Иванов","конструктор",1965,15000.50,
              "Петров","оператор",1967,12350.50};
  struct inform *him;              // указатель на структуру
  printf("адрес mas[0]: %u   mas[1]: %u \n",&mas[0],&mas[1]);
  him=mas;                         //  him содержит адрес mas[0]
  printf("адрес mas[0]: %u   mas[1]: %u \n",him,him+1);
  printf("him->okl = %.2f   (*him).okl = %.2f \n",him-   >okl, (*him)okl);
  him++;                           //  him содержит адрес следующей стр-ры
  printf("him->okl = %.2f   (*him).okl = %.2f \n",him-   >okl, *him).okl);
  ++him->okl;                      //  увеличение значения поля okl на 1
  printf("him->okl = %.2f   (*him).okl = %.2f \n",him-   >okl, *him).okl);
}
```

Синтаксис, используемый при описании указателя на структуру, такой же, как и для других структурированных данных:

struct имя_структуры * имя_указателя;

Таким образом, раз him - указатель на структуру inform, то *him есть сама структура, а (*him).okl - одно из полей структуры.

В выражении (*him).okl используются скобки, так как приоритет операции (.) выше чем *. Для упрощения доступа к элементам структуры (избежания ошибок, связанных с учетом приоритета) была введена операция -> (операция косвенного получения элемента). Эта операция имеет следующую форму записи:

имя_указателя -> имя_поля_структуры;

Этот оператор состоит из двух знаков - и >. Операция -> имеет наивысший приоритет, наряду с операциями () [ ] и, таким образом, чтобы перейти к полю okl следующей структуры, необходимо записать:

(++him)->okl; или (him++)->okl;

Рассмотрим еще один пример объявления структуры:

```
struct
{  char * str;
   float f;
} *pr;
```

Как и выше, *pr->str есть содержимое объекта, на который ссылается str; *pr->str++ передвинет указатель pr->str после считывания объекта, на который он указывал; (*pr->str) ++ увеличит значение объекта, на который ссылается str; *pr ++->str передвинет указатель pr после получения значения, на которое указывает str.

Структуры и функции

В функцию можно наряду со стандартными типами данных передавать и структуры.

```
struct men                  // обьявление шаблона структуры Men
{ char *String;
  int Number;
  int Marks[10];
};
men sum(men);          //  прототип 1
void sum (men*);       //  прототип 2
```

Выше приведены два прототипа функции. В первом описывается функция, в которую передается копия переменной типа структура men и которая возвратит в основную программу значение того же типа.

Второй прототип описывает функцию в которую передается указатель на структуру. В функции выполнятся преобразования структуры расположенной в памяти по указателю, который в нее передан. Поэтому функция ничего не возвращает. Этот вариант функции лучше нежели первый, он будет работать быстрее и требует меньшего количества памяти.

Существует, по крайней мере, три подхода передачи структуры функции: передавать компоненты по отдельности, передавать всю структуру целиком и передавать указатель на структуру. Каждый из отмеченных подходов имеет свои положительные и отрицательные стороны.

Передача функции компонент структуры. В общем случае, элемент структуры является переменной (массивом) некоторого типа, и может быть передан как аргумент функции. Рассмотрим следующую несложную программу, выполняющую суммирование окладов нескольких профессий, и нахождение max из них:

```
#include <stdio.h>
float summa(float,float *);
struct str
{  char *prof[5];          // профессия
   float okl[5],sm;        // оклад и сумма
} ok=   {"проф1","проф2","3","4","", 1.5,2.06,31.4,3.0,0,0};
```

```
void main(void)
{ float max;
  int i;
  for(i=0;i<5;i++) max=summa(ok.okl[i],&ok.sm);
  printf("max значение =%5.2f\n сумма = %5.2f",max,ok.sm);
}
float summa(float i,float *j)
{ static float max=0;
  *j+=i;                    //вычисление суммы
  max=(max>i)?max:i;  //поиск max значения
  return max;
}
```

Рассматриваемая программа состоит из двух функций: main() и summa(). Функции summa() в качестве параметров передаются один из суммируемых окладов (ok.okl[i]) и адрес элемента структуры, куда помещается сумма (&ok.sm). Функция summa() также возвращает вычисленное максимальное значение оклада.

Передача всей структуры в функцию. В этом случае выполняется передача данных всей структуры в функцию (в стек) со всеми полями, которые в нее входят (массивы, другие структуры и т. д.).

При этом никакие изменения структуры внутри функции не влияют на структуру, используемую в качестве аргумента. При использовании структуры в качестве параметра, тип аргумента должен соответствовать типу параметров. Это можно сделать определив структуру глобально, а затем использовать ее для объявления необходимых структурных переменных и параметров.

```
#include <stdio.h>
float sr_ball(struct str );
struct str
{ char prof[15];
  int oc[4];
};
void main(void)
{ float bl;
  struct str  s= {"Иванов",5,4,3,5};
  bl=sr_ball(s);
  printf("средний балл =%5.2f",bl);
}
float sr_ball(struct str st)
{
  return (st.oc[1]+st.oc[2]+st.oc[3]+st.oc[4])/4;
}
```

Передача структуры через указатель будет рассмотрена позже.

Примеры программ

Пример . Среди абитуриентов, сдавших вступительные экзамены в институт, определить количество абитуриентов, проживающих в городе Минске и сдавших экзамены со средним баллом не ниже 8, распечатать их фамилии в алфавитном порядке.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>
#define n 3
void main(void)
{ struct str
  {    char *gr;          // город
       char fm[15];       //фамилия
       int oc[3];
  } s[n];
 int i,j,k=0;
 char c;
 clrscr();
 printf("\nВведите информацию в массив структур");
 for(i=0;i<n;i++)              // цикл ввода структур в массив
 { printf("\n введите город :");
   fflush(stdin);
   s[i].gr=(char *)malloc(10);  //отводим память для ввода города
   gets(s[i].gr);              //ввод города
   if (!*s[i].gr) break;   // выход по пустой строке
   printf("\n введите фамилию :");
   gets(s[i].fm);              //ввод фамилии
   printf("\n введите оценки (ф  м  л) :");
   scanf("%d%d%d",s[i].oc,s[i].oc+1,s[i].oc+2);
 }
 i--;  // последняя стр-ра  s[i] - пустая или i==n
 for(c='A';c<='Я';c++)
 for(j=0;j<=i;j++)
 if(!strcmp(s[j].gr,"Минск") && s[j].fm[0]==c)
 if((s[j].oc[0]+s[j].oc[1]+s[j].oc[2])/3>=8)
 { printf("\n%s",s[j].fm);
   k++;   // подсчет числа абитуриентов удовл. условию задачи
 }
 printf("\n Итого абитуриентов = %d",k);
}
```

Пример . вложенных структур

```
#include <stdio.h>
#define DL 30    // max число символов в строке char
```

```
struct name
{    char fm[DL];     // фамилия
     char im;         // имя
     char ot;         // отчество
};
struct inform
{   name nm;       // переменная типа на структура name
    char prof[DL];   // профессия
    int god;         // год рождения
    float okl;       // оклад
};
void main(void)
{ static inform mas[4]=
        { "Иванов",'И','И',"конструктор",1989,15000.50,
          "Петров",'П','П',"оператор",1987,12350.50};
  inform *him;                  // указатель на структуру  inform
  printf("адрес mas[0]: %u   mas[1]: %u \n",&mas[0],&mas[1]);
  him=mas;                      // him содержит адрес   mas[0]
  printf("адрес mas[0]: %u   mas[1]: %u \n",him,him+1);
  printf("him->okl = %.2f   (*him).okl = %.2f \n",him->okl,(*him).okl);
  him++;                       // him содержит адрес следующей стр-ры   mas[1]
  (++him)->okl++;              // him->okl==12351.5
  printf("him->okl = %.2f   (*him).okl = %.2f \n",him->okl,(*him).okl);
  him--;
  printf("him->nm.fm = %.20s   (*him).nm.fm = %.20s \n",
                                    (him)->nm.fm,(*--him).nm.fm);
// в этом случае выполняется переход к mas[0]----------┘
// затем вывод ----------------------------┘
}
```

Пример . Нахождение площади прямоугольной фигуры по 2точкам с координатами (x1,y1) и (x2,y2).

```
#include <stdio.h>
#include <conio.h>
struct rect analiz(struct rect); // здесь наличие слова struct обязательно
void swap(int n,struct rect *);

struct point
{   int x;            // структура point описывает координаты
    int y;            // (x  и  y) точки на плоскости
};
struct rect
{   point p1;         // структура rect описывает координаты
    point p2;         // 2 угловых точек (прямоуг-ка, квадрата)
};
```

```
void main(void)
{ rect pt;
  int s;
  do
  { clrscr();
    puts("введите координаты левого нижнего угла фигуры");
    scanf("%d%d",&pt.p1.x,&pt.p1.y);
    puts("введите координаты правого верхнего угла фигуры");
    scanf("%d%d",&pt.p2.x,&pt.p2.y);
    pt=analiz(pt);      // анализ и коррекция координат угловых точек
  } while(pt.p1.x>pt.p2.x && pt.p1.y>pt.p2.y);
  s=(pt.p2.x-pt.p1.x)*(pt.p2.y-pt.p1.y);   // вычисление площади фигуры
  printf("площадь фигуры S= %d",s);
  getch();
}
rect analiz(rect ptt)
{ void (*f)(int,rect *)={swap};       // указатель на функцию   swap
 rect *ptr;
 ptr=&ptt;                          // ptr содержит адрес стр-ры ptt
 if (ptt.p1.x>ptt.p2.x) (*f)(1,ptr);  // x1>x2 выполняем в (*f)  x1<->x2
 if (ptt.p1.y>ptt.p2.y) (*f)(2,ptr);  // y1>y2 выполняем в (*f)  y1<->y2
 return ptt;
}
void swap(int n,rect *ptr)  // замена одноименных координат 2 точек
{ int i;                  // из функции не требуется возврата т.к.
 switch (n)               // замена выполняется по адресу стр-ры ptt
 { case 1 : i=ptr->p1.x; ptr->p1.x=ptr->p2.x; ptr->p2.x=i; break;
   case 2 : i=ptr->p1.y; ptr->p1.y=ptr->p2.y; ptr->p2.y=i; break;
 }
}
```

Поля бит

Существует несколько разновидностей структур. Одна из них - это поля бит. Поле представляет собой последовательность соседних двоичных разрядов (бит) внутри одного целого значения. Каждое поле может иметь тип unsigned int или signed int и размещается в машинном слове целиком, а вся группа полей может выходить за пределы машинного слова. Поле бит – особый тип структуры определяющий какую длину имеет каждый ее элемент. Общий вид объявления полей бит имеет вид:

```
struct имя стуктуры
{   тип имя 1: длина;
            . . .
    тип имя n : длина;
};
```

Наиболее распространенный подход к использованию полей можно

показать на примере объявления следующей структуры:

```
struct pole
{  int p1:1;
   unsigned  p2:2;
   int:6;
   int p3:4
} pl;
```

Объявление такого вида обеспечивает размещение стуктуры (полей бит) в памяти следующим образом. В полях типа signed крайний левый бит является знаковым. Таким образом, поле signed int p:1 может быть использовано для хранения значений -1 и 0, так как любое не нулевое значение поля p будет интерпретироваться как -1. Поле signed int p:2, в отличие от int p:1, может принимать три значения -1,0,1. В объявлении структуры имеется еще два поля (int:6; int ёр3:4). Первое указывает, что структура содержит 6 неиспользуемых бит, второе предназначено для чисел в диапазоне от –7 до 7.

В Borland C самый левый бит является знаковым. Поля могут не иметь имени; с помощью безымянного поля (задаваемого только двоеточием и шириной) организуется пропуск требуемого количества разрядов. Ширина равная нулю, используется тогда когда необходимо выйти на границу следующего слова.

Все особенности, связанные с использованием полей, например: может ли поле байт перейти границу слова, зависят от аппаратной реализации.

При определенном удобстве работа с полями может повлечь некоторые трудности. Они связаны, например, с тем, что на одних машинах поля размещаются слева направо, на других - справа налево. Это в свою очередь влечет некоторые трудности при перенесении программ. Поля не могут быть массивами и не имеют адресов и, следовательно, операция & к ним не применима.

Объединения

Объединение представляет собой структурированную переменную с несколько иным способом размещения элементов в памяти. Если в структуре (как и в массиве) элементы расположены последовательно друг за другом, то в объединении "параллельно". То есть для их размещения выделяется одна общая память, в которой они перекрывают друг друга и имеют в ней общий адрес. Размерность ее определяется максимальной размерностью элемента объединения. Таким образом, при использовании объединений появляется возможность работы с данными различных типов и размеров в одной и той же области памяти. Синтаксис объединения полностью совпадает с синтаксисом структуры, только ключевое слово struct заменяется на union:

Спецификация доступа к элементам объединения, как и у полей бит, полностью соответствует тому, как это осуществляется у обычных структур:

имя_объединения . имя поля объединения;

или при работе с указателями:

имя_указателя _на_объединение-> имя_поля_объединения;

Рассмотрим пример объявления и некоторых вариантов использования объединения:

```
union un_type
{  int i;          // пример объявления
   double d;    // шаблона объединения
   char c;
};
un_type un;              // переменная типа объединения
un_type  un_mas[6]; // массив из шести элементов
un_type *pt;             // указатель на переменную типа un_type
```

Первое описание приводит к выделению памяти под одну переменную un. При этом компилятор выделяет достаточно памяти для размещения max из переменных описанных в шаблоне un_type, т.е. 8 байт (тип double). Для массива un_mas[6] выделяется 6x8 байт памяти. Механизм использования объединения может быть рассмотрен на следующем примере:

```
un.i=14;          // 14 записывается в переменную un; исп-ся 2 байта
un.d=3.4;         // 14 стирается, 3.4 записывается; исп-ся 8 байтов
un.c='k';         // 3,4 стирается, записывается k; используется 1 байт
```

В каждый момент времени запоминается только одно значение; нельзя записать char и int одновременно в одно объединение даже, если для этого достаточно памяти. Приведенный ниже фрагмент демонстрирует ошибочное использование полей объединения:

```
un.d=2.4;
un.i=14;
f=5.2*un.d;              // ошибка
```

Ошибка заключается в том, что в момент использования un.d его значение 2.4 затерто оператором un.i=14;

Как отмечалось выше, с объединениями можно использовать операцию -> аналогично, как это делалось со структурами.

```
pt=&un;
f=pt->d;          // аналогично f=un.d
```

Объединение может быть использовано не только для экономии памяти. Если записать в один элемент объединения некоторое значение, то через другой элемент это же значение можно прочитать уже в другой форме представления Таким образом форму представления данных в памяти можно менять совершенно свободно.

Переменные с изменяемой структурой

Очень часто некоторые объекты программы относятся к одному и тому же классу, отличаясь лишь некоторыми деталями. Рассмотрим, например, представление геометрических фигур. Общая информация о фигурах может включать такие элементы, как площадь, периметр. Однако соответствующая информация о геометрических размерах может оказаться различной в зависимости от их формы.

Рассмотрим пример, в котором информация о геометрических фигурах

представляется на основе комбинированного использования структуры и объединения.

```
struct figure
{   double area,perimetr;   // общие компоненты
    int type;               // признак компонента
    union                   // перечисление компонент
    {  double radius;       // окружность
       double a[2];         // прямоугольник
       double b[3];         // треугольник
    } geom_fig;
}  fig1, fig2;
```

В общем случае каждый объект типа figure будет состоять из трех компонентов: area, perimetr, type. Компонент type называется меткой активного компонента, так как он используется для указания, какой из компонентов объединения geom_fig является активным в данный момент. Такая структура называется переменной структурой, потому что ее компоненты меняются в зависимости от значения метки активного компонента (значение type).

Отметим, что вместо компоненты type типа int, целесообразно было бы использовать перечисляемый тип. Например, такой

enum figure_chess { CIRCLE, BOX, TRIANGLE } ;

Константы CIRCLE, BOX, TRIANGLE получат значения соответственно равные 0, 1, 2. Переменная type может быть объявлена как имеющая перечислимый тип :

enum figure_chess type;

В этом случае компилятор СИ предупредит программиста о потенциально ошибочных присвоениях, таких, например, как

figure.type = 40;

В общем случае переменная структуры будет состоять из трех частей: набор общих компонент, метки активного компонента и части с меняющимися компонентами. Общая форма переменной структуры, имеет следующий вид:

```
struct имя_структуры
{   общие компоненты;
    метка активного компонента;
    union
    { описание компоненты 1 ;
      описание компоненты 2 ;
              .    .    .
       описание компоненты n ;
    } идентификатор-объединения ;
} идентификатор-структуры ;
```

Пример определения переменной структуры:

```
struct inform
```

```
{           // общая информация
    char  name [25];              //  имя
    int   age;                    //  возраст
          // метка активного компонента;
    otdel otd;                    //  номер отдела
          // переменная часть
    union
    {  struct          //  работает
       {  char dolgnost[8];       // должность
          int oklad;              // оклад
       }  work_info;
       struct        //  уволен
       {  int  date;              // дата увольнения
       }  uvol_info;
    }  var_info;
} rabot;
```

Обращаться к компонентам структуры можно:
rabot.name,
rabot.otd,
rabot.var_info.work_info.oklad.
rabot.var_info.uvol_info.date.

Организация списков и их обработка

Динамические переменные чаще всего реализуются как связанные структуры, списки.

Список – это набор динамических элементов (чаще всего структурных переменных) связанных между собой каким либо способом.

Списки бывают линейными и кольцевыми, односвязными и двусвязными.

Динамические переменные, как правило, имеют тип “структуры”, так как должны содержать, помимо значения (целого, вещественного и т.д.), ссылку на другую динамическую переменную. Чтобы связать динамические переменные в цепочку, надо в каждой компоненте иметь ссылку на предыдущую компоненту, так как это не массив и различные компоненты могут быть размещены в произвольных областях памяти.

Описание структуры и указателя в этом случае может иметь вид:

```
typedef struct element      //  структура элемента хранения
{  float  val;              //  элемент списка
   struct snd *n ;          //  указатель на элемент хранения
} SPIS;
SPIS *p;                    //  указатель текущего элемента
SPIS *dl;                   //  указатель на начало списка
```

Для выделения памяти под элементы хранения необходимо пользоваться функцией malloc(sizeof(element)) или calloc(l,sizeof(element)). Формирование списка в связанном хранении может осуществляется операторами:

```
p=malloc(sizeof(element);
p->val=10;   p->n=NULL;
dl=malloc(sizeof(element));
dl->val=7;   dl->n=p;
```

В последнем элементе хранения (конец списка) указатель на соседний элемент имеет значение NULL. Получаемый список изображен на рис.9.

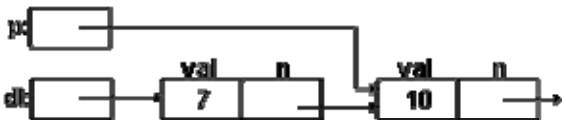


Рис. 9 . Связное хранение линейного списка.

При работе со списками на практике чаще всего приходится выполнять следующие операции:

- поиск элемента с заданным свойством;
- определение первого элемента в линейном списке;
- добавление нового элемента до или после указанного узла;
- исключение определенного элемента из списка;
- упорядочивание узлов линейного списка.

Ниже приведены схемы демонстрирующие некоторые из операций над списками.

Операции со списками при связном хранении

Удаление элемента, следующего за узлом, на который указывает p (см. рис. 10). Исключение элемента из цепочки данных сводится к изменению одной единственной ссылки и не требует перемещения остальных элементов, как это имело бы место, например, для массивов.

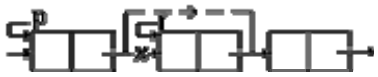


Рис. 10. Схема удаления элемента из списка.

Добавление нового элемента в цепочку данных. Для этого достаточно изменить одну ссылку. Ниже показана вставка нового узла со значением new за элементом, определенным указателем p (см. рис. 11):

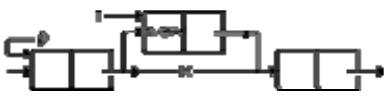


Рис. 11. Схема вставки элемента в список.

Стек

Стек - это конечная последовательность некоторых однотипных

элементов - скалярных переменных, массивов, структур или объединений, среди которых могут быть и одинаковые. Размещение новых элементов в стек и удаление существующих из стека производится только с одного его конца, называемого вершиной стека.

Стек предполагает добавление и удаление из него элементов так, что стек является динамической, постоянно меняющейся структурой. Согласно определению, в стеке имеется только одно место для размещения новых элементов - его вершина. Последний размещаемый элемент находится в вершине стека. Стек иногда называется списком с организацией LIFO - ”последний размещенный извлекается первым”. Если необходимо поддерживать информацию о промежуточных состояниях стека, то она должна размещаться вне стека. Внутри самого стека эта информация не поддерживается.

Допустимыми операциями над стеком являются:

- проверка стека на пустоту;
- добавление нового элемента на вершину стека;
- удаление элемента в вершины стека.

Стек можно представить как стопку книг на столе, где добавление или взятие новой книги возможно только сверху.

Ниже приводится текст программы, иллюстрирующий работу со стеком:

```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>
#include <string.h>

struct zap {char inf[50];
            zap *l;};
void push(zap **);
void pop(zap **);
void see(zap *);
void sort1(zap *);
void sort2(zap **);
void (*f[])(zap **)={push,pop,sort2};
void (*ff[])(zap *)={see,sort1};

main()
{ zap *s;
  char l;
  clrscr();
  s=NULL;
  while(1)
  {  puts("\nвид операции:\n  1-создать/добавить\n  2-удалить");
     puts("  3-просмотреть\n  4-сортировка 1\n  5-сортировка 2");
     puts("  0-окончить");
     fflush(stdin);
```

```
      l=getch();
      switch(l)
      { case '1': (*f[0])(&s); break;
        case '2': if(s) (*f[1])(&s); break;
        case '3': (*ff[0])(s); break;
        case '4': if (s) (*ff[1])(s); break;
        case '5': if (s) (*f[2])(&s); break;
        case '0': return;
        default: printf("Ошибка, повторите \n");
      }
   }
   return 0;
}

// функция создания/добавления в стек
void push(zap **s)
{ zap *s1=*s;
   do
   {  if(!(*s=(zap *) calloc(1,sizeof(zap))))
      { puts("Нет памяти");
        return;
      }
      puts("Введите информацию в inf");
      fflush(stdin);
      gets((*s)->inf);
      (*s)->l=s1;
      s1=*s;
      puts("Продолжить (y/n)");
      fflush(stdin);
   } while(getch()=='y');
}

// функция просмотра элементов стека
void see(zap *s)
{ zap *s1;
   s1=s;
   if (!s)
   {  puts("Стек пуст");
      return;
   }
   do
   {  printf("%s\n",s1->inf);
      s1=s1->l;
   } while(s1);
   puts("Вывод стека закончен");
```

```
}

// функция удаления последнего элемента стека
void pop(zap **s)
{ zap *s1;
  s1=*s;
  *s=(*s)->l;
  free(s1);
  puts("Последний элемент стека удален");
  return;
}

// функция сортировки элементов стека
// перестановкой содержимого элементов стека
void sort1(zap *s)
{ zap *s1,*s2;
  for(;s->l;s=s->l)
  { s1=s;                          // элемент стека для упорядочивания
    for(s2=s->l;s2;s2=s2->l)  // перебор последующих эл-тов справа от i-го
    if(strcmp(s1->inf,s2->inf)>0)
    s1=s2;                  // найдено новое меньшее значение
                            // по новому адресу
    if(s1!=s)
    { strcpy(s2->inf,s1->inf);
      strcpy(s1->inf,s->inf);
      strcpy(s->inf,s2->inf);
    }
  }
  puts("Сортировка стека выполнена");
}

// функция сортировки элементов стека
// перестановкой адресов элементов стека
void sort2(zap **s)
{ zap *ss,*s1,*s2,*s3,*s4=NULL;
  ss=( zap *) calloc(1,sizeof(zap));
  ss->l=*s;                        // ссылка на вершину стека
  for(;ss->l->l;)
  { s1=ss->l;                      // элемент стека для упорядочивания
    for(s2=ss->l; s2->l; s2=s2->l)   // перебор последующих эл-тов стека
    if(strcmp(s1->inf,s2->l->inf)>0) // найдено новое меньшее значение
    { s1=s2->l;                  // адрес текущего минимального эл-та
      s3=s2;                     // адрес элемента перед минимальным
    }
    if(s1!=ss->l)
```

```
    { s3->l=s1->l;
      s1->l=ss->l;
      ss->l=s1;
    }
    ss=ss->l;
    if(!s4) s4=s1;          //новая вершина стека
  }
  puts("Сортировка стека выполнена");
  *s=s4;
}
```

Построение обратной польской записи

Одной из главных причин, лежащих в основе появления языков программирования высокого уровня, явились вычислительные задачи, требующие больших объёмов рутинных вычислений. Поэтому к языкам программирования предъявлялись требования максимального приближения формы записи вычислений к естественному языку математики. В этой связи одной из первых областей системного программирования сформировалось исследование способов трансляции выражений. Здесь получены многочисленные результаты, однако наибольшее распространение получил метод трансляции с помощью обратной польской записи, которую предложил польский математик Я.Лукашевич.

Арифметическое выражение вида:

(A+B)*(C+D)-E

в форме обратной польской записи будет иметь вид:

AB+CD+*E-

Характерные особенности этого выражения состоят в следовании символов операций за символами операндов и в отсутствии скобок.

Обратная польская запись обладает рядом замечательных свойств, которые превращают ее в идеальный промежуточный язык при трансляции. Во-первых, вычисление выражения, записанного в обратной польской записи, может проводиться путем однократного просмотра, что является весьма удобным при генерации объектного кода программ. Во-вторых, получение обратной польской записи из исходного выражения может осуществляться весьма просто на основе простого алгоритма, предложенного Дейкстрой. Для этого вводится понятие стекового приоритета операций(табл. 10.):

Таблица 10.

Операция	Приоритет
(	0
)	1
+ -	2
* /	3

Просматривается исходная строка символов слева направо, операнды переписываются в выходную строку, а знаки операций заносятся в стек на

основе следующих соображений:

а) если стек пуст, то операция из исходной строки переписывается в стек;

б) операция выталкивает из стека все операции с большим или равным приоритетом в выходную строку;

в) если очередной символ из исходной строки есть открывающая скобка, то он проталкивается в стек;

г) закрывающая круглая скобка выталкивает все операции из стека до ближайшей открывающей скобки, сами скобки в выходную строку не переписываются, а уничтожают друг друга.

Рассмотрим текст программы, выполняющей рассмотренные выше действия.

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
struct st                // Описание элемента стека
{  char c;
   st *next;
};
st *push(st *,char);      // занесение в стек
char pop(st **);          //  чтение из стека
int PRIOR(char);          // возвращает приоритет операции

void main(void)
{
  st *OPR=NULL;           // стек операций пуст
  char in[80],out[80];    // входная и выходная строки
  int k,point;
  do
  {  puts("Введите выражение(в конце '='):");
     fflush(stdin);
     gets(in);                              // ввод арифметического выражения
     k=point=0;
     while(in[k]!='\0' && in[k]!='=') // пока не дойдем до '='
     {  if(in[k]==')')                      // если очередной символ - ')'
        { while((OPR->c)!='(')              // то удаляем из стека в
          out[point++]=pop(&OPR);   // выходную строку все знаки операций
                                             //  до открывающей скобки
          pop(&OPR); // Удаляем из стека открывающую скобку
        }
        if(in[k]>='a' && in[k]<='z')    // если символ - буква ,то
        out[point++]=in[k];             // заносим её в выходную строку
        if(in[k]=='(')                  // если очередной символ - '(' ,то
        OPR=push(OPR,'(');              // заносим её в стек
        if(in[k]=='+' || in[k]=='-' || in[k]=='/' || in[k]=='*')
        {               //  Если следующий символ - знак операции ,то все на-
```

```
                    //  ходящиеся в стеке операции с большим или равным
                    //  приоритетом переписываются в выходную строку
          while((OPR!=NULL)&&(PRIOR(OPR->c)>=PRIOR(in[k])))
          out[point++]=pop(&OPR);
          OPR=push(OPR,in[k]); // запись в стек новой операцию
        }
        k++;        // переход к следующему символу входной строки
    }
    while(OPR!=NULL)          // после рассмотрения всего выражения
    out[point++]=pop(&OPR); // перезапись операции
    out[point]='\0';                // из стека в выходную строку
    printf("\n%s\n",out);          // и печатаем её
    fflush(stdin);
    puts("\nПовторить(y/n)?");
 } while(getch()!='n');
}
 // push записывает в стек символ a, возвращает
 // указатель на новую вершину стека
st *push(st *head,char a)
{  st *PTR;
   if(!(PTR=(st*) malloc(sizeof(*PTR))))
   {  puts("Нет памяти"); exit(-1);}
      PTR->c=a;            //
      PTR->next=head;   //
      return PTR;           // PTR -новая вершина стека
   }
 // pop   удаляет символ с вершины стека. Возвращает
 // удаляемый символ. Изменяет указатель на вершину стека
char pop(st **head)
{  st *PTR;
   char a;
   if(!(*head)) return '\0';//Если стек пуст, возвращаем \0
   PTR=*head;             // в PTR - адрес вершины стека
   a=PTR->c;
   *head=PTR->next;     // Изменяем адрес вершины стека
   free(PTR);              // Освобождение памяти
   return a;                  // Возврат символа с вершины стека
}

 // Функция PRIOR возвращает приоритет арифм. операции
int PRIOR(char a)
{  switch(a)
   { case '*':case '/':return 3;
     case '-':case '+':return 2;
     case '(':return 1;
```

```
  } return 0;
}
```

Односвязный линейный список, очередь

Односвязный список (очередь) - такая упорядоченная структура данных, которые могут удаляться с одного ее конца (**начало** очереди), а добавляются в другой ее конец (**конец** очереди).

Очередь организована по принципу FIFO - ”первый вошел - первый вышел”. Для очереди определены следующие операции:

- проверка очереди на пустоту;
- добавление нового элемента в конец (хвост) очереди;
- удаление элемента из начала (головы) очереди;
- поиск и модификация элемента в очереди.
- упорядочивание элементов очереди.

Ниже приводится текст программы, иллюстрирующий работу с однонаправленной очередью.

```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>
#include <string.h>
struct zap
{  char inf[50];
   zap *nx;
};
void add(zap **);
void del(zap **);
void del_any(zap **,char *);
void see(zap *);
void sort(zap **);

main()
{ zap *h,*t;
  char l,*st;
  st=(char *)malloc(10);
  h=NULL;
  clrscr();
  while(1)
  { puts("вид операции: 1-создать очередь");
    puts("              2-вывод содержимого очереди");
    puts("              3-удаление элемента из очереди");
    puts("              4-удаление любого элемента из очереди");
    puts("              5-сортировка очереди");
    puts("              0-окончить");
    fflush(stdin);
    switch(getch())
```

```
    { case '1': add(&h); break;
      case '2': see(h); break;
      case '3': if(h) del(&h); break;
      case '4': if(h)
                { fflush(stdin);
                  puts("Введите информацию для удаления ");
                  gets(st);
                  del_any(&h,st);
                } break;
      case '5': if (h) sort(&h); break;
      case '0': return;
      default: printf("Ошибка, повторите \n");
    }
  }
}

// функция создания очереди
void add(zap **h)
{ zap *n;
  puts("Создание очереди \n");
  do
  { if(!(n=(zap *) calloc(1,sizeof(zap))))
    { puts("Нет свободной памяти");
      return;
    }
    puts("Введите информацию в inf");
    scanf("%s",n->inf);
    if (!*h)            // очередь еще не создана
        *h=n;           // хвост очереди
    else
    {  n->nx=*h;    // очередной элемент очереди
      *h=n;           // передвигаем указатель на хвост
    }
    puts("Продолжить (y/n): ");
    fflush(stdin);
  } while(getch()=='y');
}

// функция вывода содержимого очереди
void see(zap *h)
{ puts("Вывод содержимого очереди \n");
  if (!h)
  { puts("Очередь пуста");
    return;
  }
```

```
  do
  { printf("%s\n",h->inf);
    h=h->nx;
  } while(h);
   return;
}

// функция удаления первого элемента очереди
void del(zap **h)
{ zap *f,*pr;
  if (!(*h)->nx)          // в очереди только один элемент
  { free(*h);
    *h=NULL;              // очередь пуста
    return;
  }
  pr=*h;
  while(pr->nx->nx)
  pr=pr->nx;
  free(pr->nx);          // удаление первого элемента из очереди
  pr->nx=NULL;
  return;
}

// функция удаления любого элемента очереди
void del_any(zap **h,char *st)
{ zap *f,*pr;
  if (!(*h)->nx &&        // в очереди только один элемент
     (!strcmp((*h)->inf,st) || *st=='\0'))
  { free(*h);
    *h=NULL;              // очередь пуста
     return;
  }
  pr=*h;
  f=pr;                   // далее f предыдущий к  pr элемент
  while(pr)
  { if (!strcmp(pr->inf,st)) // найден элемент со строкой  st
    {  if (pr==*h) {*h=pr->nx; free(pr); f=pr=*h;}
       else
       { f->nx=pr->nx;   // обходим элемент  pr
         free(pr);           // удаляем элемент  pr очереди
         pr=f->nx;           // выбор следующего элемента очереди pr
       }
    }
    else {f=pr; pr=pr->nx;}  // переход к следующему элементу
  }
```

```
}

// функция сортировки содержимого очереди
void sort(zap **h)
{ zap *s1=*h,*s2,*s3,*s4,*hh=NULL;
  s1=*h;                                  // ссылка на хвост очереди
  s4=( zap *) calloc(1,sizeof(zap));
  for(; s1->nx; s1=s1->nx)                // выбор исходного элемента очереди
  { for(s2=s1->nx; s2; s3=s3->nx, s2=s2->nx) // перебор последующих за S1
    { if (s2==s1->nx) s3=s1;              // S3-элемент расположенный перед S2
      if(strcmp(s1->inf,s2->inf)>0)  // найдено новое меньшее значение
      { s3->nx=s2->nx;
        s2->nx=s1;                        // элемент с min становится перед S1
        s4->nx=s2;                        // S4- элемент расположенный перед S1
        s1=s2;                            // новый адрес S1- (после замены S1<->S2)
      }
    }
    if(!hh)
    { hh=s1;          // модификация текущего указателя на хвост очереди
      free(s4);
    }
    s4=s1;
  }
  *h=hh;              // возврат возможно измененного указателя на хвост
   puts("Сортировка выполнена");
}
```

Двусвязный линейный список

Связанное хранение линейного списка называется списком с двумя связями или двусвязным списком, если каждый элемент хранения имеет два компонента указателя (ссылки на предыдущий и последующий элементы линейного списка).

В программе двусвязный список можно реализовать с помощью описаний:

```
typedef struct ndd
{ float val;            //  значение элемента
  struct ndd * n;     //  указатель на следующий элемент
  struct ndd * m;    //  указатель на предыдующий элемент
} NDD;
NDD  *dl, *p, *r;
```

Графическая интерпретация списка с двумя связями приведена на рис. 12.

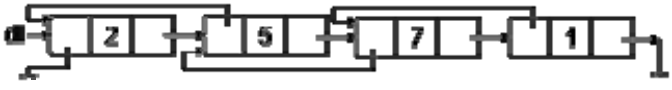


Рис. 12 . Схема хранения двусвязного списка.

Вставка нового узла со значением new за элементом, определяемым указателем p, осуществляется при помощи операторов:

```
r=malloc(NDD);
r->val=new;
r->n=p->n;
(p->n)->m=r;
p->=r;
```

Удаление элемента, следующего за узлом, на который указывает p

```
p->n=r;
p->n=(p->n)->n;
( (p->n)->n )->m=p;
free(r);
```

Ниже приводится текст программы, иллюстрирующий работу с двунаправленным списком.

```
#include <stdio.h>
#include <alloc.h>
#include <conio.h>
#include <string.h>
struct zap
{ char inf[50];
    zap *pr;
    zap *nx;
};
void add(zap **, zap **);
void ins(zap **, char *);
void del_any(zap **, zap **, char *);
void see(zap *, int);
void sort(zap **, zap **);

void main(void)
{ zap *h,*g;        // указатели на хвост и голову очереди
  char l,*st;
  st=(char *)malloc(10);
  h=g=NULL;
  clrscr();
  while(1)
  { puts("вид операции: 1-создать очередь");
    puts("                2-вывод содержимого очереди");
    puts("                3-вставка нового элемента в очередь");
    puts("                4-удаление любого элемента из очереди");
    puts("                5-сортировка очереди");
```

```
    puts("                    0-окончить");
    fflush(stdin);
    switch(getch())
    { case '1': add(&h,&g); break;
      case '2': clrscr();
                puts("0-просмотр с хвоста\n1- просмотр с головы");
                l=getch();
                if(l=='0') see(h,0);
                else see(g,1);
                break;
      case '3': if(h)
                { gets(st);
                  ins(&h,st);
                } break;
      case '4': if(h)
                { gets(st);
                  del_any(&h,&g,st);
                } break;
      case '5': if (h) sort(&h,&g); break;
      case '0': return;
      default: printf("Ошибка, повторите \n");
    }
  }
}

// функция создания очереди  и добавления (только) в хвост очереди
void add(zap **ph,zap **pg)
{ zap *n;
  puts("Создание очереди \n");
  do
  {  if(!(n=(zap *) calloc(1,sizeof(zap))))
     { puts("Нет свободной памяти");
       return;
     }
     puts("Введите информацию в inf");
     scanf("%s",n->inf);
     if (!*ph || !*pg)  // очередь еще не создана
     {  *ph=n;          // указатель на хвост очереди
        *pg=n;          // указатель на голову очереди
     }
     else
     {  n->nx=*ph;         // указатель на элемент (хвост) очереди
        (*ph)->pr=n;       // хвост указывает на новый элемент
       *ph=n;             // передвигаем указатель хвоста на новый элемент
     }
```

```
    puts("Продолжить (y/n): ");
    fflush(stdin);
  } while(getch()=='y');
}

// функция вывода содержимого очереди
// вывод начинать с хвоста (i==0) или головы (i==1)
void see(zap *p,int i)
{  puts("Вывод содержимого очереди \n");
   if (!p)
   { puts("Очередь пуста");
     return;
   }
   do
   {  printf("%s\n",p->inf);
      if(!i) p=p->nx;
      else p=p->pr;
  } while(p);
  return;
}

// функция добавления элемента в очередь  (до головы очереди)
// добавление работает правильно только если очередь упорядочена
// с хвоста по возрастанию  [ (хвост) 1 2 5 7 9 (голова)  вставить 4 ]
void ins(zap **ph,char *s)
{ zap *p=*ph,*n;
  while(p && strcmp(p->inf,s)<0) p=p->nx;
  if(!(n=(zap *) calloc(1,sizeof(zap))))
  { puts("Нет свободной памяти");
    return;
  }
  strcpy(n->inf,s);   //  копирует s n->inf
  p->pr->nx=n;        //  предыдущий элемент очереди указывает
                      //  на вводимый элементт (n)
  n->nx=p;            //  новый элемент указывает на следующий
                      //  элемент очереди p
  n->pr=p->pr;        //  новый элемент указывает на предыдующий
                      //  элемент очереди p
  p->pr=n;            //  последующий элемент очереди указывает на
                      //  вводимый элемент
}

// функция удаления любого одного элемента очереди
// p1- указатель на хвост  p2- на голову очереди
void del_any(zap **p1,zap **p2,char *st)
```

```
{ zap *ph=*p1;
  if (!*p1 || !p2)
  { puts("Очередь пуста");
    return;
  }
  if (ph->nx==NULL &&       // в очереди только один элемент
    (!strcmp(ph->inf,st) || *st=='\0'))
  { free(ph);
    *p1=*p2=NULL;       // очередь пуста
     return;
  }
  while(ph && strcmp(ph->inf,st)) ph=ph->nx;
  if (!strcmp(ph->inf,st))  // найден элемент со строкой  st
  { if(ph==*p1)             // удаляемая вершина - хвост очереди
    { *p1=(*p1)->nx;        // новый указатель на хвост очереди
      (*p1)->pr=NULL;
    }
    else if(ph==*p2)                // удаляемая вершина - голова очереди
          { *p2=(*p2)->pr;          // новый указатель на голову очереди
            (*p2)->nx=NULL;
          }
          else
          { ph->pr->nx=ph->nx;   // обходим элемент  pr
            ph->nx->pr=ph->pr;
          }
    free(ph);                       // удаляем элемент  pr  очереди
  }
}

// функция сортировки содержимого очереди
void sort(zap **ph,zap **pg)
{ zap *s1,*s2,*s3;

 for(s2=*pg ;s2; s2=s2->pr)
 for(s1=*ph; s1->nx; s1=s1->nx)
 { if(strcmp(s1->nx->inf,s1->inf)>0)
   {  s3=s1->nx;                    // s3- вершина следующая за s1
      if(!s1->pr) *ph=s1->nx;       // модификация хвоста очереди
      s1->nx=s1->nx->nx;            // s1-nx указывает через вершину
      if(s1->nx) s1->nx->pr=s1;     // s1->nx выше был модифицирован
      else s2=*pg=s1;               // s1->nx==NULL -коррекция головы
      s3->pr=s1->pr;
      s3->nx=s1;
      if (s3->pr) s3->pr->nx=s3;
      s1->pr=s3;
```

```
      s1=s1->pr;                    // откат для s1=s1->nx в цикле for
    }
   }
   puts("Сортировка выполнена");
}
```

Циклический список, кольцо

Связанное хранение линейного списка называется циклическим списком или кольцом, если его последний элемент указывает на первый элемент списка, а указатель на список - на последний элемент списка.

Двусвязный циклический список

Как и в случае линейного списка, элементы циклического списка могут иметь несколько ссылок. Текст программы, иллюстрирующий работу с двунаправленным циклическим списком, приведен ниже.

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <conio.h>
#include <dos.h>
struct zap
{ char inf[50];
  zap *l,*r;
};
void see(zap *);
zap *sozd(zap *);
zap *del(zap *);
void srt(zap *);
zap *srt_uk(zap *);
zap * napr(char,zap *);

void main(void)
{ zap *s=NULL;
  char l;
  clrscr();
  while(1)
  { puts("вид операции: 1-создать кольцо");
    puts("              2-вывод содержимого кольца");
    puts("              3-удаление элемента из кольца");
    puts("              4-сортировка (изменяя указатели на элементы)");
    puts("              5-сортировка (изменяя содержимое элементов)");
    puts("              0-выход");
    fflush(stdin);
```

```
   switch(getch())
   { case '1': s=sozd(s); break;
     case '2': if(s) see(s);
                  else puts("Кольцо пустое"); getch(); break;
     case '3': if(s) s=del(s); break;
     case '4': if(s) s=srt_uk(s); break;
     case '5': if(s) srt(s); break;
     case '0': return;
   }
   clrscr();
  }
}

// функция создания/добавления в кольцо
// добавление выполняется вправо от элемента входа в кольцо
zap *sozd(zap *s)
{ zap *s1,*s2;
 if (!s)
 { if(!(s=(zap *) malloc(sizeof(zap))))
   { puts("Нет свободной памяти");
     return NULL;
   }
   puts("Введите информацию в inf");
   scanf("%s",s->inf);
   s->l=s;
   s->r=s;
   s1=s;
 }
 else s1=s->r;      // кольцо уже существует
 do
 { if((s2=(zap *) calloc(1,sizeof(zap)))==NULL)
   { puts("Нет свободной памяти");
     return NULL;
   }
   puts("Введите информацию в inf");
   scanf("%s",s2->inf);
   s1->l=s2;
   s2->r=s1;
   s1=s2;
   puts("Продолжить (y/n): ");
   fflush(stdin);
 } while(getch()=='y');
 s2->l=s;
 s->r=s2;
 return(s);
```

```
}

// функция вывода содержимого кольца
void see(zap *s)
{ zap *s1;
  char p;
  s1=s;
  puts("r - по часовой, l - против часовой\n");
  fflush(stdin);
  switch(p=getch())
  { case 'r': case 'R':
    case 'l': case 'L':
              do
              { printf("%s\n",s1->inf);
                 s1=napr(p,s1);
              } while(s1!=s); break;
  }
  puts("Вывод кольца закончен");
  return;
}

// функция удаления элемента кольца
zap *del(zap *s)
{ zap *s1;
  char in[50];
  s1=s;
  puts("Введите информацию в inf");
  scanf("%s",in);
  do
  { if(strcmp(s1->inf,in)) s1=s1->r;
    else
    { if (s1->r==s1)    // элемент в кольце один
          { free(s1); return NULL; }
      if(s==s1) s=s->r; // удаляемый эл-т - точка входа
      s1->l->r=s1->r;   // исключаем вершину из кольца
      s1->r->l=s1->l;
      free(s1);
      return s;
    }
  } while(s1!=s);
  printf("Записи с информацией = %s в кольце нет \n",in);
  getch();
//  return s;
}
```

```
// сортировка информации в кольце
void srt(zap *s)
{zap *s1,*s2;
 char a[50],c;
 int i;
 puts("направление r - по часовой, l - против часовой\n");
 fflush(stdin);
 c=getch();
 s1=s;                    // исходный элемент для замены
 do
 { strcpy(a,s1->inf);          // исходная информ. для замены
  s2=s1;
  do
  { s2=napr(c,s2);            //
   if(strcmp(a,s2->inf)>0)
   { strcpy(s1->inf,s2->inf);
     strcpy(s2->inf,a);
     strcpy(a,s1->inf);
   }
  } while(napr(c,s2)!=s);
  s1=napr(c,s1);;
 } while(napr(c,s1)!=s);
}

// сортировка информации в кольце (методом "отбора")
// рассмотрен только случай сортировки вправо (по возрастанию)
 zap * srt_uk(zap *s)
{ zap *s1,*s2,*s3;
 int i;
 s1=s;
 do
 { s2=s1->r; s3=s1;      // s3 адрес элемента кольца с min значением
  do                     // блок отбора меньшего (его адрес в s3)
  { if(strcmp(s3->inf,s2->inf)>0)
    s3=s2;               // s3 премещаем на узел с меньшей строкой
    s2=s2->r;            // поиск далее по кольцу
  } while(s2!=s);        // пока не прошли по кольцу
  if(s3!=s1)
  {   if(s==s1) s=s3;   // новая точка входа в кольцо
      s3->l->r=s3->r;  // исключение элемента s3
      s3->r->l=s3->l;  // из кольца
      s1->l->r=s3;      // вставляем
      s3->r=s1;         // элемент s3
      s3->l=s1->l;      //  перед
      s1->l=s3;         //  s1
```

```
    }
    else
    s1=s1->r;
  } while(s1->r!=s);
  return s;
}

zap * napr(char c,zap * s)
{ switch(c)
  { case 'r': case 'R': return s->r;
    case 'l': case 'L': return s->l;
  }
}
```

Примеры программ

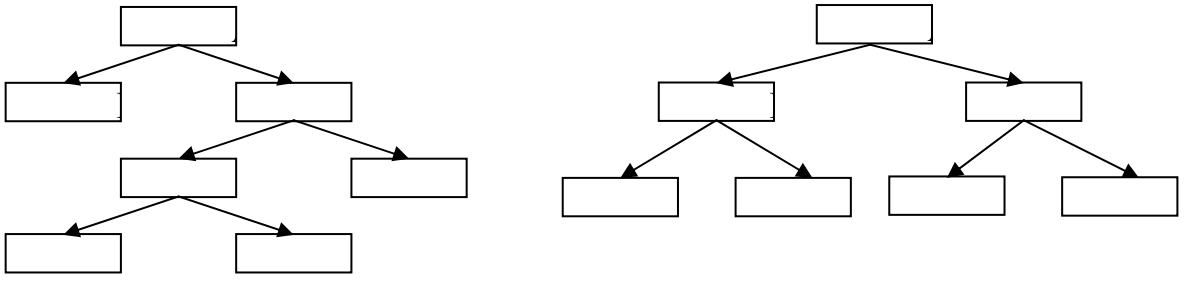
Пример . Требуется ввести некоторую последовательность символов, заканчивающуюся нажатием клавиши Enter и напечатать ее в обратном порядке (т.е. если на входе будет "строка a1b2-3j" то на выходе должно быть "3j-2b1a акортс". В приводимой ниже программе все символы вводимой последовательности, записываются в стек, а затем содержимое стека считывается. В примере реализована основная особенность стека – последний элемент занесенный в стек, будет извлечен из стека первым.

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

typedef struct st          // объявление типа STACK
{  char ch;
   st *ps;
} STACK;

main()
{  STACK *p,*q;
   char a;
   p=0;
   do                  //  заполнение стека
   { a=getchar();
     q=(st*)malloc(sizeof(st));
     q->ps=p;
     p=q;
     q->ch=a;
   }while(a!=13);
   printf("\n");
   do                  //  печать стека
   { p=q->ps;
```

```
    free(q);
        q=p;
    printf("%c",p->ch);
  } while(p->ps);
  return 0;
}
```

Деревья

Дерево - непустое конечное множество элементов, один из которых называется ***корнем***, а остальные делятся на несколько непересекающихся подмножеств, каждое из которых также является деревом. Одним из разновидностей деревьев являются **бинарные деревья**. Бинарное дерево имеет один корень и два непересекающихся подмножества, каждое из которых само является бинарным деревом. Эти подмножества называются **левым** и **правым поддеревьями** исходного дерева. Ниже, на рис. 13, приведены графические изображения бинарных деревьев. Если один или более узлов дерева имеют более двух ссылок, то такое дерево не является бинарным.



а б

Рис. 13

На рис. 13,а A-корень дерева, B (C) - корень левого (правого) поддерева, то говорят, что A - отец, B (C) - левый (правый) сын. Узел, не имеющий сыновей B, F,G, E - **лист**. Узел i - **предок** узла j (a j - потомок i), если i - либо отец j, либо отец некоторого предка j. Если каждый узел бинарного дерева, не являющийся листом, имеет непустые правые и левые поддеревья, то дерево называется **строго бинарным деревом (**рис. 13,б). Строго бинарное дерево с n листами всегда содержит 2n-1 узлов.

Уровень узла в бинарном дереве может быть определен следующим образом. Корень дерева имеет **уровень** 0, уровень любого другого узла дерева на 1 больше уровня его отца. **Глубина** бинарного дерева - максимальный уровень листа дерева (длина максимального пути от корня к листу). **Полное бинарное дерево уровня n** - дерево, в котором каждый узел уровня n является листом, и каждый узел уровня меньше n имеет непустые левое и правое поддеревья. **Почти полное бинарное дерево** - бинарное дерево, для которого существует неотрицательное целое k такое, что:

- каждый лист в дереве имеет уровень k или k+1;
- если узел дерева имеет правого потомка уровня k+1, то все его левые потомки, также листья, имеют уровень k+1.

Бинарные деревья с успехом могут быть использованы, например, при сравнении и организации хранения очередной порции входной информации с информацией, введенной ранее, и при этом в каждой точке сравнения может быть принято одно из двух возможных решений. Так как информация вводится в произвольном порядке, то нет возможности предварительно упорядочить ее и применить бинарный поиск. При использовании линейного поиска время пропорционально квадрату количества анализируемых слов. Каким же образом, не затрачивая большое количество времени, организовать эффективное хранение и обработку входной информации. Один из способов постоянно поддерживать упорядоченность имеющейся информации: это перестановка ее при каждом новом вводе информации, что требует существенных временных затрат. Построение бинарного дерева осуществляется на основе *лексикографического упорядочивания* входной информации. Различным элементам входной информации ставятся в соответствие различные узлы бинарного дерева, каждый из которых является структурой, содержащей:

- информационную часть;
- указатель на левый сыновний узел;
- указатель на правый сыновний узел.

Лексикографическое упорядочивание информации в бинарном дереве заключается в следующем. Считывается первая информация и помещается в узел который становится корнем бинарного дерева с пустыми левым и правым поддеревьями. Затем каждая вводимая порция информации сравнивается с информацией содержащейся в корне. Если значения совпадают, то, например, наращиваем число повторов и переходим к новому вводу информации. Если же введенная информация меньше значения в корне, то процесс повторяется для левого поддерева, иначе для правого. Так продолжается до тех пор, пока не встретится дубликат или пока не будет достигнуто пустое поддерево. В этом случае число помещается в новый узел данного места дерева.

Следующая операция **прохождение** бинарного дерева. Существуют три метода обхода бинарного дерева: в **прямом** порядке, в **симметричном** порядке и в **обратном** порядке. Чтобы пройти непустое бинарное дерево в *прямом* порядке надо:

Попасть в корень.
Пройти в прямом порядке левое поддерево.
Пройти в прямом порядке правое поддерево.
Для прохождения в *симметричном* порядке:
Пройти в симметричном порядке левое поддерево.
Попасть в корень.
Пройти в симметричном порядке правое поддерево.
Для прохождения в *симметричном* порядке:
Пройти в обратном порядке левое поддерево.
Пройти в обратном порядке правое поддерево.
Попасть в корень.

Ниже приведен текст программы, иллюстрирующий работу с бинарным деревом.

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <conio.h>
#include <dos.h>
#define N 20
#define M 20
struct der
{ char *inf;      // информационное поле*/
  int n;          // число встреч инф. поля в бинарном дереве*/
  der *l,*r;};  // указатель на левое и правое поддерево*/

void see_1(der *);
void see_2(der *);
der *sozd(der *);
void add(der *);
der *del(der *);

void main(void)
{ der *dr;
 dr=NULL;        // адрес корня бинарного дерева

 clrscr();
 while(1)
 { puts("вид операции: 1-создать дерево");
   puts("              2-рекурсивный вывод содержимого дерева");
   puts("              3-нерукурсивный вывод содержимого дерева");
   puts("              4-добавление элементов в дерево");
   puts("              5-удаление любого элемента из дерева");
   puts("              6-выход");
   fflush(stdin);
   switch(getch())
   { case '1': dr=sozd(dr); break;
     case '2': see_1(dr); getch(); break;
     case '3': see_2(dr); getch(); break;
     case '4': add(dr); break;
     case '5': del(dr); break;
     case '6': return;
   }
   clrscr();
 }
}

// создание бинарного дерева
der *sozd(der *dr)
```

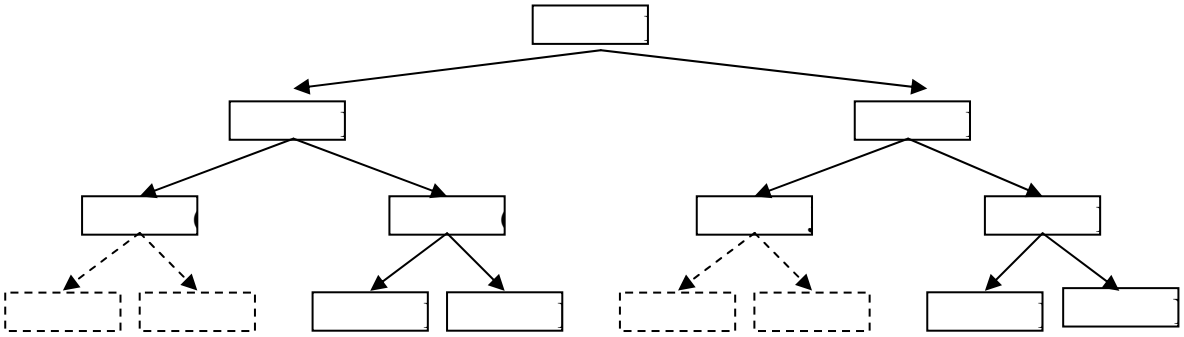
```
{ if (dr)
  { puts("Бинарное дерево уже создано");
    return (dr);
  }
  if (!(dr=(der *) calloc(1,sizeof(der))))
  { puts("Нет свободной памяти");
    getch();
    return NULL;
  }
  puts("Введите информацию в корень дерева");
  dr->inf=(char *) calloc(1,sizeof(char)*N);
  gets(dr->inf);
  dr->n=1;       // число повторов информации в дереве
  return dr;
}

// функция добавления узлов в бинарное дерево
void add(der *dr)
{ der *dr1,*dr2;
  char *st;              // строка для анализа информации
  int k;                 // результат сравнения двух строк
  int ind;
  if (!dr)
  { puts("Нет корня дерева \n");
    getch();
    return;
  }
  do
  { puts("Введите информацию в очередной узел дерева   (0 - выходl)");
    st=(char *) calloc(1,sizeof(char)*N);  //память под символьную инф.
    gets(st);
    if(!*st) return;      // выход в функцию  main
    dr1=dr;
    ind=0;               // 1 - признак выхода из цикла поиска
    do
    { if(!(k=strcmp(st,dr1->inf)))
      { dr1->n++;        // увеличение числа встреч информации узла
        ind=1;           // для выхода из цикла do ... while
      }
      else
      {  if (k<0)          // введ. строка  <  строки в анализируемом узле
         { if (dr1->l) dr1=dr1->l;  // считываем новый узел дерева
           else ind=1;     // выход из цикла do ... while
         }
         else             // введ. строка  >  строки в анализируемом узле
```

```
          { if (dr1->r) dr1=dr1->r;  // считываем новый узел дерева
            else ind=1;     // выход из цикла do ... while
          }
       }
     } while(ind==0);
     if (k)       // не найден узел с аналогичной информацией
     { if (!(dr2=(struct der *) calloc(1,sizeof(struct der))))
       { puts("Нет свободной памяти");
         return;
       }
       if (k<0) dr1->l=dr2;   // ссылка в dr1 налево
       else dr1->r=dr2;       // ..... направо
       dr2->inf=(char *) calloc(1,sizeof(char)*N);
       strcpy(dr2->inf,st);   // заполнение нового узла  dr2
       dr2->n=1;
     }
     free(st);
   } while(1);        // любое условие т.к. выход из цикло по return
}

// рекурсивный вывод содержимого бинарного дерева
void see_1(der *dr1)
{ if(dr1)
  { //printf("узел содержит :   %s , число встреч %d\n",dr1->inf,dr1->n);
    if (dr1->l) see_1(dr1->l);     // вывод левой ветви дерева
    //printf("узел содержит :   %s , число встреч %d\n",dr1->inf,dr1->n);
    if (dr1->r) see_1(dr1->r);     // вывод правой ветви дерева
    //printf("узел содержит :   %s , число встреч %d\n",dr1->inf,dr1->n);
  }
}

// нерекурсивный вывод содержимого бинарного дерева
// используя стек для занесения адресов узлов дерева
void see_2(der *dr1)
{ struct stek
  { der *d;
    stek *s;} *st,*st1=NULL;
  int pr=1;
  for(int i=0;i<2;i++)   // в стек заносятся 2 элемента содержащие указатель
  { st=(stek *)calloc(1,sizeof(stek));  // на корень дерева для прохода по
    st->d=dr1;                          // левому и правому поддеревьям
    st->s=st1;                          // указатель на стек вниз
    st1=st;
  }
    printf("узел содержит :   %s , число встреч %d\n",dr1->inf,dr1->n);
```

```
   while(st)
   { do
     { if(pr && dr1->l) dr1=dr1->l;    // переход на узел слева
       else if (dr1->r) dr1=dr1->r;      // переход на узел справа
       pr=1;                          // сброс принудительного движения вправо
       if(dr1->l && dr1->r)             // узел с 2 связями вниз
       { st1=st;
         st=(stek *)calloc(1,sizeof(stek));
         st->d=dr1;                     // указатель на найденый узел
         st->s=st1;                     // указатель на стек вниз
       }
       printf("узел содержит :   %s , число встреч %d\n",dr1->inf,dr1->n);
     } while(dr1->l || dr1->r);
     dr1=st->d;               // возврат на узел ветвления
     st1=st->s;               // в стеке адрес узла выше удаляемого
     free(st);                // удаление из стека указателя на узел
                              // после прохода через него налево
     st=st1;
     if(dr1->r) pr=0;         // признак принудительного перехода на
                              // узел расположенный справа от dr1, т.к.
                              // dr1->inf уже выведен при проходе слева
   }
}

// функция удаления узла дерева
der *del(der *dr)
{ struct der *dr1,*dr2,*dr3;
  char *st;              // строка для анализа информации
  int k;                 // результат сравнения двух строк
  int ind;
  if(!dr)
  { puts("Дерево не создано \n");
    return NULL;
  }
  puts("Введите информацию в для поиска удаляемого узла");
  st=(char *) malloc(sizeof(char)*N);
  fflush(stdin);
  gets(st);                    //строка для поиска узла в дереве
  if(!*st) return NULL;    // выход в функцию  main
  dr2=dr1=dr;
  ind=0;                       // 1 - признак выхода из цикла поиска
  do                           // блок поиска удаляемого из дерева узла
  { if (!(k=strcmp(st,dr1->inf)))
      ind=1;                   // узел со строкой st найден
    if (k<0)                   // введ. строка  <  строки в анализируемом узле
```

```
    { if (dr1->l)
      { dr2=dr1;              // запоминаем текущий узел
        dr1=dr1->l;           // считываем новый левый узел дерева
      }
      else ind=1;             // выход из цикла do ... while
    }
    if (k>0)                  // введ. строка  >  строки в анализируемом узле
    { if (dr1->r)
      { dr2=dr1;                  // запоминаем текущий узел
        dr1=dr1->r;               // считываем новый правый узел дерева
      }
      else ind=1;                 // выход из цикла do ... while
    }
  } while(!ind);
  free(st);
  if (k)
  { puts("Требуемый узел не найден \n");
    getch();
    return dr;
  }
  else
  { k=strcmp(dr1->inf,dr2->inf);
    dr3=dr1;
    if (k<0)                  // удаляемая вершина < предыдущей
    { dr3=dr1->r;         // поиск ссылки NULL влево
      while(dr3->l) dr3=dr3->l;
      dr2->l=dr1->r;     // сдвиг ветви начинающейся с адреса dr1->r влево
      dr3->l=dr1->l;
    }
    else                      // удаляемая вершина > предыдущей
    { dr3=dr1->l;         // поиск ссылки NULL вправо
      while(dr3->r) dr3=dr3->r;
      dr2->r=dr1->l;     // сдвиг ветви начинающейся с адреса dr1->l вправо
      dr3->r=dr1->r;
    }
  }
}
```

Кроме рассмотренного выше узлового представления, бинарное дерево (почти полное) может быть представлено с помощью массива рис. 14. Узлы почти полного бинарного дерева могут быть занумерованы таким образом, что номер, назначенный левому сыну, равен 2k, где k - номер, присвоенный узлу отца, а номер, присвоенный правому сыну, 2k+1. Дерево с n листами представляется массивом размером 2n-1, если дерево строго бинарное, и 2n иначе. Корень



индекс в массиве

0	1	2	3	4	5	6	7	8	9	0	1	2	3	4
I	E	K	C	G	J	M			F	H			L	

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5

номер узла в бинарном дереве

Рис.14. Представление бинарного дерева в виде массива

дерева содержится в элементе с номером 1. Таким образом левый сын узла, расположенного в позиции k массива, находится в позиции 2k этого массива, а правый сын - в позиции 2k+1 соответственно. Аналогично позиция отца в массиве может быть получена путем округления до целого значения n/2 или (n+1)/2 (где n - позиция левого сына). Такое представление почти полного бинарного дерева в виде массива может быть расширено до общего представления любого бинарного дерева. При этом элемент массива выделяется вне зависимости от того, будет ли он содержать узел дерева. Следовательно, необходимо отметить элементы массива, соответствующие пустым (несуществующим) узлам дерева. Это может быть выполнено либо, например, занесением в эти элементы значений, не допустимых для узлов дерева, либо дополнительным массивом с информацией о таких узлах.

Файлы

Рассмотрим вначале некоторые функции относящиеся к первой группе. Для системы ввода/вывода ANSI требуется заголовок stdio.h. Файлы используются для размещения данных, предназначенных для длительного хранения. Каждому файлу присваивается уникальное имя для обращения к этому файлу. Для выполнения операций с файлами используется указатель на файл. Указатель на файл – это переменная идентифицирующая конкретный дисковый файл (его адрес) и используется для организации ввода/вывода. Указатель на файл имеет вид:

FILE *fp;

Прежде чем производить действия с файлом (чтение, запись) файл должен быть открыт. Функция fopen() открывает для использования поток, связывает файл с данным потоком и возвращает указатель на открытый файл. Функция fopen() имеет следующий прототип:

FILE *fopen(const char **физическое_имя_файла*, const char **режим*);

Режим указывает на строку содержащую режим открытия файла. Возможны следующие режимы (табл. 11):

Таблица 11.

режим	Действие
“r”	Открывает файл для чтения
“w”	Создает файл для записи
“a”	Открывает файл для дозаписи в конец файла
“rb”	Открывает двоичный файл для чтения
“wb”	Открывает двоичный файл для записи
“ab”	Открывает двоичный файл для дозаписи в конец файла
“r+”	Открывает текстовый файл для чтения/записи
“w+”	Создает текстовый файл для чтения/записи
“a+”	Открывает или создает текстовый файл для чтения/записи
“r+b”	Открывает двоичный файл для чтения/записи
“w+b”	Открывает двоичный файл для чтения/записи
“a+b”	Открывает или создает двоичный файл для чтения/записи
“rt”	Открывает текстовый файл для чтения
“wt”	Открывает текстовый файл для записи
“at”	Открывает текстовый файл для дозаписи в конец файла
“r+t”	Открывает текстовый файл для чтения
“w+t”	Открывает текстовый файл для чтения/записи
“a+t”	Открывает или создает текстовый файл для чтения/записи

Функция fopen() указатель идентифицирующий файл и в дальнейшем использующийся большинством функций файловой системы. В случае если спецификация файла задана неверно, то fopen() возвращает указатель NULL.

Функция fclose() разрывает связь указателя с файлом и закрывает его к дальнейшему использованию, до тех пор пока он вновь не будет используя функцию fopen().

Наряду с доступом к информации в файле используя указатель типа FILE* может быть использован подход, основанный на использовании ***дескриптора*** (номера) файла. Дескриптор файла имеет тип int. Всякий раз, когда осуществляется ввод/вывод, идентификация файла осуществляется по его номеру. Для обеспечения возможности работы с файлами, как и в предыдущем случае, файл должен быть открыт. Для этого имеется две функции: open и creat.

int open(const char *name, int flags, unsigned mode);

Функция open() открывает файл с именем name и устанавливает режим доступа к нему в соответствии со значением аргумента flags. Аргумент flags представляет собой комбинацию основного режима доступа и модификаторов.

O_RDONLY - открыть только для чтения;

O_WRONLY - открыть только для pfgbcb;

O_RDRW - открыть для записи/чтения;

Выбрав одно из значений основного режима, можно объединить его, используя операцию | (ИЛИ) с одним или большим числом модификаторов:

O_NDELAY - не используется, включен для совместимости с UNIX:

O_APPEND - перемещает указатель в конец файла перед очередной записью:

O_CREAT - если файл не существует, то создает его в соответствии с mode;

O_TRUNC - существующий файл урезает до 0, сохраняя его атрибуты:

O_EXCL - при использовании с O_GREAT не будет создаваться выходной файл если файл с таким именем уже существует;

O_BINARY - открывается двоичный файл;

O_TEXT - открывается текстовый файл.

Аргумент mode требуется только при использовании модификатора O_CREAT и принимает одно из следующих трех значений:

S_IWRITE доступ по записи;

S_IREAD доступ по чтению;

S_IWRITE|S_IREAD доступ по записи /чтению.

Функция возвращает либо дескриптор открытого файла, либо –1 в случае ошибки, при этом errno принимает одно из четырех значений:

ENOENT файл не существует;

EMFILE слишком много открытых файлов;

EACCESS доступ запрещен;

EINVACC недействительный код доступа.

Пример открытия файла с использованием функции open():

```
int fd;
if ((fd=open(filename,O_WRONLY)==-1)
{  printf(”ошибка открытия файла\n”);
   exit(1);
}
```

Функция close() разрывает связь между файловым дескриптором и открытым файлом. При этом файл закрывается, а дескриптор освобождается для его использования с другим файлом. Она аналогична функции fclose() с той разницей, что не приводит к выталкиванию содержимого буфера в файл. Завершение программы при помощи exit() или return в функции main() закрывает все открытые файлы.

Примеры программ

Пример . Создать два файла f1 - с четными f2 с нечетными числами используя функцию объединить оба файла в один f3.

```
#include <stdio.h>
void fcopy(FILE *,FILE *);
void main(void)
{ FILE *f1,*f2,*f3;
  int i,j,k;
  if (!(f1=fopen("aa","w+")) ||       // открытие файлов для
     !(f2=fopen("bb","w+")))          // записи и чтения
  { puts("файл не может быть создан");
```

```
  return;
 }
 for(i=1;i<10;i++)
 { if (!(i%2)) fprintf(f1,"%d",i);   // запись четных чисел
   else fprintf(f2,"%d",i);          // запись нечетных чисел
 }
 rewind(f1);                 // возврат УТПФ в начало файла и сброс
 rewind(f2);                 // признака всех ошибок
 f3=fopen("cc","w");         // открытие файла для записи
 fcopy(f1,f3);               // копирование  f1 -->  f3
 fcopy(f2,f3);               // добавление    f2 -->  f3
 fclose(f1);
 fclose(f2);
 fclose(f3);
}
        // ф-ция копирования информации из fi в fp
void fcopy(FILE *fi,FILE *fp)
{ int c;
  // вариант 1 – используется цикл while для считывания из файла fi
//  считывание выполняется по одному байту
//  while((c=getc(fi))!=EOF)  // вариант 1
//  putc(c,fp);

  // вариант 2 – используется цикл do{}while для считывания из файла fi
 do                          // вариант 2
  { fread(&c,1,1,fi);
    if(feof(fi)) break;
    fwrite(&c,1,1,fp);
  } while(!feof(fi));
}
```

Пример . Написать программу добавления в текстовый, упорядоченный по возрастанию файл, чисел вводимых с клавиатуры не нарушая его упорядоченности.

```
#include <stdio.h>
void main(void)
{ FILE *f;
  int ms[]={5,7,10,11,14};
  int i,j,k,kk;
  fpos_t l1,l2;              // тип позиция в файле
  f=fopen("aaa","w+");
  for(i=0; i<6;i+)
  fprintf(f,"%3d",ms[i]); // для записи каждого числа отводится 3 позиции
  while(1)
  { scanf("%d",&i);
```

```
    if (i==999) break;       // выход из программы
    rewind(f);               // установка УТПФ в начало файла и сброс
    do                       // признака конца файла
    { fgetpos(f,&l1);        // УТПФ наначало поля считываемого числа
      fscanf(f,"%d",&j);
      if (feof(f) || j>i)  break; // достигнут конец файла или считано число
                                  // большее чем введенное с клавиатуры.
    } while(1);
    if (feof(f) && j<i)          // EOF и в файле нет числа > чем введенное
    {  rewind(f);
       fseek(f,0,2);             // выход на конец файла
       kk=fprintf(f,"%3d",i);  // дозапись в конец файла
       continue;
    }
    rewind(f);
    fseek(f,-3,2);               // УТПФ на последний элемент файла
    k=1;
   do
    { fgetpos(f,&l2);            // сдвиг всех чисел в массиве до
     fscanf(f,"%d",&j);          // l2 позиция УТПФ числа которое > чем i
     rewind(f);
     l2+=3;
     fsetpos(f,&l2);
     fprintf(f,"%3d",j);
     k++;
     fseek(f,-3*k-3,2);
    } while(l1!=l2);
    fsetpos(f,&l2);
    fprintf(f,"%3d",i);    // запись i на место числа с коптрого
  }                        // произведен сдвиг всех чисел вниз
    fclose(f);
}
```

Пример . Пусть имеются 2 файла (file1.txt и file2.txt) оба упорядочены по возрастанию, переписать их в результирующий file3.txt без нарушения упорядоченности.

```
#include <stdio.h>
#include <values.h>
void main(void)
{ FILE *f1,*f2,*f3;
  int i1,i2,j1,j2;
  char cc;
  if (!(f1=fopen("file1.txt","r")) ||
     !(f2=fopen("file2.txt","r")) ||
     !(f3=fopen("file3.txt","w")))
```

```
{ puts("ошибка открытия файла");
  return;
}
fscanf(f1,"%d",&i1);          // предварительное считывание начальных
fscanf(f2,"%d",&i2);          // значений из каждого яайла
do
{ if (i1<i2)                  // выбранный эл-т 1 файла < чем 2 файла
  { while(!feof(f1) && i1<i2)
    { fprintf(f3,"%d ",i1);
 j1=fscanf(f1,"%d",&i1);
    }
    if (feof(f1) && i1<i2 && j1>0)
    { fprintf(f3,"%d ",i1);   // дозапись последнего элемента из файла
                              // если после последнего числа стоит EOF
                              // а не пробел
 i1=MAXINT;
    }
  }
  else
  { while(!feof(f2) && i1>=i2)
    { fprintf(f3,"%d ",i2);
 j2=fscanf(f2,"%d",&i2);
    }
    if (feof(f2) && i1>=i2 && j2>0)
    { fprintf(f3,"%d ",i2);   // как и выше для f1
 i2=MAXINT;
    }
  }
} while(!feof(f1) || !feof(f2));
fclose(f1);
fclose(f2);
fclose(f3);
}
```

Пример . Пусть имеются 2 файла (f1.txt упорядочен по возрастанию и f2.txt - по убыванию), переписать их в результирующий файл f3.txt упорядочивая информацию в нем по возрастанию

```
#include <stdio.h>
#include <conio.h>
#include <values.h>
void main(void)
{ FILE *f1,*f2,*f3;
  int i1,i2,j1,j2,n;
  long n2;
  char cc;
```

```
clrscr();
if (!(f1=fopen("f1.txt","r")) ||
   !(f2=fopen("f2.txt","r")) ||
   !(f3=fopen("f3.txt","w+")))
{ puts("ошибка открытия файла");
 return;
}
fscanf(f1,"%d",&i1);          // предварительное считывание начальных
fseek(f2,0,2);
fgetpos(f2,&n2);              //
n=1;
fseek(f2,-2*n+++1,2);
fscanf(f2,"%d",&i2);          // значений из каждого яайла
do
{ if(i1<i2)                   // выбранный эл-т 1 файла < чем 2 файла
  { while(!feof(f1) && i1<i2)
    {  fprintf(f3,"%d ",i1);
       j1=fscanf(f1,"%d",&i1);
    }
    if(feof(f1))
    { if(i1<i2 && j1>0)
       fprintf(f3,"%d ",i1);   // дозапись последнего элемента из файла если
                               // после последнего числа стоит EOF а не пробел
       i1=MAXINT;
    }
  }
  else
  { while(n2-2*n+1>=0 && i1>=i2)
     {  fprintf(f3,"%d ",i2);
        fseek(f2,-2*n+++1,2);
        j2=fscanf(f2,"%d",&i2);
     }
     if(n2-2*n+1<=0 && i1>=i2 && j2>0) //достигнуто начало файла
    {  fprintf(f3,"%d ",i2);                    // как и выше для f1
       n++;
       i2=MAXINT;
    }
  }
} while(!feof(f1) || n2-2*n+1>=0);  // n2-2*n если после посл. цифры
fclose(f1);                         // стоит пробел  а не признак EOF
fclose(f2);
rewind(f3);
while(1)
{  fscanf(f3,"%d",&i1);
   if(feof(f3)) break;
```

```
    printf("%3d",i1);
  }
  fclose(f3);
 }
```

Пример . Разработать программу выполняющую обработку файловой информации. Содержимое файла – структуры. В программе выполняются следующие функции : создание файла, добавление и удаление информации, модификация ее, упорядочивание по одному из полей структуры.

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#include <string.h>
void sort(char *,int);
void add(char *);
void del(char *);
void print(char *);
struct str swap(FILE *,struct str,struct str,fpos_t *,fpos_t *);

struct inf
{  char tp[15];
   unsigned short pr;
};
struct str
{  inf in;
   double hd;
   int mem;
};
FILE *f;
void main(void)
{ char name[]="baza.txt";
  int i,k,pl;
  clrscr();

  while(1)
  {  puts("\n1- add \n2-sort \n3-del \n4-print \n5-end\n
            выберите режим работы");
    scanf("%d",&k);
    switch(k)
    { case 1: add(name); break;
      case 2: do{ scanf("%d",&pl);} while(pl<1 || pl>4);
                  sort(name,pl); break;
      case 3: del(name); break;
      case 4: print(name); break;
```

```
        case 5: return;
      }
    }
  }

void add(char *name)
{ double a;
  str *s;
  if ((f=fopen(name,"ab"))==NULL)
  { puts(" ошибка  открытия файла");
    return;
  }
  if ((s=(str *)calloc(1,sizeof(str)))==NULL)
  { puts("ошибка при выделении памяти");
    fclose(f);
    return;
  }
  do
  { fflush(stdin);
    puts("введите тип компъютера ");
    gets(s->in.tp); if(!strcmp(s->in.tp,"")) break;
    puts("введите частоту проц. , объем  HDD   и  памяти");
    scanf("%d",&(s->in.pr));
    scanf("%lf",&a);        // ???????/
    s->hd=a;
    scanf("%d",&s->mem);
    fwrite(s,sizeof(str),1,f);   // дозапись структуры в конец файла
  } while(1);
  fclose(f);
}

void del(char *name)
{ str st,stt,s={"",0,0,0};
  long l;
  if ((f=fopen(name,"r+b"))==NULL)
  { puts(" ошибка  открытия файла");
    return;
  }
  scanf("%s%d%lf%d",st.in.tp,&st.in.pr,&st.hd,&st.mem);
  do
  { fread(&stt,sizeof(str),1,f);
    if (feof(f)) break;
    if (! strcmp(st.in.tp,stt.in.tp) && st.in.pr==stt.in.pr &&
   st.hd==stt.hd && st.mem==stt.mem)
     {
```

```
      do
      { fread(&stt,sizeof(str),1,f);
   if (feof(f)) break;
   l=ftell(f);
   fseek(f,l-(long)(2*sizeof(str)),0);
   fwrite(&stt,sizeof(str),1,f);
   fseek(f,l,0);
     } while (!feof(f));
    }
  } while(1);
  rewind(f);
  fseek(f,-(long)(sizeof(str)),2);
  fwrite(&s,sizeof(str),1,f);
  fclose(f);
}

void sort(char *name,int pr)
{ str st,stt;
  int ind;
  fpos_t *l1,*l2;
  if ((f=fopen(name,"r+b"))==NULL)
  { puts(" ошибка  открытия файла");
    return;
  }
  do
  { fgetpos(f,l1);
    fread(&st,sizeof(str),1,f);
    if (feof(f)) break;
    do
    { fgetpos(f,l2);
      fread(&stt,sizeof(str),1,f);
      if (feof(f)) break;
      ind=0;                    // сброс индикатора перестановки
      switch(pr)
      { case 1: if (strcmp(st.in.tp,stt.in.tp)>0) // сравнение по названию
                  ind=1;          // установка индикатора перестановки
                  break;
        case 2: if (st.in.pr>stt.in.pr) ind=1;  // сравнение по процессору
                  break;
        case 3: if (st.hd>stt.hd) ind=1;        // сравнение по HDD
                  break;
        case 4: break;                          // добавить
      }
      if(ind)
      {  st=swap(f,st,stt,l1,l2);
```

```
        *l2+=(long)sizeof(str);
         fsetpos(f,l2);  // УТПФ на следующую запись stt в файле
      }
    } while(!feof(f));
     *l1+=(long)sizeof(str);
     fsetpos(f,l1);       // УТПФ на следующую запись st в файле
  } while(1);
  fclose(f);
}

void print(char *name)
{ str st;
  int k=0;
  if ((f=fopen(name,"rb"))==NULL)
  { puts(" ошибка  открытия файла");
    return;
  }
  clrscr();
  puts("   тип компъютера    процессор     HDD    RAM");
  do
  { k++;              // счетчик числа выведенных на экран записей
    fread(&st,sizeof(str),1,f);
    if (feof(f)) break;
    if (k>=10)         // выведено 5 записей
    {  k=0;            // сброс четчика
       getch();        // задержка вывода на экран
       clrscr();
       puts("   тип компъютера    процессор     HDD    RAM");
    }
    printf("\n%20s%10d%8.2lf%8d",st.in.tp,st.in.pr,st.hd,st.mem);
  } while(!feof(f));
  fclose(f);
}
str swap(FILE *ff,struct str s1,struct str s2,fpos_t *l1,fpos_t *l2)
{ fsetpos(ff,l1);
   fwrite(&s2,sizeof(str),1,ff);    // вторая стр-ра по первому адресу
   fsetpos(ff,l2);
   fwrite(&s1,sizeof(str),1,ff);    // первая стр-ра по второму адресу
   return s2;
}
```

Литература

1. Керниган Б., Ритчи Д. Фьэр А. Язык программирования С(С++). Задачи по языку С(С++). –М.: Финансы и статистика, 1985.

2. Керниган Б., Ритчи Д. Язык программирования С(С++). –М.: Финансы и статистика, 1992.

3. Юлин В.А., Булатова И.Р. Приглашение к С(С++). –Мн.: Выш. шк, 1991.

4 . Касаткин А.И., Вальвачев А.Н. От Turbo C к Borland C++. Мн.: Выш. шк, 1992.

5. Касаткин А.И. Управление ресурсами. Мн.: Выш. шк, 1992.

6. Касаткин А.И. Системное программирование. Мн.: Выш. шк, 1993.

7. Романовская Л.М., Русс Т.В., Свитковский С.Г. Программирование в среде С(С++) для ПЭВМ. М.: Финансы и статистика, 1992.

10. Герберт Шилд. Программирование на Borland C++. –Мн.: ООО "Попурри" 1998г. – 800с.

11. Скляров В.А. Программирование на языках С(С++) и С(С++)++. –М.: Высш. шк. 1999г. –288 с.

Содержание

Св. план , поз.

Учебное издание

Луцик Юрий Александрович
Анна Михайловна Ковальчук,
Ирина Викторовна Лукьянова

МЕТОДИЧЕСКОЕ ПОСОБИЕ

по курсу

ОСНОВЫ АЛГОРИТМИЗАЦИИ И ПРОГРАММИРОВАНИЯ

для студентов специальности
” Вычислительные машины системы и сети”

Редактор
Корректор

Подписано в печать . . . Формат 60x84 1/16.
Бумага Печать офсетная Усл. печ. л.
Уч.-изд. л. Тираж экз. Заказ

Белорусский государственный университет информатики и радиоэлектроники
Отпечатано в БГУИР. Лицензия ЛП № 156.22027, Минск, П Бровки, 6