Госкомитет Российской Федерации по высшей школе
Тамбовский Государственный Технический Университет

Ю.Ю.Громов, С.И.Татаренко

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ СИ

Утверждено ученым советом университета в качестве учебного пособия

Тамбов, 1994

УДК 519.682.2

ББК з973-018.2(Си)я73

Громов Ю.Ю.,Татаренко С.И. Программирование на языке СИ: Учебное пособие. -Тамбов,1995.- 169 с.

В пособии приведено подробное описание наиболее распространенного языка программирования СИ для персональных компьютеров, совместимых с IBM PC, и описано применение средств языка на примерах задач работы со списками.

Учебное пособие предназначено для студентов всех специальностей, аспирантов и инженерно-технических работников использующих вычислительную технику. Может быть использовано как справочное пособие для широкого круга программистов, как профессионалов, имеющих большой опыт работы на СИ, так и начинающих программировать на СИ.

Ил. 33, табл. 8, библиография назв.

Утверждено ученым советом университета

Рецензенты: д.ф-м.н., проф. Афанасьев А.П.

ОГЛАВЛЕНИЕ

В htm-формат книгу преобразовали С.Ю.Севастьянов и Е.С.Севастьянова
Со стороны Центра Информационных Технологий - Иван Махлин.

1.Описание Языка СИ

1.1. Элементы Языка СИ

1.1.1. Используемые символы

Множество символов используемых в языке СИ можно разделить на пять групп.

1. Символы, используемые для образования ключевых слов и идентификаторов (табл.1). В эту группу входят прописные и строчные буквы английского алфавита, а также символ подчеркивания. Следует отметить, что одинаковые прописные и строчные буквы считаются различными символами, так как имеют различные коды.

Таблица 1

Прописные буквы латинского алфавита	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Строчные буквы латинского алфавита	a b c d e f g h i j k l m n o p q r s t u v w x y z
Символ подчеркивания	_

2. Группа прописных и строчных букв русского алфавита и арабские цифры (табл.2).

Таблица 2

Прописные буквы русского алфавита	А Б В Г Д Е Ж З И К Л М Н О П Р С Т У Ф Х Ц Ч Ш Щ Ы Ь Э Ю Я
Строчные буквы русского алфавита	а б в г д е ж з и к л м н о п р с т у ф х ц ч ш щ ъ ы ь э ю я
Арабские цифры	0 1 2 3 4 5 6 7 8 9

3. Знаки нумерации и специальные символы (табл. 3). Эти символы используются с одной стороны для организации процесса вычислений, а с другой - для передачи компилятору определенного набора инструкций.

Таблица 2

Символ	**Наименование**	**Символ**	**Наименование**
,	запятая	)	круглая скобка правая
.	точка	(	круглая скобка левая
;	точка с запятой	}	фигурная скобка правая
:	двоеточие	{	фигурная скобка левая
?	вопросительный знак	<	меньше
'	апостроф	>	больше
!	восклицательный знак	[	квадратная скобка
|	вертикальная черта	]	квадратная скобка
/	дробная черта	#	номер
\	обратная черта	%	процент

~	тильда	&	амперсанд
*	звездочка	^	логическое не
+	плюс	=	равно
-	мину	"	кавычки

4. Управляющие и разделительные символы. К той группе символов относятся: пробел, символы табуляции, перевода строки, возврата каретки, новая страница и новая строка. Эти символы отделяют друг от друга объекты, определяемые пользователем, к которым относятся константы и идентификаторы. Последовательность разделительных символов рассматривается компилятором как один символ (последовательность пробелов).

5. Кроме выделенных групп символов в языке СИ широко используются так называемые, управляющие последовательности, т.е. специальные символьные комбинации, используемые в функциях ввода и вывода информации. Управляющая последовательность строится на основе использования обратной дробной черты (\) (обязательный первый символ) и комбинацией латинских букв и цифр (табл.4).

Таблица 4

Управляющая последовательность	**Наименование**	**Шеснадцатеричная замена**
\a	Звонок	007
\b	Возврат на шаг	008
\t	Горизонтальная табуляция	009
\n	Переход на новую строку	00A
\v	Вертикальная табуляция	00B
\r	Возврат каретки	00C
\f	Перевод формата	00D
\"	Кавычки	022
\'	Апостроф	027
\0	Ноль-символ	000
\\	Обратная дробная черта	05C
\ddd	Символ набора кодов ПЭВМ в восьмеричном представлении	
\xddd	Символ набора кодов ПЭВМ в шестнадцатеричном представлении	

Последовательности вида \ddd и \xddd (здесь d обозначает цифру) позволяет представить символ из набора кодов ПЭВМ как последовательность восьмеричных или шестнадцатеричных цифр соответственно. Например символ возврата каретки может быть представлен различными способами:

\r - общая управляющая последовательность,

\015 - восьмеричная управляющая последовательность,

\x00D - шестнадцатеричная управляющая последовательность.

Следует отметить, что в строковых константах всегда обязательно задавать все три цифры в управляющей последовательности. Например отдельную управляющую последовательность \n (переход на новую строку) можно представить как \010 или \xA, но в строковых константах необходимо задавать все три цифры, в противном случае символ или символы следующие за управляющей последовательностью будут рассматриваться как ее недостающая часть. Например:

"ABCDE\x009FGH" данная строковая команда будет напечатана с использованием определенных функций языка СИ, как два слова ABCDE FGH, разделенные 8-ю пробелами, в этом случае если указать неполную управляющую строку"ABCDE\x09FGH",то на печати появится ABCDE=|=GH, так как компилятор воспримет последовательность \x09F как символ "=+=".

Отметим тот факт, что, если обратная дробная черта предшествует символу не являющемуся управляющей последовательностью (т.е. не включенному в табл.4) и не являющемуся цифрой, то эта черта игнорируется, а сам символ представляется как литеральный. Например:

символ \h представляется символом h в строковой или символьной константе.

Кроме определения управляющей последовательности, символ обратной дробной черты (\) используется также как символ продолжения. Если за (\) следует (\n), то оба символа игнорируются, а следующая строка является продолжением предыдущей. Это свойство может быть использовано для записи длинных строк.

1.1.2. Константы

Константами называются перечисление величин в программе. В языке СИ разделяют четыре типа констант: целые константы, константы с плавающей запятой, символьные константы и строковыми литералы.

Целая константа: это десятичное, восьмеричное или шестнадцатеричное число, которое представляет целую величину в одной из следующих форм: десятичной, восьмеричной или шестнадцатеричной.

Десятичная константа состоит из одной или нескольких десятичных цифр, причем первая цифра не должна быть нулем (в противном случае число будет воспринято как восьмеричное).

Восьмеричная константа состоит из обязательного нуля и одной или нескольких восьмеричных цифр (среди цифр должны отсутствовать восьмерка и девятка, так как эти цифры не входят в восьмеричную систему счисления).

Шестнадцатеричная константа начинается с обязательной последовательности 0x или 0X и содержит одну или несколько шестнадцатеричных цифр (цифры представляющие собой набор цифр шеснадцатеричной системы счисления: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)

```
Примеры целых констант:
   Десятичная      Восьмеричная       Шестнадцатеричная
   константа       константа          константа
       16              020                0x10
```

```
127             0117            0x2B
240             0360            0XF0
```

Если требуется сформировать отрицательную целую константу, то используют знак "-" перед записью константы (который будет называться унарным минусом). Например: -0x2A, -088, -16 .

Каждой целой константе присваивается тип, определяющий преобразования, которые должны быть выполнены, если константа используется в выражениях. Тип константы определяется следующим образом:

- десятичные константы рассматриваются как величины со знаком, и им присваивается тип int (целая) или long (длинная целая) в соответствии со значением константы. Если константа меньше 32768, то ей присваивается тип int в противном случае long.

- восьмеричным и шестнадцатеричным константам присваивается тип int, unsigned int (беззнаковая целая), long или unsigned long в зависимости от значения константы согласно табл 5.

Таблица 5

Диапазон шестнадцатеричных констант	**Диапазон восьмеричных констант**	**Тип**
0x0 - 0x7FFF	0 - 077777	int
0X8000 - 0XFFFF	0100000 - 0177777	unsigned int
0X10000 - 0X7FFFFFFF	0200000 - 017777777777	long
0X80000000 - 0XFFFFFFFF	020000000000 - 037777777777	unsigned long

Для того чтобы любую целую константу определить типом long, достаточно в конце константы поставить букву "l" или "L". Пример:

5l, 6l, 128L, 0105L, OX2A11L.

Константа с плавающей точкой - десятичное число, представленное в виде действительной величины с десятичной точкой или экспонентой. Формат имеет вид:

[цифры].[цифры] [E|e [+|-] цифры] .

Число с плавающей точкой состоит из целой и дробные части и (или) экспоненты. Константы с плавающей точкой представляют положительные величины удвоенной точности (имеют тип double). Для определения отрицательной величины необходимо сформировать константное выражение, состоящее из знака минуса и положительной константы.

Примеры: 115.75, 1.5E-2, -0.025, .075, -0.85E2

Символьная константа - представляется символом заключенном в апострофы. Управляющая последовательность рассматривается как одиночный символ, допустимо ее использовать в символьных константах. Значением символьной константы является числовой код символа. Примеры:

' '- пробел ,

'Q'- буква Q ,

'\n' - символ новой строки ,

'\\' - обратная дробная черта ,

'\v' - вертикальная табуляция .

Символьные константы имеют тип int и при преобразовании типов дополняются знаком.

Строковая константа (литерал) - последовательность символов (включая строковые и прописные буквы русского и латинского а также цифры) заключенные в кавычки (") . Например: "Школа N 35", "город Тамбов", "YZPT КОД".

Отметим, что все управляющие символы, кавычка ("), обратная дробная черта (\) и символ новой строки в строковом литерале и в символьной константе представляются соответствующими управляющими последовательностями. Каждая управляющая последовательность представляется как один символ. Например, при печати литерала "Школа \n N 35" его часть "Школа" будет напечатана на одной строке, а вторая часть "N 35" на следующей строке.

Символы строкового литерала сохраняются в области оперативной памяти. В конец каждого строкового литерала компилятором добавляется нулевой символ, представляемый управляющей последовательностью \0.

Строковый литерал имеет тип char[] . Это означает, что строка рассматривается как массив символов. Отметим важную особенность, число элементов массива равно числу символов в строке плюс 1, так как нулевой символ (символ конца строки) также является элементом массива. Все строковые литералы рассматриваются компилятором как различные объекты. Строковые литералы могут располагаться на нескольких строках. Такие литералы формируются на основе использования обратной дробной черты и клавиши ввод. Обратная черта с символом новой строки игнорируется компилятором, что приводит к тому, что следующая строка является продолжением предыдущей. Например:

"строка неопределенной \n

длины"

полностью идентична литералу

"строка неопределенной длинны" .

Для сцепления строковых литералов можно использовать символ (или символы) пробела. Если в программе встречаются два или более строковых литерала, разделенные только пробелами, то они будут рассматриваться как одна символьная строка. Этот принцип можно использовать для формирования строковых литералов занимающих более одной строки.

1.1.3. Идентификатор

Идентификатором называется последовательность цифр и букв, а также специальных символов, при условии, что первой стоит буква или специальный символ. Для образования идентификаторов могут быть использованы строчные или прописные буквы латинского алфавита. В качестве специального символа может использоваться символ подчеркивание (_). Два идентификатора для образования которых используются совпадающие строчные и прописные буквы, считаются различными. Например: abc, ABC, A128B, a128b .

Важной особенностью является то, что компилятор допускает любое количество символов в идентификаторе, хотя значимыми являются первые 31 символ. Идентификатор создается на этапе объявления переменной, функции, структуры и т.п. после этого его можно использовать в последующих операторах разрабатываемой программы. Следует отметить важные особенности при выборе идентификатора.

Во первых, идентификатор не должен совпадать с ключевыми словами, с зарезервированными словами и именами функций библиотеки компилятора языка СИ.

Во вторых, следует обратить особое внимание на использование символа (_) подчеркивание в качестве первого символа идентификатора, поскольку идентификаторы построенные таким образом, что, с одной стороны, могут совпадать с именами системных функций и (или) переменных, а с другой стороны, при использовании таких идентификаторов программы могут оказаться непереносимыми, т.е. их нельзя использовать на компьютерах других типов.

В третьих, на идентификаторы используемые для определения внешних переменных, должны быть наложены ограничения, формируемые используемым редактором связей (отметим, что использование различных версий редактора связей, или различных редакторов накладывает различные требования на имена внешних переменных).

1.1.4. Ключевые слова

Ключевые слова - это зарезервированные идентификаторы, которые наделены определенным смыслом. Их можно использовать только в соответствии со значением известным компилятору языка СИ.

Приведем список ключевых слов

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

Кроме того в рассматриваемой версии реализации языка СИ, зарезервированными словами являются :

_asm, fortran, near, far, cdecl, huge, paskal, interrupt .

Ключевые слова far, huge, near позволяют определить размеры указателей на области памяти. Ключевые слова _asm, cdelc, fortran, pascal служат для организации связи с функциями написанными на других языках, а также для использования команд языка ассемблера непосредственно в теле разрабатываемой программы на языке СИ.

Ключевые слова не могут быть использованы в качестве идентификаторов.

1.1.5. Использование комментариев в тексте программы

Комментарий - это набор символов, которые игнорируются компилятором, на этот набор символов, однако, накладываются следующие ограничения. Внутри набора символов, который представляет комментарий не может быть специальных символов определяющих начало и конец комментариев, соответственно (/* и */). Отметим, что комментарии могут заменить как одну строку, так и несколько. Например:

```
     /* комментарии к программе */
     /* начало алгоритма */
      или
     /* комментарии  можно  записать в следующем виде,  однако надо
  быть осторожным, чтобы внутри последовательности, которая игнорируется
компилятором,не попались операторы программы, которые также будут
игнорироваться */
```

Неправильное определение комментариев.

```
     /* комментарии к алгоритму /* решение краевой задачи */ */
      или
     /* комментарии к алгоритму решения */ краевой задачи */
```

1.2. Типы Данных И Их Объявление

Важное отличие языка СИ от других языков (PL1, FORTRAN, и др.) является отсутствие принципа умолчания, что приводит к необходимости объявления всех переменных используемых в программе явно вместе с указанием соответствующих им типов.

Объявления переменной имеет следующий формат:

```
[спецафикатор-класа-памяти]  спецификатор-типа
описатель [=инициатор] [,описатель [= инициатор] ]...
```

Описатель - идентификатор простой переменной либо более сложная конструкция с квадратными скобками, круглыми скобками или звездочкой (набором звездочек).

Спецификатор типа - одно или несколько ключевых слов, определяющие тип объявляемой переменной. В языке СИ имеется стандартный набор типов данных, используя который можно сконструировать новые (уникальные) типы данных.

Инициатор - задает начальное значение или список начальных значений, которые (которое) присваивается переменной при объявлении.

Спецификатор класса памяти - определяется одним из четырех ключевых слов языка СИ: auto, extern, register, static, и указывает,каким образом будет распределяться память под объявляемую переменную, с одной стороны, а с другой, область видимости этой переменной, т.е., из каких частей программы можно к ней обратиться.

1.2.1 Категории типов данных

Ключевые слова для определения основных типов данных

```
Целые типы :                Плавающие типы:
  char                        float
  int                         double
  short                       long  double
  long
  signed
  unsigned
```

Переменная любого типа может быть объявлена как немодифицируемая. Это достигается добавлением ключевого слова const к спецификатору-типа. Объекты с типом const представляют собой данные используемые только для чтения, т.е. этой переменной не может быть присвоено новое значение. Отметим, что если после слова const отсутствует спецификатор-типа, то подразумевается спецификатор типа int. Если ключевое слово const стоит перед объявлением составных типов (массив, структура, смесь, перечисление), то это приводит к тому, что каждый элемент также должен являться немодифицируемым, т.е. значение ему может быть присвоено только один раз.

Примеры:

```
const double A=2.128E-2;
const B=286; (подразумевается const int B=286)
```

Примеры объявления составных данных будут рассмотрены ниже.

1.2.2. Целый тип данных

Для определения данных целого типа используются различные ключевые слова, которые определяют диапазон значений и размер области памяти, выделяемой под переменные (табл. 6).

Таблица 6

Тип	Размер памяти в байтах	Диапазон значений
char	1	от -128 до 127
int	Для IBM XT,AT,SX,DX 2	
short	2	от -32768 до 32767
long	4	от -2 147 483 648 до 2 147 483 647
unsigned shar	1	от 0 до 255
unsigned int	Для IBM XT,AT,SX,DX 2	
unsigned short	2	от 0 до 65535
unsigned long	4	от 0 до 4 294 967 295

Отметим, что ключевые слова signed и unsigned необязательны. Они указывают, как интерпретируется нулевой бит объявляемой переменной, т.е., если указано ключевое слово unsigned, то нулевой бит интерпретируется как часть числа, в противном случае нулевой бит интерпретируется как знаковый. В случае отсутствия ключевого слова unsigned целая переменная считается знаковой. В том случае, если спецификатор типа состоит из ключевого типа signed или unsigned и далее следует идентификатор переменной, то она будет рассматриваться как переменная типа int. Например:

```
unsigned int n;
```

```
unsigned int b;
int c;        (подразумевается  signed   int c  );
unsigned d;   (подразумевается  unsigned int d  );
signed f;     (подразумевается  signed   int f  ).
```

Отметим, что модификатор-типа char используется для представления символа (из массива представление символов) или для объявления строковых литералов. Значением объекта типа char является код (размером 1 байт), соответствующий представляемому символу. Для представления символов русского алфавита, модификатор типа идентификатора данных имеет вид unsigned char, так как коды русских букв превышают величину 127.

Следует сделать следующее замечание: в языке СИ не определено представление в памяти и диапазон значений для идентификаторов с модификаторами-типа int и unsigned int. Размер памяти для переменной с модификатором типа signed int определяется длиной машинного слова, которое имеет различный размер на разных машинах. Так, на 16-ти разрядных машинах размер слова равен 2-м байтам, на 32-х разрядных машинах соответственно 4-м байтам, т.е. тип int эквивалентен типам short int, или long int в зависимости от архитектуры используемой ПЭВМ. Таким образом, одна и та же программа может правильно работать на одном компьютере и неправильно на другом. Для определения длины памяти занимаемой переменной можно использовать операцию sizeof языка СИ, возвращающую значение длины указанного модификатора-типа.

Например:

```
a = sizeof(int);
b = sizeof(long int);
c = sizeof(unsigned long);
d = sizeof(short);
```

Отметим также, что восьмеричные и шестнадцатеричные константы также могут иметь модификатор unsigned. Это достигается указанием префикса u или U после константы, константа без этого префикса считается знаковой.

Например:

```
0xA8C   (int  signed   );
01786l  (long signed   );
0xF7u   (int  unsigned );
```

1.2.3. Данные плавающего типа

Для переменных, представляющих число с плавающей точкой используются следующие модификаторы-типа : float, double, long double (в некоторых реализациях языка long double СИ отсутствует).

Величина с модификатором-типа float занимает 4 байта. Из них 1 байт отводится для знака, 8 бит для избыточной экспоненты и 23 бита для мантиссы. Отметим, что старший бит мантиссы всегда равен 1, поэтому он не заполняется, в связи с этим диапазон значений переменной с плавающей точкой приблизительно равен от 3.14E-38 до 3.14E+38.

Величина типа double занимает 8 бит в памяти. Ее формат аналогичен формату float. Биты памяти распределяются следующим образом: 1 бит для знака, 11 бит для экспоненты и 52

бита для мантиссы. С учетом опущенного старшего бита мантиссы диапазон значений равен от 1.7E-308 до 1.7E+308.

Примеры:

```
float f, a, b;
double x,y;
```

1.2.4. Указатели

Указатель - это адрес памяти, распределяемой для размещения идентификатора (в качестве идентификатора может выступать имя переменной, массива, структуры, строкового литерала). В том случае, 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:

спецификатор-типа [модификатор] * описатель .

Спецификатор-типа задает тип объекта и может быть любого основного типа, типа структуры, смеси (об этом будет сказано ниже). Задавая вместо спецификатора-типа ключевое слово void, можно своеобразным образом отсрочить спецификацию типа, на который ссылается указатель. Переменная, объявляемая как указатель на тип void, может быть использована для ссылки на объект любого типа. Однако для того, чтобы можно было выполнить арифметические и логические операции над указателями или над объектами, на которые они указывают, необходимо при выполнении каждой операции явно определить тип объектов. Такие определения типов может быть выполнено с помощью операции приведения типов.

В качестве модификаторов при объявлении указателя могут выступать ключевые слова const, near, far, huge. Ключевое слово const указывает, что указатель не может быть изменен в программе. Размер переменной объявленной как указатель, зависит от архитектуры компьютера и от используемой модели памяти, для которой будет компилироваться программа. Указатели на различные типы данных не обязательно должны иметь одинаковую длину.

Для модификации размера указателя можно использовать ключевые слова near, far, huge.

Примеры:

```
unsigned int * a; /* переменная  a  представляет собой указатель
                  на тип unsigned int (целые числа без знака) */
double * x;       /* переменная  x  указывает  на  тип  данных с
                  плавающей  точкой  удвоенной  точности      */
char * fuffer ;   /*  объявляется  указатель с именем fuffer
                   который указывает на  переменную типа char */
double nomer;
void *addres;
addres = & nomer;
(double *)addres ++;
/* Переменная addres объявлена как указатель на объект любого типа. Поэтому
ей можно присвоить адрес любого объекта (& - операция  вычисления адреса).
Однако, как было отмечено выше, ни одна арифмитическая операция не может быть
выполнена над указателем,  пока
```

```
  не будет явно определен тип данных,  на которые он указывает. Это
  можно сделать,  используя операцию приведения типа (double *) для
  преобразования addres к указателю на тип double, а затем увеличение адреса.
*/
    const * dr;
  /* Переменная  dr  объявлена как указатель на константное выражение, т.е.
значение указателя может изменяться в процессе выполнения программы, а
величина, на которую он указывает, нет. */
    unsigned char * const w = &obj.
  /* Переменная w объявлена как константный указатель на данные типа char
unsigned.  Это означает, что на протяжение всей программы
  w  будет указывать на одну и ту же область памяти.  Содержание же
  этой области может быть изменено. */
```

1.2.5. Переменные перечислимого типа

Переменная, которая может принимать значение из некоторого списка значений, называется переменной перечислимого типа или перечислением.

Объявление перечисления начинается с ключевого слова enum и имеет два формата представления.

Формат 1. enum [имя-тега-перечисления] {список-перечисления} описатель[,описатель...];

Формат 2. enum имя-тега-перечисления описатель [,описатель..];

Объявление перечисления задает тип переменной перечисления и определяет список именованных констант, называемый списком-перечисления. Значением каждого имени списка является некоторое целое число.

Переменная типа перечисления может принимать значения одной из именованных констант списка. Именованные константы списка имеют тип int. Таким образом, память соответствующая переменной перечисления, это память необходимая для размещения значения типа int.

Переменная типа enum могут использоваться в индексных выражениях и как операнды в арифметических операциях и в операциях отношения.

В первом формате 1 имена и значения перечисления задаются в списке перечислений. Необязательное имя-тега-перечисления, это идентификатор, который именует тег перечисления, определенный списком перечисления. Описатель именует переменную перечисления. В объявлении может быть задана более чем одна переменная типа перечисления.

Список-перечисления содержит одну или несколько конструкций вида:

идентификатор [= константное выражение]

Каждый идентификатор именует элемент перечисления. Все идентификаторы в списке enum должны быть уникальными. В случае отсутствия константного выражения первому идентификатору соответствует значение 0, следующему идентификатору - значение 1 и т.д. Имя константы перечисления эквивалентно ее значению.

Идентификатор, связанный с константным выражением, принимает значение, задаваемое этим константным выражением. Константное выражение должно иметь тип int и может быть как положительным, так и отрицательным. Следующему идентификатору в списке присваивается значение, равное константному выражению плюс 1, если этот идентификатор не имеет своего константного выражения. Использование элементов перечисления должно подчиняться следующим правилам:

1. Переменная может содержать повторяющиеся значения.

2. Идентификаторы в списке перечисления должны быть отличны от всех других идентификаторов в той же области видимости, включая имена обычных переменных и идентификаторы из других списков перечислений.

3. Имена типов перечислений должны быть отличны от других имен типов перечислений, структур и смесей в этой же области видимости.

4. Значение может следовать за последним элементом списка перечисления.

Пример:

```
enum week { SUB = 0,     /*     0   */
            VOS = 0,     /*     0   */
            POND,        /*     1   */
            VTOR,        /*     2   */
            SRED,        /*     3   */
            HETV,        /*     4   */
            PJAT         /*     5   */
            }   rab_ned ;
```

В данном примере объявлен перечислимый тег week, с соответствующим множеством значений, и объявлена переменная rab_ned имеющая тип week.

Во втором формате используется имя тега перечисления для ссылки на тип перечисления, определяемый где-то в другом месте. Имя тега перечисления должно относится к уже определенному тегу перечисления в пределах текущей области видимости. Так как тег перечисления объявлен где-то в другом месте, список перечисления не представлен в объявлении.

Пример:

enum week rab1;

В объявлении указателя на тип данных перечисления и объявляемых typedef для типов перечисления можно использовать имя тега перечисления до того, как данный тег перечисления определен. Однако определение перечисления должно предшествовать любому действию используемого указателя на тип объявления typedef. Объявление без последующего списка описателей описывает тег, или, если так можно сказать, шаблон перечисления.

1.2.6. Массивы

Массивы - это группа элементов одинакового типа (double, float, int и т.п.). Из объявления массива компилятор должен получить информацию о типе элементов массива и их количестве. Объявление массива имеет два формата:

спецификатор-типа описатель [константное - выражение];

спецификатор-типа описатель [ ];

Описатель - это идентификатор массива .

Спецификатор-типа задает тип элементов объявляемого массива. Элементами массива не могут быть функции и элементы типа void.

Константное-выражение в квадратных скобках задает количество элементов массива. Константное-выражение при объявлении массива может быть опущено в следующих случаях:

- при объявлении массив инициализируется,

- массив объявлен как формальный параметр функции,

- массив объявлен как ссылка на массив, явно определенный в другом файле.

В языке СИ определены только одномерные массивы, но поскольку элементом массива может быть массив, можно определить и многомерные массивы. Они формализуются списком константных-выражений следующих за идентификатором массива, причем каждое константное-выражение заключается в свои квадратные скобки.

Каждое константное-выражение в квадратных скобках определяет число элементов по данному измерению массива, так что объявление двухмерного массива содержит два константных-выражения, трехмерного - три и т.д. Отметим, что в языке СИ первый элемент массива имеет индекс равный 0.

Примеры:

```
int a[2][3]; /* представлено в виде матрицы
                a[0][0]  a[0][1] a[0][2]
                a[1][0]  a[1][1] a[1][2]      */
double b[10]; /* вектор из 10  элементов имеющих тип double */
int w[3][3] = { { 2, 3, 4 },
                { 3, 4, 8 },
                { 1, 0, 9 } };
```

В последнем примере объявлен массив w[3][3]. Списки, выделенные в фигурные скобки, соответствуют строкам массива, в случае отсутствия скобок инициализация будет выполнена неправильно.

В языке СИ можно использовать сечения массива, как и в других языках высокого уровня (PL1 и т.п.), однако на использование сечений накладывается ряд ограничений. Сечения формируются вследствие опускания одной или нескольких пар квадратных скобок. Пары квадратных скобок можно отбрасывать только справа налево и строго последовательно. Сечения массивов используются при организации вычислительного процесса в функциях языка СИ, разрабатываемых пользователем.

Примеры:

int s[2][3];

Если при обращении к некоторой функции написать s[0], то будет передаваться нулевая строка массива s.

int b[2][3][4];

При обращении к массиву b можно написать, например, b[1][2] и будет передаваться вектор из четырех элементов, а обращение b[1] даст двухмерный массив размером 3 на 4. Нельзя написать b[2][4], подразумевая, что передаваться будет вектор, потому что это не соответствует ограничению наложенному на использование сечений массива.

Пример объявления символьного массива.

char str[] = "объявление символьного массива";

Следует учитывать, что в символьном литерале находится на один элемент больше, так как последний из элементов является управляющей последовательностью '\0'.

1.2.7. Структуры

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. Тип структуры определяется записью вида:

struct { список определений }

В структуре обязательно должен быть указан хотя бы один компонент. Определение структур имеет следующий вид:

тип-данных описатель;

где тип-данных указывает тип структуры для объектов, определяемых в описателях. В простейшей форме описатели представляют собой идентификаторы или массивы.

Пример:

```
struct { double x,y; } s1, s2, sm[9];
struct {  int   year;
          char  moth, day; } date1, date2;
```

Переменные s1, s2 определяются как структуры, каждая из которых состоит из двух компонент x и y. Переменная sm определяется как массив из девяти структур. Каждая из двух переменных date1, date2 состоит из трех компонентов year, moth, day. >p>Существует и другой способ ассоциирования имени с типом структуры, он основан на использовании тега структуры. Тег структуры аналогичен тегу перечислимого типа. Тег структуры определяется следующим образом:

struct тег { список описаний; };

где тег является идентификатором.

В приведенном ниже примере идентификатор student описывается как тег структуры:

```
struct student { char name[25];
```

```
                int  id, age;
                char  prp;          };
```

Тег структуры используется для последующего объявления структур данного вида в форме:

struct тег список-идентификаторов;

Пример:

struct studeut st1,st2;

Использование тегов структуры необходимо для описания рекурсивных структур. Ниже рассматривается использование рекурсивных тегов структуры.

```
struct node { int data;
              struct node * next; }   st1_node;
```

Тег структуры node действительно является рекурсивным, так как он используется в своем собственном описании, т.е. в формализации указателя next. Структуры не могут быть прямо рекурсивными, т.е. структура node не может содержать компоненту, являющуюся структурой node, но любая структура может иметь компоненту, являющуюся указателем на свой тип, как и сделано в приведенном примере.

Доступ к компонентам структуры осуществляется с помощью указания имени структуры и следующего через точку имени выделенного компонента, например:

```
st1.name="Иванов";
st2.id=st1.id;
st1_node.data=st1.age;
```

1.2.8. Объединения (смеси)

Объединение подобно структуре, однако в каждый момент времени может использоваться (или другими словами быть ответным) только один из элементов объединения. Тип объединения может задаваться в следующем виде:

```
union {   описание элемента 1;
          ...
          описание элемента n; };
```

Главной особенностью объединения является то, что для каждого из объявленных элементов выделяется одна и та же область памяти, т.е. они перекрываются. Хотя доступ к этой области памяти возможен с использованием любого из элементов, элемент для этой цели должен выбираться так, чтобы полученный результат не был бессмысленным.

Доступ к элементам объединения осуществляется тем же способом, что и к структурам. Тег объединения может быть формализован точно так же, как и тег структуры.

Объединение применяется для следующих целей:

- инициализации используемого объекта памяти, если в каждый момент времени только один объект из многих является активным;

- интерпретации основного представления объекта одного типа, как если бы этому объекту был присвоен другой тип.

Память, которая соответствует переменной типа объединения, определяется величиной, необходимой для размещения наиболее длинного элемента объединения. Когда используется элемент меньшей длины, то переменная типа объединения может содержать неиспользуемую память. Все элементы объединения хранятся в одной и той же области памяти, начиная с одного адреса.

Пример:

```
union {   char  fio[30];
          char  adres[80];
          int   vozrast;
          int   telefon;   } inform;
union {   int ax;
          char al[2];      }   ua;
```

При использовании объекта infor типа union можно обрабатывать только тот элемент который получил значение, т.е. после присвоения значения элементу inform.fio, не имеет смысла обращаться к другим элементам. Объединение ua позволяет получить отдельный доступ к младшему ua.al[0] и к старшему ua.al[1] байтам двухбайтного числа ua.ax .

1.2.9. Поля битов

Элементом структуры может быть битовое поле, обеспечивающее доступ к отдельным битам памяти. Вне структур битовые поля объявлять нельзя. Нельзя также организовывать массивы битовых полей и нельзя применять к полям операцию определения адреса. В общем случае тип структуры с битовым полем задается в следующем виде:

```
struct { unsigned идентификатор 1 :  длина-поля  1;
         unsigned идентификатор 2 :  длина-поля  2;    }
```

длинна - поля задается целым выражением или константой. Эта константа определяет число битов, отведенное соответствующему полю. Поле нулевой длинны обозначает выравнивание на границу следующего слова.

Пример:

```
 struct { unsigned a1 :  1;
          unsigned a2 :  2;
          unsigned a3 :  5;
          unsigned a4 :  2;  } prim;
```

Структуры битовых полей могут содержать и знаковые компоненты. Такие компоненты автоматически размещаются на соответствующих границах слов, при этом некоторые биты слов могут оставаться неиспользованными.

Ссылки на поле битов выполняются точно так же, как и компоненты общих структур. Само же битовое поле рассматривается как целое число, максимальное значение которого определяется длиной поля.

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

Пример определения переменной структуры с именем helth_record

```
struct { /* общая информация */
          char  name [25];   /* имя       */
          int   age;         /* возраст   */
          char  sex;         /* пол       */
          /*   метка   активного   компонента  */
          /*   (семейное положение)            */
          enum merital_status ins;
          /* переменная часть */
          union  {   /* холост        */
                     /* нет компонент */
                   struct {  /* состоит в браке */
                             char marripge_date[8];
                             char spouse_name[25];
                             int  no_children;
                           }  marriage_info;
                    /* разведен */
                    char date_divorced[8];
                  }   marital_info;
           } health_record;

  enum marital_status { SINGLE, /* холост   */
                        MARRIGO, /* женат    */
                        DIVOREED /* разведен */
                        } ;
```

Обращаться к компонентам структуры можно при помощи ссылок:

```
helth_record.neme,
helth_record.ins,
helth_record.marriage_info.marriage_date .
```

1.2.11. Определение объектов и типов

Как уже говорилось выше, все переменные используемые в программах на языке СИ, должны быть объявлены. Тип объявляемой переменной зависит от того, какое ключевое слово используется в качестве спецификатора типа и является ли описатель простым идентификатором или же комбинацией идентификатора с модификатором указателя (звездочка), массива (квадратные скобки) или функции (круглые скобки).

При объявлении простой переменной, структуры, смеси или объединения, а также перечисления, описатель - это простой идентификатор. Для объявления указателя, массива или функции идентификатор модифицируется соответствующим образом: звездочкой слева, квадратными или круглыми скобками справа.

Отметим важную особенность языка СИ, при объявлении можно использовать одновременно более одного модификатора, что дает возможность создавать множество различных сложных описателей типов.

Однако надо помнить, что некоторые комбинации модификаторов недопустимы:

- элементами массивов не могут быть функции,

- функции не могут возвращать массивы или функции.

При инициализации сложных описателей квадратные и круглые скобки (справа от идентификатора) имеют приоритет перед звездочкой (слева от идентификатора). Квадратные или круглые скобки имеют один и тот же приоритет и раскрываются слева направо. Спецификатор типа рассматривается на последнем шаге, когда описатель уже полностью проинтерпретирован. Можно использовать круглые скобки, чтобы поменять порядок интерпретации на необходимый.

Для интерпретации сложных описаний предлагается простое правило, которое звучит как "изнутри наружу", и состоит из четырех шагов.

1. Начать с идентификатора и посмотреть вправо, есть ли квадратные или круглые скобки.

2. Если они есть, то проинтерпретировать эту часть описателя и затем посмотреть налево в поиске звездочки.

3. Если на любой стадии справа встретится закрывающая круглая скобка, то вначале необходимо применить все эти правила внутри круглых скобок, а затем продолжить интерпретацию.

4. Интерпретировать спецификатор типа.

Примеры:

```
int    * ( * comp [10]) ();
 6     5   3   1    2    4
```

В данном примере объявляется переменная comp (1), как массив из десяти (2) указателей (3) на функции (4), возвращающие указатели (5) на целые значения (6).

```
char  * ( * ( * var ) () ) [10];
  7   6   4   2  1     3     5
```

Переменная var (1) объявлена как указатель (2) на функцию (3) возвращающую указатель (4) на массив (5) из 10 элементов, которые являются указателями (6) на значения типа char.

Кроме объявлений переменных различных типов, имеется возможность объявить типы. Это можно сделать двумя способами. Первый способ - указать имя тега при объявлении структуры, объединения или перечисления, а затем использовать это имя в объявлении переменных и функций в качестве ссылки на этот тег. Второй - использовать для объявления типа ключевое слово typedef.

При объявлении с ключевым словом typedef, идентификатор стоящий на месте описываемого объекта, является именем вводимого в рассмотрение типа данных, и далее этот тип может быть использован для объявления переменных.

Отметим, что любой тип может быть объявлен с использованием ключевого слова typedef, включая типы указателя, функции или массива. Имя с ключевым словом typedef для типов указателя, структуры, объединения может быть объявлено прежде чем эти типы будут определенны, но в пределах видимости объявителя.

Примеры:

```
 typedef double (* MATH)( );
   /* MATH - новое имя типа, представляющее указатель на
   функцию, возвращающую значения типа double          */
   MATH cos;
   /* cos  указатель на функцию, возвращающую
   значения типа  double                                */
   /* Можно провести эквивалентное объявление          */
   double (* cos)( );

typedef char FIO[40]
    /* FIO - массив из сорока символов                 */
FIO person;
    /*  Переменная person  -  массив из сорока символов  */
    /*  Это  эквивалентно объявлению                    */
char person[40];
```

При объявлении переменных и типов здесь были использованы имена типов (MATH FIO). Помимо этого, имена типов могут еще использоваться в трех случаях: в списке формальных параметров, в объявлении функций, в операциях приведения типов и в операции sizeof (операция приведения типа).

Именами типов для основных типов, типов перечисления, структуры и смеси являются спецификаторы типов для этих типов. Имена типов для типов указателя массива и функции задаются при помощи абстрактных описателей следующим образом:

спецификатор-типа абстрактный-описатель;

Абстрактный-описатель - это описатель без идентификатора, состоящий из одного или более модификаторов указателя, массива или функции. Модификатор указателя (*) всегда задается перед идентификатором в описателе, а модификаторы массива [] и функции () - после него. Таким образом, чтобы правильно интерпретировать абстрактный описатель, нужно начать интерпретацию с подразумеваемого идентификатора.

Абстрактные описатели могут быть сложными. Скобки в сложных абстрактных описателе задают порядок интерпретации подобно тому, как это делалось при интерпретации сложных описателей в объявлениях.

1.2.12. Инициализация данных

При объявлении переменной ей можно присвоить начальное значение, присоединяя инициатор к описателю. Инициатор начинается со знака "=" и имеет следующие формы.

Формат 1: = инициатор;

Формат 2: = { список - инициаторов };

Формат 1 используется при инициализации переменных основных типов и указателей, а формат 2 - при инициализации составных объектов.

Примеры:

char tol = 'N';

Переменная tol инициализируется символом 'N'.

const long megabute = (1024 * 1024);

Немодифицируемая переменная megabute инициализируется константным выражением после чего она не может быть изменена.

static int b[2][2] = {1,2,3,4};

Инициализируется двухмерный массив b целых величин элементам массива присваиваются значения из списка. Эта же инициализация может быть выполнена следующим образом :

static int b[2][2] = { { 1,2 }, { 3,4 } };

При инициализации массива можно опустить одну или несколько размерностей

static int b[3[] = { { 1,2 }, { 3,4 } };

Если при инициализации указано меньше значений для строк, то оставшиеся элементы инициализируются 0, т.е. при описании

static int b[2][2] = { { 1,2 }, { 3 } };

элементы первой строки получат значения 1 и 2, а второй 3 и 0.

При инициализации составных объектов, нужно внимательно следить за использованием скобок и списков инициализаторов.

Примеры:

```
struct complex { double real;
                 double imag; }  comp [2][3] =
     { { {1,1}, {2,3}, {4,5} },
       { {6,7}, {8,9}, {10,11} } };
```

В данном примере инициализируется массив структур comp из двух строк и трех столбцов, где каждая структура состоит из двух элементов real и imag.

```
struct complex comp2 [2][3] =
   { {1,1},{2,3},{4,5}, {6,7},{8,9},{10,11} };
```

В этом примере компилятор интерпретирует рассматриваемые фигурные скобки следующим образом:

- первая левая фигурная скобка - начало составного инициатора для массива comp2;

- вторая левая фигурная скобка - начало инициализации первой строки массива comp2[0]. Значения 1,1 присваиваются двум элементам первой структуры;

- первая правая скобка (после 1) указывает компилятору, что список инициаторов для строки массива окончен, и элементы оставшихся структур в строке comp[0] автоматически инициализируются нулем;

- аналогично список {2,3} инициализирует первую структуру в строке comp[1], а оставшиеся структуры массива обращаются в нули;

- на следующий список инициализаторов {4,5} компилятор будет сообщать о возможной ошибке так как строка 3 в массиве comp2 отсутствует.

При инициализации объединения задается значение первого элемента объединения в соответствии с его типом.

Пример:

```
union tab { unsigned char name[10];
            int tab1;
          }          pers  =  {'A','H','T','O','H'};
```

Инициализируется переменная pers.name, и так как это массив, для его инициализации требуется список значений в фигурных скобках. Первые пять элементов массива инициализируются значениями из списка, остальные нулями.

Инициализацию массива символов можно выполнить путем использования строкового литерала.

char stroka[ ] = "привет";

Инициализируется массив символов из 7 элементов, последним элементом (седьмым) будет символ '\0', которым завершаются все строковые литералы.

В том случае, если задается размер массива, а строковый литерал длиннее, чем размер массива, то лишние символы отбрасываются.

Следующее объявление инициализирует переменную stroka как массив, состоящий из семи элементов.

char stroka[5] = "привет";

В переменную stroka попадают первые пять элементов литерала, а символы 'Т' и '\0' отбрасываются.

Если строка короче, чем размер массива, то оставшиеся элементы массива заполняются нулями.

Отметим, что инициализация переменной типа tab может иметь следующий вид:

union tab pers1 = "Антон";

и, таким образом, в символьный массив попадут символы:

'A','H','T','O','H','\0',

а остальные элементы будут инициализированы нулем.

1.3. Выражения И Присваивания

1.3.1. Операнды и операции

Комбинация знаков операций и операндов, результатом которой является определенное значение, называется выражением. Знаки операций определяют действия, которые должны быть выполнены над операндами. Каждый операнд в выражении может быть выражением. Значение выражения зависит от расположения знаков операций и круглых скобок в выражении, а также от приоритета выполнения операций.

В языке СИ присваивание также является выражением, и значением такого выражения является величина, которая присваивается.

При вычислении выражений тип каждого операнда может быть преобразован к другому типу. Преобразования типов могут быть неявными, при выполнении операций и вызовов функций, или явными, при выполнении операций приведения типов.

Операнд - это константа, литерал, идентификатор, вызов функции, индексное выражение, выражение выбора элемента или более сложное выражение, сформированное комбинацией операндов, знаков операций и круглых скобок. Любой операнд, который имеет константное значение, называется константным выражением. Каждый операнд имеет тип.

Если в качестве операнда используется константа, то ему соответствует значение и тип представляющей его константы. Целая константа может быть типа int, long, unsigned int, unsigned long, в зависимости от ее значения и от формы записи. Символьная константа имеет тип int. Константа с плавающей точкой всегда имеет тип double.

Строковый литерал состоит из последовательности символов, заключенных в кавычки, и представляется в памяти как массив элементов типа char, инициализируемый указанной последовательностью символов. Значением строкового литерала является адрес первого элемента строки и синтаксически строковый литерал является немодифицируемым указателем на тип char. Строковые литералы могут быть использованы в качестве операндов в выражениях, допускающих величины типа указателей. Однако так как строки не являются переменными, их нельзя использовать в левой части операции присваивания.

Следует помнить, что последним символом строки всегда является нулевой символ, который автоматически добавляется при хранении строки в памяти.

Идентификаторы переменных и функций. Каждый идентификатор имеет тип, который устанавливается при его объявлении. Значение идентификатора зависит от типа следующим образом:

- идентификаторы объектов целых и плавающих типов представляют значения соответствующего типа;

- идентификатор объекта типа enum представлен значением одной константы из множества значений констант в перечислении. Значением идентификатора является константное значение. Тип значения есть int, что следует из определения перечисления;

- идентификатор объекта типа struct или union представляет значение, определенное структурой или объединением;

- идентификатор, объявляемый как указатель, представляет указатель на значение, заданное в объявлении типа;

- идентификатор, объявляемый как массив, представляет указатель, значение которого является адресом первого элемента массива. Тип адресуемых указателем величин - это тип элементов массива. Отметим, что адрес массива не может быть изменен во время выполнения программы, хотя значение отдельных элементов может изменяться. Значение указателя, представляемое идентификатором массива, не является переменной и поэтому идентификатор массива не может появляться в левой части оператора присваивания.

- идентификатор, объявляемый как функция, представляет указатель, значение которого является адресом функции, возвращающей значения определенного типа. Адрес функции не изменяется во время выполнения программы, меняется только возвращаемое значение. Таким образом, идентификаторы функций не могут появляться в левой части операции присваивания.

Вызов функций состоит из выражения, за которым следует необязательный список выражений в круглых скобках:

выражение-1 ([список выражений])

Значением выражения-1 должен быть адрес функции (например, идентификатор функции). Значения каждого выражения из списка выражений передается в функцию в качестве фактического аргумента. Операнд, являющийся вызовом функции, имеет тип и значение возвращаемого функцией значения.

Индексное выражение задает элемент массива и имеет вид:

выражение-1 [выражение-2]

Тип индексного выражения является типом элементов массива, а значение представляет величину, адрес которой вычисляется с помощью значений выражение-1 и выражение-2.

Обычно выражение-1 - это указатель, например, идентификатор массива, а выражение-2 - это целая величина. Однако требуется только, чтобы одно из выражений было указателем, а второе целочисленной величиной. Поэтому выражение-1 может быть целочисленной величиной, а выражение-2 указателем. В любом случае выражение-2 должно быть заключено в квадратные скобки. Хотя индексное выражение обычно используется для ссылок на элементы массива, тем не менее индекс может появляться с любым указателем.

Индексные выражения для ссылки на элементы одномерного массива вычисляются путем сложения целой величины со значениями указателя с последующим применением к результату операции разадресации (*).

Так как одно из выражений, указанных в индексном выражении, является указателем, то при сложении используются правила адресной арифметики, согласно которым целая величина преобразуется к адресному представлению, путем умножения ее на размер типа, адресуемого указателем. Пусть, например, идентификатор arr объявлен как массив элементов типа double.

double arr[10];

Таким образом, чтобы получить доступ к i-тому элементу массива arr можно написать arr[i], что, в силу сказанного выше, эквивалентно i[a]. При этом величина i умножается на размер типа double и представляет собой адрес i-го элемента массива arr от его начала. Затем это значение складывается со значением указателя arr, что в свою очередь дает адрес i-го элемента массива. К полученному адресу применяется операция разадресации, т.е. осуществляется выборка элемента массива arr по сформированному адресу.

Таким образом, результатом индексного выражения arr[i] (или i[arr]) является значение i-го элемента массива.

Выражение с несколькими индексами ссылается на элементы многомерных массивов. Многомерный массив - это массив, элементами которого являются массивы. Например, первым элементом трехмерного массива является массив с двумя измерениями.

Для ссылки на элемент многомерного массива индексное выражение должно иметь несколько индексов заключенных к квадратные скобки:

выражение-1 [выражение-2][выражение-3] ...

Такое индексное выражение интерпретируется слева направо, т.е. вначале рассматривается первое индексное выражение:

выражение-1 [выражение-2]

Результат этого выражения есть адресное выражение, с которым складывается выражение-3 и т.д. Операция разадресации осуществляется после вычисления последнего индексного выражения. Отметим, что операция разадресации не применяется, если значение последнего указателя адресует величину типа массива.

Пример:

int mass [2][5][3];

Рассмотрим процесс вычисления индексного выражения mass[1][2][2].

1. Вычисляется выражения mass[1]. Ссылка индекс 1 умножается на размер элемента этого массива, элементом же этого массива является двухмерный массив содержащий 5x3 элементов, имеющих тип int. Получаемое значение складывается со значением указателя mass. Результат является указатель на второй двухмерный массив размером (5x3) в трехмерном массиве mass.

2. Второй индекс 2 указывает на размер массива из трех элементов типа int, и складывается с адресом, соответствующим mass [1].

3. Так как каждый элемент трехмерного массива - это величина типа int, то индекс 2 увеличивается на размер типа int перед сложением с адресом mass [1][2].

4. Наконец, выполняется разадресация полученного указателя. Результирующим выражением будет элемент типа int.

Если было бы указано mass [1][2], то результатом был бы указатель на массив из трех элементов типа int. Соответственно значением индексного выражения mass [1] является указатель на двухмерный массив.

Выражение выбора элемента применяется, если в качестве операнда надо использовать элемент структуры или объединения. Такое выражение имеет значение и тип выбранного элемента. Рассмотрим две формы выражения выбора элемента:

выражение.идентификатор ,

выражение->идентификатор .

В первой форме выражение представляет величину типа struct или union, а идентификатор - это имя элемента структуры или объединения. Во второй форме выражение должно иметь значение адреса структуры или объединения, а идентификатор - именем выбираемого элемента структуры или объединения.

Обе формы выражения выбора элемента дают одинаковый результат. Действительно, запись, включающая знак операции выбора (->), является сокращенной версией записи с точкой для случая, когда выражению стоящему перед точкой предшествует операция разадресации (*), примененная к указателю, т.е. запись

выражение -> идентификатор

эквивалентна записи

(* выражение) . идентификатор

в случае, если выражение является указателем.

Пример:

```
struct tree { float          num;
              int            spisoc[5];
              struct tree   *left;      }  tr[5] , elem ;
       elem.left = & elem;
```

В приведенном примере используется операция выбора (.) для доступа к элементу left структурной переменной elem. Таким образом элементу left структурной переменной elem присваивается адрес самой переменной elem, т.е. переменная elem хранит ссылку на себя саму.

Приведение типов это изменение (преобразование) типа объекта. Для выполнения преобразования необходимо перед объектом записать в скобках нужный тип:

(имя-типа) операнд.

Приведение типов используются для преобразования объектов одного скалярного типа в другой скалярный тип. Однако выражению с приведением типа не может быть присвоено другое значение.

Пример:

```
int i;
bouble x;
b = (double)i+2.0;
```

В этом примере целая переменная i с помощью операции приведения типов приводится к плавающему типу, а затем уже участвует в вычислении выражения.

Константное выражение - это выражение, результатом которого является константа. Операндом константного выражения могут быть целые константы, символьные константы, константы с плавающей точкой, константы перечисления, выражения приведения типов, выражения с операцией sizeof и другие константные выражения. Однако на использование знаков операций в константных выражениях налагаются следующие ограничения:

1. В константных выражениях нельзя использовать операции присваивания и последовательного вычисления (,) .

2. Операция "адрес" (&) может быть использована только при некоторых инициализациях.

Выражения со знаками операций могут участвовать в выражениях как операнды. Выражения со знаками операций могут быть унарными (с одним операндом), бинарными (с двумя операндами) и тернарными (с тремя операндами).

Унарное выражение состоит из операнда и предшествующего ему знаку унарной операции и имеет следующий формат:

знак-унарной-операции операнд .

Бинарное выражения состоит из двух операндов, разделенных знаком бинарной операции:

операнд1 знак-бинарной-операции операнд2 .

Тернарное выражение состоит из трех операндов, разделенных знаками тернарной операции (?) и (:), и имеет формат:

операнд1 ? операнд2 : операнд3 .

Операции. По количеству операндов, участвующих в операции, операции подразделяются на унарные, бинарные и тернарные.

В языке Си имеются следующие унарные операции:

- арифметическое отрицание (отрицание и дополнение);

~ побитовое логическое отрицание (дополнение);

! логическое отрицание;

* разадресация (косвенная адресация);

& вычисление адреса;

+ унарный плюс;

++ увеличение (инкремент);

-- уменьшение (декремент);

sizeof размер .

Унарные операции выполняются справа налево.

Операции увеличения и уменьшения увеличивают или уменьшают значение операнда на единицу и могут быть записаны как справа так и слева от операнда. Если знак операции записан перед операндом (префиксная форма), то изменение операнда происходит до его использования в выражении. Если знак операции записан после операнда (постфиксная форма), то операнд вначале используется в выражении, а затем происходит его изменение.

В отличие от унарных, бинарные операции, список которых приведен в табл.7, выполняются слева направо.

Таблица 7

Знак операции	**Операция**	**Группа операций**
*	Умножение	Мультипликативные
/	Деление	
%	Остаток от деления	
+	Сложение	Аддитивные
-	Вычитание	
<<	Сдвиг влево	Операции сдвига
>>	Сдвиг вправо	
<	Меньше	Операции отношения
<=	Меньше или равно	
>=	Больше или равно	
==	Равно	
!=	Не равно	
&	Поразрядное И	Поразрядные операции
|	Поразрядное ИЛИ	
^	Поразрядное исключающее ИЛИ	
&&	Логическое И	Логические операции
||	Логическое ИЛИ	
,	Последовательное вычисление	Последовательного вычисления
=	Присваивание	Операции присваивания
*=	Умножение с присваиванием	
/=	Деление с	

	присваиванием	
%=	Остаток от деления с присваиванием	
-=	Вычитание с присваиванием	
+=	Сложение с присваиванием	
<<=	Сдвиг влево с присваиванием	
>>=	Сдвиг вправо присваиванием	
&=	Поразрядное И с присваиванием	
|=	Поразрядное ИЛИ с присваиванием	
^=	Поразрядное исключающее ИЛИ с присваиванием	

Левый операнд операции присваивания должен быть выражением, ссылающимся на область памяти (но не объектом объявленным с ключевым словом const), такие выражения называются леводопустимыми к ним относятся:

- идентификаторы данных целого и плавающего типов, типов указателя, структуры, объединения;

- индексные выражения, исключая выражения имеющие тип массива или функции;

- выражения выбора элемента (->) и (.), если выбранный элемент является леводопустимым;

- выражения унарной операции разадресации (*), за исключением выражений, ссылающихся на массив или функцию;

- выражение приведения типа если результирующий тип не превышает размера первоначального типа.

При записи выражений следует помнить, что символы (*), (&), (!), (+) могут\ обозначать унарную или бинарную операцию.

1.3.2. Преобразования при вычислении выражений

При выполнении операций производится автоматическое преобразование типов, чтобы привести операнды выражений к общему типу или чтобы расширить короткие величины до размера целых величин, используемых в машинных командах. Выполнение преобразования зависит от специфики операций и от типа операнда или операндов.

Рассмотрим общие арифметические преобразования.

1. Операнды типа float преобразуются к типу double.

2. Если один операнд long double, то второй преобразуется к этому же типу.

3. Если один операнд double, то второй также преобразуется к типу double.

4. Любые операнды типа char и short преобразуются к типу int.

5. Любые операнды unsigned char или unsigned short преобразуются к типу unsigned int.

6. Если один операнд типа unsigned long, то второй преобразуется к типу unsigned long.

7. Если один операнд типа long, то второй преобразуется к типу long.

8. Если один операнд типа unsigned int, то второй операнд преобразуется к этому же типу.

Таким образом, можно отметить, что при вычислении выражений операнды преобразуются к типу того операнда, который имеет наибольший размер.

Пример:

```
double          ft,sd;
unsigned char   ch;
unsigned long   in;
int             i;
....
   sd=ft*(i+ch/in);
```

При выполнении оператора присваивания правила преобразования будут использоваться следующим образом. Операнд ch преобразуется к unsigned int (правило 5). Затем он преобразуется к типу unsigned long (правило 6). По этому же правилу i преобразуется к unsigned long и результат операции, заключенной в круглые скобки будет иметь тип unsigned long. Затем он преобразуется к типу double (правило 3) и результат всего выражения будет иметь тип double.

1.3.3. Операции отрицания и дополнения

Операция арифметического отрицания (-) вырабатывает отрицание своего операнда. Операнд должен быть целой или плавающей величиной. При выполнении осуществляются обычные арифметические преобразования.

Пример:

```
double  u = 5;
u = -u;          /* переменной u присваивается ее отрицание,
                       т.е. u  принимает значение -5     */
```

Операция логического отрицания "НЕ" (!) вырабатывает значение 0, если операнд есть истина (не нуль), и значение 1, если операнд равен нулю (0). Результат имеет тип int. Операнд должен быть целого или плавающего типа или типа указатель.

Пример:

```
int t, z=0;
```

```
t=!z;
```

Переменная t получит значение равное 1, так как переменная z имела значение равное 0 (ложно).

Операция двоичного дополнения (~) вырабатывает двоичное дополнение своего операнда. Операнд должен быть целого типа. Осуществляется обычное арифметическое преобразование, результат имеет тип операнда после преобразования.

Пример:

```
char            b = '9';
unsigned char   f;
 b = ~f;
```

Шестнадцатеричное значение символа '9' равно 39. В результате операции ~f будет получено шестнадцатеричное значение C6, что соответствует символу 'ц'.

1.3.4. Операции разадресации и адреса

Эти операции используются для работы с переменными типа указатель.

Операция разадресации (*) осуществляет косвенный доступ к адресуемой величине через указатель. Операнд должен быть указателем. Результатом операции является величина, на которую указывает операнд. Типом результата является тип величины, адресуемой указателем. Результат не определен, если указатель содержит недопустимый адрес.

Рассмотрим типичные ситуации, когда указатель содержит недопустимый адрес:

- указатель является нулевым;

- указатель определяет адрес такого объекта, который не является активным в момент ссылки;

- указатель определяет адрес, который не выровнен до типа объекта, на который он указывает;

- указатель определяет адрес, не используемый выполняющейся программой.

Операция адрес (&) дает адрес своего операнда. Операндом может быть любое именуемое выражение. Имя функции или массива также может быть операндом операции "адрес", хотя в этом случае знак операции является лишним, так как имена массивов и функций являются адресами. Результатом операции адрес является указатель на операнд. Тип, адресуемый указателем, является типом операнда.

Операция адрес не может применятся к элементам структуры, являющимися полями битов, и к объектам с классом памяти register.

Примеры:

```
int t, f=0,  * adress;
adress = &t; /* переменной adress, объявляемой как
                указатель, присваивается адрес переменной t */
```

```
* adress =f; /* переменной находящейся по адресу, содержащемуся
               в переменной adress, присваивается значение
               переменной f,  т.е.  0 , что эквивалентно
               t=f;      т.е.   t=0;                      */
```

1.3.5. Операция sizeof

С помощью операции sizeof можно определить размер памяти которая соответствует идентификатору или типу. Операция sizeof имеет следующий формат:

sizeof(выражение) .

В качестве выражения может быть использован любой идентификатор, либо имя типа, заключенное в скобки. Отметим, что не может быть использовано имя типа void, а идентификатор не может относится к полю битов или быть именем функции.

Если в качестве выражения указанно имя массива, то результатом является размер всего массива (т.е. произведение числа элементов на длину типа), а не размер указателя, соответствующего идентификатору массива.

Когда sizeof применяются к имени типа структуры или объединения или к идентификатору имеющему тип структуры или объединения, то результатом является фактический размер структуры или объединения, который может включать участки памяти, используемые для выравнивания элементов структуры или объединения. Таким образом, этот результат может не соответствовать размеру, получаемому путем сложения размеров элементов структуры.

Пример:

```
struct     { char   h;
             int    b;
             double f;
           } str;
int a1;
a1  = sizeof(str);
```

Переменная a1 получит значение, равное 12, в то же время если сложить длины всех используемых в структуре типов, то получим, что длина структуры str равна 7.

Несоответствие имеет место в виду того, что после размещения в памяти первой переменной h длинной 1 байт, добавляется 1 байт для выравнивания адреса переменной b на границу слова (слово имеет длину 2 байта для машин серии IBM PC AT /286/287), далее осуществляется выравнивание адреса переменной f на границу двойного слова (4 байта), таким образом в результате операций выравнивания для размещения структуры в оперативной памяти требуется на 5 байт больше.

В связи с этим целесообразно рекомендовать при объявлении структур и объединения располагать их элементы в порядке убывания длины типов, т.е. приведенную выше структуру следует записать в следующем виде:

```
struct  { double f;
          int    b;
          char   h;
        } str;
```

1.3.6. Мультипликативные операции

К этому классу операций относятся операции умножения (*), деления (/) и получение остатка от деления (%). Операндами операции (%) должны быть целые числа. Отметим, что типы операндов операций умножения и деления могут отличаться, и для них справедливы правила преобразования типов. Типом результата является тип операндов после преобразования.

Операция умножения (*) выполняет умножение операндов.

```
int  i=5;
float f=0.2;
double g,z;
    g=f*i;
```

Тип произведения i и f преобразуется к типу double, затем результат присваивается переменной g.

Операция деления (/) выполняет деление первого операнда на второй. Если две целые величины не делятся нацело, то результат округляется в сторону нуля.

При попытке деления на ноль выдается сообщение во время выполнения.

```
int i=49, j=10,  n, m;
n = i/j;                  /* результат   4   */
m = i/(-j);               /* результат  -4   */
```

Операция остаток от деления (%) дает остаток от деления первого операнда на второй.

Знак результата зависит от конкретной реализации. В данной реализации знак результата совпадает со знаком делимого. Если второй операнд равен нулю, то выдается сообщение.

```
int n = 49, m = 10, i, j, k, l;
i = n % m;              /*    9    */
j = n % (-m);           /*    9    */
k = (-n) % m;           /*    -9   */
l = (-n) % (-m);        /*    -9   */
```

1.3.7. Аддитивные операции

К аддитивным операциям относятся сложение (+) и вычитание (-). Операнды могут быть целого или плавающего типов. В некоторых случаях над операндами аддитивных операций выполняются общие арифметические преобразования. Однако преобразования, выполняемые при аддитивных операциях, не обеспечивают обработку ситуаций переполнения и потери значимости. Информация теряется, если результат аддитивной операции не может быть представлен типом операндов после преобразования. При этом сообщение об ошибке не выдается.

Пример:

```
int   i=30000, j=30000, k;
       k=i+j;
```

В результате сложения k получит значение равное -5536.

Результатом выполнения операции сложения является сумма двух операндов. Операнды могут быть целого или плавающего типа или один операнд может быть указателем, а второй - целой величиной.

Когда целая величина складывается с указателем, то целая величина преобразуется путем умножения ее на размер памяти, занимаемой величиной, адресуемой указателем.

Когда преобразованная целая величина складывается с величиной указателя, то результатом является указатель, адресующий ячейку памяти, расположенную на целую величину дальше от исходного адреса. Новое значение указателя адресует тот же самый тип данных, что и исходный указатель.

Операция вычитания (-) вычитает второй операнд из первого. Возможна следующая комбинация операндов:

1. Оба операнда целого или плавающего типа.

2. Оба операнда являются указателями на один и тот же тип.

3. Первый операнд является указателем, а второй - целым.

Отметим, что операции сложения и вычитания над адресами в единицах, отличных от длины типа, могут привести к непредсказуемым результатам.

Пример:

```
double d[10],* u;
int i;
u = d+2;  /*  u  указывает на третий элемент массива */
i = u-d;  /*  i  принимает значение равное 2         */
```

1.3.8. Операции сдвига

Операции сдвига осуществляют смещение операнда влево (<<) или вправо (>>) на число битов, задаваемое вторым операндом. Оба операнда должны быть целыми величинами. Выполняются обычные арифметические преобразования. При сдвиге влево правые освобождающиеся биты устанавливаются в нуль. При сдвиге вправо метод заполнения освобождающихся левых битов зависит от типа первого операнда. Если тип unsigned, то свободные левые биты устанавливаются в нуль. В противном случае они заполняются копией знакового бита. Результат операции сдвига не определен, если второй операнд отрицательный.

Преобразования, выполненные операциями сдвига, не обеспечивают обработку ситуаций переполнения и потери значимости. Информация теряется, если результат операции сдвига не может быть представлен типом первого операнда, после преобразования.

Отметим, что сдвиг влево соответствует умножению первого операнда на степень числа 2, равную второму операнду, а сдвиг вправо соответствует делению первого операнда на 2 в степени, равной второму операнду.

Примеры:

```
int  i=0x1234, j,  k ;
```

```
k = i<<4 ;          /*    k = 0x0234   */
j = i<<8 ;          /*    j = 0x3400   */
i = j>>8 ;          /*    i = 0x0034   */
```

1.3.9. Поразрядные операции

К поразрядным операциям относятся: операция поразрядного логического "И" (&), операция поразрядного логического "ИЛИ" (|), операция поразрядного "исключающего ИЛИ" (^).

Операнды поразрядных операций могут быть любого целого типа. При необходимости над операндами выполняются преобразования по умолчанию, тип результата - это тип операндов после преобразования.

Операция поразрядного логического И (&) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба сравниваемых бита единицы, то соответствующий бит результата устанавливается в 1, в противном случае в 0.

Операция поразрядного логического ИЛИ (|) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если любой (или оба) из сравниваемых битов равен 1, то соответствующий бит результата устанавливается в 1, в противном случае результирующий бит равен 0.

Операция поразрядного исключающего ИЛИ (^) сравнивает каждый бит первого операнда с соответствующими битами второго операнда. Если один из сравниваемых битов равен 0, а второй бит равен 1, то соответствующий бит результата устанавливается в 1, в противном случае, т.е. когда оба бита равны 1 или 0, бит результата устанавливается в 0.

Пример.

```
int  i=0x45FF,     /*      i= 0100 0101 1111 1111   */
     j=0x00FF;             j= 0000 0000 1111 1111   */
char r;
  r = i^j;       /* r=0x4500 = 0100 0101 0000 0000   */
  r = i|j;       /* r=0x45FF = 0100 0101 0000 0000   */
  r = i&j;       /* r=0x00FF = 0000 0000 1111 1111    */
```

1.3.10. Логические операции

К логическим операциям относятся операция логического И (&&) и операция логического ИЛИ (||). Операнды логических операций могут быть целого типа, плавающего типа или типа указателя, при этом в каждой операции могут участвовать операнды различных типов.

Операнды логических выражений вычисляются слева направо. Если значения первого операнда достаточно, чтобы определить результат операции, то второй операнд не вычисляется.

Логические операции не вызывают стандартных арифметических преобразований. Они оценивают каждый операнд с точки зрения его эквивалентности нулю. Результатом логической операции является 0 или 1, тип результата int.

Операция логического И (&&) вырабатывает значение 1, если оба операнда имеют нулевые значения. Если один из операндов равен 0, то результат также равен 0. Если значение первого операнда равно 0, то второй операнд не вычисляется.

Операция логического ИЛИ (||) выполняет над операндами операцию включающего ИЛИ. Она вырабатывает значение 0, если оба операнда имеют значение 0, если какой-либо из операндов имеет ненулевое значение, то результат операции равен 1. Если первый операнд имеет ненулевое значение, то второй операнд не вычисляется.

1.3.11. Операция последовательного вычисления

Операция последовательного вычисления обозначается запятой (,) и используется для вычисления двух и более выражений там, где по синтаксису допустимо только одно выражение. Эта операция вычисляет два операнда слева направо. При выполнении операции последовательного вычисления, преобразование типов не производится. Операнды могут быть любых типов. Результат операции имеет значения и тип второго операнда. Отметим, что запятая может использоваться также как символ разделитель, поэтому необходимо по контексту различать, запятую, используемую в качестве разделителя или знака операции.

1.3.12. Условная операция

В языке СИ имеется одна тернарная операция - условная операция, которая имеет следующий формат:

операнд-1 ? операнд-2 : операнд-3

Операнд-1 должен быть целого или плавающего типа или быть указателем. Он оценивается с точки зрения его эквивалентности 0. Если операнд-1 не равен 0, то вычисляется операнд-2 и его значение является результатом операции. Если операнд-1 равен 0, то вычисляется операнд-3 и его значение является результатом операции. Следует отметить, что вычисляется либо операнд-2, либо операнд-3, но не оба. Тип результата зависит от типов операнда-2 и операнда-3, следующим образом.

1. Если операнд-2 или операнд-3 имеет целый или плавающий тип (отметим, что их типы могут отличаться), то выполняются обычные арифметические преобразования. Типом результата является тип операнда после преобразования.

2. Если операнд-2 и операнд-3 имеют один и тот же тип структуры, объединения или указателя, то тип результата будет тем же самым типом структуры, объединения или указателя.

3. Если оба операнда имеют тип void, то результат имеет тип void.

4. Если один операнд является указателем на объект любого типа, а другой операнд является указателем на vold, то указатель на объект преобразуется к указателю на vold, который и будет типом результата.

5. Если один из операндов является указателем, а другой константным выражением со значением 0, то типом результата будет тип указателя.

Пример:

max = (d<=b) ? b : d;

Переменной max присваивается максимальное значение переменных d и b.

1.3.13. Операции увеличения и уменьшения

Операции увеличения (++) и уменьшения (--) являются унарными операциями присваивания. Они соответственно увеличивают или уменьшают значения операнда на единицу. Операнд может быть целого или плавающего типа или типа указатель и должен быть модифицируемым. Операнд целого или плавающего типа увеличиваются (уменьшаются) на единицу. Тип результата соответствует типу операнда. Операнд адресного типа увеличивается или уменьшается на размер объекта, который он адресует. В языке допускается префиксная или постфиксная формы операций увеличения (уменьшения), поэтому значения выражения, использующего операции увеличения (уменьшения) зависит от того, какая из форм указанных операций используется.

Если знак операции стоит перед операндом (префиксная форма записи), то изменение операнда происходит до его использования в выражении и результатом операции является увеличенное или уменьшенное значение операнда.

В том случае если знак операции стоит после операнда (постфиксная форма записи), то операнд вначале используется для вычисления выражения, а затем происходит изменение операнда.

Примеры:

```
int t=1, s=2, z, f;
     z=(t++)*5;
```

Вначале происходит умножение t*5, а затем увеличение t. В результате получится z=5, t=2.

```
     f=(++s)/3;
```

Вначале значение s увеличивается, а затем используется в операции деления. В результате получим s=3, f=1.

В случае, если операции увеличения и уменьшения используются как самостоятельные операторы, префиксная и постфиксная формы записи становятся эквивалентными.

z++; /* эквивалентно */ ++z;

1.3.14. Простое присваивание

Операция простого присваивания используется для замены значения левого операнда, значением правого операнда. При присваивании производится преобразование типа правого операнда к типу левого операнда по правилам, упомянутым раньше. Левый операнд должен быть модифицируемым.

Пример:

```
int t;
char f;
```

```
long z;
t=f+z;
```

Значение переменной f преобразуется к типу long, вычисляется f+z ,результат преобразуется к типу int и затем присваивается переменной t.

1.3.15. Составное присваивание

Кроме простого присваивания, имеется целая группа операций присваивания, которые объединяют простое присваивание с одной из бинарных операций. Такие операции называются составными операциями присваивания и имеют вид:

(операнд-1) (бинарная операция) = (операнд-2) .

Составное присваивание по результату эквивалентно следующему простому присваиванию:

(операнд-1) = (операнд-1) (бинарное операция) (операнд-2) .

Отметим, что выражение составного присваивания с точки зрения реализации не эквивалентно простому присваиванию, так как в последнем операнд-1 вычисляется дважды.

Каждая операция составного присваивания выполняет преобразования, которые осуществляются соответствующей бинарной операцией. Левым операндом операций (+=) (-=) может быть указатель, в то время как правый операнд должен быть целым числом.

Примеры:

```
double  arr[4]={ 2.0, 3.3, 5.2, 7.5 } ;
double  b=3.0;
b+=arr[2];      /*  эквивалентно   b=b+arr[2]            */
arr[3]/=b+1;    /*  эквивалентно   arr[3]=arr[3]/(b+1)   */
```

Заметим, что при втором присваивании использование составного присваивания дает более заметный выигрыш во времени выполнения, так как левый операнд является индексным выражением.

1.3.16. Приоритеты операций и порядок вычислений

В языке СИ операции с высшими приоритетами вычисляются первыми. Наивысшим приоритетом является приоритет равный 1. Приоритеты и порядок операций приведены в табл. 8.

Таблица 8

Приоритет	Знак операции	Типы операции	Порядок выполнения
2	() [] . ->	Выражение	Слева направо
1	- ~ ! * & ++ -- sizeof приведение	Унарные	Справа налево

	типов		
3	* / %	Мультипликативные	Слева направо
4	+ -	Аддитивные	
5	<< >>	Сдвиг	
6	< > <= >=	Отношение	
7	== !=	Отношение (равенство)	
8	&	Поразрядное И	
9	^	Поразрядное исключающее ИЛИ	
10	|	Поразрядное ИЛИ	
11	&&	Логическое И	
12	||	Логическое ИЛИ	
13	? :	Условная	
14	= *= /= %= += -= &= |= >>= <<= ^=	Простое и составное присваивание	Справа налево
15	,	Последовательное вычисление	Слева направо

1.3.17. Побочные эффекты

Операции присваивания в сложных выражениях могут вызывать побочные эффекты, так как они изменяют значение переменной. Побочный эффект может возникать и при вызове функции, если он содержит прямое или косвенное присваивание (через указатель). Это связано с тем, что аргументы функции могут вычисляться в любом порядке. Например, побочный эффект имеет место в следующем вызове функции:

prog (a,a=k*2);

В зависимости от того, какой аргумент вычисляется первым, в функцию могут быть переданы различные значения.

Порядок вычисления операндов некоторых операций зависит от реализации и поэтому могут возникать разные побочные эффекты, если в одном из операндов используется операции увеличения или уменьшения, а также другие операции присваивания.

Например, выражение i*j+(j++)+(--i) может принимать различные значения при обработке разными компиляторами. Чтобы избежать недоразумений при выполнении побочных эффектов необходимо придерживаться следующих правил.

1. Не использовать операции присваивания переменной в вызове функции, если эта переменная участвует в формировании других аргументов функции.

2. Не использовать операции присваивания переменной в выражении, если эта переменная используется в выражении более одного раза.

1.3.18. Преобразование типов

При выполнении операций происходят неявные преобразования типов в следующих случаях:

- при выполнении операций осуществляются обычные арифметические преобразования (которые были рассмотрены выше);

- при выполнении операций присваивания, если значение одного типа присваивается переменной другого типа;

- при передаче аргументов функции.

Кроме того, в Си есть возможность явного приведения значения одного типа к другому.

В операциях присваивания тип значения, которое присваивается, преобразуется к типу переменной, получающей это значение. Допускается преобразования целых и плавающих типов, даже если такое преобразование ведет к потере информации.

Преобразование целых типов со знаком. Целое со знаком преобразуется к более короткому целому со знаком, посредством усечения старших битов. Целая со знаком преобразуется к более длинному целому со знаком, путем размножения знака. При преобразовании целого со знаком к целому без знака, целое со знаком преобразуется к размеру целого без знака и результат рассматривается как значение без знака.

Преобразование целого со знаком к плавающему типу происходит без потери] информации, за исключением случая преобразования значения типа long int или unsigned long int к типу float, когда точность часто может быть потеряна.

Преобразование целых типов без знака. Целое без знака преобразуется к более короткому целому без знака или со знаком путем усечения старших битов. Целое без знака преобразуется к более длинному целому без знака или со знаком путем дополнения нулей слева.

Когда целое без знака преобразуется к целому со знаком того же размера, битовое представление не изменяется. Поэтому значение, которое оно представляет, изменяется, если знаковый бит установлен (равен 1), т.е. когда исходное целое без знака больше чем максимальное положительное целое со знаком, такой же длины.

Целые значения без знака преобразуются к плавающему типу, путем преобразования целого без знака к значению типа signed long, а затем значение signed long преобразуется в плавающий тип. Преобразования из unsigned long к типу float, double или long double производятся с потерей информации, если преобразуемое значение больше, чем максимальное положительное значение, которое может быть представлено для типа long.

Преобразования плавающих типов. Величины типа float преобразуются к типу double без изменения значения. Величины double и long double преобразуются к float с некоторой потерей точности. Если значение слишком велико для float, то происходит потеря значимости, о чем сообщается во время выполнения.

При преобразовании величины с плавающей точкой к целым типам она сначала преобразуется к типу long (дробная часть плавающей величины при этом отбрасывается), а затем величина типа long преобразуется к требуемому целому типу. Если значение слишком велико для long, то результат преобразования не определен.

Преобразования из float, double или long double к типу unsigned long производится с потерей точности, если преобразуемое значение больше, чем максимально возможное положительное значение, представленное типом long.

Преобразование типов указателя. Указатель на величину одного типа может быть преобразован к указателю на величину другого типа. Однако результат может быть не определен из-за отличий в требованиях к выравниванию и размерах для различных типов.

Указатель на тип void может быть преобразован к указателю на любой тип, и указатель на любой тип может быть преобразован к указателю на тип void без ограничений. Значение указателя может быть преобразовано к целой величине. Метод преобразования зависит от размера указателя и размера целого типа следующим образом:

- если размер указателя меньше размера целого типа или равен ему, то указатель преобразуется точно так же, как целое без знака;

- если указатель больше, чем размер целого типа, то указатель сначала преобразуется к указателю с тем же размером, что и целый тип, и затем преобразуется к целому типу.

Целый тип может быть преобразован к адресному типу по следующим правилам:

- если целый тип того же размера, что и указатель, то целая величина просто рассматривается как указатель (целое без знака);

- если размер целого типа отличен от размера указателя, то целый тип сначала преобразуется к размеру указателя (используются способы преобразования, описанные выше), а затем полученное значение трактуется как указатель.

Преобразования при вызове функции. Преобразования, выполняемые над аргументами при вызове функции, зависят от того, был ли задан прототип функции (объявление "вперед") со списком объявлений типов аргументов.

Если задан прототип функции и он включает объявление типов аргументов, то над аргументами в вызове функции выполняются только обычные арифметические преобразования.

Эти преобразования выполняются независимо для каждого аргумента. Величины типа float преобразуются к double, величины типа char и short преобразуются к int, величины типов unsigned char и unsigned short преобразуются к unsigned int. Могут быть также выполнены неявные преобразования переменных типа указатель. Задавая прототипы функций, можно переопределить эти неявные преобразования и позволить компилятору выполнить контроль типов.

Преобразования при приведении типов. Явное преобразование типов может быть осуществлено посредством операции приведения типов, которая имеет формат:

(имя-типа) операнд .

В приведенной записи имя-типа задает тип, к которому должен быть преобразован операнд.

Пример:

```
int     i=2;
long    l=2;
double    d;
float     f;
d=(double)i * (double)l;
f=(float)d;
```

В данном примере величины i,l,d будут явно преобразовываться к указанным в круглых скобках типам.

1.4. Операторы

Все операторы языка СИ могут быть условно разделены на следующие категории:

- условные операторы, к которым относятся оператор условия if и оператор выбора switch;

- операторы цикла (for,while,do while);

- операторы перехода (break, continue, return, goto);

- другие операторы (оператор "выражение", пустой оператор).

Операторы в программе могут объединяться в составные операторы с помощью фигурных скобок. Любой оператор в программе может быть помечен меткой, состоящей из имени и следующего за ним двоеточия.

Все операторы языка СИ, кроме составных операторов, заканчиваются точкой с запятой ";".

1.4.1. Оператор выражение

Любое выражение, которое заканчивается точкой с запятой, является оператором.

Выполнение оператора выражение заключается в вычислении выражения. Полученное значение выражения никак не используется, поэтому, как правило, такие выражения вызывают побочные эффекты. Заметим, что вызвать функцию, невозвращающую значения можно только при помощи оператора выражения. Правила вычисления выражений были сформулированы выше.

Примеры:

++ i;

Этот оператор представляет выражение, которое увеличивает значение переменной i на единицу.

a=cos(b * 5);

Этот оператор представляет выражение, включающее в себя операции присваивания и вызова функции.

a(x,y);

Этот оператор представляет выражение состоящее из вызова функции.

1.4.2. Пустой оператор

Пустой оператор состоит только из точки с запятой. При выполнении этого оператора ничего не происходит. Он обычно используется в следующих случаях:

- в операторах do, for, while, if в строках, когда место оператора не требуется, но по синтаксису требуется хотя бы один оператор;

- при необходимости пометить фигурную скобку.

Синтаксис языка СИ требует, чтобы после метки обязательно следовал оператор. Фигурная же скобка оператором не является. Поэтому, если надо передать управление на фигурную скобку, необходимо использовать пустой оператор.

Пример:

```
int main ( )
{
     :
      { if (...) goto a;    /* переход на скобку */
          { ...
          }
 a:;  }
      return 0;
  }
```

1.4.3. Составной оператор

Составной оператор представляет собой несколько операторов и объявлений, заключенных в фигурные скобки:

```
{  [объявление]
    :
    оператор; [оператор];
    :
}
```

Заметим, что в конце составного оператора точка с запятой не ставится.

Выполнение составного оператора заключается в последовательном выполнении составляющих его операторов.

Пример:

```
int main ()
{
  int    q,b;
  double t,d;
    :
    if (...)
      {
        int    e,g;
        double f,q;
        :
       }
    :
    return (0);
}
```

Переменные e,g,f,q будут уничтожены после выполнения составного оператора. Отметим, что переменная q является локальной в составном операторе, т.е. она никоим образом не связана с переменной q объявленной в начале функции main с типом int. Отметим также, что выражение стоящее после return может быть заключено в круглые скобки, хотя наличие последних необязательно.

1.4.4. Оператор if

Формат оператора:

if (выражение) оператор-1; [else оператор-2;]

Выполнение оператора if начинается с вычисления выражения.

Далее выполнение осуществляется по следующей схеме:

- если выражение истинно (т.е. отлично от 0), то выполняется оператор-1.

- если выражение ложно (т.е. равно 0),то выполняется оператор-2.

- если выражение ложно и отсутствует оператор-2 (в квадратные скобки заключена необязательная конструкция), то выполняется следующий за if оператор.

После выполнения оператора if значение передается на следующий оператор программы, если последовательность выполнения операторов программы не будет принудительно нарушена использованием операторов перехода.

Пример:

```
if (i < j)   i++:
else { j = i-3;   i++; }
```

Этот пример иллюстрирует также и тот факт, что на месте оператор-1, так же как и на месте оператор-2 могут находиться сложные конструкции.

Допускается использование вложенных операторов if. Оператор if может быть включен в конструкцию if или в конструкцию else другого оператора if. Чтобы сделать программу более читабельной, рекомендуется группировать операторы и конструкции во вложенных операторах if, используя фигурные скобки. Если же фигурные скобки опущены, то

компилятор связывает каждое ключевое слово else с наиболее близким if, для которого нет else.

Примеры:

```
int main ( )
 {
   int t=2, b=7, r=3;
     if (t>b)
      {
          if (b < r)  r=b;
      }
     else r=t;
   return (0);
 }
```

В результате выполнения этой программы r станет равным 2.

Если же в программе опустить фигурные скобки, стоящие после оператора if, то программа будет иметь следующий вид:

```
int main  ( )
 {
   int t=2,b=7,r=3;
   if ( a>b )
       if ( b < c ) t=b;
       else       r=t;
   return (0);
  }
```

В этом случае r получит значение равное 3, так как ключевое слово else относится ко второму оператору if, который не выполняется, поскольку не выполняется условие, проверяемое в первом операторе if.

Следующий фрагмент иллюстрирует вложенные операторы if:

```
char ZNAC;
int x,y,z;
  :
 if (ZNAC == '-') x = y - z;
 else  if (ZNAC == '+') x = y + z;
       else if (ZNAC == '*') x = y * z;
             else if (ZNAC == '/') x = y / z;
                   else ...
```

Из рассмотрения этого примера можно сделать вывод, что конструкции использующие вложенные операторы if, являются довольно громоздкими и не всегда достаточно надежными. Другим способом организации выбора из множества различных вариантов является использование специального оператора выбора switch.

1.4.5. Оператор switch

Оператор switch предназначен для организации выбора из множества различных вариантов. Формат оператора следующий:

```
switch ( выражение )
{   [объявление]
```

```
            :
        [ case  константное-выражение1]: [ список-операторов1]
        [ case  константное-выражение2]: [ список-операторов2]
            :
            :
        [ default: [ список операторов ]]
}
```

Выражение, следующее за ключевым словом switch в круглых скобках, может быть любым выражением, допустимыми в языке СИ, значение которого должно быть целым. Отметим, что можно использовать явное приведение к целому типу, однако необходимо помнить о тех ограничениях и рекомендациях, о которых говорилось выше.

Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора smitch состоит из нескольких операторов, помеченных ключевым словом case с последующим константным-выражением. Следует отметить, что использование целого константного выражения является существенным недостатком, присущим рассмотренному оператору.

Так как константное выражение вычисляется во время трансляции, оно не может содержать переменные или вызовы функций. Обычно в качестве константного выражения используются целые или символьные константы.

Все константные выражения в операторе switch должны быть уникальны. Кроме операторов, помеченных ключевым словом case, может быть, но обязательно один, фрагмент помеченный ключевым словом default.

Список операторов может быть пустым, либо содержать один или более операторов. Причем в операторе switch не требуется заключать последовательность операторов в фигурные скобки.

Отметим также, что в операторе switch можно использовать свои локальные переменные, объявления которых находятся перед первым ключевым словом case, однако в объявлениях не должна использоваться инициализация.

Схема выполнения оператора switch следующая:

- вычисляется выражение в круглых скобках;

- вычисленные значения последовательно сравниваются с константными выражениями, следующими за ключевыми словами case;

- если одно из константных выражений совпадает со значением выражения, то управление передается на оператор, помеченный соответствующим ключевым словом case;

- если ни одно из константных выражений не равно выражению, то управление передается на оператор, помеченный ключевым словом default, а в случае его отсутствия управление передается на следующий после switch оператор.

Отметим интересную особенность использования оператора switch: конструкция со словом default может быть не последней в теле оператора switch. Ключевые слова case и default в теле оператора switch существенны только при начальной проверке, когда определяется начальная точка выполнения тела оператора switch. Все операторы, между

начальным оператором и концом тела, выполняются вне зависимости от ключевых слов, если только какой-то из операторов не передаст управления из тела оператора switch. Таким образом, программист должен сам позаботится о выходе из case, если это необходимо. Чаще всего для этого используется оператор break.

Для того, чтобы выполнить одни и те же действия для различных значений выражения, можно пометить один и тот же оператор несколькими ключевыми словами case.

Пример:

```
 int i=2;
 switch (i)
   {
    case 1: i += 2;
    case 2: i *= 3;
    case 0: i /= 2;
    case 4: i -= 5;
    default:      ;
}
```

Выполнение оператора switch начинается с оператора, помеченного case 2. Таким образом, переменная i получает значение, равное 6, далее выполняется оператор, помеченный ключевым словом case 0, а затем case 4, переменная i примет значение 3, а затем значение -2. Оператор, помеченный ключевым словом default, не изменяет значения переменной.

Рассмотрим ранее приведенный пример, в котором иллюстрировалось использование вложенных операторов if, переписанной теперь с использованием оператора switch.

```
char ZNAC;
int x,y,z;
switch (ZNAC)
{
     case '+':  x = y + z;    break;
     case '-':  x = y - z;    break;
     case '*':  x = y * z;    break;
     case '/':  x = u / z;    break;
     default : ;
   }
```

Использование оператора break позволяет в необходимый момент прервать последовательность выполняемых операторов в теле оператора switch, путем передачи управления оператору, следующему за switch.

Отметим, что в теле оператора switch можно использовать вложенные операторы switch, при этом в ключевых словах case можно использовать одинаковые константные выражения.

Пример:

```
  :
switch (a)
 {
   case 1: b=c; break;
   case 2:
       switch (d)
       {   case 0:  f=s;  break;
```

```
            case 1:  f=9;  break;
            case 2:  f-=9; break;
         }
      case 3: b-=c; break;
      :
   }
```

1.4.6. Оператор break

Оператор break обеспечивает прекращение выполнения самого внутреннего из объединяющих его операторов switch, do, for, while. После выполнения оператора break управление передается оператору, следующему за прерванным.

1.4.7. Оператор for

Оператор for - это наиболее общий способ организации цикла. Он имеет следующий формат:

for (выражение 1 ; выражение 2 ; выражение 3) тело

Выражение 1 обычно используется для установления начального значения переменных, управляющих циклом. Выражение 2 - это выражение, определяющее условие, при котором тело цикла будет выполняться. Выражение 3 определяет изменение переменных, управляющих циклом после каждого выполнения тела цикла.

Схема выполнения оператора for:

1. Вычисляется выражение 1.

2. Вычисляется выражение 2.

3. Если значения выражения 2 отлично от нуля (истина), выполняется тело цикла, вычисляется выражение 3 и осуществляется переход к пункту 2, если выражение 2 равно нулю (ложь), то управление передается на оператор, следующий за оператором for.

Существенно то, что проверка условия всегда выполняется в начале цикла. Это значит, что тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным.

Пример:

```
int main()
{ int i,b;
   for (i=1; i<10; i++)
      b=i*i;
   return 0;
}
```

В этом примере вычисляются квадраты чисел от 1 до 9.

Некоторые варианты использования оператора for повышают его гибкость за счет возможности использования нескольких переменных, управляющих циклом.

Пример:

```
int main()
{  int top,  bot;
   char string[100],  temp;
   for ( top=0, bot=100 ; top < bot ; top++, bot--)
   {  temp=string[top];
      string[bot]=temp;
   }
   return 0;
 }
```

В этом примере, реализующем запись строки символов в обратном порядке, для управления циклом используются две переменные top и bot. Отметим, что на месте выражение 1 и выражение 3 здесь используются несколько выражений, записанных через запятую, и выполняемых последовательно.

Другим вариантом использования оператора for является бесконечный цикл. Для организации такого цикла можно использовать пустое условное выражение, а для выхода из цикла обычно используют дополнительное условие и оператор break.

Пример:

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

Так как согласно синтаксису языка Си оператор может быть пустым, тело оператора for также может быть пустым. Такая форма оператора может быть использована для организации поиска.

Пример:

for (i=0; t[i]<10 ; i++) ;

В данном примере переменная цикла i принимает значение номера первого элемента массива t, значение которого больше 10.

1.4.8. Оператор while

Оператор цикла while называется циклом с предусловием и имеет следующий формат:

while (выражение) тело ;

В качестве выражения допускается использовать любое выражение языка Си, а в качестве тела любой оператор, в том числе пустой или составной. Схема выполнения оператора while следующая:

1. Вычисляется выражение.

2. Если выражение ложно, то выполнение оператора while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполняется тело оператора while.

3. Процесс повторяется с пункта 1.

Оператор цикла вида

for (выражение-1; выражение-2; выражение-3) тело ;

может быть заменен оператором while следующим образом:

```
выражение-1;
while (выражение-2)
{  тело
   выражение-3;
}
```

Так же как и при выполнении оператора for, в операторе while вначале происходит проверка условия. Поэтому оператор while удобно использовать в ситуациях, когда тело оператора не всегда нужно выполнять.

Внутри операторов for и while можно использовать локальные переменные, которые должны быть объявлены с определением соответствующих типов.

1.4.9. Оператор do while

Оператор цикла do while называется оператором цикла с постусловием и используется в тех случаях, когда необходимо выполнить тело цикла хотя бы один раз. Формат оператора имеет следующий вид:

do тело while (выражение);

Схема выполнения оператора do while :

1. Выполняется тело цикла (которое может быть составным оператором).

2. Вычисляется выражение.

3. Если выражение ложно, то выполнение оператора do while заканчивается и выполняется следующий по порядку оператор. Если выражение истинно, то выполнение оператора продолжается с пункта 1.

Чтобы прервать выполнение цикла до того, как условие станет ложным, можно использовать оператор break.

Операторы while и do while могут быть вложенными.

Пример:

```
int i,j,k;
...
i=0;  j=0;  k=0;
do { i++;
     j--;
     while (a[k] < i)  k++;
    }
while (i<30 && j<-30);
```

1.4.10. Оператор continue

Оператор continue, как и оператор break, используется только внутри операторов цикла, но в отличие от него выполнение программы продолжается не с оператора, следующего за прерванным оператором, а с начала прерванного оператора. Формат оператора следующий:

continue;

Пример:

```
int main()
{   int a,b;
    for (a=1,b=0; a<100; b+=a,a++)
    {  if (b%2) continue;
       ...    /* обработка четных сумм  */
    }
    return 0;
}
```

Когда сумма чисел от 1 до a становится нечетной, оператор continue передает управление на очередную итерацию цикла for, не выполняя операторы обработки четных сумм.

Оператор continue, как и оператор break, прерывает самый внутренний из объемлющих его циклов.

1.4.11. Оператор return

Оператор return завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию, в точку, непосредственно следующую за вызовом. Функция main передает управление операционной системе. Формат оператора:

return [выражение] ;

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Выражение может быть заключено в круглые скобки, хотя их наличие не обязательно.

Если в какой-либо функции отсутствует оператор return, то передача управления в вызывающую функцию происходит после выполнения последнего оператора вызываемой функции. При этом возвращаемое значение не определено. Если функция не должна иметь возвращаемого значения, то ее нужно объявлять с типом void.

Таким образом, использование оператора return необходимо либо для немедленного выхода из функции, либо для передачи возвращаемого значения.

Пример:

```
int sum (int a, int b)
{   renurn (a+b);    }
```

Функция sum имеет два формальных параметра a и b типа int, и возвращает значение типа int, о чем говорит описатель, стоящий перед именем функции. Возвращаемое оператором return значение равно сумме фактических параметров.

Пример:

```
void prov (int a, double b)
{  double c;
   if (a<3) return;
   else if (b>10) return;
        else { c=a+b;
               if ((2*c-b)==11) return;
              }
 }
```

В этом примере оператор return используется для выхода из функции в случае выполнения одного из проверяемых условий.

1.4.12. Оператор goto

Использование оператора безусловного перехода goto в практике программирования на языке СИ настоятельно не рекомендуется, так как он затрудняет понимание программ и возможность их модификаций.

Формат этого оператора следующий:

```
goto имя-метки;
          ...
имя-метки: оператор;
```

Оператор goto передает управление на оператор, помеченный меткой имя-метки. Помеченный оператор должен находиться в той же функции, что и оператор goto, а используемая метка должна быть уникальной, т.е. одно имя-метки не может быть использовано для разных операторов программы. Имя-метки - это идентификатор.

Любой оператор в составном операторе может иметь свою метку. Используя оператор goto, можно передавать управление внутрь составного оператора. Но нужно быть осторожным при входе в составной оператор, содержащий объявления переменных с инициализацией, так как объявления располагаются перед выполняемыми операторами и значения объявленных переменных при таком переходе будут не определены.

1.5.1. Определение и вызов функций

Мощность языка СИ во многом определяется легкостью и гибкостью в определении и использовании функций в СИ-программах. В отличие от других языков программирования высокого уровня в языке СИ нет деления на процедуры, подпрограммы и функции, здесь вся программа строится только из функций.

Функция - это совокупность объявлений и операторов, обычно предназначенная для решения определенной задачи. Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова. В любой программе на СИ должна быть функция с именем main (главная функция), именно с этой функции, в каком бы месте программы она не находилась, начинается выполнение программы.

При вызове функции ей при помощи аргументов (формальных параметров) могут быть переданы некоторые значения (фактические параметры), используемые во время выполнения функции. Функция может возвращать некоторое (одно !) значение. Это возвращаемое значение и есть результат выполнения функции, который при выполнении программы подставляется в точку вызова функции, где бы этот вызов ни встретился. Допускается также использовать функции не имеющие аргументов и функции не возвращающие никаких значений. Действие таких функций может состоять, например, в изменении значений некоторых переменных, выводе на печать некоторых текстов и т.п..

С использованием функций в языке СИ связаны три понятия - определение функции (описание действий, выполняемых функцией), объявление функции (задание формы обращения к функции) и вызов функции.

Определение функции задает тип возвращаемого значения, имя функции, типы и число формальных параметров, а также объявления переменных и операторы, называемые телом функции, и определяющие действие функции. В определении функции также может быть задан класс памяти.

Пример:

```
int rus (unsigned char r)
{  if (r>='A' && c<=' ')
      return 1;
   else
      return 0;
}
```

В данном примере определена функция с именем rus, имеющая один параметр с именем r и типом unsigned char. Функция возвращает целое значение, равное 1, если параметр функции является буквой русского алфавита, или 0 в противном случае.

В языке СИ нет требования, чтобы определение функции обязательно предшествовало ее вызову. Определения используемых функций могут следовать за определением функции main, перед ним, или находится в другом файле.

Однако для того, чтобы компилятор мог осуществить проверку соответствия типов передаваемых фактических параметров типам формальных параметров до вызова функции нужно поместить объявление (прототип) функции.

Объявление функции имеет такой же вид, что и определение функции, с той лишь разницей, что тело функции отсутствует, и имена формальных параметров тоже могут быть опущены. Для функции, определенной в последнем примере, прототип может иметь вид

int rus (unsigned char r); или rus (unsigned char);

В программах на языке СИ широко используются, так называемые, библиотечные функции, т.е. функции предварительно разработанные и записанные в библиотеки. Прототипы библиотечных функций находятся в специальных заголовочных файлах, поставляемых вместе с библиотеками в составе систем программирования, и включаются в программу с помощью директивы #include.

Если объявление функции не задано, то по умолчанию строится прототип функции на основе анализа первой ссылки на функцию, будь то вызов функции или определение. Однако такой прототип не всегда согласуется с последующим определением или вызовом функции. Рекомендуется всегда задавать прототип функции. Это позволит компилятору либо выдавать диагностические сообщения, при неправильном использовании функции, либо корректным образом регулировать несоответствие аргументов устанавливаемое при выполнении программы.

Объявление параметров функции при ее определении может быть выполнено в так называемом "старом стиле", при котором в скобках после имени функции следуют только имена параметров, а после скобок объявления типов параметров. Например, функция rus из предыдущего примера может быть определена следующим образом:

```
int rus (r)
unsigned char r;
{ ...     /*  тело функции   */      ...  }
```

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. Подробно классы памяти будут рассмотрены в следующем разделе.

Спецификатор-типа функции задает тип возвращаемого значения и может задавать любой тип. Если спецификатор-типа не задан, то предполагается, что функция возвращает значение типа int.

Функция не может возвращать массив или функцию, но может возвращать указатель на любой тип, в том числе и на массив и на функцию. Тип возвращаемого значения, задаваемый в определении функции, должен соответствовать типу в объявлении этой функции.

Функция возвращает значение если ее выполнение заканчивается оператором return, содержащим некоторое выражение. Указанное выражение вычисляется, преобразуется, если необходимо, к типу возвращаемого значения и возвращается в точку вызова функции в качестве результата. Если оператор return не содержит выражения или выполнение функции завершается после выполнения последнего ее оператора (без выполнения оператора return), то возвращаемое значение не определено. Для функций, не использующих возвращаемое значение, должен быть использован тип void, указывающий на отсутствие возвращаемого значения. Если функция определена как функция, возвращающая некоторое значение, а в операторе return при выходе из нее отсутствует выражение, то поведение вызывающей функции после передачи ей управления может быть непредсказуемым.

Список-формальных-параметров - это последовательность объявлений формальных параметров, разделенная запятыми. Формальные параметры - это переменные, используемые внутри тела функции и получающие значение при вызове функции путем копирования в них значений соответствующих фактических параметров. Список-формальных-параметров может заканчиваться запятой (,) или запятой с многоточием (,...),

это означает, что число аргументов функции переменно. Однако предполагается, что функция имеет, по крайней мере, столько обязательных аргументов, сколько формальных параметров задано перед последней запятой в списке параметров. Такой функции может быть передано большее число аргументов, но над дополнительными аргументами не проводится контроль типов.

Если функция не использует параметров, то наличие круглых скобок обязательно, а вместо списка параметров рекомендуется указать слово void.

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. Тип формального параметра может быть любым основным типом, структурой, объединением, перечислением, указателем или массивом. Если тип формального параметра не указан, то этому параметру присваивается тип int.

Для формального параметра можно задавать класс памяти register, при этом для величин типа int спецификатор типа можно опустить.

Идентификаторы формальных параметров используются в теле функции в качестве ссылок на переданные значения. Эти идентификаторы не могут быть переопределены в блоке, образующем тело функции, но могут быть переопределены во внутреннем блоке внутри тела функции.

При передаче параметров в функцию, если необходимо, выполняются обычные арифметические преобразования для каждого формального параметра и каждого фактического параметра независимо. После преобразования формальный параметр не может быть короче чем int, т.е. объявление формального параметра с типом char равносильно его объявлению с типом int. А параметры, представляющие собой действительные числа, имеют тип double.

Преобразованный тип каждого формального параметра определяет, как интерпретируются аргументы, помещаемые при вызове функции в стек. Несоответствие типов фактических аргументов и формальных параметров может быть причиной неверной интерпретации.

Тело функции - это составной оператор, содержащий операторы, определяющие действие функции.

Все переменные, объявленные в теле функции без указания класса памяти, имеют класс памяти auto, т.е. они являются локальными. При вызове функции локальным переменным отводится память в стеке и производится их инициализация. Управление передается первому оператору тела функции и начинается выполнение функции, которое продолжается до тех пор, пока не встретится оператор return или последний оператор тела функции. Управление при этом возвращается в точку, следующую за точкой вызова, а локальные переменные становятся недоступными. При новом вызове функции для локальных переменных память распределяется вновь, и поэтому старые значения локальных переменных теряются.

Параметры функции передаются по значению и могут рассматриваться как локальные переменные, для которых выделяется память при вызове функции и производится инициализация значениями фактических параметров. При выходе из функции значения этих переменных теряются. Поскольку передача параметров происходит по значению, в

теле функции нельзя изменить значения переменных в вызывающей функции, являющихся фактическими параметрами. Однако, если в качестве параметра передать указатель на некоторую переменную, то используя операцию разадресации можно изменить значение этой переменной.

Пример:

```
/*    Неправильное использование параметров   */
void change (int x, int y)
{    int k=x;
         x=y;
         y=k;
}
```

В данной функции значения переменных x и y, являющихся формальными параметрами, меняются местами, но поскольку эти переменные существуют только внутри функции change, значения фактических параметров, используемых при вызове функции, останутся неизменными. Для того чтобы менялись местами значения фактических аргументов можно использовать функцию приведенную в следующем примере.

Пример:

```
/*     Правильное использование параметров   */
 void change (int *x, int *y)
 {  int k=*x;
       *x=*y;
       *y=k;
 }
```

При вызове такой функции в качестве фактических параметров должны быть использованы не значения переменных, а их адреса

change (&a,&b);

Если требуется вызвать функцию до ее определения в рассматриваемом файле, или определение функции находится в другом исходном файле, то вызов функции следует предварять объявлением этой функции. Объявление (прототип) функции имеет следующий формат:

[спецификатор-класса-памяти] [спецификатор-типа] имя-функции ([список-формальных-параметров]) [,список-имен-функций];

В отличие от определения функции, в прототипе за заголовком сразу же следует точка с запятой, а тело функции отсутствует. Если несколько разных функций возвращают значения одинакового типа и имеют одинаковые списки формальных параметров, то эти функции можно объявить в одном прототипе, указав имя одной из функций в качестве имени-функции, а все другие поместить в список-имен-функций, причем каждая функция должна сопровождаться списком формальных параметров. Правила использования остальных элементов формата такие же, как при определении функции. Имена формальных параметров при объявлении функции можно не указывать, а если они указаны, то их область действия распространяется только до конца объявления.

Прототип - это явное объявление функции, которое предшествует определению функции. Тип возвращаемого значения при объявлении функции должен соответствовать типу возвращаемого значения в определении функции.

Если прототип функции не задан, а встретился вызов функции, то строится неявный прототип из анализа формы вызова функции. Тип возвращаемого значения создаваемого прототипа int, а список типов и числа параметров функции формируется на основании типов и числа фактических параметров используемых при данном вызове.

Таким образом, прототип функции необходимо задавать в следующих случаях:

1. Функция возвращает значение типа, отличного от int.

2. Требуется проинициализировать некоторый указатель на функцию до того, как эта функция будет определена.

Наличие в прототипе полного списка типов аргументов параметров позволяет выполнить проверку соответствия типов фактических параметров при вызове функции типам формальных параметров, и, если необходимо, выполнить соответствующие преобразования.

В прототипе можно указать, что число параметров функции переменно, или что функция не имеет параметров.

Если прототип задан с классом памяти static, то и определение функции должно иметь класс памяти static. Если спецификатор класса памяти не указан, то подразумевается класс памяти extern.

Вызов функции имеет следующий формат:

адресное-выражение ([список-выражений])

Поскольку синтаксически имя функции является адресом начала тела функции, в качестве обращения к функции может быть использовано адресное-выражение (в том числе и имя функции или разадресация указателя на функцию), имеющее значение адреса функции.

Список-выражений представляет собой список фактических параметров, передаваемых в функцию. Этот список может быть и пустым, но наличие круглых скобок обязательно.

Фактический параметр может быть величиной любого основного типа, структурой, объединением, перечислением или указателем на объект любого типа. Массив и функция не могут быть использованы в качестве фактических параметров, но можно использовать указатели на эти объекты.

Выполнение вызова функции происходит следующим образом:

1. Вычисляются выражения в списке выражений и подвергаются обычным арифметическим преобразованиям. Затем, если известен прототип функции, тип полученного фактического аргумента сравнивается с типом соответствующего формального параметра. Если они не совпадают, то либо производится преобразование типов, либо формируется сообщение об ошибке. Число выражений в списке выражений должно совпадать с числом формальных параметров, если только функция не имеет

переменного числа параметров. В последнем случае проверке подлежат только обязательные параметры. Если в прототипе функции указано, что ей не требуются параметры, а при вызове они указаны, формируется сообщение об ошибке.

2. Происходит присваивание значений фактических параметров соответствующим формальным параметрам.

3. Управление передается на первый оператор функции.

4. Выполнение оператора return в теле функции возвращает управление и возможно, значение в вызывающую функцию. При отсутствии оператора return управление возвращается после выполнения последнего оператора тела функции, а возвращаемое значение не определено.

Адресное выражение, стоящее перед скобками определяет адрес вызываемой функции. Это значит что функция может быть вызвана через указатель на функцию.

Пример:

int (*fun)(int x, int *y);

Здесь объявлена переменная fun как указатель 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 (intx,int *y);

будет интерпретироваться как объявление функции fun возвращающей указатель на int.

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 * .

Указатель на функцию может быть передан в качестве параметра функции. При этом разадресация происходит во время вызова функции, на которую ссылается указатель на функцию. Присвоить значение указателю на функцию можно в операторе присваивания, употребив имя функции без списка параметров.

Пример:

```
double (*fun1)(int x, int y);
double fun2(int k, int l);
   fun1=fun2;          /* инициализация указателя на функцию */
   (*fun1)(2,7);       /* обращение к функции                */
```

В рассмотренном примере указатель на функцию fun1 описан как указатель на функцию с двумя параметрами, возвращающую значение типа double, и также описана функция fun2. В противном случае, т.е. когда указателю на функцию присваивается функция описанная иначе чем указатель, произойдет ошибка.

Рассмотрим пример использования указателя на функцию в качестве параметра функции вычисляющей производную от функции cos(x).

Пример:

```
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);

Любая функция в программе на языке СИ может быть вызвана рекурсивно, т.е. она может вызывать саму себя. Компилятор допускает любое число рекурсивных вызовов. 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.

Классический пример рекурсии - это математическое определение факториала n! :

```
n! = 1 при n=0;
     n*(n-1)!  при    n>1 .
```

Функция, вычисляющая факториал, будет иметь следующий вид:

```
long fakt(int n)
{
  return ( (n==1) ? 1 : n*fakt(n-1)  );
}
```

Хотя компилятор языка СИ не ограничивает число рекурсивных вызовов функций, это число ограничивается ресурсом памяти компьютера и при слишком большом числе рекурсивных вызовов может произойти переполнение стека.

1.5.2. Вызов функции с переменным числом параметров

При вызове функции с переменным числом параметров в вызове этой функции задается любое требуемое число аргументов. В объявлении и определении такой функции переменное число аргументов задается многоточием в конце списка формальных параметров или списка типов аргументов.

Все аргументы, заданные в вызове функции, размещаются в стеке. Количество формальных параметров, объявленных для функции, определяется числом аргументов, которые берутся из стека и присваиваются формальным параметрам. Программист отвечает за правильность выбора дополнительных аргументов из стека и определение числа аргументов, находящихся в стеке.

Примерами функций с переменным числом параметров являются функции из библиотеки функций языка СИ, осуществляющие операции ввода-вывода информации (printf,scanf и т.п.). Подробно эти функции рассмотрены во третьей части книги.

Программист может разрабатывать свои функции с переменным числом параметров. Для обеспечения удобного способа доступа к аргументам функции с переменным числом параметров имеются три макроопределения (макросы) va_start, va_arg, va_end, находящиеся в заголовочном файле stdarg.h. Эти макросы указывают на то, что функция, разработанная пользователем, имеет некоторое число обязательных аргументов, за которыми следует переменное число необязательных аргументов. Обязательные аргументы доступны через свои имена как при вызове обычной функции. Для извлечения необязательных аргументов используются макросы va_start, va_arg, va_end в следующем порядке.

Макрос va_start предназначен для установки аргумента arg_ptr на начало списка необязательных параметров и имеет вид функции с двумя параметрами:

void va_start(arg_ptr,prav_param);

Параметр prav_param должен быть последним обязательным параметром вызываемой функции, а указатель arg_prt должен быть объявлен с предопределением в списке переменных типа va_list в виде:

va_list arg_ptr;

Макрос va_start должен быть использован до первого использования макроса va_arg.

Макрокоманда va_arg обеспечивает доступ к текущему параметру вызываемой функции и тоже имеет вид функции с двумя параметрами

type_arg va_arg(arg_ptr,type);

Эта макрокоманда извлекает значение типа type по адресу, заданному указателем arg_ptr, увеличивает значение указателя arg_ptr на длину использованного параметра (длина type) и таким образом параметр arg_ptr будет указывать на следующий параметр вызываемой функции. Макрокоманда va_arg используется столько раз, сколько необходимо для извлечения всех параметров вызываемой функции.

Макрос va_end используется по окончании обработки всех параметров функции и устанавливает указатель списка необязательных параметров на ноль (NULL).

Рассмотрим применение этих макросов для обработки параметров функции вычисляющей среднее значение произвольной последовательности целых чисел. Поскольку функция имеет переменное число параметров будем считать концом списка значение равное -1. Поскольку в списке должен быть хотя бы один элемент, у функции будет один обязательный параметр.

Пример:

```
#include
int main()
{ int n;
  int sred_znach(int,...);
  n=sred_znach(2,3,4,-1);
                 /* вызов с четырьмя параметрами */
  printf("n=%d",n);
  n=sred_znach(5,6,7,8,9,-1);
                /* вызов с шестью параметрами   */
  printf("n=%d",n);
  return (0);
}

int sred_znach(int x,...);
{
   int i=0,  j=0,  sum=0;
   va_list uk_arg;
   va_start(uk_arg,x);  /*  установка указателя uk_arg на  */
                        /*  первый необязятельный параметр */
   if (x!=-1) sum=x;    /*  проверка на пустоту списка     */
   else return (0);
   j++;
   while ( (i=va_arg(uk_arg,int))!=-1)
                               /* выборка очередного   */
   {                           /* параметра и проверка */
     sum+=i;                   /* на конец списка      */
     j++;
   }
   va_end(uk_arg);         /* закрытие списка параметров   */
   return (sum/j);
}
```

1.5.3. Передача параметров функции main

Функция main, с которой начинается выполнение СИ-программы, может быть определена с параметрами, которые передаются из внешнего окружения, например, из командной строки. Во внешнем окружении действуют свои правила представления данных, а точнее, все данные представляются в виде строк символов. Для передачи этих строк в функцию main используются два параметра, первый параметр служит для передачи числа передаваемых строк, второй для передачи самих строк. Общепринятые (но не обязательные) имена этих параметров argc и argv. Параметр argc имеет тип int, его

значение формируется из анализа командной строки и равно количеству слов в командной строке, включая и имя вызываемой программы (под словом понимается любой текст не содержащий символа пробел). Параметр argv это массив указателей на строки, каждая из которых содержит одно слово из командной строки. Если слово должно содержать символ пробел, то при записи его в командную строку оно должно быть заключено в кавычки.

Функция main может иметь и третий параметр, который принято называть argp, и который служит для передачи в функцию main параметров операционной системы (среды) в которой выполняется СИ-программа.

Заголовок функции main имеет вид:

int main (int argc, char *argv[], char *argp[])

Если, например, командная строка СИ-программы имеет вид:

A:\>cprog working 'C program' 1

то аргументы argc, argv, argp представляются в памяти как показано в схеме на рис.1.

```
argc  [ 4  ]
argv  [    ]--> [    ]-->   [A:\cprog.exe\0]
                [    ]-->   [working\0]
                [    ]-->   [C program\0]
                [    ]-->   [1\0]
                [NULL]
argp  [    ]--> [    ]-->   [path=A:\;C:\\0]
                [    ]-->   [lib=D:\LIB\0]
                [    ]-->   [include=D:\INCLUDE\0]
                [    ]-->   [conspec=C:\COMMAND.COM\]
                [NULL]
```

Рис.1. Схема размещения параметров командной строки

Операционная система поддерживает передачу значений для параметров argc, argv, argp, а на пользователе лежит ответственность за передачу и использование фактических аргументов функции main.

Следующий пример представляет программу печати фактических аргументов, передаваемых в функцию main из операционной системы и параметров операционной системы.

```
Пример:
int main ( int argc, char *argv[], char *argp[])
{  int i=0;
   printf ("\n Имя программы %s", argv[0]);
   for    (i=1; i>=argc; i++)
   printf ("\n аргумент %d равен %s", argv[i]);
   printf ("\n   Параметры операционной системы:");
   while  (*argp)
     {  printf ("\n %s",*argp);
        argp++;
     }
   return (0);
}
```

Доступ к параметрам операционной системы можно также получить при помощи библиотечной функции geteuv, ее прототип имеет следующий вид:

char *geteuv (const char *varname);

Аргумент этой функции задает имя параметра среды, указатель на значение которой выдаст функция geteuv. Если указанный параметр не определен в среде в данный момент, то возвращаемое значение NULL.

Используя указатель, полученный функцией geteuv, можно только прочитать значение параметра операционной системы, но нельзя его изменить. Для изменения значения параметра системы предназначена функция puteuv.

Компилятор языка СИ строит СИ-программу таким образом, что вначале работы программы выполняется некоторая инициализация, включающая, кроме всего прочего, обработку аргументов, передаваемых функции main, и передачу ей значений параметров среды. Эти действия выполняются библиотечными функциями _setargv и _seteuv, которые всегда помещаются компилятором перед функцией main.

Если СИ-программа не использует передачу аргументов и значений параметров операционной системы, то целесообразно запретить использование библиотечных функций _setargv и _seteuv поместив в СИ-программу перед функцией main функции с такими же именами, но не выполняющие никаких действий (заглушки). Начало программы в этом случае будет иметь вид:

```
_setargv()
 {  return ;     /* пустая функция */
 }
-seteuv()
 {  return ;     /* пустая функция */
 }
int main()
 {      /* главная функция без аргументов */
 ...
 ...
 renurn (0);
 }
```

В приведенной программе при вызове библиотечных функций _setargv и _seteuv будут использованы функции помещенные в программу пользователем и не выполняющие никаких действий. Это заметно снизит размер получаемого exe-файла.

1.6.1. Исходные файлы и объявление переменных

Обычная СИ-программа представляет собой определение функции main, которая для выполнения необходимых действий вызывает другие функции. Приведенные выше примеры программ представляли собой один исходный файл, содержащий все необходимые для выполнения программы функции. Связь между функциями осуществлялась по данным посредством передачи параметров и возврата значений функций. Но компилятор языка СИ позволяет также разбить программу на несколько отдельных частей (исходных файлов), оттранслировать каждую часть отдельно, и затем объединить все части в один выполняемый файл при помощи редактора связей.

При такой структуре исходной программы функции, находящиеся в разных исходных файлах могут использовать глобальные внешние переменные. Все функции в языке Си по определению внешние и всегда доступны из любых файлов. Например, если программа состоит из двух исходных файлов, как показано на рис.2., то функция main может вызывать любую из трех функций fun1, fun2, fun3, а каждая из этих функций может вызывать любую другую.

```
main ()
{   ...
}
fun1()
{   ...
}
```

file1.c

```
fun2()
{   ...
}
fun3()
{   ...
}
```

file2.c

Рис.2. Пример программы из двух файлов

Для того, чтобы определяемая функция могла выполнять какие либо действия, она должна использовать переменные. В языке СИ все переменные должны быть объявлены до их использования. Объявления устанавливают соответствие имени и атрибутов переменной, функции или типа. Определение переменной вызывает выделение памяти для хранения ее значения. Класс выделяемой памяти определяется спецификатором класса памяти, и определяет время жизни и область видимости переменной, связанные с понятием блока программы.

В языке СИ блоком считается последовательность объявлений, определений и операторов, заключенная в фигурные скобки. Существуют два вида блоков - составной оператор и определение функции, состоящее из составного оператора, являющегося телом функции, и предшествующего телу заголовка функции (в который входят имя функции, типы возвращаемого значения и формальных параметров). Блоки могут включать в себя составные операторы, но не определения функций. Внутренний блок называется вложенным, а внешний блок - объемлющим.

Время жизни - это интервал времени выполнения программы, в течение которого программный объект (переменная или функция) существует. Время жизни переменной может быть локальным или глобальным. Переменная с глобальным временем жизни имеет распределенную для нее память и определенное значение на протяжении всего времени выполнения программы, начиная с момента выполнения объявления этой переменной. Переменная с локальным временем жизни имеет распределенную для него память и определенное значение только во время выполнения блока, в котором эта переменная определена или объявлена. При каждом входе в блок для локальной переменной распределяется новая память, которая освобождается при выходе из блока.

Все функции в СИ имеют глобальное время жизни и существуют в течение всего времени выполнения программы.

Область видимости - это часть текста программы, в которой может быть использован данный объект. Объект считается видимым в блоке или в исходном файле, если в этом блоке или файле известны имя и тип объекта. Объект может быть видимым в пределах блока, исходного файла или во всех исходных файлах, образующих программу. Это зависит от того, на каком уровне объявлен объект: на внутреннем, т.е. внутри некоторого блока, или на внешнем, т.е. вне всех блоков.

Если объект объявлен внутри блока, то он видим в этом блоке, и во всех внутренних блоках. Если объект объявлен на внешнем уровне, то он видим от точки его объявления до конца данного исходного файла.

Объект может быть сделан глобально видимым с помощью соответствующих объявлений во всех исходных файлах, образующих программу.

Спецификатор класса памяти в объявлении переменной может быть auto, register, static или extern. Если класс памяти не указан, то он определяется по умолчанию из контекста объявления.

Объекты классов auto и register имеют локальное время жизни. Спецификаторы static и extern определяют объекты с глобальным временем жизни.

При объявлении переменной на внутреннем уровне может быть использован любой из четырех спецификаторов класса памяти, а если он не указан, то подразумевается класс памяти auto.

Переменная с классом памяти auto имеет локальное время жизни и видна только в блоке, в котором объявлена. Память для такой переменной выделяется при входе в блок и освобождается при выходе из блока. При повторном входе в блок этой переменной может быть выделен другой участок памяти.

Переменная с классом памяти auto автоматически не инициализируется. Она должна быть проинициализирована явно при объявлении путем присвоения ей начального значения. Значение неинициализированной переменной с классом памяти auto считается неопределенным.

Спецификатор класса памяти register предписывает компилятору распределить память для переменной в регистре, если это представляется возможным. Использование регистровой памяти обычно приводит к сокращению времени доступа к переменной. Переменная, объявленная с классом памяти register, имеет ту же область видимости, что и переменная auto. Число регистров, которые можно использовать для значений переменных, ограничено возможностями компьютера, и в том случае, если компилятор не имеет в распоряжении свободных регистров, то переменной выделяется память как для класса auto. Класс памяти register может быть указан только для переменных с типом int или указателей с размером, равным размеру int.

Переменные, объявленные на внутреннем уровне со спецификатором класса памяти static, обеспечиваю возможность сохранить значение переменной при выходе из блока и использовать его при повторном входе в блок. Такая переменная имеет глобальное время жизни и область видимости внутри блока, в котором она объявлена. В отличие от переменных с классом auto, память для которых выделяется в стеке, для переменных с классом static память выделяется в сегменте данных, и поэтому их значение сохраняется при выходе из блока.

```
   Пример:
/*  объявления переменной i на внутреннем уровне
          с  классом памяти static.                    */
/*   исходный файл file1.c       */
   main()
      { ...
      }
   fun1()
```

```
      {  static int i=0; ...
      }
 /* исходный файл   file2.c       */
   fun2()
      {  static int i=0; ...
      }
   fun3()
      {  static int i=0; ...
      }
```

В приведенном примере объявлены три разные переменные с классом памяти static, имеющие одинаковые имена i. Каждая из этих переменных имеет глобальное время жизни, но видима только в том блоке (функции), в которой она объявлена. Эти переменные можно использовать для подсчета числа обращений к каждой из трех функций.

Переменные класса памяти static могут быть инициализированы константным выражением. Если явной инициализации нет, то такой переменной присваивается нулевое значение. При инициализации константным адресным выражением можно использовать адреса любых внешних объектов, кроме адресов объектов с классом памяти auto, так как адрес последних не является константой и изменяется при каждом входе в блок. Инициализация выполняется один раз при первом входе в блок.

Переменная, объявленная локально с классом памяти extern, является ссылкой на переменную с тем же самым именем, определенную глобально в одном из исходных файлов программы. Цель такого объявления состоит в том, чтобы сделать определение переменной глобального уровня видимым внутри блока.

```
   Пример:
/*   объявления  переменной  i,  являющейся  именем внешнего
     массива длинных целых чисел, на локальном уровне      */
/*      исходный файл    file1.c          */
   main()
      { ...
      }
   fun1()
      { extern long i[]; ...
      }
/*     исходный файл      file2.c         */
   long i[MAX]={0};
   fun2()
      {  ...
      }
   fun3()
      {  ...
      }
```

Объявление переменной i[] как extern в приведенном примере делает ее видимой внутри функции fun1. Определение этой переменной находится в файле file2.c на глобальном уровне и должно быть только одно, в то время как объявлений с классом памяти extern может быть несколько.

Объявление с классом памяти extern требуется при необходимости использовать переменную, описанную в текущем исходном файле, но ниже по тексту программы, т.е. до выполнения ее глобального определения. Следующий пример иллюстрирует такое использование переменной с именем st.

```
Пример:
main()
{ extern int st[]; ...
}
static int st[MAX]={0};
fun1()
{   ...
}
```

Объявление переменной со спецификатором extern информирует компилятор о том, что память для переменной выделять не требуется, так как это выполнено где-то в другом месте программы.

При объявлении переменных на глобальном уровне может быть использован спецификатор класса памяти static или extern, а так же можно объявлять переменные без указания класса памяти. Классы памяти auto и register для глобального объявления недопустимы.

Объявление переменных на глобальном уровне - это или определение переменных, или ссылки на определения, сделанные в другом месте программы. Объявление глобальной переменной, которое инициализирует эту переменную (явно или неявно), является определением переменной. Определение на глобальном уровне может задаваться в следующих формах:

1. Переменная объявлена с классом памяти static. Такая переменная может быть инициализирована явно константным выражением, или по умолчанию нулевым значением. То есть обявления static int i=0 и static int i эквивалентны, и в обоих случаях переменной i будет присвоено значение 0.

2. Переменная объявлена без указания класса памяти, но с явной инициализацией. Такой переменной по умолчанию присваивается класс памяти static. То есть объявления int i=1 и static int i=1 будут эквивалентны.

Переменная объявленная глобально видима в пределах остатка исходного файла, в котором она определена. Выше своего описания и в других исходных файлах эта переменная невидима (если только она не объявлена с классом extern).

Глобальная переменная может быть определена только один раз в пределах своей области видимости. В другом исходном файле может быть объявлена другая глобальная переменная с таким же именем и с классом памяти static, конфликта при этом не возникает, так как каждая из этих переменных будет видимой только в своем исходном файле.

Спецификатор класса памяти extern для глобальных переменных используется, как и для локального объявления, в качестве ссылки на переменную, объявленную в другом месте программы, т.е. для расширения области видимости переменной. При таком объявлении область видимости переменной расширяется до конца исходного файла, в котором сделано объявление.

В объявлениях с классом памяти extern не допускается инициализация, так как эти объявления ссылаются на уже существующие и определенные ранее переменные.

Переменная, на которую делается ссылка с помощью спецификатора extern, может быть определена только один раз в одном из исходных файлов программы.

1.6.2. Объявления функций

Функции всегда определяются глобально. Они могут быть объявлены с классом памяти static или extern. Объявления функций на локальном и глобальном уровнях имеют одинаковый смысл.

Правила определения области видимости для функций отличаются от правил видимости для переменных и состоят в следующем.

1. Функция, объявленная как static, видима в пределах того файла, в котором она определена. Каждая функция может вызвать другую функцию с классом памяти static из своего исходного файла, но не может вызвать функцию определенную с классом static в другом исходном файле. Разные функции с классом памяти static имеющие одинаковые имена могут быть определены в разных исходных файлах, и это не ведет к конфликту.

2. Функция, объявленная с классом памяти extern, видима в пределах всех исходных файлов программы. Любая функция может вызывать функции с классом памяти extern.

3. Если в объявлении функции отсутствует спецификатор класса памяти, то по умолчанию принимается класс extern.

Все объекты с классом памяти extern компилятор помещает в объектном файле в специальную таблицу внешних ссылок, которая используется редактором связей для разрешения внешних ссылок. Часть внешних ссылок порождается компилятором при обращениях к библиотечным функциям СИ, поэтому для разрешения этих ссылок редактору связей должны быть доступны соответствующие библиотеки функций.

1.6.3. Время жизни и область видимости программных объектов

Время жизни переменной (глобальной или локальной) определяется по следующим правилам.

1. Переменная, объявленная глобально (т.е. вне всех блоков), существует на протяжении всего времени выполнения программы.

2. Локальные переменные (т.е. объявленные внутри блока) с классом памяти register или auto, имеют время жизни только на период выполнения того блока, в котором они объявлены. Если локальная переменная объявлена с классом памяти static или extern, то она имеет время жизни на период выполнения всей программы.

Видимость переменных и функций в программе определяется следующими правилами.

1. Переменная, объявленная или определенная глобально, видима от точки объявления или определения до конца исходного файла. Можно сделать переменную видимой и в других исходных файлах, для чего в этих файлах следует ее объявить с классом памяти extern.

2. Переменная, объявленная или определенная локально, видима от точки объявления или определения до конца текущего блока. Такая переменная называется локальной.

3. Переменные из объемлющих блоков, включая переменные объявленные на глобальном уровне, видимы во внутренних блоках. Эту видимость называют вложенной. Если

переменная, объявленная внутри блока, имеет то же имя, что и переменная, объявленная в объемлющем блоке, то это разные переменные, и переменная из объемлющего блока во внутреннем блоке будет невидимой.

4. Функции с классом памяти static видимы только в исходном файле, в котором они определены. Всякие другие функции видимы во всей программе.

Метки в функциях видимы на протяжении всей функции.

Имена формальных параметров, объявленные в списке параметров прототипа функции, видимы только от точки объявления параметра до конца объявления функции.

1.6.4. Инициализация глобальных и локальных переменных

При инициализации необходимо придерживаться следующих правил:

1. Объявления содержащие спецификатор класса памяти extern не могут содержать инициаторов.

2. Глобальные переменные всегда инициализируются, и если это не сделано явно, то они инициализируются нулевым значением.

3. Переменная с классом памяти static может быть инициализирована константным выражением. Инициализация для них выполняется один раз перед началом программы. Если явная инициализация отсутствует, то переменная инициализируется нулевым значением.

4. Инициализация переменных с классом памяти auto или register выполняется всякий раз при входе в блок, в котором они объявлены. Если инициализация переменных в объявлении отсутствует, то их начальное значение не определено.

5. Начальными значениями для глобальных переменных и для переменных с классом памяти static должны быть константные выражения. Адреса таких переменных являются константами и эти константы можно использовать для инициализации объявленных глобально указателей. Адреса переменных с классом памяти auto или register не являются константами и их нельзя использовать в инициаторах.

```
 Пример:
int global_var;
int func(void)
{ int local_var;                      /* по умолчанию auto  */
  static int *local_ptr=&local_var;   /*  так неправильно   */
  static int *global_ptr=&global_var; /*  а так правильно   */
  register int *reg_ptr=&local_var;   /*  и так правильно   */
}
```

В приведенном примере глобальная переменная global_var имеет глобальное время жизни и постоянный адрес в памяти, и этот адрес можно использовать для инициализации статического указателя global_ptr. Локальная переменная local_var, имеющая класс памяти auto размещается в памяти только на время работы функции func, адрес этой переменной не является константой и не может быть использован для инициализации статической переменной local_ptr. Для инициализации локальной регистровой переменной reg_ptr можно использовать неконстантные выражения, и, в частности, адрес переменной local_ptr.

1.7.1. Методы доступа к элементам массивов

В языке СИ между указателями и массивами существует тесная связь. Например, когда объявляется массив в виде int array[25], то этим определяется не только выделение памяти для двадцати пяти элементов массива, но и для указателя с именем array, значение которого равно адресу первого по счету (нулевого) элемента массива, т.е. сам массив остается безымянным, а доступ к элементам массива осуществляется через указатель с именем array. С точки зрения синтаксиса языка указатель arrey является константой, значение которой можно использовать в выражениях, но изменить это значение нельзя.

Поскольку имя массива является указателем допустимо, например, такое присваивание:

```
int arrey[25];
int *ptr;
ptr=array;
```

Здесь указатель ptr устанавливается на адрес первого элемента массива, причем присваивание ptr=arrey можно записать в эквивалентной форме ptr=&arrey[0].

Для доступа к элементам массива существует два различных способа. Первый способ связан с использованием обычных индексных выражений в квадратных скобках, например, array[16]=3 или array[i+2]=7. При таком способе доступа записываются два выражения, причем второе выражение заключается в квадратные скобки. Одно из этих выражений должно быть указателем, а второе - выражением целого типа. Последовательность записи этих выражений может быть любой, но в квадратных скобках записывается выражение следующее вторым. Поэтому записи array[16] и 16[array] будут эквивалентными и обозначают элемент массива с номером шестнадцать. Указатель используемый в индексном выражении не обязательно должен быть константой, указывающей на какой-либо массив, это может быть и переменная. В частности после выполнения присваивания ptr=array доступ к шестнадцатому элементу массива можно получить с помощью указателя ptr в форме ptr[16] или 16[ptr].

Второй способ доступа к элементам массива связан с использованием адресных выражений и операции разадресации в форме *(array+16)=3 или *(array+i+2)=7. При таком способе доступа адресное выражение равное адресу шестнадцатого элемента массива тоже может быть записано разными способами *(array+16) или *(16+array).

При реализации на компьютере первый способ приводится ко второму, т.е. индексное выражение преобразуется к адресному. Для приведенных примеров array[16] и 16[array] преобразуются в *(array+16).

Для доступа к начальному элементу массива (т.е. к элементу с нулевым индексом) можно использовать просто значение указателя array или ptr. Любое из присваиваний

```
*array = 2;
array[0] = 2;
*(array+0) = 2;
*ptr = 2;
ptr[0] = 2;
*(ptr+0) = 2;
```

присваивает начальному элементу массива значение 2, но быстрее всего выполнятся присваивания *array=2 и *ptr=2, так как в них не требуется выполнять операции сложения.

1.7.2. Указатели на многомерные массивы

Указатели на многомерные массивы в языке СИ - это массивы массивов, т.е. такие массивы, элементами которых являются массивы. При объявлении таких массивов в памяти компьютера создается несколько различных объектов. Например при выполнении объявления двумерного массива int arr2[4][3] в памяти выделяется участок для хранения значения переменной arr, которая является указателем на массив из четырех указателей. Для этого массива из четырех указателей тоже выделяется память. Каждый из этих четырех указателей содержит адрес массива из трех элементов типа int, и, следовательно, в памяти компьютера выделяется четыре участка для хранения четырех массивов чисел типа int, каждый из которых состоит из трех элементов. Такое выделение памяти показано на схеме на рис.3.

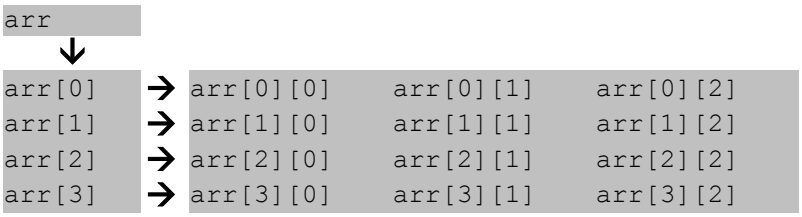

Рис.3. Распределение памяти для двумерного массива.

Таким образом, объявление arr2[4][3] порождает в программе три разных объекта: указатель с идентификатором arr, безымянный массив из четырех указателей и безымянный массив из двенадцати чисел типа int. Для доступа к безымянным массивам используются адресные выражения с указателем arr. Доступ к элементам массива указателей осуществляется с указанием одного индексного выражения в форме arr2[2] или *(arr2+2). Для доступа к элементам двумерного массива чисел типа int должны быть использованы два индексных выражения в форме arr2[1][2] или эквивалентных ей *(*(arr2+1)+2) и (*(arr2+1))[2]. Следует учитывать, что с точки зрения синтаксиса языка СИ указатель arr и указатели arr[0], arr[1], arr[2], arr[3] являются константами и их значения нельзя изменять во время выполнения программы.

Размещение трехмерного массива происходит аналогично и объявление float arr3[3][4][5] порождает в программе кроме самого трехмерного массива из шестидесяти чисел типа float массив из четырех указателей на тип float, массив из трех указателей на массив указателей на float, и указатель на массив массивов указателей на float.

При размещении элементов многомерных массивов они располагаются в памяти подряд по строкам, т.е. быстрее всего изменяется последний индекс, а медленнее - первый. Такой порядок дает возможность обращаться к любому элементу многомерного массива, используя адрес его начального элемента и только одно индексное выражение.

Например, обращение к элементу arr2[1][2] можно осуществить с помощью указателя ptr2, объявленного в форме int *ptr2=arr2[0] как обращение ptr2[1*4+2] (здесь 1 и 2 это индексы используемого элемента, а 4 это число элементов в строке) или как ptr2[6]. Заметим, что внешне похожее обращение arr2[6] выполнить невозможно так как указателя с индексом 6 не существует.

Для обращения к элементу arr3[2][3][4] из трехмерного массива тоже можно использовать указатель, описанный как float *ptr3=arr3[0][0] с одним индексным выражением в форме ptr3[3*2+4*3+4] или ptr3[22].

Далее приведена функция, позволяющая найти минимальный элемент в трехмерном массиве. В функцию передается адрес начального элемента и размеры массива, возвращаемое значение - указатель на структуру, содержащую индексы минимального элемента.

```
struct INDEX {  int i,
                int j,
                int k }  min_index ;

struct INDEX  * find_min (int *ptr1, int l, int m int n)
{     int  min, i, j, k, ind;
      min=*ptr1;
      min_index.i=min_index.j=min_index.k=0;
      for (i=0; i*(ptr1+ind)
           {  min=*(ptr1+ind);
              min_index.i=i;
              min_index.j=j;
              min_index.k=k;
           }
      }
      return &min_index;
}
```

1.7.3. Операции с указателями

Над указателями можно выполнять унарные операции: инкремент и декремент. При выполнении операций ++ и -- значение указателя увеличивается или уменьшается на длину типа, на который ссылается используемый указатель.

Пример:

```
int *ptr, a[10];
   ptr=&a[5];
   ptr++;      /* равно адресу элемента a[6] */
   ptr--;      /* равно адресу элемента a[5] */
```

В бинарных операциях сложения и вычитания могут участвовать указатель и величина типа int. При этом результатом операции будет указатель на исходный тип, а его значение будет на указанное число элементов больше или меньше исходного.

Пример:

```
int *ptr1, *ptr2, a[10];
int i=2;
   ptr1=a+(i+4);   /*  равно адресу элемента a[6] */
   ptr2=ptr1-i;    /*  равно адресу элемента a[4] */
```

В операции вычитания могут участвовать два указателя на один и тот же тип. Результат такой операции имеет тип int и равен числу элементов исходного типа между уменьшаемым и вычитаемым, причем если первый адрес младше, то результат имеет отрицательное значение.

Пример:

```
int *ptr1, *ptr2, a[10];
int i;
   ptr1=a+4;
```

```
ptr2=a+9;
i=ptr1-ptr2;  /*   равно 5   */
i=ptr2-ptr1;  /*   равно -5  */
```

Значения двух указателей на одинаковые типы можно сравнивать в операциях ==, !=, <, <=, >, >= при этом значения указателей рассматриваются просто как целые числа, а результат сравнения равен 0 (ложь) или 1 (истина).

Пример:

```
int *ptr1, *ptr2, a[10];
   ptr1=a+5;
   ptr2=a+7;
   if (prt1>ptr2) a[3]=4;
```

В данном примере значение ptr1 меньше значения ptr2 и поэтому оператор a[3]=4 не будет выполнен.

1.7.4. Массивы указателей

В языке СИ элементы массивов могут иметь любой тип, и, в частности, могут быть указателями на любой тип. Рассмотрим несколько примеров с использованием указателей.

Следующие объявления переменных

```
int a[]={10,11,12,13,14,};
int *p[]={a, a+1, a+2, a+2, a+3, a+4};
int **pp=p;
```

порождают программные объекты, представленные на схеме на рис.4.

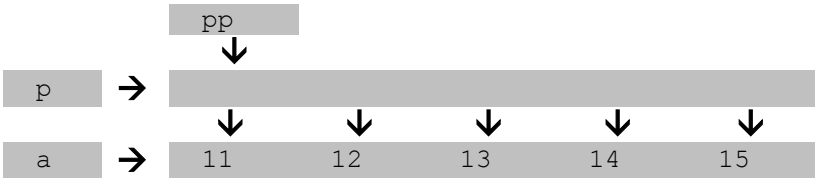

Рис.4. Схема размещения переменных при объявлении.

При выполнении операции pp-p получим нулевое значение, так как ссылки pp и p равны и указывают на начальный элемент массива указателей, связанного с указателем p (на элемент p[0]).

После выполнения операции pp+=2 схема изменится и примет вид, изображенный на рис.5.

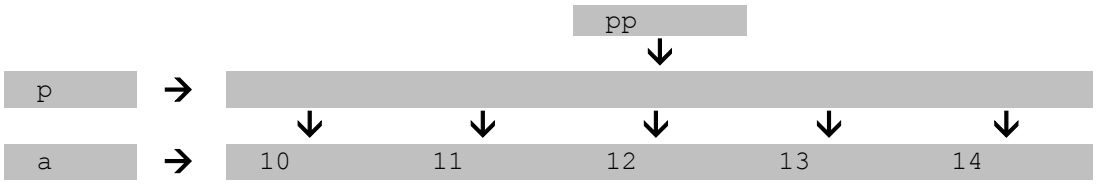

Рис.5. Схема размещения переменных после выполнения операции pp+=2.

Результатом выполнения вычитания pp-p будет 2, так как значение pp есть адрес третьего элемента массива p. Ссылка *pp-a тоже дает значение 2, так как обращение *pp есть адрес

третьего элемента массива a, а обращение a есть адрес начального элемента массива a. При обращении с помощью ссылки **pp получим 12 - это значение третьего элемента массива a. Ссылка *pp++ даст значение четвертого элемента массива p т.е. адрес четвертого элемента массива a.

Если считать, что pp=p, то обращение *++pp это значение первого элемента массива a (т.е. значение 11), операция ++*pp изменит содержимое указателя p[0], таким образом, что он станет равным значению адреса элемента a[1].

Сложные обращения раскрываются изнутри. Например обращение *(++(*pp)) можно разбить на следующие действия: *pp дает значение начального элемента массива p[0], далее это значение инкременируется ++(*p) в результате чего указатель p[0] станет равен значению адреса элемента a[1], и последнее действие это выборка значения по полученному адресу, т.е. значение 11.

В предыдущих примерах был использован одномерный массив, рассмотрим теперь пример с многомерным массивом и указателями. Следующие объявления переменных

```
int a[3][3]={  { 11,12,13 },
               { 21,22,23 },
               { 31,32,33 }   };
int *pa[3]={ a,a[1],a[2] };
int *p=a[0];
```

порождают в программе объекты представленные на схеме на рис.6.

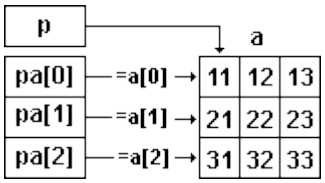

Рис.6. Схема размещения указателей на двумерный массив.

Согласно этой схеме доступ к элементу a[0][0] получить по указателям a, p, pa при помощи следующих ссылок: a[0][0], *a, **a[0], *p, **pa, *p[0].

Рассмотрим теперь пример с использованием строк символов. Объявления переменных

```
char *c[]={ "abs", "dx", "yes", "no" };
char **cp[]={ c+3, c+2 , c+1 , c };
char ***cpp=cp;
```

можно изобразить схемой представленной на рис.7.

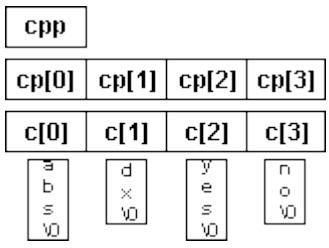

Рис.7. Схема размещения указателей на строки.

1.7.5. Динамическое размещение массивов

При динамическом распределении памяти для массивов следует описать соответствующий указатель и присваивать ему значение при помощи функции calloc. Одномерный массив a[10] из элементов типа float можно создать следующим образом

```
float *a;
a=(float*)(calloc(10,sizeof(float));
```

Для создания двумерного массива вначале нужно распределить память для массива указателей на одномерные массивы, а затем распределять память для одномерных массивов. Пусть, например, требуется создать массив a[n][m], это можно сделать при помощи следующего фрагмента программы:

```
#include
main ()
    {  double **a;
       int n,m,i;
       scanf("%d %d",&n,&m);
       a=(double **)calloc(m,sizeof(double *));
       for (i=0; i<=m; i++)
           a[i]=(double *)calloc(n,sizeof(double));
      . . . . . .
      }
```

Аналогичным образом можно распределить память и для трехмерного массива размером n,m,l. Следует только помнить, что ненужную для дальнейшего выполнения программы память следует освобождать при помощи функции free.

```
#include
main ()
    {  long ***a;
       int n,m,l,i,j;
       scanf("%d %d %d",&n,&m,&l);
       /* --------  распределение памяти -------- */
       a=(long ***)calloc(m,sizeof(long **));
       for (i=0; i<=m; i++)
          {    a[i]=(long **)calloc(n,sizeof(long *));
               for (j=0; i<=l; j++)
               a[i][j]=(long *)calloc(l,sizeof(long));
           }
      . . . . . .
       /* --------- освобождение памяти ----------*/
       for (i=0; i<=m; i++)
           {   for (j=0; j<=l; j++)
               free (a[i][j]);
               free (a[i]);
            }
        free (a);
      }
```

Рассмотрим еще один интересный пример, в котором память для массивов распределяется в вызываемой функции, а используется в вызывающей. В таком случае в вызываемую функцию требуется передавать указатели, которым будут присвоены адреса выделяемой для массивов памяти.

```
  Пример:
#include
main()
{   int vvod(double ***, long **);
    double **a;     /*  указатель для массива  a[n][m]   */
```

```
   long *b;        /*  указатель для массива  b[n]      */
   vvod (&a,&b);
    ..   /* в функцию vvod передаются адреса указателей, */
    ..   /* а не их значения                              */
    ..
}
int vvod(double ***a, long **b)
{   int n,m,i,j;
    scanf (" %d %d ",&n,&m);
    *a=(double **)calloc(n,sizeof(double *));
    *b=(long *)calloc(n,sizeof(long));
    for (i=0; i<=n; i++)
        *a[i]=(double *)calloc(m,sizeof(double));
    .....
 }
```

Отметим также то обстоятельство, что указатель на массив не обязательно должен показывать на начальный элемент некоторого массива. Он может быть сдвинут так, что начальный элемент будет иметь индекс отличный от нуля, причем он может быть как положительным так и отрицательным.

Пример:

```
#include
int main()
{  float *q, **b;
   int i, j, k, n, m;
   scanf("%d %d",&n,&m);
   q=(float *)calloc(m,sizeof(float));
/* сейчас указатель q показывает на начало массива    */
   q[0]=22.3;
   q-=5;
/* теперь начальный элемент массива имеет индекс 5,   */
/* а конечный элемент индекс n-5                      */
   q[5]=1.5;
/* сдвиг индекса не приводит к перераспределению      */
/* массива в памяти и изменится начальный элемент     */
   q[6]=2.5;   /*  -  это второй элемент              */
   q[7]=3.5;   /*  -  это третий элемент              */
   q+=5;
/* теперь начальный элемент вновь имеет индекс 0,     */
/* а значения элементов q[0], q[1], q[2] равны        */
/* соответственно 1.5, 2.5, 3.5                       */
   q+=2;
/* теперь начальный элемент имеет индекс -2,          */
/* следующий -1, затем 0 и т.д. по порядку            */
   q[-2]=8.2;
   q[-1]=4.5;
   q-=2;
/* возвращаем начальную индексацию, три первых        */
/* элемента массива q[0], q[1], q[2], имеют           */
/* значения 8.2, 4.5, 3.5                             */
   q--;
/* вновь изменим индексацию .                         */
/* Для освобождения области памяти в которой размещен */
/* массив q используется функция free(q), но поскольку */
/* значение указателя q смещено, то выполнение     */
/* функции free(q) приведет к непредсказуемым последствиям. */
/* Для правильного выполнения этой функции    */
/* указатель q должен быть возвращен в первоначальное */
/* положение                                          */
   free(++q);
```

```
 /* Рассмотрим возможность изменения индексации и      */
 /* освобождения памяти для двумерного массива         */
    b=(float **)calloc(m,sizeof(float *));
    for (i=0; i < m; i++)
        b[i]=(float *)calloc(n,sizeof(float));
 /* После распределения памяти начальным элементом      */
 /* массива будет элемент b[0][0]                       */
 /* Выполним сдвиг индексов так, чтобы начальным        */
 /* элементом стал элемент b[1][1]                      */
    for (i=0; i < m ; i++) --b[i];
    b--;
 /* Теперь присвоим каждому элементу массива сумму его  */
 /* индексов                                            */
    for (i=1; i<=m; i++)
        for (j=1; j<=n; j++)
            b[i][j]=(float)(i+j);
 /* Обратите внимание на начальные значения счетчиков   */
 /* циклов i и j, он начинаются с 1 а не с 0            */
 /* Возвратимся к прежней индексации                    */
    for (i=1; i<=m; i++) ++b[i];
    b++;
 /* Выполним освобождение памяти                        */
    for (i=0; i < m; i++)  free(b[i]);
    free(b);
    ...
    ...
    return 0;
 }
```

В качестве последнего примера рассмотрим динамическое распределение памяти для массива указателей на функции, имеющие один входной параметр типа double и возвращающие значение типа double.

Пример:

```
#include
#include
 double cos(double);
 double sin(double);
 double tan(double);
int main()
{ double (*(*masfun))(double);
  double x=0.5, y;
  int i;
  masfun=(double(*(*))(double))
           calloc(3,sizeof(double(*(*))(double)));
  masfun[0]=cos;
  masfun[1]=sin;
  masfun[2]=tan;
  for (i=0; i<3; i++);
  {  y=masfun[i](x);
     printf("\n x=%g  y=%g",x,y);
  }
  return 0;
}
```

1.8. Директивы Препроцессора

Директивы препроцессора представляют собой инструкции, записанные в тексте программы на СИ, и выполняемые до трансляции программы. Директивы препроцессора позволяют изменить текст программы, например, заменить некоторые лексемы в тексте, вставить текст из другого файла, запретить трансляцию части текста и т.п. Все директивы препроцессора начинаются со знака #. После директив препроцессора точка с запятой не ставятся.

1.8.1. Директива #include

Директива #include включает в текст программы содержимое указанного файла. Эта директива имеет две формы:

```
#include "имя файла"
#include <имя файла>
```

Имя файла должно соответствовать соглашениям операционной системы и может состоять либо только из имени файла, либо из имени файла с предшествующим ему маршрутом. Если имя файла указано в кавычках, то поиск файла осуществляется в соответствии с заданным маршрутом, а при его отсутствии в текущем каталоге. Если имя файла задано в угловых скобках, то поиск файла производится в стандартных директориях операционной системы, задаваемых командой PATH.

Директива #include может быть вложенной, т.е. во включаемом файле тоже может содержаться директива #include, которая замещается после включения файла, содержащего эту директиву.

Директива #include широко используется для включения в программу так называемых заголовочных файлов, содержащих прототипы библиотечных функций, и поэтому большинство программ на СИ начинаются с этой директивы.

1.8.2. Директива #define

Директива #define служит для замены часто использующихся констант, ключевых слов, операторов или выражений некоторыми идентификаторами. Идентификаторы, заменяющие текстовые или числовые константы, называют именованными константами. Идентификаторы, заменяющие фрагменты программ, называют макроопределениями, причем макроопределения могут иметь аргументы.

Директива #define имеет две синтаксические формы:

```
#define  идентификатор текст
#define  идентификатор (список параметров) текст
```

Эта директива заменяет все последующие вхождения идентификатора на текст. Такой процесс называется макроподстановкой. Текст может представлять собой любой фрагмент программы на СИ, а также может и отсутствовать. В последнем случае все экземпляры идентификатора удаляются из программы.

Пример:

```
#define WIDTH 80
#define LENGTH (WIDTH+10)
```

Эти директивы изменят в тексте программы каждое слово WIDTH на число 80, а каждое слово LENGTH на выражение (80+10) вместе с окружающими его скобками.

Скобки, содержащиеся в макроопределении, позволяют избежать недоразумений, связанных с порядком вычисления операций. Например, при отсутствии скобок выражение t=LENGTH*7 будет преобразовано в выражение t=80+10*7, а не в выражение t=(80+10)*7, как это получается при наличии скобок, и в результате получится 780, а не 630.

Во второй синтаксической форме в директиве #define имеется список формальных параметров, который может содержать один или несколько идентификаторов, разделенных запятыми. Формальные параметры в тексте макроопределения отмечают позиции на которые должны быть подставлены фактические аргументы макровызова. Каждый формальный параметр может появиться в тексте макроопределения несколько раз.

При макровызове вслед за идентификатором записывается список фактических аргументов, количество которых должно совпадать с количеством формальных параметров.

```
Пример:
#define  MAX(x,y) ((x)>(y))?(x):(y)
Эта директива заменит фрагмент
            t=MAX(i,s[i]);
на фрагмент
            t=((i)>(s[i])?(i):(s[i]);
```

Как и в предыдущем примере, круглые скобки, в которые заключены формальные параметры макроопределения, позволяют избежать ошибок связанных с неправильным порядком выполнения операций, если фактические аргументы являются выражениями.

```
Например, при наличии скобок фрагмент
              t=MAX(i&j,s[i]||j);
будет заменен на фрагмент
             t=((i&j)>(s[i]||j)?(i&j):(s[i]||j);
а при отсутствии скобок - на фрагмент
             t=(i&j>s[i]||j)?i&j:s[i]||j;
в котором условное выражение вычисляется в совершенно другом порядке.
```

1.8.3. Директива #undef

Директива #undef используется для отмены действия директивы #define. Синтаксис этой директивы следующий #undef идентификатор

Директива отменяет действие текущего определения #define для указанного идентификатора. Не является ошибкой использование директивы #undef для идентификатора, который не был определен директивой #define.

Пример:

```
#undef WIDTH
#undef MAX
```

Эти директивы отменяют определение именованной константы WIDTH и макроопределения MAX.

2. Организация списков и их обработка

2.1. Линейные списки

2.1.1. Методы организации и хранения линейных списков

Линейный список - это конечная последовательность однотипных элементов (узлов), возможно, с повторениями. Количество элементов в последовательности называется длиной списка, причем длина в процессе работы программы может изменяться.

Линейный список F, состоящий из элементов D1,D2,...,Dn, записывают в виде последовательности значений заключенной в угловые скобки F=, или представляют графически (см.рис.12).

Рис.12. Изображение линейного списка.

Например, F1=< 2,3,1>,F2=< 7,7,7,2,1,12 >, F3=< >. Длина списков F1, F2, F3 равна соответственно 3,6,0.

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

- найти элемент с заданным свойством;

- определить первый элемент в линейном списке;

- вставить дополнительный элемент до или после указанного узла;

- исключить определенный элемент из списка;

- упорядочить узлы линейного списка в определенном порядке.

В реальных языках программирования нет какой-либо структуры данных для представления линейного списка так, чтобы все указанные операции над ним выполнялись в одинаковой степени эффективно. Поэтому при работе с линейными списками важным является представление используемых в программе линейных списков таким образом, чтобы была обеспечена максимальная эффективность и по времени выполнения программы, и по объему требуемой памяти.

Методы хранения линейных списков разделяются на методы последовательного и связанного хранения. Рассмотрим простейшие варианты этих методов для списка с целыми значениями F=<7,10>.

При последовательном хранении элементы линейного списка размещаются в массиве d фиксированных размеров, например, 100, и длина списка указывается в переменной l, т.е. в программе необходимо иметь объявления вида

```
float d[100];   int l;
```

Размер массива 100 ограничивает максимальные размеры линейного списка. Список F в массиве d формируется так:

```
d[0]=7; d[1]=10; l=2;
```

Полученный список хранится в памяти согласно схеме на рис.13.

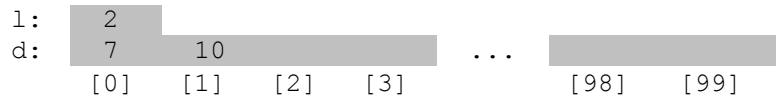

Рис.13. Последовательное хранение линейного списка.

При связанном хранении в качестве элементов хранения используются структуры, связанные по одной из компонент в цепочку, на начало которой (первую структуру) указывает указатель dl. Структура образующая элемент хранения, должна кроме соответствующего элемента списка содержать и указатель на соседний элемент хранения.

Описание структуры и указателя в этом случае может имееть вид:

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

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

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

В последнем элементе хранения (конец списка) указатель на соседний элемент имеет значение NULL. Получаемый список изображен на рис.14.

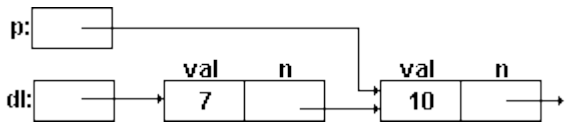

Рис.14. Связное хранение линейного списка.

2.1.2. Операции со списками при последовательном хранении

При выборе метода хранения линейного списка следует учитывать, какие операции будут выполняться и с какой частотой, время их выполнения и объем памяти, требуемый для хранения списка.

Пусть имеется линейный список с целыми значениями и для его хранения используется массив d (с числом элементов 100), а количество элементов в списке указывается

переменной l. Реализация указанных ранее операций над списком представляется следующими фрагментами программ которые используют объявления:

```
          float d[100];
          int i,j,l;
     1) печать значения первого элемента (узла)
        if (i<0 || i>l) printf("\n нет элемента");
        else printf("d[%d]=%f ",i,d[i]);
     2) удаление элемента, следующего за i-тым  узлом
        if (i>=l) printf("\n нет следующего ");
        l--;
        for (j=i+1;j<="1" узла i-того соседей обоих печать 3)
d[j]="d[j+1];">=l) printf("\n нет соседа");
        else printf("\n %d  %d",d[i-1],d[i+1]);
     4) добавление нового элемента new за i-тым узлом
        if (i==l || i>l) printf("\n нельзя добавить");
        else
        {    for (j=l; j>i+1; j--) d[j+1]=d[j];
             d[i+1]=new; l++;
        }
     5) частичное упорядочение списка с элементами  K1,K2,...,Kl  в
     список K1',K2',...,Ks,K1,Kt",...,Kt", s+t+1=l так, чтобы K1'= K1.

     { int t=1;
       float aux;
       for (i=2; i<=l; i++)
         if (d[i]=2; j--) d[j]=d[j-1];
               t++;
               d[i]=aux;
           }
     }
```

Схема движения индексов i,j,t и значения aux=d[i] при выполнении приведенного фрагмента программы приведена на рис.15.

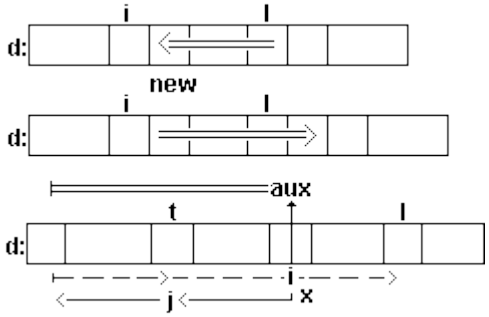

Рис.15. Движение индексов при выполнении операций над списком в последовательном хранении.

Количество действий Q, требуемых для выполнения приведенных операций над списком, определяется соотношениями: для операций 1 и 2 - Q=1; для операций 3,4 - Q=l; для операции 5 - Q=l*l.

Заметим, что вообще операцию 5 можно выполнить при количестве действий порядка l, а операции 3 и 4 для включения и исключения элементов в конце списка, часто встречающиеся при работе со стеками, - при количестве действий 1.

Более сложная организация операций требуется при размещении в массиве d нескольких списков, или при размещении списка без привязки его начала к первому элементу массива.

2.1.3. Операции со списками при связном хранении

При простом связанном хранении каждый элемент списка представляет собой структуру nd, состоящую из двух элементов: val - предназначен для хранения элемента списка, n - для указателя на структуру, содержащую следующий элемент списка. На первый элемент списка указывает указатель dl. Для всех операций над списком используется описание:

```
typedef struct nd
  {  float val;
     struct nd * n;  }  ND;
int i,j;
ND * dl, * r, * p;
```

Для реализации операций могут использоваться следующие фрагменты программ:

1) печать значения i-го элемента

```
r=dl;j=1;
while(r!=NULL && j++n ;
if (r==NULL) printf("\n нет узла %d ",i);
else printf("\n элемент %d равен %f ",i,r->val);
```

2) печать обоих соседей узла(элемента), определяемого указателем p (см. рис.16)

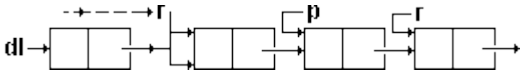

Рис.16. Схема выбора соседних элементов.

```
if((r=p->n)==NULL) printf("\n нет соседа справа");
else  printf("\n  сосед  справа  %f",  r->val);
if(dl==p) printf("\n нет соседа слева" );
else { r=dl;
       while( r->n!=p ) r=r->n;
       printf("\n левый сосед %f", r->val);
     }
```

3) удаление элемента, следующего за узлом, на который указывает p (см. рис.17)

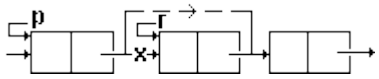

Рис.17. Схема удаления элемента из списка.

```
if ((r=p->n)==NULL) printf("\n  нет  следующего");
p->n=r->n; free(r->n);
```

4) вставка нового узла со значением new за элементом, определенным указателем p (см. рис.18)

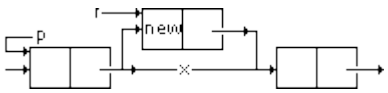

Рис.18. Схема вставки элемента в список.

```
r=malloc(1,sizeof(ND));
r->n=p->n;   r->val=new;   p->n=r;
```

5) частичное упорядочение списка в последовательность значений , s+t+1=l, так что K1'=K1; после упорядочения указатель v указывает на элемент K1' (см. рис.19)

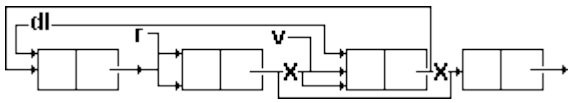

Рис.19. Схема частичного упорядочения списка.

```
ND *v;
float k1;
k1=dl->val;
r=dl;
while( r->n!=NULL )
{ v=r->n;
   if (v->valn=v->n;
       v->n=dl;
       dl=v;
     }
    else r=v;
}
```

Количество действий, требуемых для выполнения указанных операций над списком в связанном хранении, оценивается соотношениями: для операций 1 и 2 - Q=l; для операций 3 и 4 - Q=1; для операции 5 - Q=l.

2.1.4. Организация двусвязных списков

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 метода связанного хранения списка F=< 2,5,7,1 > как списка с двумя связями приведена на рис.20.

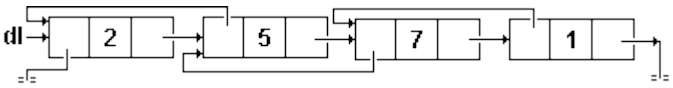

Рис.20. Схема хранения двусвязного списка.

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);
```

Связанное хранение линейного списка называется циклическим списком, если его последний указывает на первый элемент, а указатель dl - на последний элемент списка.

Схема циклического хранение списка F=< 2,5,7,1 > приведена на рис.21.

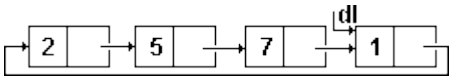

Рис.21. Схема циклического хранения списка.

При решении конкретных задач могут возникать разные виды связанного хранения.

Пусть на входе задана последовательность целых чисел B1,B2,...,Bn из интервала от 1 до 9999, и пусть Fi (1<I по возрастанию. Составить процедуру для формирования Fn в связанном хранении и возвращения указателя на его начало.

При решении задачи в каждый момент времени имеем упорядоченный список Fi и при вводе элемента Bi+1 вставляем его в нужное место списка Fi, получая упорядоченный список Fi+1. Здесь возможны три варианта: в списке нет элементов; число вставляется в начало списка; число вставляется в конец списка. Чтобы унифицировать все возможные варианты, начальный список организуем как связанный список из двух элементов <0,1000>.

Рассмотрим программу решения поставленной задачи, в которой указатели dl, r, p, v имеют следующее значение: dl указывает начало списка; p, v - два соседних узла; r фиксирует узел, содержащий очередное введенное значение in.

```
#include
#include
typedef struct str1
  { float val;
    struct str1 *n; }  ND;
main()
 {  ND *arrange(void);
    ND *p;
    p=arrange();
    while(p!=NULL)
      {
       printf("\n %f ",p->val);
       p=p->n;
      }
 }
ND *arrange() /*   формирование упорядоченного списка */
 {  ND *dl, *r, *p, *v;
    float in=1;
    char *is;
    dl=malloc(sizeof(ND));
    dl->val=0;                    /*  первый элемент     */
    dl->n=r=malloc(sizeof(ND));
    r->val=10000; r->n=NULL;      /*  последний элемент  */
    while(1)
    {
       scanf(" %s",is);
       if(* is=='q') break;
       in=atof(is);
       r=malloc(sizeof(ND));
       r->val=in;
```

```
            p=dl;
            v=p->n;
            while(v->valn;
            }
            r->n=v;
            p->n=r;
        }
        return(dl);
    }
```

2.1.5. Стеки и очереди

В зависимости от метода доступа к элементам линейного списка различают разновидности линейных списков называемые стеком, очередью и двусторонней очередью.

Стек - это конечная последовательность некоторых однотипных элементов - скалярных переменных, массивов, структур или объединений, среди которых могут быть и одинаковые. Стек обозначается в виде: S= и представляет динамическую структуру данных; ее количество элементов заранее не указывается и в процессе работы, как правило изменяется. Если в стеке элементов нет, то он называется пустым и обозначается S=< >.

Допустимыми операциями над стеком являются:

- проверка стека на пустоту S=< >,

- добавление нового элемента Sn+1 в конец стека - преобразование < S1,...,Sn> в < S1,...,Sn+1>;

- изъятие последнего элемента из стека - преобразование < S1,...,Sn-1,Sn> в < S1,...,Sn-1>;

- доступ к его последнему элементу Sn, если стек не пуст.

Таким образом, операции добавления и удаления элемента, а также доступа к элементу выполняются только в конце списка. Стек можно представить как стопку книг на столе, где добавление или взятие новой книги возможно только сверху.

Очередь - это линейный список, где элементы удаляются из начала списка, а добавляются в конце списка (как обыкновенная очередь в магазине).

Двусторонняя очередь - это линейный список, у которого операции добавления и удаления элементов и доступа к элементам возможны как вначале так и в конце списка. Такую очередь можно представить как последовательность книг стоящих на полке, так что доступ к ним возможен с обоих концов.

Реализация стеков и очередей в программе может быть выполнена в виде последовательного или связанного хранения. Рассмотрим примеры организации стека этими способами.

Одной из форм представления выражений является польская инверсная запись, задающая выражение так, что операции в нем записываются в порядке выполнения, а операнды находятся непосредственно перед операцией.

Например, выражение

```
(6+8)*5-6/2
```

в польской инверсной записи имеет вид

```
6 8 + 5 * 6 2 / -
```

Особенность такой записи состоит в том, что значение выражения можно вычислить за один просмотр записи слева направо, используя стек, который до этого должен быть пуст. Каждое новое число заносится в стек, а операции выполняются над верхними элементами стека, заменяя эти элементы результатом операции. Для приведенного выражения динамика изменения стека будет иметь вид

```
S = < >;  <6>;  <6,8>;  <14>;  <14,5>;  <70>;
    <70,6>;  <70,6,2>;  <70,3>;  <67>.
```

Ниже приведена функция eval, которая вычисляет значение выражения, заданного в массиве m в форме польской инверсной записи, причем m[i]>0 означает неотрицательное число, а значения m[i]<0 - операции. В качестве кодировки операций сложения, вычитания, умножения и деления выбраны отрицательные числа -1, -2, -3, -4. Для организации последовательного хранения стека используется внутренний массив stack. Параметрами функции являются входной массив a и его длина l.

```
float eval (float *m, int l)
{ int p,n,i;
  float stack[50],c;
   for(i=0; i < l ;i++)
   if ((n=m[i])<0)
     {  c=st[p--];
        switch(n)
        { case -1:  stack[p]+=c;  break;
          case -2:  stack[p]-=c;  break;
          case -3:  stack[p]*=c;  break;
          case -4:  stack[p]/=c;
        }
     }
   else stack[++p]=n;
   return(stack[p]);
}
```

Рассмотрим другую задачу. Пусть требуется ввести некоторую последовательность символов, заканчивающуюся точкой, и напечатать ее в обратном порядке (т.е. если на входе будет "ABcEr-1." то на выходе должно быть "1-rEcBA"). Представленная ниже программа сначала вводит все символы последовательности, записывая их в стек, а затем содержимое стека печатается в обратном порядке. Это основная особенность стека - чем позже элемент занесен в стек, тем раньше он будет извлечен из стека. Реализация стека выполнена в связанном хранении при помощи указателей p и q на тип, именованный именем STACK.

```
#include
typedef struct st           /* объявление типа STACK */
  { char ch;
    struct st *ps;   } STACK;
main()
{   STACK *p,*q;
    char a;
```

```
    p=NULL;
    do                          /*  заполнение стека    */
    { a=getch();
      q=malloc(sizeof(STR1));
       q->ps=p; p=q;
       q->ch=a;
    }  while(a!='.');
    do                          /*  печать стека        */
    { p=q->ps;free(q);q=p;
      printf("%c",p->ch);
    } while(p->ps!=NULL);
}
```

2.1.6. Сжатое и индексное хранение линейных списков

При хранении больших объемов информации в форме линейных списков нежелательно хранить элементы с одинаковым значением, поэтому используют различные методы сжатия списков.

Сжатое хранение. Пусть в списке B= несколько элементов имеют одинаковое значение V, а список B'= получается из B заменой каждого элемента Ki на пару Ki'=(i,Ki). Пусть далее B"=< K1",K2",...,Km" > - подсписок B', получающийся вычеркиванием всех пар Ki=(i,V). Сжатым хранением B является метод хранения B", в котором элементы со значением V умалчиваются. Различают последовательное сжатое хранение и связанное сжатое хранение. Например, для списка B=, содержащего несколько узлов со значением X, последовательное сжатое и связанное сжатое хранения, с умалчиванием элементов со значением X, представлены на рис.22,23.

```
1,C     3,Y     6,S     7,H     9,T
```

Рис.22. Последовательное сжатое хранение списка.

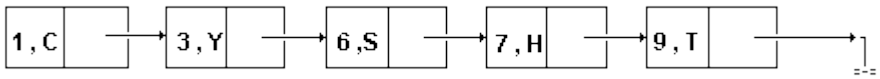

Рис.23. Связное сжатое хранение списка.

Достоинство сжатого хранения списка при большом числе элементов со значением V заключается в возможности уменьшения объема памяти для его хранения.

Поиск i-го элемента в связанном сжатом хранении осуществляется методом полного просмотра, при последовательном хранении - методом бинарного поиска.

Преимущества и недостатки последовательного сжатого и связанного сжатого хранений аналогичны преимуществам и недостаткам последовательного и связанного хранений.

Рассмотрим следующую задачу. На входе заданы две последовательности целых чисел M=< M1,M2,...,M10000 >, N=< N1,N2,...,N10000 >, причем 92% элементов последовательности M равны нулю. Составить программу для вычисления суммы произведений Mi * Ni, i=1,2,...,10000.

Предположим, что список M хранится последовательно сжато в массиве структур m с объявлением:

```
struct
{ int nm;
  float val; }   m[10000];
```

Для определения конца списка добавим еще один элемент с порядковым номером m[j].nm=10001, который называется стоппером (stopper) и располагается за последним элементом сжатого хранения списка в массиве m.

Программа для нахождения искомой суммы имеет вид:

```
# include
main()
{ int i,j=0;
  float inp,sum=0;
  struct                          /* объявление  массива  */
  { int nm;                       /* структур             */
    float val;  }   m[10000];

  for(i=0;i<10000;i++)            /*  чтение списка    M  */
     { scanf("%f",&inp);
       if (inp!=0)
        { m[j].nm=i;
          m[j++].val=inp;
        }
      }
  m[j].nm=10001;                  /*    stopper           */
  for(i=0,j=0; i<10000; i++)
     { scanf("%f",&inp);          /*  чтение списка    N  */

       if(i==m[j].nm)             /*  вычисление суммы    */
         sum+=m[j++].val*inp;
     }
  printf( "сумма произведений Mi*Ni равна %f",sum);
}
```

Индексное хранение используется для уменьшения времени поиска нужного элемента в списке и заключается в следующем. Исходный список B = < K1,K2, ...,Kn > разбивается на несколько подсписков B1,B2, ...,Bm таким образом, что каждый элемент списка B попадает только в один из подсписков, и дополнительно используется индексный список с M элементами, указывающими на начало списков B1,B2, ...,Bm.

Считается, что список хранится индексно с помощью подсписков B1,B2, ...,Bm и индексного списка X = < ADG1,ADG2,... ADGm >, где ADGj - адрес начала подсписка Bj, j=1,M.

При индексном хранении элемент K подсписка Bj имеет индекс j. Для получения индексного хранения исходный список B часто преобразуется в список B' путем включения в каждый узел еще и его порядкового номера в исходном списке B, а в j-ый элемент индексного списка X, кроме ADGj, может включаться некоторая дополнительная информация о подсписке Bj. Разбиение списка B на подсписки осуществляется так, чтобы все элементы B, обладающие определенным свойством Pj, попадали в один подсписок Bj.

Достоинством индексного хранения является то, что для нахождения элемента K с заданным свойством Pj достаточно просмотреть только элементы подсписка Bj; его начало находится по индексному списку X, так как для любого K, принадлежащего Bi, при i не равном j свойство Pj не выполняется.

В разбиении B часто используется индексная функция G(K), вычисляющая по элементу K его индекс j, т.е. G(K)=j. Функция G обычно зависит от позиции K, обозначаемой поз.K, в подсписке B или от значения определенной части компоненты K - ее ключа.

Рассмотрим список B=< K1,K2, ...,K9 > с элементами

```
K1=(17,Y),   K2=(23,H),    K3=(60,I),   K4=(90,S),    K5=(66,T),
K6=(77,T),   K7=(50,U),    K8=(88,W),   K9=(30,S).
```

Если для разбиения этого списка на подсписки в качестве индексной функции взять Ga(K)=1+(поз.K-1)/3, то список разделится на три подсписка:

```
B1a=<(17,Y),(23,H),(60,I)>,
B2a=<(90,S),(66,T),(77,T)>,
B3a=<(50,U),(88,W),(30,S)>.
```

Добавляя всюду еще и начальную позицию элемента в списке, получаем:

```
B1a'=<(1,17,Y),(2,23,H),(3,60,I)>,
B2a'=<(4,90,S),(5,66,T),(6,77,T)>,
B3a'=<(7,50,U),(8,88,W),(9,30,S)>.
```

Если в качестве индексной функции выбрать другую функцию Gb(K)=1+(поз.K-1)%3, то получим списки:

```
B1b"=<(1,17,Y),(4,90,S),(7,50,U)>,
B2b"=<(2,23,H),(5,66,T),(8,88,U)>,
B3b"=<(3,60,I),(6,77,T),(9,30,S)>.
```

Теперь для нахождения узла K6 достаточно просмотреть только одну из трех последовательностей (списков). При использовании функции Ga(K) это список B2a', а при функции Gb(K) список B3b".

Для индексной функции Gc(K)=1+K1/100, где K1 - первая компонента элемента K, находим:

```
B1=<(17,Y),(23,H),(60,I),(90,S)>,
B2=<(66,T),(77,T)>,
B3=<(50,U),(88,W)>,
B4=<(30,S)>.
```

Чтобы найти здесь узел K с первым компонентом-ключом K1=77, достаточно просмотреть список B2.

При реализации индексного хранения применяется методика A для хранения индексного списка X (функция Ga(X)) и методика C для хранения подсписков B1,B2,...,Bm (функция Gc(Bi)), т.е. используется, так называемое, A-C индексное хранение.

В практике часто используется последовательно-связанное индексное хранение. Так как обычно длина списка индексов известна, то его удобно хранить последовательно, обеспечивая прямой доступ к любому элементу списка индексов. Подсписки B1,B2,...,Bm хранятся связанно, что упрощает вставку и удаление узлов(элементов). В частности, подобный метод хранения используется в ЕС ЭВМ для организации, так называемых, индексно-последовательных наборов данных, в которых доступ к отдельным записям возможен как последовательно, так и при помощи ключа.

Последовательно-связанное индексное хранение для приведенного примера изображено на рис.24, где X=.

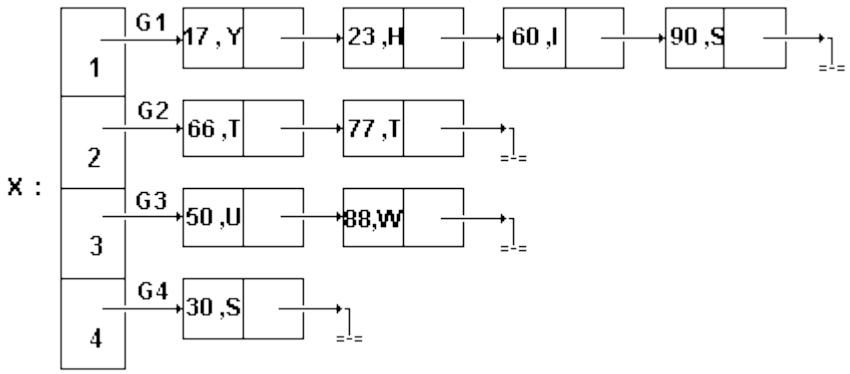

Рис.24. Последовательно-связанное индексное хранение списка.

Рассмотрим еще одну задачу. На входе задана последовательность целых положительных чисел, заканчивающаяся нулем. Составить процедуру для ввода этой последовательности и организации ее последовательно-связанного индексного хранения таким образом, чтобы числа, совпадающие в двух последних цифрах, помещались в один подсписок.

Выберем в качестве индексной функции G(K)=K%100+1, а в качестве индексного списка X - массив из 100 элементов. Следующая функция решает поставленную задачу:

```
#include
#include
typedef struct nd
        {  float val;
           struct nd *n; }   ND;
int index (ND *x[100])
{  ND *p;
   int i,j=0;
   float inp;
   for (i=0; i<100; i++)  x[i]=NULL;
   scanf("%d",&inp);
   while (inp!=0)
   {   j++;
       p=malloc(sizeof(ND));
       i=inp%100+1;
       p->val=inp;
       p->n=x[i];
       x[i]=p;
       scanf("%d",&inp);
   }
   return j;
}
```

Возвращаемым значением функции index будет число обработанных элементов списка.

Для индексного списка также может использоваться индексное хранение. Пусть, например, имеется список B= с элементами

```
K1=(338,Z),  K2=(145,A),  K3=(136,H),  K4=(214,I),  K5 =(146,C),
K6=(334,Y),  K7=(333,P),  K8=(127,G),  K9=(310,O),  K10=(322,X).
```

Требуется разделить его на семь подсписков, т.е. X= таким образом, чтобы в каждый список B1,B2,...,B7 попадали элементы, совпадающие в первой компоненте первыми двумя цифрами. Список X, в свою очередь, будем индексировать списком индексов Y=, чтобы в каждый список Y1,Y2,Y3 попадали элементы из X, у которых в первой компоненте совпадают первые цифры. Если списки B1,B2,...,B7 хранить связанно, а

списки индексов X,Y индексно, то такой способ хранения списка B называется связанно-связанным связанным индексным хранением. Графическое изображение этого хранения приведено на рис.25.

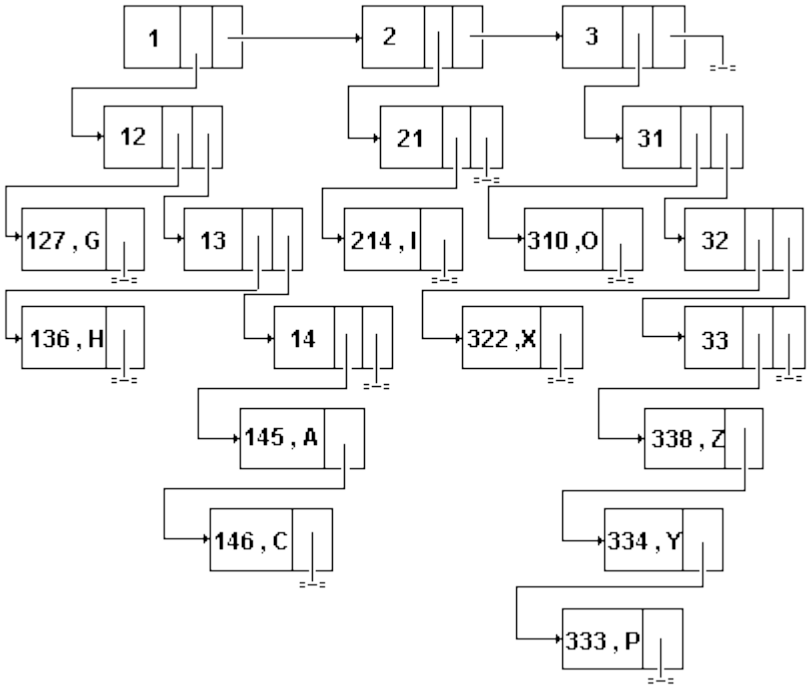

Рис.25. Связанно-связанное связанное индексное хранение списка.

2.2. Сортировка И Слияние Списков

При работе со списками очень часто возникает необходимость перестановки элементов списка в определенном порядке. Такая задача называется сортировкой списка и для ее решения существуют различные методы. Рассмотрим некоторые из них.

2.2.1. Пузырьковая сортировка

Задача сортировки заключается в следующем: задан список целых чисел (простейший случай) B=< K1, K2,..., Kn >. Требуется переставить элементы списка B так, чтобы получить упорядоченный список B'=< K'1, K'2,...,K'n >, в котором для любого 1<=i<=n элемент K'(i) <= K'(i+1).

При обменной сортировке упорядоченный список B' получается из B систематическим обменом пары рядом стоящих элементов, не отвечающих требуемому порядку, пока такие пары существуют.

Наиболее простой метод систематического обмена соседних элементов с неправильным порядком при просмотре всего списка слева на право определяет пузырьковую сортировку: максимальные элементы как бы всплывают в конце списка.

Пример:

```
B=<20,-5,10,8,7>,   исходный список;
B1=<-5,10,8,7,20>,  первый просмотр;
B2=<-5,8,7,10,20>,  второй просмотр;
```

```
B3=<-5,7,8,10,20>,  третий просмотр.
```

В последующих примерах будем считать, что сортируется одномерный массив (либо его часть от индекса n до индекса m) в порядке возрастания элементов.

Нижеприведенная функция bubble сортирует входной массив методом пузырьковой сортировки.

```
/*   сортировка   пузырьковым   методом          */
float * bubble(float * a, int m, int n)
{
char is=1;
int i;
float c;
  while(is)
  {   is=0;
      for (i=m+1; i<=n; i++)
      if ( a[i] < a[i-1] )
      {    c=a[i];
           a[i]=a[i-1];
           a[i-1]=c;
           is=1;
      }
  }
  return(a);
}
```

Пузырьковая сортировка выполняется при количестве действий Q=(n-m)*(n-m) и не требует дополнительной памяти.

2.2.2. Сортировка вставкой

Упорядоченный массив B' получается из B следующим образом: сначала он состоит из единственного элемента K1; далее для i=2,...,N выполняется вставка узла Ki в B' так, что B' остается упорядоченным списком длины i.

Например, для начального списка B=< 20,-5,10,8,7 > имеем:

```
B=< 20,-5,10,8,7>  B'=< >
B=< -5,10,8,7 >    B'=< 20 >
B=< 10,8,7 >       B'=< -5,20 >
B=< 8,7  >         B'=< -5,10,20 >
B=< 7 >            B'=< -5,8,10,20 >
B=< >              B'=< -5,7,8,10,20 >
```

Функция insert реализует сортировку вставкой.

```
/*    сортировка  методом   вставки             */
float *insert(float *s, int m, int n)
{
int i,j,k;
float aux;
for (i=m+1; i<=n; i++)
     { aux=s[i];
       for (k=m; k<=i && s[k]=k; j--)  s[j+1]=s[j];
       s[k]=aux;
     }
     return(a);
}
```

Здесь оба списка B и B' размещаются в массиве s, причем список B занимает часть s с индексами от i до n, а B' - часть s с индексами от m до i-1 (см. рис.26).

При сортировке вставкой требуется Q=(n-m)*(n-m) сравнений и не требуется дополнительной памяти.

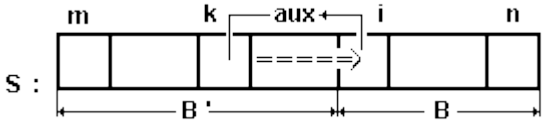

Рис.26. Схема движения индексов при сортировке вставкой.

2.2.3. Сортировка посредством выбора

Упорядоченный список B' получается из B многократным применением выборки из B минимального элемента, удалением этого элемента из B и добавлением его в конец списка B', который первоначально должен быть пустым.

Например:

```
B=<20,10,8,-5,7>, B'=< >
B=<20,10,8,7>,    B'=<-5>
B=<20,10,8>,      B'=<-5,7>
B=<20,10>,        B'=<-5,7,8>
B=<20>,           B'=<-5,7,8,10>
B=< >,            B'=<-5,7,8,10,20> .
```

Функция select упорядочивает массив s сортировкой посредством выбора.

```
/*    сортировка  методом  выбора          */
double *select( double *s, int m, int n)
{
  int i,j;
  double c;
  for (i=m; is[j])
       {  c=s[i];
          s[i]=s[j];
          s[j]=c;
       }
  return(s);
}
```

Здесь, как и в предыдущем примере оба списка B и B' размещаются в разных частях массива s (см. рис.27). При сортировке посредством выбора требуется Q=(n-m)*(n-m) действий и не требуется дополнительной памяти.

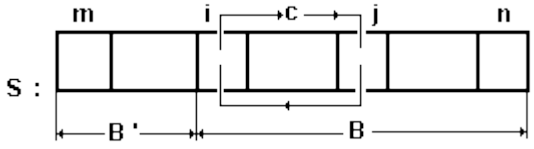

Рис.27. Схема движения индексов при сортировке выбором.

Сортировка квадратичной выборкой. Исходный список B из N элементов делится на M подсписков B1,B2,...,Bm, где M равно квадратному корню из N, и в каждом B1 находится минимальный элемент G1. Наименьший элемент всего списка B определяется как

минимальный элемент Gj в списке , и выбранный элемент Gj заменяется новым наименьшим из списка Bj. Количество действий, требуемое для сортировки квадратичной выборкой, несколько меньше, чем в предыдущих методах Q= N*N, но требуется дополнительная память для хранения списка G.

2.2.4. Слияние списков

Упорядоченные списки A и B длин M и N сливаются в один упорядоченный список C длины M+N, если каждый элемент из A и B входит в C точно один раз. Так, слияние списков A=<6,17,23,39,47> и B=<19,25,38,60> из 5 и 4 элементов дает в качестве результата список C=<6,17,19,23,25,38,39,47,60> из 9 элементов.

Для слияния списков A и B список C сначала полагается пустым, а затем к нему последовательно приписывается первый узел из A или B, оказавшийся меньшим и отсутствующий в C.

Составим функцию для слияния двух упорядоченных, расположенных рядом частей массива s. Параметром этой функции будет исходный массив s с выделенными в нем двумя расположенными рядом упорядоченными подмассивами: первый с индекса low до индекса low+l, второй с индекса low+l+1 до индекса up, где переменные low, l, up указывают месторасположения подмассивов. Функция merge осуществляет слияние этих подмассивов, образуя на их месте упорядоченный массив с индексами от low до up (см. рис.28).

```
/*   слияние списков */
double *merge(double *s, int low, int up, int l)
{
double *b,*c,v;
  int i,j,k;
  b=calloc(l,sizeof(double));
  c=calloc(up+1-l,sizeof(double));
  for(i=low;is[up-1]) ?
                          (s[low+l-1]+1) : (s[up-1]+1)));
  i=(j=0);
  k=low;
  while(b[i]<V||C[J]< } (s); return k++; s[k]="b[i++];" else if(b[i]
```

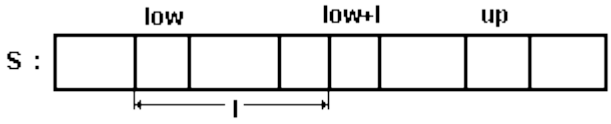

Рис.28. Схема движения индексов при слиянии списков.

2.2.5. Сортировка списков путем слияния

Для получения упорядоченного списка B' последовательность значений B= разделяют на N списков B1=, B2=,...,Bn=, длина каждого из которых 1. Затем осуществляется функция прохода, при которой M>=2 упорядоченных списков B1,B2,...,Bm заменяется на M/2 (или (M+1)/2) упорядоченных списков, B(2i-1)-ого и B(2i)-ого (2i<=M) и добавлением Bm при нечетном M. Проход повторяется до тех пор пока не получится одна последовательность длины N.

Приведем пример сортировки списка путем использования слияния, отделяя последовательности косой чертой, а элементы запятой.
Пример:

```
 9 / 7 / 18 / 3 / 52 / 4 / 6 / 8 / 5 / 13 / 42 / 30 / 35 / 26;
7,9 / 3,18 / 4 / 52 / 6 / 8 / 54 / 13 / 30 / 42 / 26 / 35;
3,7,9,18 /  4,6,8,52 /  5,13,30,42 /  26,35;
3,4,6,7,8,9,18,52 /   5,13,26,30,35,42;
3,4,5,6,7,8,9,13,18,26,30,35,42,52.
```

Количество действий, требуемое для сортировки слиянием, равно Q=N*log2(N), так как за один проход выполняется N сравнений, а всего необходимо осуществить
log2(N) проходов. Сортировка слиянием является очень эффективной и часто применяется для больших N, даже при использовании внешней памяти.
Функция smerge упорядочивает массив s сортировкой слиянием, используя описанную ранее функцию merge.

```
/* сортировка слиянием */
double *smerge (double *s, int m, int n)
{ int l,low,up;
  double *merge (double *, int, int, int);
  l=1;
  while(l<=(n-m))
  {  low=m;
     up=m-1;
     while (l+up < n)
     {   up=(low+2*l-1 < n) ? (low+2*l-1) : n ;
         merge (s,low,up,l);
         low=up-1;
     }
     l*=2;
  }
  return(a);
}
```

Для сортировки массива путем слияния удобно использовать рекурсию. Составим рекурсивную функцию srecmg для сортировки массива либо его части. При каждом вызове сортируемый массив делится на две равных части, каждая из которых сортируется отдельно, а затем происходит их слияние, как это показано на рис.29.

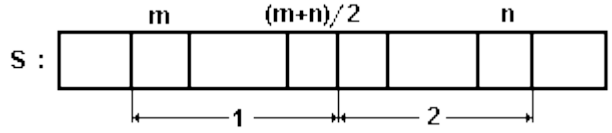

Рис.29. Схема сортировки слиянием.

```
/*    рекурсивная   сортировка слиянием  1/2          */
double *srecmg (double *a, int m, int n)
{ double * merge (double *, int, int, int);
  double * smerge (double *, int, int);
   int i;
  if (n>m)
     { i=(n+m)/2;
       srecmg(a,m,i);
       srecmg(a,i+1,n);
       merge(a,m,n,(n-m)/2+1);
     }
  return (a);
}
```

2.2.6. Быстрая и распределяющая сортировки

Быстрая сортировка состоит в том, что список B=< K1,K2,...,Kn> реорганизуется в список B',< K1 >,B", где B' - подсписок B с элементами, не большими K1, а B" - подсписок B с элементами, большими K1. В списке B',< K1 >,B" элемент K1 расположен на месте, на котором он должен быть в результирующем отсортированном списке. Далее к спискам B' и B" снова применяется упорядочивание быстрой сортировкой. Приведем в качестве примера сортировку списка, отделяя упорядоченные элементы косой чертой, а элементы Ki знаками < и >.

Пример:

```
      9, 7, 18, 3, 52, 4, 6, 8, 5, 13, 42, 30, 35, 26
      7, 3, 4, 6, 8, 5/ <9>/ 18, 52, 13, 42, 30, 35, 26
      3, 4, 6, 5/<7>/ 8/ 9/ 13/ <18>/ 52, 42, 30, 35, 26
      <3>/ 4, 6, 5/ 7/ 8/ 9/ 13/ 18/ 42, 30, 35, 26/ <52>
      3/ <4>/ 6, 5/ 7/ 8/ 9/ 13/ 18/ 30, 35, 26/ <42>/ 52
      3/ 4/ 5/ <6>/ 7/ 8/ 9/ 13/ 18/ 26/ <30>/ 35/ 42/ 52
```

Время работы по сортировке списка методом быстрой сортировки зависит от упорядоченности списка. Оно будет минимальным, если на каждом шаге разбиения получаются подсписки B' и B" приблизительно равной длины, и тогда требуется около N*log2(N) шагов. Если список близок к упорядоченному, то требуется около
(N*N)/2 шагов.

Рекурсивная функция quick упорядочивает участок массива s быстрой сортировкой.

```
      /*               быстрая сортировка             */
      double * quick(double *s,int low,int hi)
      { double cnt,aux;
        int i,j;
        if (hi>low)
           { i=low;
             j=hi;
             cnt=s[i];
             while(i < j)
             {  if (s[i+1]<=cnt)
                {  s[i]=s[i+1];
                   s[i+1]=cnt;
                   i++;
                }
                else
                {  if (s[j]<=cnt)
                   {  aux=s[j];
                      s[j]=s[i+1];
                      s[i+1]=aux;
                    }
                   j--;
                }
             }
             quick(s,low,i-1);
             quick(s,i+1,hi);
           }
        return(s);
     }
```

Здесь используются два индекса i и j, проходящие части массива навстречу друг другу (см. рис.30). При этом i всегда фиксирует разделяющий элемент cnt=s[low], слева от которого находятся числа, не большие cnt, а справа от i - числа, большие cnt. Возможны три случая: при s[i+1]<=cnt; при s[i+1]>cnt и s[j]<=cnt; при s[i+1]>cnt и s[j]>cnt. По окончании работы i=j, и cnt=s[i] устанавливается на своем месте.

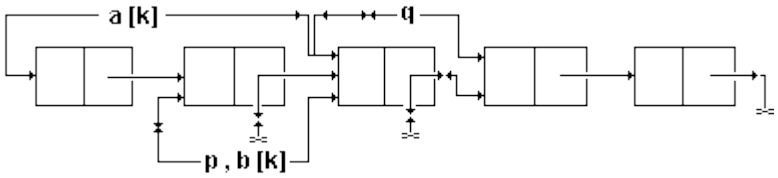

Рис.30. Схема быстрой сортировки.

Быстрая сортировка требует дополнительной памяти порядка log2(N) для выполнения рекурсивной функции quick (неявный стек).
Оценка среднего количества действий, необходимых для выполнения быстрой сортировки списка из N различных чисел, получена как оценка отношения числа различных возможных последовательностей из N различных чисел, равного N!, и общего количества действий C(N), необходимых для выполнения быстрой сортировки
всех различных последовательностей. Доказано, что C(N)/N! < 2*N*ln(N).
Распределяющая сортировка. Предположим, что элементы линейного списка B есть T-разрядные положительные десятичные числа D(j,n) - j-я справа цифра в десятичном числе n>=0, т.е. D(j,n)=floor(n/m)%10, где m=10^(j-1). Пусть B0,B1,...,B9 - вспомогательные списки (карманы), вначале пустые.
Для реализации распределяющей сортировки выполняется процедура, состоящая из двух процессов, называемых распределение и сборка для j=1,2,...,T.
РАСПРЕДЕЛЕНИЕ заключается в том, что элемент Ki (i=1,N) из B добавляется как последний в список Bm, где m=D(j,Ki), и таким образом получаем десять списков,
в каждом из которых j-тые разряды чисел одинаковы и равны m.
СБОРКА объединяет списки B0,B1,...,B9 в этом же порядке, образуя один список B.
Рассмотрим реализацию распределяющей сортировки при T=2 для списка:
B=<09,07,18,03,52,04,06,08,05,13,42,30,35,26> .

```
                РАСПРЕДЕЛЕНИЕ-1:
     B0=<30>,  B1=< >,  B2=<52,42>, B3=<03,13>, B4=<04>,
     B5=<05,35>,  B6=<06,26>,B7=<07>, B8=<18,08>, B9=<09>.
                СБОРКА-1:
     B=<30,52,42,03,13,04,05,35,06,26,07,18,08,09>
                РАСПРЕДЕЛЕНИЕ-2:
     B0=<03,04,05,06,07,08,09>, B1=<13,18>, B2=<26>,
     B3=<30,35>, B4=<42>, B5=<52>, B6=< >,B7=< >,B8=< >, B9=< >.
                СБОРКА-2:
     B=<03,04,05,06,07,08,09,13,18,26,30,35,42,52>.
```

Количество действий, необходимых для сортировки N T-цифровых чисел, определяется как Q(N*T). Недостатком этого метода является необходимость использования дополнительной памяти под карманы.
Однако можно исходный список представить как связанный и сортировку организовать так, чтобы для карманов B0,B1,...,B9 не использовать дополнительной
памяти, элементы списка не перемещать, а с помощью перестановки указателей присоединять их к тому или иному карману.
В представленной ниже программе функция pocket реализует распределяющую сортировку связанного линейного списка (указатель q), в котором содержатся T-разрядные десятичные положительные числа, без использования дополнительной памяти; в функции a[i], b[i] указывают соответственно на первый и на последний
элементы кармана Bi.

```
     /*   вызов  распределяющей  сортировки  списка    */
     #include
     #include
     typedef struct str
     { long val;
       struct str *n; } SP1;
```

```
main()
{ int i;
  SP1 *q=malloc(sizeof(SP1)),*r;
  SP1 *pocket(SP1 * ,int );
  long a[14]={ 0,7,18,3,52,4,6,8,5,13,42,30,35,26 };
  q->n=NULL;
  q->val=a[0];
  r=q;
  printf(" %d",a[0]);
  for(i=1;i<14;i++)      /*  формирование списка  */
    {   r->n=malloc(sizeof(SP1));
        r->val=a[i];
        (r->n)->n=NULL;
        r=r->n;
        printf(" %d",a[i]);
    }
  r=pocket(q,2);
  printf("\n");          /*  печать результатов   */
  while (r!=NULL)
    {   printf(" %d",r->val);
        r=r->n;
    }
}
/*    распределяющая  сортировка  списка          */
SP1 *pocket(SP1 *q,int t)
{ int i,j,k,m=1;
  SP1 *r, *gg, *a[10], *b[10];
  gg=q;
  for(j=1;j<=t;j++)
    {  for(i=0;i<=9;i++) a[i]=(b[i]=NULL);
       while(q!=NULL)
         {  k=((int)(q->val/m))%(int)10;
            r=b[k];
            if (a[k]==NULL) a[k]=q;
            else r->n=q;
            r=b[k]=q;
            q=q->n;
            r->n=NULL;
         }
      for(i=0;i<=9;i++)
      if (a[i]!=NULL) break;
      q=a[i];
      r=b[i];
      for(k=i+1;k<=9;k++)
      if(a[k]!=NULL)
         {  r->n=a[k];
            r=b[k];
         }
      m=m*10;
    }
  return (gg);
}
```

На рис.31 показана схема включения очередного элемента списка в К-й карман.

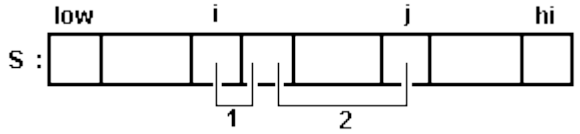

Рис.31. Схема включения очередного элемента списка в карман.

Разновидностью распределяющей сортировки является битовая сортировка. В ней элементы списка интерпретируются как двоичные числа, и D(j,n) обозначает j-ю справа двоичную цифру числа n. При этой сортировке в процессе РАСПРЕДЕЛЕНИЕ требуются только два вспомогательных кармана B0 и B1; их можно разместить в одном массиве, двигая списки B0 и B1 навстречу друг другу и отмечая точку встречи. Для осуществления СБОРКИ нужно за списком B0 написать инвертированный
список B1.
Так как выделение j-го бита требует только операций сдвига, то битовая сортировка хорошо подходит для внешней сортировки на магнитных лентах и дисках.
Разновидностью битовой сортировки является бинарная сортировка. Здесь из всех элементов списка B= выделяются его минимальный и максимальный элементы и находится их среднее арифметическое m=(MIN+MAX)/2. Список B разбивается на подсписки B1 и B2, причем в B1 попадают элементы, не большие m,
а в список B2 - элементы, большие m. Потом для непустых подсписков B1 и B2 сортировка продолжается рекурсивно.

2.3.1. Последовательный поиск

Задача поиска. Пусть заданы линейные списки: список элементов B=<K1,K2,K3,...,Kn> и список ключей V= (в простейшем случае это целые числа). Требуется для каждого значения Vi из V найти множество всех совпадающих с ним элементов из B. Чаще всего встречается ситуация когда V содержит один элемент, а в B имеется не более одного такого элемента.

Эффективность некоторого алгоритма поиска A оценивается максимальным Max{A} и средним Avg{A} количествами сравнений, необходимых для нахождения элемента V в B. Если Pi - относительная частота использования элемента Ki в B, а Si - количество сравнений, необходимое для его поиска, то

$$Max\{A\} = \max\{ S_i, i=1,n \} \; ; \quad Avg\{A\} = \sum_{i=1}^{n} P_i S_i \; .$$

Последовательный поиск предусматривает последовательный просмотр всех элементов списка B в порядке их расположения, пока не найдется элемент равный V. Если достоверно неизвестно, что такой элемент имеется в списке, то необходимо следить за тем, чтобы поиск не вышел за пределы списка, что достигается использованием стоппера.

Очевидно, что Max последовательного поиска равен N. Если частота использования каждого элемента списка одинакова, т.е. P=1/N, то Avg последовательного поиска равно N/2. При различной частоте использования элементов Avg можно улучшить, если поместить часто встречаемые элементы в начало списка.

Пусть во входном потоке задано 100 целых чисел K1,K2,... K100 и ключ V. Составим программу для последовательного хранения элементов Ki и поиска среди них элемента,

равного V, причем такого элемента может и не быть в списке. Без использования стоппера программа может быть реализована следующим образом:

```
/*    последовательный поиск без стоппера    */
#include
main()
{
int k[100],v,i;
for (i=0;i<100;i++)
scanf("%d",&k[i]);
scanf("%d",&v);
i=0;
while(k[i]!=v && i<100) i++;
if (k[i]==v) printf("%d %d",v,i);
else printf("%d не найден",v);
}
```

С использованием стоппера программу можно записать в виде:

```
/*  последовательный поиск со стоппером    */
#include
main()
{
int k[101],v,i;
for (i=0;i<100;i++)
scanf("%d",&k[i]);              /*   ввод данных  */
scanf("%d",&v);
k[100]=v;                       /*   стоппер      */
i=0;
while(k[i]!=v) i++;
if (i<100) printf ("%d %d",v,i);
else printf ("%d не найден",v);
}
```

2.3.2. Бинарный поиск

Для упорядоченных линейных списков существуют более эффективные алгоритмы поиска, хотя и для таких списков применим последовательный поиск. Бинарный поиск состоит в том, что ключ V сравнивается со средним элементом списка. Если эти значения окажутся равными, то искомый элемент найден, в противном случае поиск продолжается в одной из половин списка.

Нахождение элемента бинарным поиском осуществляется очень быстро. Max бинарного поиска равен log2(N), и при одинаковой частоте использования каждого элемента Avg бинарного поиска равен log2(N). Недостаток бинарного поиска заключается в необходимости последовательного хранения списка, что усложняет операции добавления и исключения элементов .

Пусть, например, во входном потоке задано 101 число, K1,K2,...,K100, V - элементы списка и ключ. Известно, что список упорядочен по возрастанию, и элемент V в списке имеется. Составим программу для ввода данных и осуществления бинарного поиска ключа V в списке K1,K2,...,K100.

```
/*    Бинарный поиск      */
#include
main()
{
int k[100],v,i,j,m;
```

```
for (i=0;i<100;i++)
scanf("%d",&k[i]);
scanf("%d",&v);
i=0;   j=100;   m=50;
while (k[m]!=v)
   {
   if (k[m] < v)  i+=m;
   else j=m-i;
   m=(i+j)/2;
   }
printf("%d  %d",v,m);
}
```

2.3.3. M-блочный поиск

Этот способ удобен при индексном хранении списка. Предполагается, что исходный упорядоченный список B длины N разбит на M подсписков B1,B2,...,Bm длины N1,N2,...,Nm, таким образом, что B=B1,B2,..,Bm.

Для нахождения ключа V, нужно сначала определить первый из списков Bi, i=1,M, последний элемент которого больше V, а потом применить последовательный поиск к списку Bi.

Хранение списков Bi может быть связным или последовательным. Если длины всех подсписков приблизительно равны и M= N, то Max M-блочного поиска равен 2 N. При одинаковой частоте использования элементов Avg M-блочного поиска равен N.

Описанный алгоритм усложняется, если не известно, действительно ли в списке имеется элемент, совпадающий с ключом V. При этом возможны случаи: либо такого элемента в списке нет, либо их несколько.

Если вместо ключа V имеется упорядоченный список ключей, то последовательный или M-блочный поиск может оказаться более удобным, чем бинарный, поскольку не требуется повторной инициализации для каждого нового ключа из списка V.

2.3.4. Методы вычисления адреса

Методы вычисления адреса. Пусть в каждом из M элементов массива T содержится элемент списка (например целое положительное число). Если имеется некоторая функция H(V), вычисляющая однозначно по элементу V его адрес - целое положительное число из интервала [0,M-1],то V можно хранить в массиве T с номером H(V) т.е. V=T(H(V)). При таком хранении поиск любого элемента происходит за постоянное время не зависящее от M.

Массив T называется массивом хеширования, а функция H - функцией хеширования.

При конкретном применении хеширования обычно имеется определенная область возможных значений элементов списка V и некоторая информация о них. На основе этого выбирается размер массива хеширования M и строится функция хеширования. Критерием для выбора M и H является возможность их эффективного использования.

Пусть нужно хранить линейный список из элементов K1,K2,..,Kn, таких, что при Ki=Kj, mod(Ki,26)= mod(Kj,26). Для хранения списка выберем массив хеширования T(26) с

пространством адресов 0-25 и функцию хеширования H(V)= mod(V,26). Массив T заполняется элементами T(H(Ki))=Ki и T(j)=0 если j (H(K1), H(K2),..,H(Kn)).

Поиск элемента V в массиве T с присваиванием Z его индекса если V содержится в T, или -1, если V не содержится в T, осуществляется следующим образом

```
int t[26],v,z,i;
i=(int)fmod((double)v,26.0);
if(t[i]==v) z=i;
else z=-1;
```

Добавление нового элемента V в список с возвращением в Z индекса элемента, где он будет храниться, реализуется фрагментом

```
z=(int)fmod((doule)v,26.0);
t[z]=v;
```

а исключение элемента V из списка присваиванием

```
t[(int)fmod((double)v,26)]=0;
```

Теперь рассмотрим более сложный случай, когда условие Ki=Kj H(Ki)=H(Kj) не выполняется. Пусть V - множество возможных элементов списка (целые положительные числа), в котором максимальное число элементов равно 6. Возьмем M=8 и в качестве функции хеширования выберем функцию H(V)=Mod(V,8).

Предположим, что B=, причем H(K1)=5, H(K2)=3, H(K3)=6, H(K4)=3, H(K5)=1, т.е. H(K2)=H(K4) хотя K2=K4. Такая ситуация называется коллизией, и в этом случае при заполнении массива хеширования требуется метод для ее разрешения. Обычно выбирается первая свободная ячейка за собственным адресом. Для нашего случая массив T[8] может иметь вид

```
 T=<0,K5,0,K2,K4,K1,K3,0> .
```

При наличии коллизий усложняются все алгоритмы работы с массивом хеширования. Рассмотрим работу с массивом T[100], т.е. с пространством адресов от 0 до 99. Пусть количество элементов N не более 99, тогда в T всегда будет хотя бы один свободный элемент равный нулю. Для объявления массива используем оператор

```
    int static t[100];
```

Добавление в массив T нового элемента Z с занесением его адреса в I и числа элементов в N выполняется так:

```
i=h(z);
while (t[i]!=0 && t[i]!=z)
if (i==99) i=0;
else i++;
if (t[i]!=z)  t[i]=z,  n++;
```

Поиск в массиве T элемента Z с присвоением I индекса Z, если Z имеется в T, или I=-1, если такого элемента нет, реализуется следующим образом:

```
i=h(z);
while (t[i]!=0 && t[i]!=z)
```

```
if (i==99) i=0;
else i++;
if (t[i]==0) i=-1;
```

При наличии коллизий исключение элемента из списка путем пометки его как пустого, т.е. t[i]=0, может привести к ошибке. Например, если из списка B исключить элемент K2, то получим массив хеширования в виде T=<0,K5,0,0,K4,K1,K3,0>, в котором невозможно найти элемент K4, поскольку H(K4)=3, а T(3)=0. В таких случаях при исключении элемента из списка можно записывать в массив хеширования некоторое значение непринадлежащее области значений элементов списка и не равное нулю. При работе с таким массивом это значение будет указывать на то, что нужно просматривать со средние ячейки.

Достоинство методов вычисления адреса состоит в том, что они самые быстрые, а недостаток в том, что порядок элементов в массиве T не совпадает с их порядком в списке, кроме того довольно сложно осуществить динамическое расширение массива T.

2.3.5. Выбор в линейных списках

Задача выбора. Задан линейный список целых, различных по значению чисел B=, требуется найти элемент, имеющий i-тое наибольшее значение в порядке убывания элементов. При i=1 задача эквивалентна поиску максимального элемента, при i=2 поиску элемента с вторым наибольшим значением.

Поставленная задача может быть получена из задачи поиска j-того минимального значения заменой i=n-j+1 и поиском i-того максимального значения. Особый интерес представляет задача выбора при i=a/n, 0<a<1, в частности, задача выбора медианы при a=1/2.

Все варианты задачи выбора легко решаются, если список B полностью отсортирован, тогда просто нужно выбрать i-тый элемент. Однако в результате полной сортировки списка B получается больше информации, чем требуется для решения поставленной задачи.

Количество действий можно уменьшить применяя сортировку выбором только частично до i-того элемента. Это можно сделать, напри мер при помощи функции findi

```
/*  выбор путем частичной сортировки  */
int findi(int *s, int n, int i)
{
   int c,j,k;
   for (k=0; k<=i; k++)
   for (j=k+1; j<=n; j++)
   if (s[k] < s[j])
   {  c=s[k];
      s[k]=s[j];
      s[j]=c;
   }
   return s[i];
}
```

Эта функция ищет элемент с индексом i, частично сортируя массив s, и выполняет при этом (n*i) сравнений. Отсюда следует, что функция findi приемлима для решения задачи при малом значении i, и малоэффективна при нахождении медианы.

Для решения задачи выбора i-того наибольшего значения в списке B модифицируем алгоритм быстрой сортировки. Список B разбиваем элементом K1 на подсписки B' и B", такие, что если Ki -B', то Ki>K1, и если Ki - B", то Ki<K1, и список B реорганизуется в список B',K1,B". Если K1 элемент располагается в списке на j-том месте и j=i, то искомый элемент найден. При j>i наибольшее значение ищется в списке B'; при j<i будем искать (i-j) значение в подсписке B".

Алгоритм выбора на базе быстрой сортировки в общем эффективен, но для улучшения алгоритма необходимо, чтобы разбиение списка на подсписки осуществлялось почти пополам. Следующий алгоритм эффективно решает задачу выбора i-того наибольшего элемента в списке B, деля его на подсписки примерно равной величины.

1. Если N<21, то выбрать i-тый наибольший элемент списка B обычной сортировкой.

2. Если N>21 разделим список на P=N/7 подсписков по 7 элементов в каждом, кроме последнего в котором mod(N,7) элементов.

3. Определим список W из медиан полученных подсписков (четвертых наибольших значений) и найдем в W его медиану M (рекурсивно при помощи данного алгоритма) т.е. (P/2+1)-й наибольший элемент.

4. С помощью элемента M разобьем список B на два подсписка B' с j элементами большими или равными M, и B" с N-j элементами меньшими M. При j>i повторим процедуру поиска сначала, но только в подсписке B'. При j=i искомый элемент найден, равен M и поиск прекращается. При j < i будем искать (i-j)-тый наибольший элемент в списке B".

```
/*   алгоритм выбора делением списка почти пополам   */
int search (int *b, int n, int i)
{
 int findi(int  *, int, int);
 int t, m, j, p, s, *w;
 if (n<21) return findi(b, n, i);     /*   шаг 1        */
 p=(int)(n/7);
 w=calloc(p+1,sizeof(int));           /*   шаги 2 и 3   */
 for (t=0; t < p; t++)
 w[t]=findi(b+7*t, 7, 3);
 if (n%7!=0)
    {   w[p]=findi(b+7*p,n%7,(n%7)/2);
        p++;
    }
 m=search(w, p, p/2);
 free (w);
 for (j=0, t=0; t < n; t++)             /*    шаг 4       */
 if (b[t]>=m) j++;
   if (j>i)
   {
     for (p=0, t=0; p < n; t++)
     if (b[t]>=m)
     { b[p]=b[t]; p++; }
     m=search(b, j, i);              /*   поиск в B"    */
   }
   if (j < i)
   {
     for (p=0, t=0; t < n; t++)
       if (b[t] < m)     b[p++]=b[t];
       m=search(b, n-j, i-j);       /*   поиск в B"    */
   }
```

```
 return m;
}
```

Рекурсивная функция search реализует алгоритм выбора i-того наибольшего значения. Для ее вызова можно использовать следующую программу

```
#include
#include
main()
{
  int search (int *b, int n, int i);
  int *b;
  int l, i, k, t;
  scanf("%d%d",&l,&i);
  printf
  ("\nВыбор %d максимального элемента из %d штук",i,l);
  b=(int *)(calloc(100,sizeof(int)));
  for (k=0; k<100; k++)
  b[k]=k;                          /* заполнение массива */
  for (k=1; k < l/4; k++)
      { t=b[k];                    /* перемешивание      */
        b[k]=b[l-k];               /* массива            */
        b[l-k]=t;
      }
  k=search(b,l,i);                 /* выбор  элемента    */
  printf ("\n выбран элемент равный %d\n\n",k);
  return 0;
}
```

Используя метод математической индукции, можно доказать, что для функции search требуется выполнить в самом неблагоприятном случае 28*N сравнений.

Действительно, если N<21, то выполнение функции findi потребует сравнений порядка N*(N-1)/2, т.е. меньше чем 28*N. Предположим, что для любого T<N количество сравнений при выполнении функции search не более 28*T и подсчитаем, сколько сравнений потребуется функции search при произвольном значении N. Для поиска медианы в каждом из подсписков функцией findi требуется не более 7*(7-1)/2=21 сравнений, а для формирования массива W в целом не более 21*(N/7)=3*N сравнений. По предположению индукции для поиска медианы в массиве W длины N/7 требуется 28*(N/7)=4*N сравнений. После удаления из B части элементов с помощью медианы в B' (или в B") останется не более N*5/7 элементов, и для удаления ненужных элементов необходимо количество сравнений порядка N. Для поиска в оставшейся части массива (в B' или B") по предположению индукции требуется не более 28*(N*5/7)=20*N сравнений. Таким образом, всего потребуется 3*N+4*N+N+20*N=28*N сравнений, т.е. выдвинутое предположение доказано.

2.4. Рекурсия

Функция называется рекурсивной, если во время ее обработки возникает ее повторный вызов, либо непосредственно, либо косвенно, путем цепочки вызовов других функций.

Прямой (непосредственной) рекурсией является вызов функции внутри тела этой функции.

```
int a()
{.....a().....}
```

Косвенной рекурсией является рекурсия, осуществляющая рекурсивный вызов функции посредством цепочки вызова других функций. Все функции, входящие в цепочку, тоже считаются рекурсивными.

Например:

```
a(){.....b().....}
b(){.....c().....}
c(){.....a().....} .
```

Все функции a,b,c являются рекурсивными, так как при вызове одной из них, осуществляется вызов других и самой себя.

Рассмотрим задачу о Ханойских башнях. Имеются три стержня с номерами 1,2,3. На стержень 1 надето n дисков различного диаметра так, что они образуют пирамиду (см.рис.31). Написать программу для печати последовательности перемещений дисков со стержня на стержень, необходимых для переноса пирамиды со стержня 1 на стержень 3 при использовании стержня 2 в качестве вспомогательного. При этом за одно перемещение должен переноситься только один диск, и диск большего диаметра не должен помещаться на диск меньшего диаметра. Доказано, что для n дисков минимальное число необходимых перемещений равно 2^n-1.

Рис.31. Задача о Ханойских башнях.

Для решения простейшего случая задачи, когда пирамида состоит только из одного диска, необходимо выполнить одно действие - перенести диск со стержня i на стержень j, что очевидно (этот перенос обозначается i -> j). Общий случай задачи изображен на рисунке, когда требуется перенести n дисков со стержня i на стержень j, считая стержень w вспомогательным. Сначала следует перенести n-1 диск со стержня i на стержень w при вспомогательном стержне j, затем перенести один диск со стержня i на стержень j и, наконец, перенести n-1 диск из w на стержень j, используя вспомогательный стержень i. Итак, задача о переносе n дисков сводится к двум задачам о переносе n-1 диска и одной простейшей задаче. Схематически это можно записать так: T(n,i,j,w) = T(n-1,i,w,j), T(1,i,j,w), T(n-1,w,j,i).

```
          i     n-1     w    n-1       j
   +      |  -------->-+|+--------->+  |
n-1|      |      I     |||    Ш     |  |
   +     / \           / \           / \
     +-/-----\-+     /     \      +-/-----\-+
   ==+----|----+===/=========\====+----|----+======
          +-------------->-------------+
                         П
```

Рис.32. Схема решения задачи о Ханойских башнях.

Ниже приведена программа, которая вводит число n и печатает список перемещений, решающая задачу о Ханойских башнях при количестве дисков n. Используется

внутренняя рекурсивная процедура tn(n,i,j,w), печатающая перемещения, необходимые для переноса n дисков со стержня i на стержень j с использованием вспомогательного стержня w при {i,j,w} = {1,3,2}.

```
/*                    ханойские башни                    */
#include
main()                                      /*   вызывающая   */
{   void tn(int, int, int, int);            /*   функция      */
    int n;
    scanf(" %d",&n);
    tn(n,1,2,3);
 }

 void tn(int n, int i, int j, int w)        /*   рекурсивная  */
 {  if (n>1)                                /*   функция      */
     {  tn (n-1,i,w,j);
        tn (1,i,j,w);
        tn (n-1,w,j,i);
      }
     else printf(" \n %d -> %d",i,j);
     return ;
 }
```

Последовательность вызовов процедуры tn при m=3 иллюстрируется древовидной структурой на рис.33. Каждый раз при вызове процедуры tn под параметры n, i, j, w выделяется память и запоминается место возврата. При возврате из процедуры tn память, выделенная под параметры n, i, j, w, освобождается и становится доступной память, выделенная под параметры n, i, j, w предыдущим вызовом, а управление передается в место возврата.

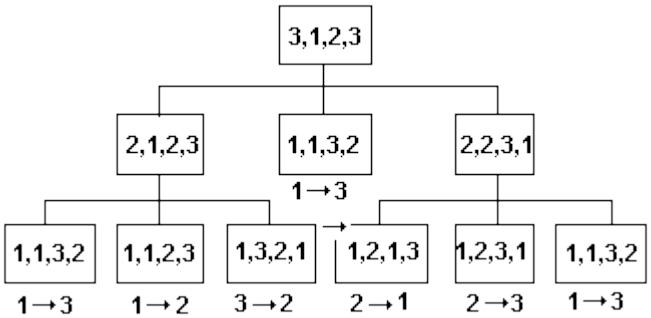

Рис.33. Последовательность вызовов процедуры tn.

Во многих случаях рекурсивные функции можно заменить на эквивалентные нерекурсивные функции или фрагменты, используя стеки для хранения точек вызова и вспомогательных переменных.

Предположим, что имеется ситуация:

```
main()              /* вызывающая  функция  */
{ ...  f() ...}
f()                 /* рекурсивная функция  */
{ ...  f() ...}
```

Здесь в функции main вызывается рекурсивная функция f. Требуется заменить описание функции f и ее вызова на эквивалентный фрагмент программы, т.е. удалить функцию f.

Пусть рекурсивная функция f имеет параметры P1,P2,...,Ps, внутренние переменные V1,V2,...,Vt и в функциях main и f имеется k обращений к функции f. Для удаления такой функции требуются следующие дополнительные объекты:

- переменные AR1,AR2,...,ARs, содержащие значения фактических параметров при вызове функции f (типы переменных должны соответствовать типам параметров P1,P2,...,Ps);

- переменная rz для вычисляемого функцией f результата (тип переменных совпадает с типом возвращаемого значения функции f);

- стек, содержащий в себе все параметры и все внутренние переменные функции f, а также переменную lr типа int, для хранения точки возврата, и переменную pst типа указатель, для хранения адреса предыдущего элемента стека;

- указатель dl для хранения адреса вершин стека;

- промежуточный указатель u для операций над стеком;

- k новых меток L1,...,Lk для обозначенных точек возврата;

- метка jf, используемая для обхода модифицированного тела функции f;

- промежуточная переменная l типа int для передачи номера точки возврата.

Чтобы получить эквивалентную нерекурсивную программу без функции f, необходимо выполнить следующие действия:

1. Убрать объявление функции f в функцию main;

2. Добавить в функции main объявления переменных AR1,AR2,...,ARs,RZ, объявления стека ST и указателей dl и u:

```
typedef struct st { P1;P2;...;Ps;V1;V2;...;Vt;
                    int lr;  struct st *pst  }  ST;
ST *dl=NULL, *u;
```

3. Модифицировать тело функции f во фрагмент программы. Для этого следует:

а) удалить заголовок функции f;

б) объявления параметров и внутренних переменных и заменить фрагментом:

```
      goto jf;
   f: a=malloc(sizeof(ST));
      a->P1=AR1; a->P2=AR2; ... ;a->Ps=ARs;
      a->lr=l; a->pst=dl;  dl=a;
```

в) в конце функции f поставить метку JF, а все обращения к формальным аргументам заменить обращением, к соответствующим элементам стека;

г) вместо каждого оператора return(y) в функции f записать фрагмент:

```
      RZ=y; l=dl->lr;
      a=dl; dl=a->pst; free(a);
```

```
switch(l)
{ case 1: goto L1;
  case 2: goto L2;
          ...
  case k: goto Lk;
}
```

4. Каждый i-тый вызов функции f (как в вызывающей функции, так и в теле функции f) вида V=f(A1,A2,...,As) заменить фрагментом программы :

```
AR1=A1; AR2=A2; ... ; ARs=As;  l=i;  goto f;
Li:  V=RZ;
```

где l=i обозначает l=1 при первом обращении к функции f, l=2 при втором обращении и т.д. Нумерация обращений может быть выполнена в произвольном порядке и требуется для возвращения в точку вызова с помощью операторов switch и goto Li; (где Li есть L1 при первой замене, Li есть L2 при второй замене и т.д.)

5. Вставить модифицированное тело функции f в конце функции main.

Для иллюстрации изложенного рассмотрим несколько вариантов реализации программы вычисляющей функцию Аккермана, которая определяется так:

```
          + X+1,                  если N=0
          | X,                    если N=1,  Y=0,
          | 0,                    если N=2,  Y=0,
A(N,X,Y)= | 1,                    если N=3,  Y=0,
          | 2,                    если N=>4, Y=0,
          + A(N-1,A(N,X,Y-1),X),  если N#0,  Y#0;

 где N,X,Y - целые неотрицательные числа.
```

Следующая программа вычисляет функцию Аккермана с использованием рекурсивной функции ackr и вспомогательной функции smacc:

```
/*      рекурсивное  вычисление функции Аккермана     */
# include
main ()                              /*  вызывающая     */
{   int x,y,n,t;                     /*  функция        */
    int ackr(int, int, int);
    scanf("%d %d %d",&n,&x,&y);
    t=ackr(n,x,y);
    printf("%d",t);
}
int smacc( int n,int x )           /*  вспомогательная  */
{   switch (n)                     /*   функция         */
     {  case 0:  return(x+1);
        case 1:  return (x);
        case 2:  return (0);
        case 3:  return (1);
        default: return (2);
      }
}
int ackr( int n, int x, int y)     /*  рекурсивная      */
{  int z;                          /*  функция          */
   int smacc( int,int);
   if(n==0 || y==0)  z=smacc(n,x);
   else { z=ackr(n,x,y-1);         /*  рекурсивные      */
          z=ackr(n-1,z,x);  }      /*  вызовы ackr(...) */
```

```
      return z;
   }
```

Модифицируя функции main и ackr в соответствии с изложенным методом получим следующую программу:

```
   /*       Эквивалентная нерекурсивная программа       */
   /*       для вычисления функции Аккермана            */
 #include
 #include
 int main()
 {  typedef struct st
    { int i,j,k,z,lr;
      struct st *pst;
    } ST;
    ST *u, *dl=NULL;
    int l,x,y,n;
    int  smacc(int,int);
    int an,ax,ay,rz,t;
    scanf("%i %i %i",&n,&x,&y);

    an=n;ax=x;ay=y;l=1;       /*  -   замена вызова    -  */
    goto ackr;                /*     t=ackr(n,x,y);       */
l1: t=rz;                     /*  -  -  -  -  -  -  -  -  */

    printf("\n %d ",t);
    goto jackr;

    /*    начало фрагмента заменяющего функцию  ackr      */
ackr:
    u=( ST *) malloc( sizeof (ST) );
    u->i=an;
    u->j=ax;
    u->k=ay;
    u->lr=l;
    u->pst=dl;
    dl=u;
    if (an==0||ay==0)
    dl->z=smacc(an,ax);
    else
        {
           an=dl->i;          /*  -   замена вызова    -  */
           ax=dl->j;          /*                          */
           ay=dl->k-1;        /*     z=ackr(n,x,y-1);     */
           l=2;               /*                          */
           goto ackr;         /*                          */
      l2:  dl->z=rz;          /*  -  -  -  -  -  -  -  -  */

           an=dl->i-1;        /*  -   замена вызова    -  */
           ax=rz;             /*                          */
           ay=dl->j;          /*     z=ackr(n-1,z,x);     */
           l=3;               /*                          */
           goto ackr;         /*                          */
      l3:  dl->z=rz;          /*  -  -  -  -  -  -  -  -  */
        }
    rz=dl->z;                 /*  -  -  -  -  -  -  -  -  */
    an=dl->i;                 /*                          */
    ax=dl->j;                 /*       замена             */
    ay=dl->k;                 /*                          */
    l=dl->lr;                 /*       оператора          */
    u=dl;                     /*                          */
    dl=u->pst;                /*       return z ;         */
    free(u);                  /*                          */
```

```
   switch(l)                  /*                          */
        {  case 1: goto l1;   /*                          */
           case 2: goto l2;   /*                          */
           case 3: goto l3;   /*                          */
        }                     /*  -  -  -  -  -  -  -  -  */
jackr:
}
int smacc( int n,int x )      /* вспомогательная функция */
{  switch (n)
        { case 0:  return(x+1);
          case 1:  return (x);
          case 2:  return (0);
          case 3:  return (1);
          default: return (2);
        }
}
```