10. МАКРОСРЕДСТВА

10.1. Макросы

В языках ассемблера и некоторых других языках используются макрокоманды. *Макрокоманда (макрос)* - это сокращенное обозначение группы команд *базового языке*, в котором используется макрокоманда (в общем случае - некоторого текста).

Применение макросов позволяет расширять базовый язык в каком-либо направлении, дополняя его новыми командами - макрокомандами, определяемыми через команды базового языка в виде *макроопределений*. Макроопределения пишут *на макроязыке*, который обычно, представляет собой надстройку над базовым языком, но может быть и самостоятельным языком. Многие языки снабжаются большими библиотеками макроопределений.

Таким путем в виде библиотеки макроопределений можно даже создавать несложные специализированные языки.

Кроме использования в программах, макросы могут служить в текстовых редакторах средством сокращенной записи и генерации любого текста.

Исходный текст на расширенном базовом языке с макрокомандами обрабатывается *макропроцессором* - программой, которая заменяет каждую макрокоманду на ее *макрорасширение* - описанный в макроопределении текст на базовом языке. Эта операция называется *макрообработкой*. На макропроцессор обычно возлагают также выполнение директивы **include** (включить), вставляющей в программу текст из заданного файла.

Макропроцессор - это одна из разновидностей *препроцессора* - вспомогательного транслятора, работающего перед основным транслятором - *процессором языка*.

Чаще всего базовым языком служит язык ассемблера. Ассемблер с макрокомандами называется *макроассемблером*. Он состоит из макропроцессора и ассемблера.

В общем случае макроязык представляет собой специализированный язык программирования для описания алгоритмов получения (генерации) текста. Подобно традиционным языкам программирования, в развитом макроязыке можно описывать и использовать локальные и глобальные переменные (*макропеременные*), в том числе массивы, присваивать им значения, использовать условные операторы, циклы, подпрограммы, в том числе рекурсивные (роль которых играют макроопределения), операторы ввода-вывода и т. п. Макрокоманду можно рассматривать как вызов подпрограммы макроязыка – *макровызов*.

В отличие от обычных языков, операторы и другие средства макроязыка и написанные на нем алгоритмы “работают” не во время исполнения программы, а во время ее трансляции, и называются поэтому средствами *периода трансляции*.

Макроопределение, в общем случае, представляет собой написанную на макроязыке подпрограмму, которая запускается макросом (макровызовом) и выполняется во время трансляции исходной программы.

Макроопределение может генерировать в зависимости от параметров макровызова и выводить в выходной файл некоторый текст - макрорасширение, но может и не порождать никакого текста, а, например, выполнять некоторую операцию с таблицей, реализованной в виде массива макропеременных.

Транслятором макроязыка является макропроцессор, который обычно реализуют в виде интерпретатора.

Средства макроязыка позволяют выполнять *условную трансляцию,* т. е. получать различные варианты программы из одного и того же исходного текста в зависимости от разнообразных условий, проверяемых во время трансляции.

Условную трансляцию можно использовать и без макросов, например, для получения различных версий программы из единого исходного текста. Таким способом, в частности, удобно создавать мобильные программы.

В некоторых макроязыках разрешается переопределять в виде макросов операторы базового языка. Тогда появляется возможность, не изменяя исходного текста, за счет переопределения операторов и перетрансляции с помощью макропроцессора преобразовать программу, в том числе перевести ее на другой язык или перенести в другую операционную систему.

В языках высокого уровня макросы применяются значительно реже, чем в языках ассемблера. Возможность использования макросов является важным достоинством языков C и C++. Макроязык для языка C/C++ очень прост и образован из *директив препроцессора*, начинающихся знаком #, например, #**include**, #**define**. Компилятор языка C/C++ состоит из препроцессора и процессора.

Макросредства языков C и C++ широко используются в библиотечных заголовочных файлах. Несмотря на свою простоту и даже примитивность, эти средства оказались весьма необходимыми и полезными, особенно для разработки сложных программ.

10.2. Макросредства языка C/C++

Директива препроцессора #**define** в общем случае представляет собой макроопределение и имеет следующий вид (между именем и скобкой пробел не пишется):

#**define** *имя(список_параметров) заменяющий_текст*

где заменяющий текст - это последовательность лексем. В частности, символическая константа, определенная с помощью директивы вида

#**define** *имя заменяющий_текст*

представляет собой макрос без параметров.

Пример 10.1. Значение константы N автоматически вычисляется и корректируется в зависимости от значения константы M:

```
#define  M  100
#define  N  (2*M+1)
```

Пример 10.2. Макрос “максимум из двух величин”. Классическим примером является макроопределение

```
#define max(x,y) (x > y ? x : y)
```

По этому определению препроцессор заменит написанный в любом месте программы вызов макроса (*макровызов)* вида max(x,y) на выражение, равное максимальному из двух параметров макроса: x или y. Например, макровызов

max (a+b, 5) заменится выражением (a+b > 5 ? a+b : 5), а использующий его оператор a = 2*max(a+b,5); заменится на

```
a = 2*( a+b > 5 ? a+b : 5);
```

Макровызов выглядит как вызов функции, и многие библиотечные “функции” языка C или C++, например getchar(), на самом деле являются макросами.

В языке C/C++ имеются следующие директивы препроцессора.

оператор-препроцессора ::= вставка | замена | макроопределение | оператор-условной-трансляции | отмена-замены

вставка ::= **#include** <имя-файла> | #**include** "имя-файла"

Эта директива заменяется содержимым указанного в ней файла. Если имя файла задано в угловых скобках <>, то файл ищется в каталогах инструментальной системы (транслятора). Если это имя задано в кавычках, то файл сначала ищется в текущем каталоге, а затем уже в системных каталогах.

замена ::= #**define** имя лексема ...
макроопределение ::= #**define** имя (имя [,имя] ...) лексема ...
отмена-замены ::= #**undef** имя

Директива #**define** действует до конца файла программы или до директивы #**undef**, которая отменяет замену.

директива-условной-трансляции ::= #**if** константное-выражение |
#**ifdef** имя |
#**ifndef** имя |
#**else** |
#**elif** константное-выражение |
#**endif**

лексема ::= служебное-слово | имя | константа | операция |

(|) | [|] | { | } | ; | , | : | ?

операция ::= префиксная-операция | бинарная-операция | операция-присв | ++ | --

Директивы условной трансляции позволяют транслировать или пропускать участки текста программы в зависимости от условий, проверяемых препроцессором. Например, следующий текст является для препроцессора условным оператором **if-else**, ветви которого текст-1 и текст-2 могут состоять из любого количества строк.

#if N < 200
текст-1
#else
текст-2
#endif

Если соблюдается условие N < 200, где N – константа, определенная директивой **#fefine**, то препроцессор обработает текст-1, а текст-2 пропустит. В противном случае, наоборот, будет транслироваться только текст-2.

Директива **#endif** заканчивает оператор условной трансляции.

Директивы #**ifdef** имя и |#**ifndef** имя представляют собой разновидности директивы **#if.** Условие #**ifdef** имя означает, что заданное имя было ранее определено директивой #**define**, а директива #**ifndef**, наоборот, проверяет, что указанное имя не было определено в предыдущем тексте

Директива #**elif** константное-выражение заменяет две директивы
#**else**
#**if** константное-выражение.

Пример 10.3. Для кодировки символов и строк консорциум ведущих мировых компаний: Apple, Xerox, Compaq, HP, IBM, Microsoft и др. в 1998 г. разработал «широкий» (wide) код - новый международный стандарт 16-битовой кодировки символов Unicode.

В связи с появлением стандарта Unicode произошли изменения и в языке C/C++. Появился новый тип данных **wchar_t -** “Unicode-символ”, определяемый в заголовочном файле string.h:

typedef unsigned short wchar_t;

В библиотеку стандартных функций языка C/C++ добавлены Unicode-функции для обработки строк с именами, начинающимися с букв wcs (от слов wide character set – набор широких символов) вместо, букв str (от слова string – строка), с которых начинаются имена аналогичных старых 8-битовых функций.

Таким образом, теперь, наряду с прототипами старых 8-битовых функций strcmp, strcpy и других:

char * strcmp (**char** *, const **char** *);

char * strcpy (**char** *, const **char** *); и т. д.,

файл string.h содержит и прототипы аналогичных Unicode-функций wcscmp, wcscpy и других:

wchar_t * strcmp (**wchar_t** *, const **wchar_t** *);

wchar_t * strcpy (**wchar_t** *, const **wchar_t** *); и т. д.

Чтобы в этой ситуации писать мобильные программы двойного назначения для 16-битовой кодировки и 8-битовой кодировки символов, надо, во-первых, заменить #**include** <string.h> на #**include** <Tchar.h>, и, во-вторых, иначе оформлять работу со строками: в описании строк вместо

char t[100]; или **wchar_t** t[100];

писать TCHAR t[100];

а в именах строковых функций вместо приставок str или wcs писать _tcs.

Используя заголовочный файл Tchar.h, можно, например, писать на языке C/C++ мобильные программы для Windows, из которых с помощью замены одной строки и перетрансляции получается версия либо для операционной системы Windows 98, работающей с однобайтовой кодировкой символов и символьных строк, либо для Windows 2000, где используется двухбайтовая кодировка Unicode.

Файл Tchar.h состоит из операторов условной трансляции, определений и макросов, заменяющих тип TCHAR на тип **wchar_t** или **char**, а вызовы _tcs-функций на вызовы wcs-функций или str-функций в зависимости от того, было ранее в программе определено имя _UNICODE или нет. Приведем в качестве примера упрощенный фрагмент файла Tchar.h:

```
#ifdef      _UNICODE
typedef  wchar_t   TCHAR
#define  _tcscmp   wcscmp
#define  _tcscpy   wcscpy
            . . .
#endif

#ifndef     _UNICODE
typedef  char          TCHAR
#define  wcscmp    strcmp
#define  wcscpy    strcpy
            . . .
#endif

TCHAR * _tcscmp (TCHAR *, const TCHAR *);
TCHAR * _tcscpy (TCHAR *, const TCHAR *);

TCHAR * _tcscmp (TCHAR *, const TCHAR *);
TCHAR * _tcscpy (TCHAR *, const TCHAR *);
        . . .
```

Если, например, в исходной программе до включения файла Tchar.h определено имя _UNICODE

```
#define          _UNICODE
                 . . .
#include     <Tchar.h>
```

то приведенный выше фрагмент файла Tchar.h подставится в следующем виде:

```
typedef          wchar_t_    TCHAR
                 . . .
TCHAR * wcscmp (TCHAR *, const TCHAR *);
TCHAR * wcscpy (TCHAR *, const TCHAR *);
```

В этом случае в программе будут использоваться UNICODE-версии символьных и строковых данных и транслятор получит UNICODE-версию программы. Если же имя _UNICODE не будет определено, то в результате трансляции получится версия программы для однобайтовых символов и строк.

Упражнения и задачи

10.1. Что произойдет, если в макроопределении **#define** имя(…) … вставить пробел между именем и открывающей скобкой?

10.2. По аналогии с макроопределением max(x,y) из примера 10.2 составить определение макроса abs(X), заменяемого на X или -X в зависимости от знака параметра X.

10.3. Задан фрагмент программы для однобайтовой кодировки символов и строк. Перепишите этот фрагмент так, чтобы можно было при необходимости путем перетрансляции получать UNICODE-версию этой же программы

```
#include <string.h>
char  sim, text1[80], text2[80];          . . .
strcpy(text1,text2);  text[10]=sim;
    .
```

ПРИМЕРЫ РЕШЕНИЯ ЗАДАЧ
к видеозанятиям 1-10

Пример 1. Составить схему и трассировочную таблицу для данной программы. Входной текст имеет вид: 28 70

```
#include <stdio.h>
void main (void)
{  int  x, y;
```

```
  scanf ("%d %d", &x, &y);
  while (x > 0 && y > 0)
    if (x > y)  x = x - y;
    else  y = y - x;
  if (y == 0)  printf ("%d", x);
  else  printf ("%d", y);
}
```

Решение. Схема программы приведена на рис. 9.1.

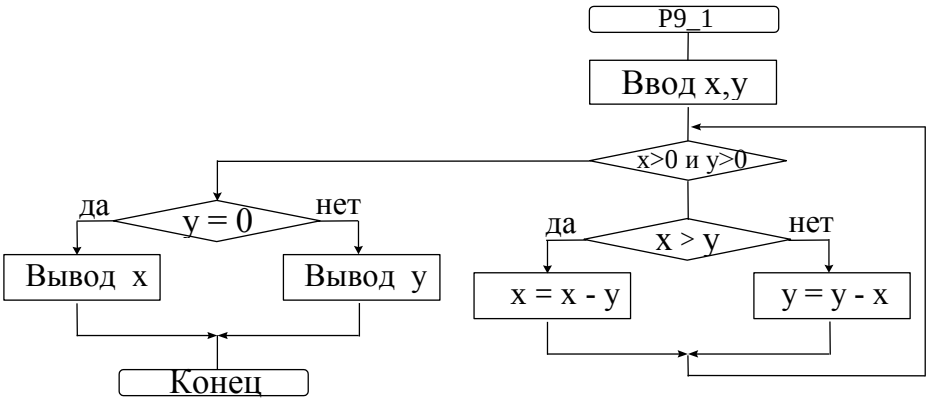

Рис. 9.1. Схема программы

Трассировочная таблица:

x =	28			14		
y =	70	42	14		0	
(x>0 && y>0) =	да	да	да	да	Нет	
(x>y) =	нет	нет	да	нет		
(y==0) =						да
Результат:						14

Пример 2. Вычислить объем памяти для данных, определенных следующим образом:

```
int  r[50];
float  x;
char t[] = "КГТУ";
```

Решение. объем = sizeof(r) + sizeof(x) + sizeof(t) =
= (50*4 + 4 + 5) байт = 209 байт = 1672 бит

Пример 3. Дана последовательность из целых чисел - количество очков каждого из 500 участников соревнований. Определить порядковые номера участников, набравших максимальное количество очков. Составить схему и С-программу.

Решение - программа 1 и схема на рис. 9.2.

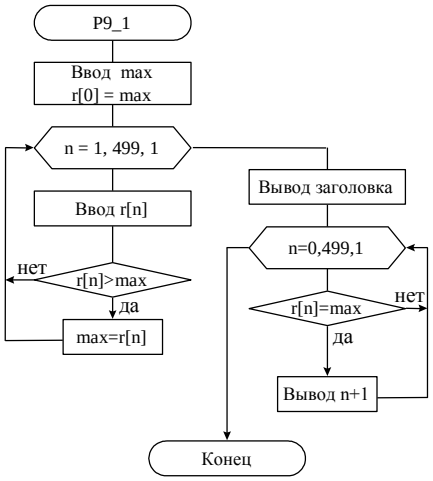

Рис. 9.2. Схема программы 9.1

```
/* Программа 1                                              */
/* Получение номеров с максимальным результатом             */
#include <stdio.h>
#define  K  500      /* Количество участников               */
void main (void)
{  int  r[K];        /* результаты участников               */
    int  max;        /* максимальный результат              */
int  n;              /* текущий номер участника             */
/* Ввод и определение максимального результата              */
   printf ("\nВведите результаты\n");
   scanf ("%d", &max); r[0] = max;
   for (n=1; n<K; n++)
   {  scanf ("%d", &r[n]);
      if (r[n] > max)  max = r[n];
   }
/* Определение номеров победителей                          */
   printf ("\nНомера победителей:\n");
   for (n=0; n<K; n++)
      if (r[n] == max) printf (" %d", n+1); /* 1..K         */
}
```

Пример 4. Составить подпрограмму подсчета количества повторений заданного символа в данной строке. Привести пример ее вызова.

Решение 1. Результат - возвращаемое значение, s - массив (см. программу 2).

```
/* Программа 2                                              */
/* Функция: количество повторений символа c в строке s      */
int kol_simv (char c, char s[])
```

```
{   int kol;        /* Значение функции                */
    int i;          /* Индекс текущего символа строки  */
    kol = 0;
    for (i = 0; s[i] != '\0'; i++)
       if (s[i] == c)  kol++;
   return kol;
}
```

Решение 2. Результат - выходной параметр kol (см. программу 3).

```
/*        Программа 3                                   */
/*Подпрограмма: kol=кол.вхождений символа с в строку s*/
void p_kol_simv (char c, char s[], int *kol)
{  int i;           /* Индекс текущего символа строки  */
   *kol = 0;
   for (i = 0; s[i] != '\0'; i++)
       if (s[i] == c)  (*kol)++;
   return;
}
```

Решение 3. Результат - выходной параметр kol, s - указатель строки, используется указатель текущего символа (см. программу 4).

```
/*                  Программа 9.4                       */
/*Подпрограмма: kol=кол.вхождений символа с в строку s */
void p_kol_simv (char c, char *s, int *kol)
{  char *i;      /* Указатель текущего символа строки  */
   *kol = 0;
   for (i = s; *i != '\0'; i++)
      if (*i == c)  (*kol)++;
   return;
}
```

Пример вызова подпрограмм kol_simv и p_kol_simv.

```
void  main (void)
{  int k;
  p_kol_simv ('a', "базар", &k);
  printf ("\nКоличество символов 'a' в слове 'базар' = %d = %d ",
           kol_simv ('a', "базар"), k);
}
```

Результаты тестирования:

Количество символов 'a' в слове "базар" = 2 = 2

Примеры реализации численных методов

Существует много численных, т. е. приближенных методов решения различных математических задач: вычисление значений функций, поиск точек их минимума или максимума, решение разнообразных уравнений и др. Выбор такого метода и его программная реализация – представляют собой,

как правило, очень непростую задачу. Реальные программы такого рода должны разрабатывать квалифицированные специалисты в этой области. Приведенные в данном разделе учебные программы служат лишь для иллюстрации используемых здесь методов и возникающих трудностей.

Ошибки вычислений

Абсолютная ошибка – это абсолютная величина разности между истинным и приближенным значением величины. *Относительной ошибкой* называют отношение абсолютной ошибки к приближенному значению величины (истинное значение обычно неизвестно и его неудобно использовать в этом определении).

Существуют три основных вида ошибок: *ошибки исходных данных*, например, из-за неточности измерений; *ошибки метода* из-за ограничений используемого метода и *ошибки округления* при выполнении операций из-за ограничения количества хранимых в компьютере значащих цифр числа.

Для уменьшения ошибок полезны следующие рекомендации.

1. Сложение и вычитание последовательности чисел лучше выполнять в порядке возрастания чисел.
2. Следует избегать вычитания почти равных чисел. Для этого желательно преобразовать соответствующую формулу.
3. Выражение вида a·(b-c) можно преобразовать к виду a·b-a·c, а выражение вида (b-c)/a можно вычислять в виде b/a – c/a. Если при этом вычитаются близкие числа, то желательно выполнять вычитание до умножения или деления, т.е. отказаться от такого преобразования.
4. Число необходимых арифметических операций желательно сводить к минимуму.

Методы вычисления элементарных функций

1. Итерационные методы. Для функции $Y = \sqrt[k]{X}$ (X>0, k>0- целое) можно применить итерационную формулу Лагранжа

$$Y_{n+1} = Y_n [(1+ 1/k) - Y_n^k / (k*X)]$$

Итерационный процесс, определяемый этой формулой, сходится, если начальное приближение $Y_0>0$ удовлетворяет условию

$$Y_0^k < (k+1)X .$$

Для итерационной формулы Ньютона

$$Y_{n+1} = 1 / k [(k-1) Y_n + X / Y_n^{k-1}]$$

в качестве начального приближения достаточно взять значение $Y_0>0$.

Итерационный процесс по каждой из этих формул заканчивается, когда два последовательных приближения удовлетворяют условию

$$|Y_{n+1} - Y_n| < \varepsilon,$$

где ε - допустимая погрешность.

2. Использование формулы Маклорена. Для многих функций удобно использовать формулу Маклорена, частный случай разложения в ряд Тейлора, например:

$Y = \sin x = x - x^3/3! + x^5/5! - ... + (-1)^{n-1} x^{2n-1}/(2n-1)! + ...$

$Y = \cos x = 1 - x^2/2! + x^4/4! - ... + (-1)^n x^{2n}/(2n)! + ...$

$Y = e^x = 1 + x + x^2/2! + ... + x^n/n! + ...$

$Y = \text{sh } x = x + x^3/3! + x^5/5! + ... + x^{2n+1}/(2n+1)! + ...$

$Y = \text{ch } x = 1 + x^2/2! + x^4/4! + ... + x^{2n}/(2n)! + ...$

$Y = \ln(1+x) = x - x^2/2 + ... + (-1)^{n-1} x^n/n + ..., \quad |x| < 1.$

Суммирование по этим формулам прекращается, когда текущее слагаемое по абсолютной величине становится меньше заданной погрешности.

Пример 5. Вычислить значения функции $F(X) = \sqrt{X}$ по итерационной формуле Ньютона для заданных A и B.

Решение. Для вычисления квадратного корня k=2, поэтому

$$Y_{n+1} = (Y_n + X / Y_n) / 2.$$

Программа 5

```
/*Программа 5. Вычисления значений функции F(X) = √X */
#include <stdio.h>
#include <math.h>
#define e 0.0001
int main()
{  int   a, b;          /* границы значений X                    */
   int   m=10;          /* количество значений X                 */
   int   y0=1;          /* начальное приближение функции         */
   float  x;            /* аргумент                              */
   float  h;            /* шаг аргумента                         */
   int    i;            /* номер значения аргумента              */
   float  y;            /* последнее приближение функции         */
   float  ypr;          /* предыдущее приближение функции        */
   float  d;            /* очередная поправка                    */
   int    n;            /* номер итерации                        */
   printf ("\nВведите границы значений X ==> ");
   scanf ("%d %d", &a, &b);
   printf ("\n    X      SQRT(X)    Y прибл.   ");
```

```
   printf (" Поправка  Кол. итераций\n\n");
   h = (float) (b – a) / m;                  /* преобразование типа */
   x = a;
   for (i=0; i<=m; i++)
   {   y=y0;
       n=0; d=1;
       while (d >= e )
       {    ypr = y;
            y = (ypr + x / ypr) / 2;
            d = fabs (y - ypr);
            n++;
       }
       printf("%6.2f %11.7f %11.7f   %9.5f %9d\n",
              x, sqrt(x), y, d, n);
       x = x + h;
   }
   return 0;
}
```

Результаты работы программы

Введите границы значений X ==> 8 9

X	SQRT(X)	Y прибл.	Поправка	Кол. итераций
8.00	2.8284271	2.8284271	0.00004	5
8.10	2.8460500	2.8460500	0.00005	5
8.20	2.8635643	2.8635643	0.00005	5
8.30	2.8809723	2.8809721	0.00005	5
8.40	2.8982756	2.8982756	0.00006	5
8.50	2.9154763	2.9154763	0.00006	5
8.60	2.9325760	2.9325759	0.00007	5
8.70	2.9495767	2.9495766	0.00007	5
8.80	2.9664799	2.9664800	0.00008	5
8.90	2.9832874	2.9832873	0.00009	5
9.00	3.0000006	3.0000007	0.00009	5

Поиск экстремума функции

Пример 6. Требуется найти координаты x и y такой точки плоскости, которая имеет минимальную сумму расстояний до трех заданных точек с координатами (a, b), (c, d), (e, f).

Решение. Искомая точка называется точкой Торичелли-Ферма для заданного треугольника. Если в треугольнике имеется угол величиной не менее 120 градусов, то искомая точка совпадает с вершиной этого угла. В

противном случае эта точка расположена таким образом, что каждая сторона треугольника видна из нее под углом 120 градусов. Найти эту точку можно простым геометрическим построением. Однако вычисление ее координат на основе геометрических соображений получается достаточно громоздким.

Будем искать точку с координатами x, y, в которой достигается минимум функции Fun(x, y), где

Fun(x, y) = Distance(x,y,a,b) + Distance(x,y,c,d) + Distance(x,y,e,f),

Distance(x, y, a, b) – расстояние между точками (x, y) и (a, b).

Функция Fun(x, y) достигает минимума в нижней точке своего графика, т. е. на дне некоторой поверхности. Эта точка единственная, что следует из геометрического смысла данной задачи.

Будем искать точку минимума методом *спирального* (*координатного*) *спуска*, начиная с некоторой точки (x0, y0). Сначала спустимся по одной координате x, изменяя ее с шагом h в сторону уменьшения значения функции, пока оно не перестанет убывать. Затем таким же образом спустимся по другой координате y. Будем повторять эти действия, каждый раз уменьшая шаг h.

Траектория точки с текущими координатами x, y напоминает спираль, описываемую и постепенно сужающуюся вокруг точки минимума. Прекратим этот процесс, когда шаг аргументов h и разность соседних значений функции станут меньше заданной погрешности EPS (прогр. 6).

Программа 6.

```
/*              Численное решение задачи нахождения
      точки Торичелли-Ферма (x, y) треугольника (a, b) (c, d) (e, f) )
 Метод: спиральный координатный спуск (поразрядное приближение)
                                                  Д.Г.Хохлов 07.10.02 */
#include <stdio.h>
#include <math.h>
#define  EPS  1E-10
typedef long double real;
real a, b, c, d, e, f, x, y, h;

/*                    Абсолютная величина x                    */
real rabs (real x)
{   return (x < 0)? –x : x;
}

/*          Расстояние между точками (x1, y1) и (x2, y2)       */
real Distance (real x1, real y1, real x2, real y2)
{    return sqrt((x1 - x2)*(x1 - x2)+ (y1 - y2)*(y1 - y2));
}
```

```
/*                    Минимизируемая функция                    */
real Fun (real x, real y)
{   return Distance(x,y,a,b) + Distance(x,y,c,d) + Distance(x,y,e,f);
}

/* Поиск минимума функции Fun спиральным координатным спуском */
void Spusk (float *x, float *y, float h)
{  real f, f1;
  f = Fun(*x, *y);
 do {
    if(Fun(*x + h, *y) > f)  h = -h;
    do {
       *x = *x + h;
       f1 = f;
       f = Fun(*x, *y);                          /* спуск по x */
    } while (f <= f1);
    if (Fun(*x, *y + h) > f)  h = -h;
    do {
       *y = *y + h;
       f1 = f;
       f = Fun(*x, *y);                          /* спуск по y */
    } while(f <= f1);
    h = h / 2;
  } while (rabs(f - f1) > EPS || rabs(h) > EPS);
}

int main()
{
    scanf("%Lf %Lf %Lf %Lf %Lf %Lf", &a, &b, &c, &d, &e, &f);
    x = (a + c + e) / 3;
    y = (b + d + f) / 3;
    h = 0.1;
    Spusk (&x, &y, h);     /* Поиск минимума Fun(x, y) спиральным спуском */
    printf("%Lf  %Lf  %Lf", x, y, Fun(x, y));
    return 0;
}
```

Динамические массивы

Массив называют *динамическим*, если его создание, т. е. выделение для него памяти, происходит во время исполнения программы.

Размер динамического массива можно вычислять в зависимости от исходных данных, если он размещается в области *динамической памяти* – так называемой *куче* (heap).

Управление *динамическим распределением памяти* в куче производится следующими стандартными функциями динамического выделения и освобождения памяти (для их использования необходима директива #**include** <stdlib.h>).

void * malloc (**unsigned** m) - возвращает указатель на начало области (блока) памяти длиной m байт. При отсутствии такого блока возвращается значение **NULL.**

void * calloc (**unsigned** n, **unsigned** m) - возвращает указатель на начало области (блока) *обнуленной* памяти, состоящего из n элементов длиной по m байт. При отсутствии такого блока возвращается значение **NULL**.

void * realloc (**void *** bl, **unsigned** m) – без изменения содержания изменяет длину ранее выделенного блока памяти с указателем bl до размера m байт. Если указатель bl равен **NULL**, считается, что память не выделялась и функция выполняется как **malloc.**

void * free (**void *** bl) – освобождает (учитывает как освободившийся) ранее выделенный блок памяти bl.

Пример 8. *Сортировка чисел*. Дано количество чисел n, за которым следуют вещественные числа $x_1, x_2, \ldots, x_n$. Требуется составить программу сортировки и вывода заданных чисел в порядке возрастания.

В программе 9 память для массива не ограничивается заранее, а выделяется динамически в зависимости от количества входных чисел.

Для сортировки чисел используем измененную подпрограмму RecQuickSort. Эта подпрограмма рассчитана на целые числа, и для сортировки вещественных чисел необходимо изменить определение переменных x, s, w следующим образом.

```
// Рекурсивная быстрая сортировка по неубыванию x[L]...x[R]
// Методы: разделение и обмен
 void RecQuickSort (float x[], int L, int R)
{  int i, j;
   float s, w;
   while (i<=j)
   //          x[L],...,x[i-1] <= s <= x[j+1],...,x[R]
   {  if(s > x[i]) i++;                  // s <= x[i]
      else if(x[j]> s) j--;              // x[j] <= s
      else { w=x[i]; x[i]=x[j]; x[j]=w;  i++;  j--; }
   }
   //       x[L], ..., x[i-1] <= s <= x[j+1], ..., x[R]
   if (j > L) RecQuickSort(x,L,j);   //Участок > 1 - сортируем
   if (i < R) RecQuickSort(x,i,R);   //Участок > 1 - сортируем
}
```

Определение функции RecQuickSort удобно разместить в отдельном файле, например, с именем qsort.c, находящемся в одной папке с программой, а в программу 9 вставить этот файл директивой #**include** “qsort.c”.

Программа 9.

```
//   Динамические массивы. Сортировка вещественного вектора
#include <stdio.h>
#include <stdlib.h>
#include "qsort.c"
int main()
{  float *x;        // вектор
   int    n;        // Количество элементов
   int    i;        // Индекс текущего элемента

   scanf("%d", &n);

   // Создание вектора (выделение памяти)
   x = (float *) malloc(n*sizeof(float)); // x = new float [n];

   // Ввод вектора
   for (i=0; i<n; ++i)
     scanf("%f", &x[i]);

   RecQuickSort (x, 0, n-1);                    // сортировка вектора


   // Вывод сортированного вектора
   for (i=0; i<n; ++i)
           printf(" %f", x[i]);

   // Уничтожение вектора  (освобождение памяти)
   free(x);                                     // delete [] x;
   return 0;
}
```

В языке C++ наряду со стандартными функциями управления памятью для создания и уничтожения динамических объектов можно использовать операции **new** – создать новый объект заданного типа, и **delete** – уничтожить ранее созданный объект. Такой вариант приведен в комментариях в программе 9.

Пример 9. *Двумерный динамический массив***.** В языке C/C++ массив представляется в памяти с помощью указателя, содержащего адрес области, в которой расположены его элементы.

В программе 9 для динамического одномерного массива этот указатель определен как **`float`** `*x`. Такое определение равносильно определению **`float`** `x[]`.

В программе 10 приведен пример обработки двумерного динамического вещественного массива. Показаны только операции по созданию и уничтожению массива, а также ввод его элементов.

Многомерный массив в языке C/C++ рассматривается как массив массивов, т. е. массив, элементами которого являются массивы меньшей размерности. Двумерный массив - матрица - рассматривается как одномерный массив строк.

Программа 10.

```
//         Динамические массивы. Обработка матрицы
#include <stdio.h>
#include <stdlib.h>

int main()
{ float   **x, y;     // Матрица и ее текущий элемент
  int m, n;           // Количество строк и столбцов
  int i;              // Индекс текущей строки
  int j;              // Индекс текущего столбца

  scanf("%d %d", &m, &n);           // Ввод размеров матрицы

  // Создание матрицы m*n (выделение памяти)
  x = (float **) malloc(m*sizeof(float *)); //x=new float* [m];
  for (i=0; i<m; ++i)
    x[i] = (float*)malloc(n*sizeof(float)); //x[i]=new float[n];

  // Ввод матрицы m по строкам
  for (i=0; i<m; ++i)
    // Ввод i-й строки
    for (j=0; j<n; ++j) {
      scanf("%f", &y);
      x[i][j] = y;
    }

  // . . .          Обработка матрицы


  // Уничтожение матрицы  m*n  (освобождение памяти)
  for (i=0; i<m; ++i)
     free(x[i]);              // delete [] x[i];
  free(x);                    // delete [] x;
  return 0;
}
```

Каждая строка матрицы является массивом и представлена указателем, содержащим адрес начала этой строки. Если элементы матрицы имеют тип **float**, то указатели на ее строки имеют тип **float** *.

Вся матрица представлена указателем, равным адресу массива указателей на ее строки. Поэтому вещественный динамический двумерный массив можно определить как указатель типа **float** **x или **float** x[][].

В программе 10 при создании матрицы из m строк и n столбцов сначала выделяется память для массива из m указателей на строки матрицы. Указателю матрицы присваивается указатель (адрес) этого массива указателей на строки.

Затем выделяется память для n элементов каждой строки и соответствующему указателю присваивается адрес начального элемента этой строки. В комментариях к программе 10 показано выделение и освобождение памяти для матрицы с помощью операций **new** и **delete** языка C++.

Использование подпрограммы в качестве параметра

Рассмотрим пример подпрограммы, параметром которой является подпрограмма. Например, параметром подпрограммы вычисления определенного интеграла может быть подынтегральная функция, программа поиска максимума или минимума функции может получать эту функцию в виде параметра и т. д.

Пример 10. Составить подпрограмму поиска корня уравнения вида F(x) = 0 на отрезке [a, b] с погрешностью, не превышающей заданного числа eps. Функция F(x) задается подпрограмме в виде параметра.

Если функция F(x) непрерывна на отрезке [a, b] и в его концах имеет значения разных знаков, то можно использовать метод дихотомии.

Чтобы использовать подпрограмму для решения разных уравнений такого вида, реализуем алгоритм этого метода в виде функции Koren, одним из параметров которой является уравнение, т. е. указатель на функцию F(x) – левую часть уравнения. Другие параметры задают границы отрезка A и B, а также допустимую погрешность eps. Значением фунции является найденный корень уравнения.

Для этого определим в программе тип FUN как указатель на вещественную функцию от вещественного аргумента:

```
typedef float(*FUN)(float);
```

Программа 11 решает два уравнения: $\cos(x) - x = 0$ и $x^3 - 2 = 0$. Левые части этих уравнений заданы в программе в виде функций:

$$F1(x) = \cos(x) - x \quad \text{и} \quad F2(x) = x^3 - 2.$$

Очевидно, что корень первого уравнения лежит между 0 и 1, поскольку $\cos(0) - 0 = 1 > 0$, а $\cos(1) - 1 < 0$. Корень второго уравнения – корень кубический из 2 – находится между 1 и 2.

Программа 11.

```
/*     Решение уравнений:  cos(x)= x   и   x*x*x = 2      */

#include <stdio.h>
#include <math.h>

typedef float(*FUN)(float); //Указатель на ф-ю float f(float)

/*       Функции - левые части уравнения вида F(x) = 0     */
float F1(float x)
{  return cos(x)-x;
}
```

```
float F2(float x)
{  return x*x*x - 2;
}

/*   Решение уравнения F(x)=0 на [a; b] с погрешностью eps */
/*   методом деления пополам                                */
float Koren (FUN f, float a, float b, float eps)
{  float l, p, s;
   l=a; p=b; s=(l+p)/2;
   while (p-l > 2*eps)
   { /* Разбиение отрезка пополам */
     if(f(l)*f(s) <= 0)  /* в левой половине меняется знак */
        p = s;
     else
        l = s;
     s=(l+p)/2;
   }
   return s;
}

void main(void)
{
   printf ("\nКорень F1 = %f",  Koren(F1, 0, 1, 1E-6));
   printf ("\nКорень F2 = %f",  Koren(F2, 1, 2, 1E-6));
}
```

В результате выполнения программы 11 получим:

Корень F1 = 0.739085
Корень F2 = 1.259921