Рассмотрим некоторые соглашения , принятые для языка Си относительно:

1- указателей (см. также л. № 3-4);

2- функций (см. л. № 4).

В языке Си относительно указателей приняты **соглашения**, например, такие:

- **всякий указатель имеет базовый тип, который совпадает с типом элемента данных, на который может ссылаться этот указатель.**

Примеры:

```
1.  int *ptr;/* здесь указатель ptr имеет базовый тип
                int */
2. double *str; /* здесь указатель str имеет базовый
                   тип double */
```

- **операция, символом которой является звездочка перед именем указателя, носит название операции *косвенной адресации* и служит для доступа к значению, расположенному по заданному адресу.**

- **операцией косвенной адресации (или получения адреса) является также операция, которая обозначается символом *амперсанда* (&) перед именем простой переменной или элемента массива. Одноместные операции * и & имеют такой же высокий приоритет, как и другие унарные операции.**

Примеры:

```
1.  ptr=&x; /*здесь указатель  ptr получает адрес
              переменной x */
2. ptr=&a[5]; /* здесь указатель  ptr получает адрес 6-
                 го элемента массива */
```

Пример 1

При выполнении следующего фрагмента программы сравнение в операторе **if** всегда будет истинно, поскольку значение указателя **ptr** совпадает с адресом переменной **x**:

```
…
  int x;
    scanf("%d", &x);
    int *ptr = &x;
    if (x == *ptr)
       printf(«Результат сравнения есть true");
       else printf(«Результат сравнения есть false");
…
```

Пример 2

//определение функции вычисления нормы произвольного вектора

```
double Norma  (int n, double x[ ])
 {
  int i;
  double s=0;
  for (i=0; i<n; i++) s+=x[i]*x[i];
  return sgrt (s);
 }
```

```
double Norma (int n, double *x)
.
.
.
//норма i-й строки
 …  <<Norma (10, A[i])
```

Применение указателей для формирования массивов с переменными размерами

Используются также функции библиотек (файлы **alloc.h** и **stdlib.h**)

функции	Прототип и краткое описание
malloc	**void *malloc(unsigned s);** /* возвращает указатель на начало области (блока) динамической памяти длиной в **s** байт. При неудачном завершении возвращает значение **NULL** */
calloc	**void *calloc(unsigned n, unsigned m);** /*возвращает указатель на начало области (блока) обнуленной динамической памяти, выделенной для размещения **n** элементов по **m** байт каждый. При неудачном завершении возвращает значение **NULL** */
realloc	**void *realloc(void *bl, unsigned ns);** /* изменяет размер блока ранее выделенной динамической памяти до размера **ns** байт, **bl** – адрес начала изменяемого блока. Если **bl** равен **NULL** (память не выделялась), то функция выполняется как **malloc** */
free	**void *free (void *bl);** /* освобождает ранее выделенный участок (блок) динамической памяти, адрес первого байта которого равен значению **bl** */

*Подробнее см. учебник **Подбельского и Фомина***

/* ввести и напечатать в обратном порядке набор вещественных чисел, количество
которых заранее не определено, а вводится до начала ввода самих числовых
значений */

Приведение к типу значения, возвращаемого ф-ей **malloc**

```
#include <stdio.h>
#include <stdlib.h>
void main ()
 {
  float *t; //указатель для выделенного блока памяти
  int i, n; //n- число элементов (вводимых чисел); i- параметр цикла
  printf("\nn="); scanf("%d", &n);
  t=(float *)malloc(n*sizeof(float)); /* динамическое выделение памяти требует
   обращения к функции malloc и явного указания необходимого числа байт, что
   требует использования ссылки на операцию sizeof . Здесь t- указатель на начало
   области, выделенной для размещения n вводимых чисел */
  for (i=0; i<n; i++) { printf("x[%d]=", i); scanf("%f", &t[i]); }
   //далее печать результатов
  for(i=n-1; i>=0; i--)
   {
    if (i%2==0) printf("\n");
    printf ("\tx[%d]=&f", I, t[i]);
```

Вид печати компилятором BC++V3.1:

n=4
X[0]=10
X[1]=20
X[2]=30
X[3]=40

x[3]=40.000000 x[2]=30.000000
x[1]=20.000000 x[0]=10.000000

Об определении динамических объектов

В отличие от заранее определенных объектов, число динамических объектов не фиксировано тем, что записано в тексте программы. По желанию динамические объекты могут создаваться и уничтожаться в процессе выполнения программы. Динамические объекты не имеют имен, а ссылка на них выполняется с помощью указателей.

О доступе к динамическим объектам

В следующем примере ***p=55;** присваивание значения объекту, ссылка на который задана указателем **p,** выполняется с помощью имени указателя

***p**. Одно и то же значение может быть присвоено более чем одной переменной – указателю. Таким образом можно ссылаться на динамический

объект с помощью более одного указателя. В таком случае про объект говорят, что он имеет **псевдоимена .**

Массивы указателей могут состоять из указателей

Правила определения:

тип *имя_массива [размер];
тип *имя_массива [ ]=инициализатор;
тип *имя_массива [размер]=инициализатор;

Примеры определения:

int *ptr[5];//массив указателей ~ **int x[5]**//просто массив

int *p[ ] = { &data[0], &data[3], &data[2] };/* значением каждого элемента может быть адрес объекта типа **int */**
//в фигурных скобках список из 3-х элементов, которые
//инициализированы адресами конкретных элементов массива **data**

/*пример сортировки с использованием массива указателей (из упомянутого учебника). Массив без перестановки его элементов одновременно упорядочивается и по возрастанию, и по убыванию */

```
#include <stdio.h>
#include <conio.h>
#define B 6
void main ()
  {
   float array []={5.0, 2.0, 3.0, 1.0, 6.0, 4.0};
   float *pmin[6], *pmax[6], *e;
   int i, j;
    for (i=0; i<B; i++) pmin[i]=pmax[i]=&array[i];
    for (i=0; i<B-1; i++)
    for (j=i+1; j<B; j++)
     {
       if (*pmin[i]<*pmin[j])  { e=pmin[i]; pmin[i]=pmin[j]; pmin[j]=e; }
       if (*pmax[i]>*pmax[j]) { e=pmax[i]; pmax[i]=pmax[j]; pmax[j]=e; }
     }
    printf ("\n По убыванию:      \n");
    for (i=0; i<B; i++)  printf ("\t%5.3f", *pmin[i]);
    printf ("\n По возрастанию:      \n");
    for (i=0; i<B; i++)  printf ("\t%5.3f", *pmax[i]);
    getch();
   }
```

Указатели на функцию

Указатель на функцию – это переменная, содержащая адрес в памяти, по которому расположена функция. Имя функции – это адрес начала программного кода функции. Указатель на функцию может быть передан другой функции в качестве аргумента , может возвращаться функцией, сохраняться в массивах и присваиваться другим указателям на функции.

Указатель на функцию как переменная вводится отдельно от определения и прототипа какой-либо функции. Для этих целей используется описание:

тип (* имя указателя) (спецификация_параметров);

здесь ***тип-*** определяет тип возвращаемого функцией значения; ***имя_указателя*** – идентификатор; ***спецификация_параметров*** – определяет состав и типы параметров функции.

Пример: ***int (* kluch) (int, float); /** определяет указатель-переменную с именем ***kluch*** на функции с параметрами типа ***int*** и ***float ,*** возвращающие значения типа ***int */***

/* иллюстрация разных способов вызова функций с использованием указателей –констант (имен функций) и указателей переменных (неконстантных указателей на функции */

```
#include <stdio.h>
void f1 (void) { printf ("выполняется f1()"); }
void f2 (void) { printf ("выполняется f2()"); }
  void main ()
{
  void ( *kluch )  (void);// kluch- указатель-переменная на функцию
   f2 ();//явный вызов функции
   kluch=f2;//настройка указателя на f2()
   (*kluch) ();//вызов f2 () по ее адресу с разыменованием указателя
   kluch=f1;// настройка указателя на f1()
   (*kluch) ();//вызов f1 () по ее адресу с разыменованием указателя
   kluch ( );//вызов f1() без явного разыменования указателя
}
```

Результат выполнения программы:

выполняется ***f2()***
выполняется ***f2()***
выполняется ***f1()***
выполняется ***f1()***

Рекурсивные вызовы функций

- Всякая функция в языке Си имеет реентерабельный (повторно входимый) код, что позволяет ей обращаться к самой себе непосредственно или через другие функции

- Такие обращения называются **рекурсивными вызовами** или **рекурсией**

- При каждом очередном рекурсивном вызове создается новая копия параметров функции, а также определенных в ее теле автоматических и регистровых переменных. Внешние и статические переменные, имеющие глобальное время существования, сохраняют при этом свои прежние значения и размещение памяти

- **Несмотря на то, что ни стандарт языка Си, ни компилятор формально не налагают никакого ограничения на количество рекурсивных обращений, тем не менее оно практически всегда существует для любых типов компьютеров, ибо каждый новый вызов требует дополнительной памяти из ресурса программного стека**

- **Если количество вызовов излишне велико, возникает переполнение сегмента стека и операционная система уже не может создать очередного экземпляра локальных объектов функции, что ведет, как правило, к аварийному завершению программы**

- В качестве примера реализации рекурсивного алгоритма рассмотрим функцию **printd()** , печатающую целое число в виде последовательности символов ASCII (т.е. цифр, образующих запись этого числа):

```
void printd(int num)
{ int i;
  if (num < 0) { putchar('-'); num = -num; }
  if ((i = num/10) != 0) printd(i);
  putchar(num % 10 + '0') ;
}
```

- Если значение переменной ***value*** равно ***123***, то в случае вызова

```
void printd(value);
```

- эта функция дважды обратится сама к себе для печати цифр заданного числа

- Классическим примером написания рекурсивной функции является вычисление факториала целого числа. Разумеется, эту задачу легко решить при помощи обычного цикла, но на этом простом примере наглядно видна идея рекурсивного алгоритма

- Текст такой функции достаточно прост:

```
/* Рекурсивное вычисление n! */
int fact(int n)
 { if (n==0) return (1);
     else    return(n*fact(n-1));
 }
```

- Если обратиться к этой функции, например, так:

```
int m;
...
m = fact(5);//продолжение ниже
```

- то, прежде чем получить значение 5!, функция должна вызвать самое себя как ***fact(4) ,*** та, в свою очередь, вызывает ***fact(3)*** . Так будет продолжаться до вызова ***fact(0)*** . Лишь после этого вызова будет получено конкретное число (единица). Затем все пойдет в обратном порядке, и в конце концов мы получим результат от обращения ***fact(5) : 120***

Порядок возвратов	**Возвращаемое значение**
return(1)	***1***
return(1*0!)	***1***
return(2*1!)	***2***
return(3*2!)	***6***
return(4*3!)	***24***
return(5*4!)	***120***