Лекция 10

Одномерные массивы.
Поиск в массиве.
Динамические массивы. Управление памятью.
Указатели и массивы. Адресная арифметика.
Символьные массивы.

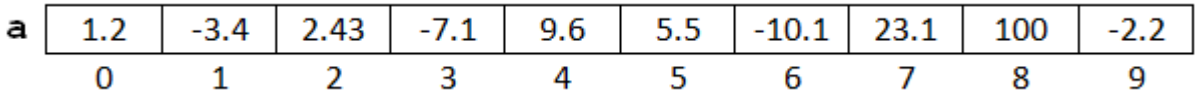
Массивы

Одномерный массив – это набор объектов одинакового типа, расположенных в *последовательной* группе ячеек памяти, доступ к которым осуществляется *по индексу*.

```
тип имя_массива[размер_массива];
```

Описание массива в программе отличается от описания простой переменной наличием после имени квадратных скобок, в которых задается количество элементов массива:

```
float a[10];
```
// описание массива из 10 вещественных чисел



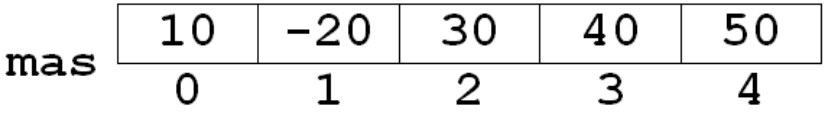
Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы, что и для простых переменных.

Память, необходимая для размещения *статического* массива, выделяется на этапе компиляции, поэтому размерность может быть задана только целой положительной константой или константным выражением.

Массивы

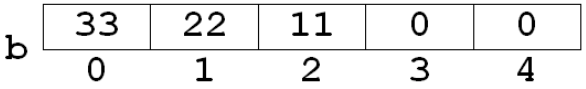
Инициализирующие значения для массивов записываются в фигурных скобках.

```
int mas[5]={10, -20, 30, 40, 50};
```

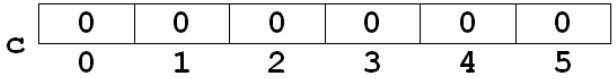


Значения элементам присваиваются по порядку. Если элементов в массиве больше, чем инициализаторов, элементы, для которых значения не указаны, обнуляются:

```
int b[5] = {33, 22, 11};
 // b[0]=33, b[1]=22, b[2]=11, b[3]=0, b[4]=0
```

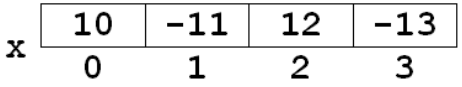



```
int c[6]={0};
// c[0]=0, c[1]=0, c[2]=0, c[3]=0, c[4]=0, c[5]=0
```



Если при описании массива не указана размерность, должен присутствовать инициализатор, в этом случае компилятор выделит память по количеству инициализирующих значений.

```
int x[]={10, -11, 12, -13};
```



Массивы

При обращении к элементам массива автоматический контроль выхода индекса за границу массива не производится, что может привести к ошибкам.

```
int b[5] = {33, 22, 11};
 // b[10]=33;
```

Пример инициализации и печати массива

```
#include <stdio.h>
const int N=10;
int x[N]= {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};
int a[N]= {0, 1, 2, 3};
int z[N]= {0}, y[N];

int main() {
 int i;
 printf("%d", x[0]); //обращение к первому элементу
 x[0]=-10;
  printf("\n massiv:\n");//обращение к элементу массива
  for (i = 0; i < N; ++i)
        printf("%d ", x[i]);
  printf("\n y-> \n");
  for (i=0; i<N; i++)
       scanf("%d",&y[i]);
   printf("\n massiv: ");
   for (i=0; i<N; i++)
       printf("%5d",y[i]);
   }
```

```
32
 massiv:
-10 27 64 18 95 14 90 70 60 37
 y->
1 0 2 -3 -45 6 7 8 -9 10

 massiv:     1    0    2   -3  -45    6    7    8   -9   10
--------------------------------
```

Поиск минимального элемента

```
int search_min (int x[N]){
int min=x[0]; //значение минимального элемента
    for(int i=1;i<N;i++)
    if (x[i]<min)
        min=x[i];
 return min; //
}
int search_imin (int x[N]) {
    int imin=0; // индекс минимума
    for(int i=0; i<N; i++)
          if (x[i]<x[imin])
                imin=i;
    return imin;
}
int main() {
int mas[N]= {1,22,3,4,-5,6,7,28,9,10};
    int t=search_min(mas);  printf("min=%d\n",t);
        t=search_imin(mas);
    printf("imin=%d min=%d",t, mas[t]);
}
```

```
min=-5
imin=4 min=-5
```

Поиск элемента в массиве

Поиск – это проверка массива на наличие какого-то значения. Существует много алгоритмов для решения этой задачи.

Самый простой для реализации алгоритм поиска (но и самый затратный по времени) – ***линейный поиск***.

```
const int N=10;
double mas[N];

int fl=0; //лог. переменная флажок
double x; // искомый элемент – ключ
    printf("x->");
    scanf("%lf",&x); //чтение ключа
    for (int i = 0; i < N; ++i) // поиск
          if (mas[i]==x)
                fl=1;
```

Если элемент найден значение **fl** будет равно 1, иначе 0

Линейный поиск и поиск с «барьером»

Линейный поиск (поиск в неупорядоченных массивах) реализуется путем перебора всех элементов массива.

```
doudle mas[N];
```

Цель поиска – найти индекс **i**, при котором **mas[i]=x**, где **x** – ключ поиска. Условие окончания линейного поиска: **mas[i]=x** или **i=N.**

```
int i=0;
while (i<N && mas[i]!=x) i++;
if (mas[i]==x) printf("yes");
```

Линейный поиск с «барьером»

В этом случае в конец массива записывают ключ **x** – **барьер. Это позволяет отказаться от проверки i<N:**

```
  doudle mas[N+1];
//…
  mas[N]=x;
  i=0;
  while (mas[i]!=x) i++;
  if (i==N) printf("Элемент не найден ");
```

Поиск в упорядоченных массивах - binsearch

(Поиск делением пополам) – исследуется не больше, чем $\log_2 N$ элементов, что намного эффективнее по сравнению с линейным поиском.

```
/* binsearch: найти x в v[0] <= v[1] <= … <= v[n-1] */
int binsearch(double x, double v[], int n) {
    int low, high, mid;
    low = 0;
    high = n - 1 ;
    while (low <= high) {
          mid = (low + high) / 2;
          if (x < v[mid])
                high = mid - 1;
          else if (x > v[mid])
                low = mid + 1 ;
          else  return mid; /* совпадение найдено */
    }
    return -1;   /* совпадения нет */
}
```

Вызов:

```
int t=binsearch(x,mas,N);
```

Динамические массивы

Динамические массивы создают с помощью операции **new**, при этом необходимо указать тип и размерность, например:

```
int n = 100:
float *p = new float [n];
```

В этой строке создается переменная-указатель на **float**, в динамической памяти отводится непрерывная область, достаточная для размещения 100 элементов вещественного типа, и адрес ее начала записывается в указатель p.

Динамические массивы ***нельзя*** при создании ***инициализировать***, и они ***не обнуляются***.

Преимущество динамических массивов состоит в том, что ***размерность может быть переменной***, то есть ***объем памяти, выделяемой под динамический массив, определяется на этапе выполнения программы***.

Динамические массивы

Доступ к элементам динамического массива осуществляется точно так же, как к статическим, например, к элементу номер 5 приведенного выше массива можно обратиться как **`p[5]`** или **`*(p+5)`**.

Альтернативный способ создания динамического массива — использование функции **`malloc`** библиотеки C:

```
int n = 100;
float *q = (float *) malloc(n * sizeof(float));
```

Операция преобразования типа, записанная перед обращением к функции **`malloc`**, требуется потому, что функция возвращает значение указателя типа **`void*`**, а инициализируется указатель на **`float`**.

Динамические массивы

Память, зарезервированная под динамический массив с помощью **new []**, должна освобождаться оператором **delete []**, а память, выделенная функцией **malloc** – посредством функции **free**, например:

```
delete [] p;
free (q);
```

При несоответствии способов выделения и освобождения памяти результат не определен.

Размерность массива в операции **delete []** не указывается, но квадратные скобки обязательны.

Частая ошибка: применение **delete** (без **[]**) к памяти, выделенной **new []** (с **[]**). Компилятор не предупреждает, программа на первый взгляд даже работает, но происходит утечка памяти.

Управление памятью

Управление памятью выполняется функциями **malloc** и **calloc.**

Функция

```
void *malloc(size_t n)
```

возвращает указатель на **n байт** неинициализированной памяти или **NULL**, если запрос удовлетворить нельзя.

Функция

```
void *calloc(size_t n, size_t size)
```

возвращает указатель на область, достаточную для хранения **массива** из **n** объектов указанного размера (**size**), или **NULL**, если запрос не удается удовлетворить. Выделенная память **обнуляется.**

К указателю на выделенную область памяти должна быть применена **операция приведения к соответствующему типу**:

```
int *ip;
ip = (int *) calloc(n, sizeof(int));
```

Функция **free(p)** освобождает область памяти, на которую указывает **p**.

Управление памятью

```
#include <stdlib.h>
#include <stdio.h>
main()
{
    int n;
      printf("n->");
      scanf("%d",&n);
    int *a=new int[n];
    double *b=(double*)malloc(n*sizeof(double));
    // обработка массивов a и b
    // ...
    delete []a;
    free(b);
    return 0;
}
```

Пример использования функций при работе с массивом

```
#include<iostream.h>
const int N=10;

float *read_mas(float *mas) // заполнение
{   for( int i=0; i<N; i++) cin>>mas[i];
    return mas;
}

void write_mas(float *mas) // печать
{ for(int i=0; i<N; i++)
     cout<<" "<<mas[i]; cout<<endl;
return;
}

main()
{ float *mas_float= new float[N];   //объявление и выделение памяти
  mas_float=read_mas(mas_float);  //  вызов функции заполнения
  write_mas(mas_float);           //  вызов функции печати
 delete [] mas_float;             //освобождение памяти
}
```

Указатели и массивы

В Си существует связь между указателями и массивами.

Объявление **`int a[10]`** определяет массив **`a`** из 10 элементов

Если **`pa`** есть указатель на **`int`**, т. е. объявлен как **`int *pa`**, то в результате присваивания

```
pa = &a[0];
```

`pa` будет указывать на нулевой элемент **`a`**, иначе говоря, **`pa`** будет содержать адрес элемента **`a[0].`**

Теперь присваивание

```
int  x = *pa;
```

будет копировать содержимое **`a[0]`** в **`x`**.

Если **`pa`** указывает на некоторый элемент массива, то **`pa+1`** по определению указывает на следующий элемент, **`pa+i`** – на i-й элемент после **`pa`**, а **`pa-i`** – на i-й элемент перед **`pa`**. Таким образом, если **`pa`** указывает на **`a[0],`** то **`*(pa+1)`** есть содержимое **`a[1], *(pa+i)`** – содержимое **`a[i].`**

Указатели и массивы

Поскольку **имя массива указывает на первый элемент**, то присваивание **`pa=&a[0]`** можно также записать в следующем виде:

`pa=a;`

Кроме этого, **`a[i]`** эквивалентно **`*(a+i)`**. Иными словами, элемент массива можно изображать как виде указателя со смещением, так и в виде имени массива с индексом.

Между именем массива и указателем на массив существует одно различие.

Указатель – это переменная, поэтому можно написать **`pa=a`** или **`pa++`**. Но **имя массива не является переменной**, и записи вроде **`a=pa`** или **`a++`** **не допускаются.**

Если имя массива передается функции, то последняя получает в качестве аргумента адрес его начального элемента. Внутри вызываемой функции этот ***аргумент является локальной переменной, содержащей адрес.***

Указатели и массивы

```
/* strlen: возвращает длину строки */
int strlen(char *s) {
    int n;
    for (n = 0; *s != '\0' ; s++)
          n++;
    return n;
}
```

Формальные параметры **`char s[]`** и **`char *s`** в определении функции эквивалентны. Предпочтение отдается последнему варианту, поскольку он более явно сообщает, что **s** есть указатель.

Адресная арифметика

1. Указатели и целые не являются взаимозаменяемыми объектами. Константа нуль – единственное исключение из этого правила: ее можно присвоить указателю, и указатель можно сравнить с нулевой константой.

Чтобы показать, что нуль – это специальное значение для указателя, вместо цифры нуль, как правило, записывают **`NULL`** – константу, определенную в файле **`<stdio.h>`.**

2. Указатели можно сравнивать при соблюдении следующего правила:

если **`p`** и **`q`** указывают на элементы одного массива, то к ним можно применять операторы отношения **==, ! =,** <, >= и т. д.

3. Указатели и целые можно складывать и вычитать. Конструкция **`p + n`** означает адрес объекта, занимающего **n-е** место после объекта, на который указывает **`p`**.

Адресная арифметика

4. Допускается также вычитание указателей. Например, если **p** и **q** указывают на элементы одного массива и **p<q**, то **q-p+1** есть число элементов от **p** до **q** включительно.

```
/* strlen: возвращает длину строки s */
int strlen(char *s) {
    char *p=s;
          while (*p != '\0' )
                p++;
          return p - s;
}
```

Символьные указатели

Указатель на строку символов можно объявить следующим образом

```
char *pmessage;
```

Присваивание

```
pmessage = "now is the time";
```

поместит в **pmessage** указатель на символьный массив, при этом сама строка *не* копируется, копируется лишь указатель на нее. Существует важное различие между следующими определениями:

```
char amessage[] = "now is the time"; /* массив */
char *pmessage = "now is the time"; /* указатель */
```

amessage – это массив, имеющий такой объем, что в нем как раз помещается указанная последовательность символов и `'\0'`. Отдельные символы внутри массива могут изменяться, но **amessage** **всегда указывает на одно и то же место памяти.**

Символьные указатели

pmessage есть указатель, указывающий на строковую константу. А значение указателя можно изменить, и тогда последний будет указывать на что-либо другое.

```
/* strcpy: копирует t в s; вариант с массивом*/
void strcpy(char *s, char *t) {
    int i;
    i = 0;
    while ((s[i] = t[i]) != '\0' )
          i++; }

/* strcpy: копирует t в s: версия 1 с указателями */
void strcpy(char *s, char *t) {
    while ((*s = *t) != '\0' ) {
          s++;
          t++;
    }
}
```

Символьные указатели

```
/* strcpy: вариант неплохого программиста*/
void strcpy(char *s, char *t) {
   while ((*s++ = *t++) != '\0') ;
}

/* strcpy: вариант опытного программиста */
void strcpy(char *s, char *t) {
   while (*s++ = *t++);
}
```

Определим функцию **strcmp(s,t).** Она сравнивает символы строк **s** и **t** и возвращает отрицательное, нулевое или положительное значение, если строка **s** соответственно лексикографически меньше, равна или больше, чем строка **t**. Результат получается вычитанием первых несовпадающих символов из **s** и **t**.

Символьные указатели

```
//strcmp: выдает < 0 при s<t, 0 при s==t, > 0 при s>t
int strcmp(char *s, char *t) {
   int i;
   for (i = 0; s[i] == t[i]; i++)
   if (s[i] == '\0')
      return 0;
   return s[i] - t[i];
}
// strcmp: вариант с указателями
int strcmp(char *s, char *t) {
   for ( ; *s == *t; s++, t++)
   if (*s == '\0')
      return 0;
return *s - *t;
}
```

<string.h> - содержит функции для обработки строк