

# Глава 5. Использование динамической памяти

## 5.1 Адресация оперативной памяти.

Минимальная адресуемая единица памяти – *байт*.



Адресация по схеме «база+смещение»:

$$A_{\text{ф}} = A_{\text{б}} + A_{\text{см}},$$

где  $A_{\text{б}}$  – адрес базы – адрес, относительно которого считают остальные адреса;

$A_{\text{см}}$  – смещение – расстояние от базового адреса до физического.

*Указатель* – тип данных, используемый для хранения *смещений*.

В памяти занимает 4 байта, адресует сегмент размером  $V = 2^{32} = 4$  Гб.

Базовый адрес = адрес сегмента.

## 5.2 Указатели и операции над ними

**[<И1>][<Тип данных>][<Тип>] [И2]\*<Имя>[=<Значение>];**

Где :

**<И1>** - признак изменчивости содержимого по адресу указателя. Задается ключевым словом **const**. При этом значение содержимого памяти, которую адресует указатель, нельзя менять. Может отсутствовать.

**<И2>** - признак изменчивости указателя. Задается ключевым словом **const**. При этом значение самого указателя нельзя менять.

Может отсутствовать.

**<Тип данных>** - тип данных, адресуемых указателем. Любой тип, определенный в C++, в том числе **void**.

**<Тип >** - тип указателя. Определяется моделью памяти. Может быть **far** или **near**. Если тип указателя не указан, принимается **near**.

# Примеры определения указателей

1) **short a, \*ptrs =&a;**

Указатель можно  
инициализировать адресом  
реальной переменной



2) **const short \*ptrs;**

Неизменяемое значение:  
можно **ptrs = &b;**; нельзя  
**\*ptrs=10;**

3) **short \*const ptrs=&a;**

Неизменяемый указатель  
можно **\*ptrs=10;**; нельзя **ptrs = &b;**

## 5.2.1 Типизированные и нетипизированные указатели

Различают указатели:

- типизированные – адресующие данные конкретного типа;
- нетипизированные – не связанные с данными определенного типа.

**Объявление типизированного указателя:**

```
int *b, *c ;
```

```
float *s , double *f;
```

```
long double *l;
```

Все указатели несут в себе сведения о размере памяти, адресуемой этим указателем

**Объявление нетипизированного указателя:**

```
void * <имя>;
```

Этот указатель создан как бы «на все случаи жизни».

Он отличается от других отсутствием сведений о размере соответствующего участка памяти. Поэтому его легко связывать с указателями других типов.

# Нулевой указатель

В C++ определена адресная константа **NULL**;

Эта константа определяет адрес, который никуда не указывает или «нулевой указатель».

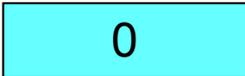
Его можно присвоить указателю любого типа.

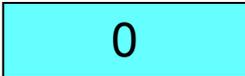
Например:

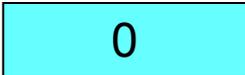
```
int *pi=NULL;
```

```
float *pf=NULL;
```

```
void *b=NULL;
```

pi 

pf 

b 

Кроме того, эту константу можно использовать в операциях сравнения при проверке логических адресных выражениях.

```
while (pi!=NULL) { ...};
```

## 5.2.2 Операции над указателями

### 1. Присваивание.

Допускается присваивать указателю значение другого указателя того же типа или нулевого указателя.

Пример:

```
int *p1,*p2;
```

```
float *p3,*p4;
```

```
void *p;...
```

```
{допустимые операции}
```

```
p1=p2;    p4=p3;    p1=NULL;  p=NULL;  ...
```

```
{недопустимые операции}
```

```
p3=p2;    p1=p3;
```

Однако, при необходимости выполнить операцию присваивания, можно использовать явное переопределение типа, для приведения указателя одного типа к другому.

```
p3=(float*)p2;
```

```
p2=(int*)p3;
```

```
p1=(int*)p;
```

Явное переопределение  
типа указателя

# Операции над указателями (2)

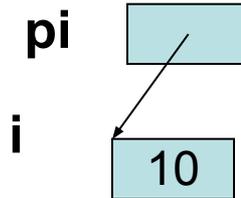
## 2. Получение адреса (&).

Результат операции – адрес некоторой области памяти, который можно присвоить указателю.

Это можно сделать:

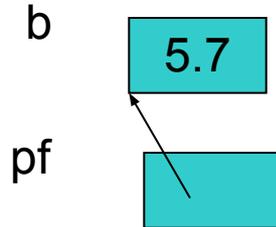
a) При помощи операции присваивания:

```
int *pi,i=10;  
... pi=&i;
```



b) Во время инициализации указателя при его определении:

```
float b=5.7;  
float *pf=&b;
```



# Операции над указателями (3)

3. *Доступ к данным по указателю (операция разыменовывания)*. Полученное значение имеет тип, совпадающий с базовым типом данных указателя.

*Нетипизированные указатели разыменовывать нельзя.*

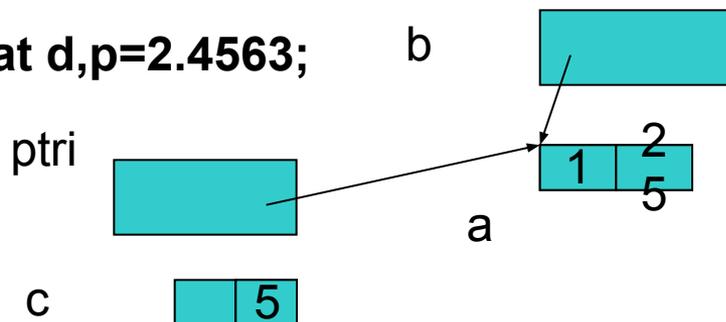
Примеры:

`short c, a=5,*ptri=&a; float d,p=2.4563;`

`void *b=&a;`

1) `c=*ptri;`

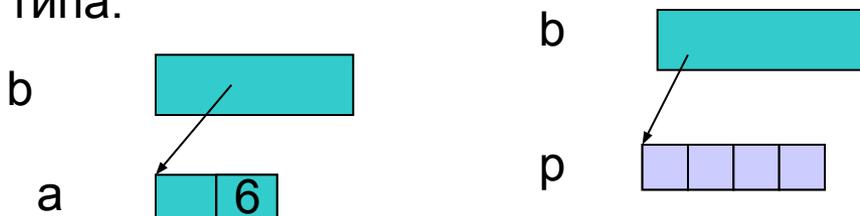
2) `*ptri=125;`



При необходимости разыменовывать нетипизированный указатель требуется выполнить явное преобразование типа.

3) `*b=6; ⇒ *(int*)b=6;`

4) `b=&p;`



`d=*b ⇒ d=(float *)b`

Явное переопределение типа указателя

# Операции над указателями (4)

## 4. Операции отношения:

проверка равенства (==) и неравенства (!=).

Примеры:

```
int sign = (p1 == p2) ;
```

```
if (p1!=NULL) {.... }
```

## 5. Арифметические операции.

В C++ над указателями разрешены операции:

- сложение и вычитание (аддитивные операции)
- инкремент или автоувеличение (++)
- декремент или автоуменьшение (--)

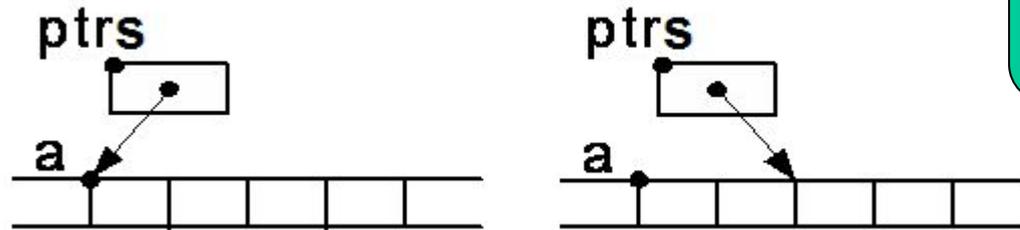
В силу особенностей выполнения арифметических операций над указателями, совокупность этих операций получила название **Адресной арифметики**

## 5.2.3 Адресная арифметика

$\langle \text{Указатель} \rangle + n \Leftrightarrow \langle \text{Адрес} \rangle + n * \text{sizeof}(\langle \text{Тип данных} \rangle)$

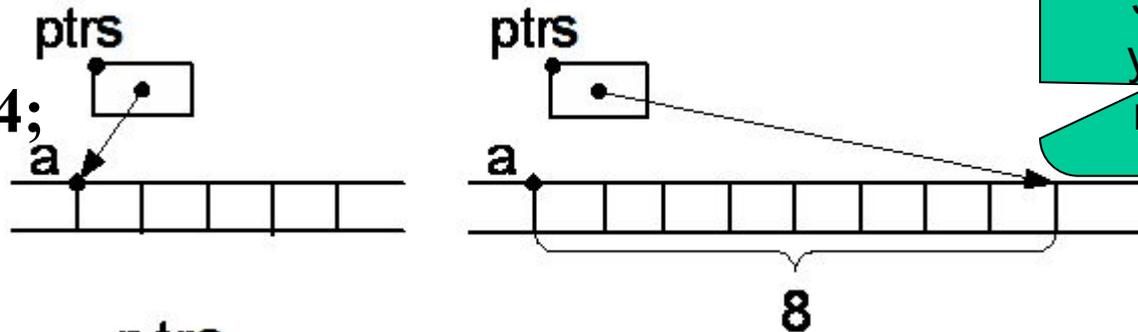
Пример: `short a, *ptrs = &a;`

1) `ptrs++;`



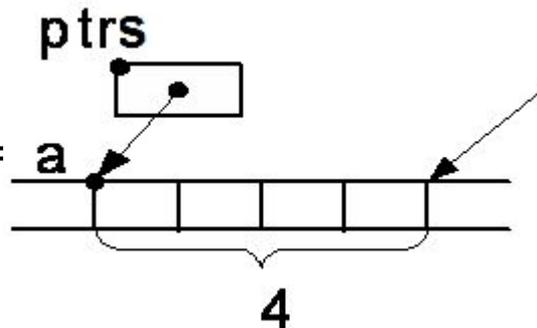
Значение указателя меняется

2) `ptrs+=4;`



Значение указателя меняется

3) `*(ptrs+2)=`



Значение указателя не меняется!!!

# Адресная арифметика (2)

Особенности результатов выполнения операций адресной арифметики связано с реализацией языка C++.

В C++ принят обратный порядок размещения объектов в памяти.

Это объясняется особенностями работы компилятора. При разборе текста, компилятор распознает и размещает в стек имена всех объектов, которые необходимо разместить в памяти.

На этапе распределения памяти имена объектов выбираются из стека и им отводятся смежные участки памяти. А так как порядок записи в стек обратен порядку чтения, размещение объектов оказывается обратным.

Например:

```
int i1=10, i2=20, i3=30;
```



← направление увеличения адресов

\*&i2=

\*&++i2=

\*p=

\*p++

++\*p=

\*--p=

++\*--p=

\*(p-1)=

# Адресная арифметика (3)

```
// Ex5_1.cpp
#include "stdafx.h"
#include <iostream.h>
int main(int argc, char*
    argv[])
{int i1=10,i2=20,i3=30;
int *p=&i2;
// Value and Address
// i1,i2,i3
cout <<"\n &i1="<<&i1;
cout<<" *&i1= " << *&i1;
cout <<"\n &i2="<<&i2;
cout << " *&i2= " << *&i2;
cout <<"\n &i3="<<&i3;
cout << " *&i3= " << *&i3;
// value i2 added 1 (+ 1)
cout <<"\n *&i2="<<*&i2<<endl;
cout<<"*&++i2"<<*&++i2<<endl;

// value i2
cout << " *p= " << *p<<endl;
// value i2 for Ukaz p
//(p up for 1)
cout <<"\n *p++ ="<< *p++;
// Value i1
cout <<"\n *p ="<< *p;
// value i1 up for 1
cout << "\n++*p =" << ++*p;
//Value i2 begin with down p
cout << "\n*--p = " << *--p;
// Value i3 , begin down p,
// i3 up
cout <<"\n++*--p ="<< ++*--p;
cout<<endl;
return 0;
}
```

# Соотношение ссылки и указателя

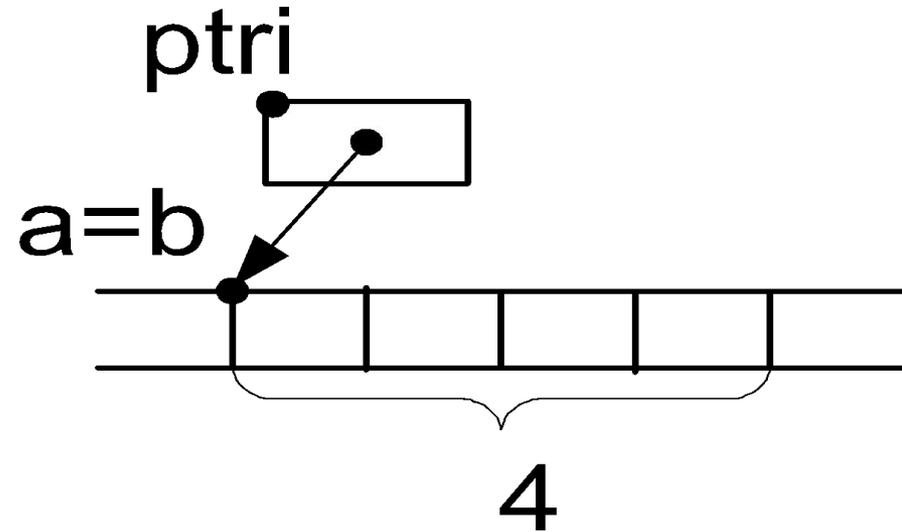
int a,

\*ptri=&a, - указатель

&b=a; - ссылка

...

a=3; ⇔ \*ptri=3; ⇔ b=3;



Основное отличие – для обращения к содержимому по указателю нужна операция разыменования,

обращение к содержимому по ссылке осуществляется по имени ссылки!!!

## 5.3 Управление динамической памятью (C)

### 1. Размещение одного значения

Выделение памяти

**void \* malloc(size\_t size);** - возвращает адрес начала области памяти; при присвоении указателю – явное преобразование типа.

Освобождение памяти

**void free(void \*block);**

Пример:

```
int *a;
```

```
if ((a = (int *) malloc(sizeof(int))) == NULL)
```

```
    { printf("Не хватает памяти для числа.");
```

```
      exit(1); }
```

```
*a=-244; *a+=10;
```

```
free(a);
```

# Управление динамической памятью (С)

## 2. Размещение нескольких значений

Выделение памяти

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

Освобождение памяти

```
void free(void *block);
```

Пример:

```
int *list;
```

```
list = (int *) calloc(3, sizeof(int));
```

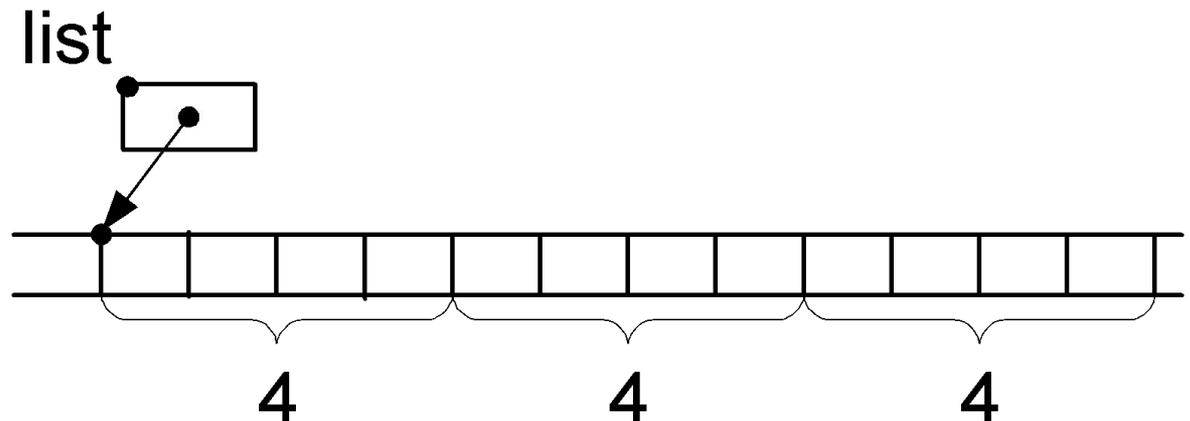
```
*list=-244;
```

```
*(list+1)=15;
```

```
*(list+2)=-45;
```

...

```
free(list);
```



# Управление динамической памятью (C++)

## 1. Одно значение

Операция выделения памяти

```
<Указатель> = new <Имя типа> [( <Значение> )];
```

Операция освобождения памяти

```
delete <Указатель>;
```

Примеры:

```
а) int *k;
```

```
    k = new int;
```

```
    *k = 85;
```

```
б) int *a;
```

```
    if ((a = new int(-244)) == NULL)
```

```
        {printf("Не хватает памяти для числа.");
```

```
          exit(1); }
```

```
    delete a;
```

# Управление динамической памятью (C++)

## 2. Несколько значений

Операция выделения памяти для n значений:

`<Указатель> = new<Имя типа>[<Количество>];`

Операция освобождения памяти:

`delete [ ] <Типизированный указатель>;`

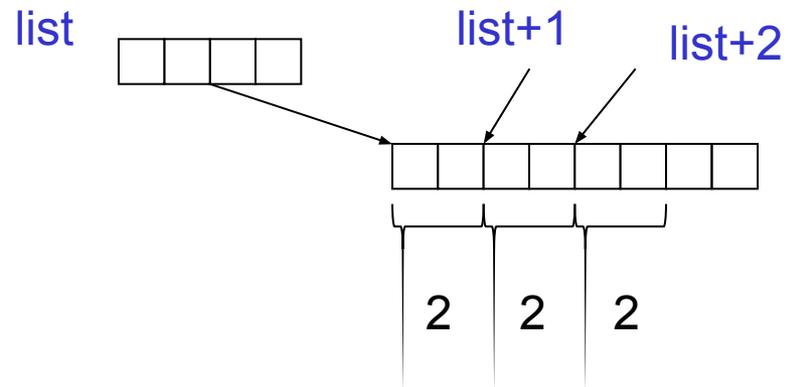
Пример:

```
short *list;
```

```
list = new short[3];
```

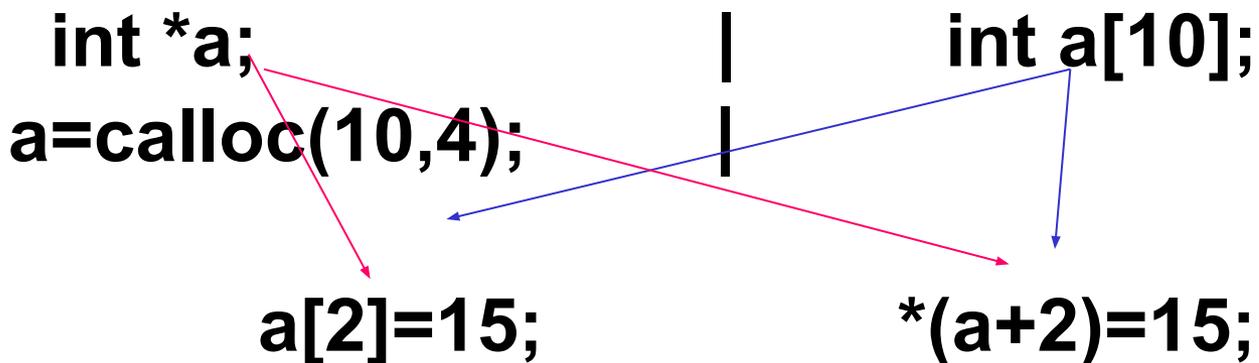
```
*list=-244; *(list+1)=15; *(list+2)=-45;
```

```
delete[ ] list;
```



## 5.4 Многомерные массивы и указатели.

Объявление массива:



Примеры:

```
int list[10];
```

По правилам C++ имя массива является его адресом.

Поэтому для адресации элементов массива независимо от способа описания можно использовать адресную арифметику:

$$(\text{list}+i) \Leftrightarrow \&(\text{list}[i])$$
$$*(\text{list}+i) \Leftrightarrow \text{list}[i]$$

# Многомерные массивы и указатели (2)

```
int m[2][3][2];
```

m -

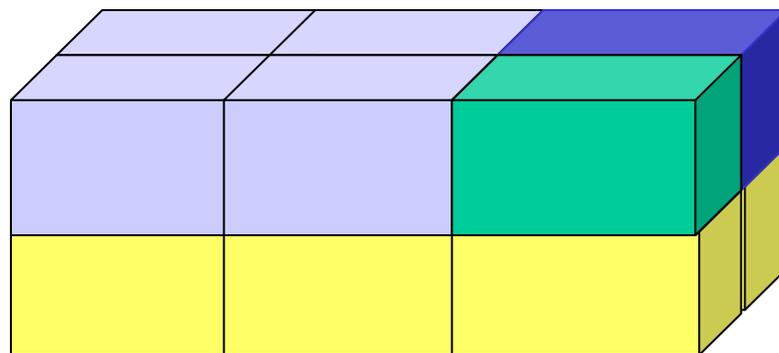
\*m => m[0][][ ]

\*\*m => m[0][0][ ]

\*\*\*m => m[0][0][0]

m[0][2][ ]

m[1] => \*(m+1)



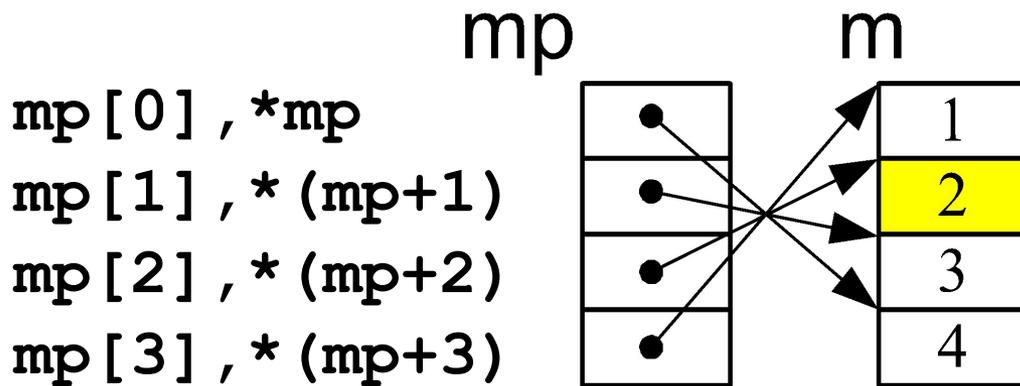
m[0][2][0] => \*(\*(\*m+0)+2)+0 => \*(\*(\*m+2))

m[i][j][k] => \*(\*(\*m+i)+j)+k => \*(\*(\*m+i)+j)+k

# Многоуровневые ссылки (Ex5\_1a)

```
int m[]={1,2,3,4};
```

```
int *mp[]={m+3,m+2,m+1,m};
```



mp[0], \*mp

mp[1], \*(mp+1)

mp[2], \*(mp+2)

mp[3], \*(mp+3)

m[1], \*(m+1)

ИЛИ

mp[0][-2],

\*(mp[0]-2),

\*( \*mp-2),

mp[1][-1],

\*(mp[1]-1),

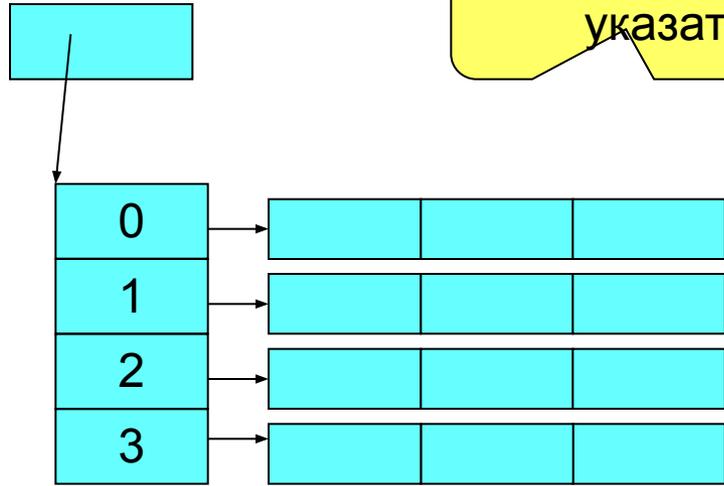
\*( \*(mp+1)-1)

# Использование указателей при обработке массивов

**Пример.** Написать программу переформирования матрицы путем сортировки каждой ее строки по возрастанию ее элементов.

*Создание динамической матрицы*

```
// Ex5_2.cpp
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
int **mas, *ptr;
int a,b,n,m,i,j,k;
void main()
{ printf("\n input n= ");
  scanf("%d",&n);
  printf("\n input m= ");
  scanf("%d",&m);
  mas=new int * [n];
  for(i=0;i<n;i++)
    mas[i]=new int [m];
```



Указатель на указатель

Одномерные массивы целого типа

Массив указателей

Выделение памяти под строки матрицы

Выделение памяти под массив указателей

# Использование указателей при обработки массивов (2)

## *Заполнение матрицы данными*

```
for (i=0 ; i<n ; i++)  
    { printf(" input  %d elem.  %d string\n", m, i) ;  
      ptr=mas [i] ;  
      for (j=0 ; j<m ; j++)  
          scanf ("%d" , mas [i]++) ;  
      mas [i]=ptr ;    }
```

Запоминаем адрес начала строки

Идем по элементам изменяя адрес элемента (адресная арифметика)

## *Печать сформированной матрицы*

```
puts ("Inputed matrix") ;  
for (i=0 ; i<n ; i++)  
    { ptr=mas [i] ;  
      for (j=0 ; j<m ; j++)  
          printf ("%3d" , *ptr++) ;  
      printf ("\n") ;  
    }
```

Восстанавливаем адрес начала строки

Используем вспомогательный указатель для перемещения по строке

# Использование указателей при обработки массивов(3)

*Сортировка строк матрицы*

```
for (i=0 ; i<n ; i++)  
    for(k=0;k<m-1; k++)  
        { ptr=mas[i];  
          for (j=0 ; j<m-1 ; ptr++ , j++)  
              if (*ptr > * (ptr+1))  
                  { b=*ptr ; *ptr=* (ptr+1) ; * (ptr+1) = b ; }  
        }
```

Использование вспомогательного указателя для перестановок

Сортировка строк матрицы методом пузырька

*Печать переформированной матрицы*

```
puts ("Sorted matrix") ;  
    for (i=0 ; i<n ; i++)  
    { ptr=mas[i] ;  
      for (j=0 ; j<m ; j++)  
          printf ("%3d" , *ptr++) ;  
      printf ("\n") ;  
    }
```

Использование вспомогательного указателя для обхода матрицы

```
for (i=0 ; i<n ; i++)  
    delete [] mas[i] ;  
delete [] mas ;
```

Удаление динамической матрицы. Идет в порядке, обратном созданию