Программирование на C++

Лекция 4

ПМИ 2 курс

Демяненко Я.М.

ЮФУ 2025

Указатели и динамическая память

- В C/C++ динамическая память управляется с помощью указателей.

- Если необходимо **в динамической памяти выделить** место для int, используется оператор **new**.

```
int* pi = new int;
*pi = 5;
(*pi)++; // *pi == 6
```

- Освобождение памяти производится с помощью оператора **delete**.

```
delete pi;
pi = nullptr;  // Хороший стиль
```

- Присваивание указателю на высвобожденную память значения nullprt защищает от ошибки при повторном (ошибочном вызове) delete.

- **Замечание.** В C++ сборщика мусора нет! Ответственность за выделение и освобождение памяти лежит на программисте.

Ошибки при работе с динамической памятью

1. Попытка разыменования нулевого указателя

```
int *pi;
*pi = 5 //ошибка
```

2. Обращение к освобожденной памяти

```
int *pi = new int;
*pi = 5
delete pi;
*pi = 6; //ошибка
delete pi; //ошибка
```

Ошибки при работе с динамической памятью

3. Утечка памяти

```
int *pi = new int;
pi = new int;

// Циклы
for(;;)
  pi = new int;
```

Ошибки при работе с динамической памятью

3. Утечка памяти

//Функции
Если в процессе работы функции выделяется динамическая память,
то ее следует **освобождать в теле той же функции**.

```
void f() {
  int* pi = new int;
}
```

Если же **указатель н**а динамически **выделенную память передается в качестве результата** работы функции, то необходимо озаботиться об **освобождении** этой памяти **вне тела** этой функции.
Такой принцип работы **не всегда очевиден** — тем и **опасен**.

```
int* f() {
  return new int;
}
```

Массивы в динамической памяти

```
int* pi = new int[10]; // Выделение памяти для 10 элементов

pi[0] = 5;
pi[1] = 3;
...
```

Для возврата этой памяти используется оператор delete[]:

```
delete[] pi;
```

Как передавать динамические массивы в функции

```
void print(const int* pi, int size) {
  for(int i = 0; i < n, i++)
    cout << pi[i] << ' ';
}

...

int* pi = new int[10];
print(pi, 10);
```

Пример. Проверить, является ли массив целых чисел симметричным относительно своей середины

```
#include <iostream>
using namespace std;
bool simm( int *a, int n );

int main() {
  int *a, n;
  cout <<" The size of the array ";
  cin>>n;
  a=new int[n];
  for (int i=0; i<n; ++i) {
    cout<<i<<" element ";
    cin>>a[i];
  }
  cout<<simm(a,n)<<endl;
  delete []a;
  return 0;
}
```

```
bool simm( int *a, int n ) {
  int *b=a+n-1;
  //указатель на последний элемент массива
  while (a<=b)
    if (*a++!=*b--)
      return false;
  return true;
}
```

Пример. Описать функцию, вычисляющую сумму элементов массива, продемонстрировать её использование для разных сегментов массива

```
int sum_arr( const int *begin, const int *end) {
  const int *pt;
  int sum=0;
  for (pt=begin; pt!=end; ++pt)
    sum+=*pt;
  return sum;
}
```

```
int main() {
  int array[] = {1,2,3,4,5,6,7,8,9,10};
  int n = sizeof array / sizeof(int);
  cout<<"The sum of all ="<<sum_arr(array, array+n)<<endl;
  cout<<"The sum of the first="<<sum_arr(array,array+3)<<endl;
  cout<<"The sum of the last 4 ="<<sum_arr(array+n-4,array+n)<<endl;
  cout<<"The sum from 3 to 7 ="<<sum_arr(array+2,array+7) <<endl;
  return 0;
}
```

Указание диапазона элементов массива

Диапазон задается полуоткрытым справа интервалом [*begin;*end).
Первый указатель определяет начало диапазона элементов массива, второй — его конец.

```
int sum_arr( const int *begin, const int *end);
```

Просмотр всех элементов диапазона организован с помощью цикла
`for (pt=begin; pt!=end; ++pt)`

Условие pt!=end основано на том, что **правая граница** диапазона является указателем на **элемент, следующий за последним элементом диапазона**

Выделение памяти для статических массивов

```
int a [3][4]; // размеры — это константы, поскольку память должна выделяться на этапе компиляции

void f(int a[][4], int m, int n) {
  ...
}
```

Вообще говоря, так писать плохо.

В таких случаях лучше использовать двумерные динамические массивы.

Двумерные массивы в динамической памяти

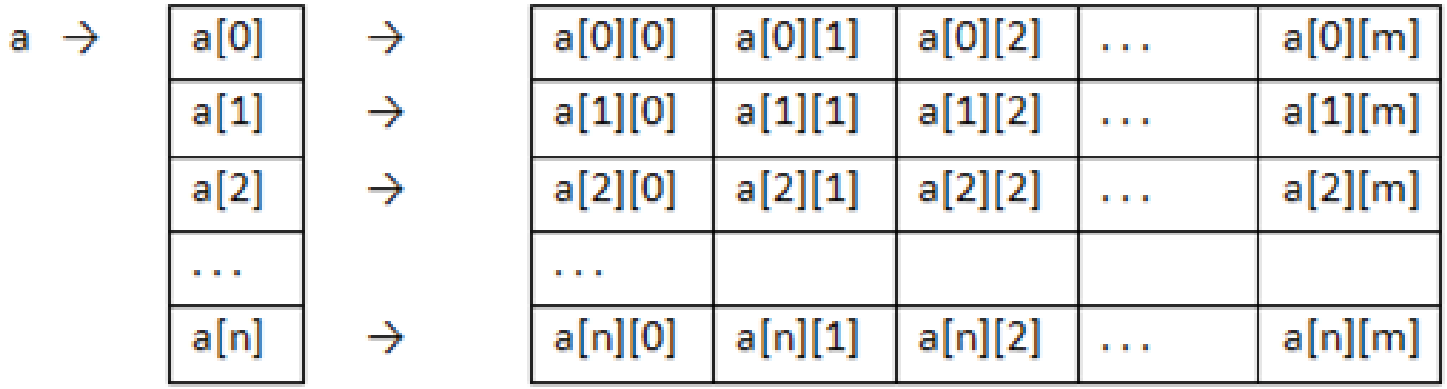
```
int** a;
// a - указатель на указатель или
// a - указатель на начало массива из int*
```

int m, n; cin >> m >> n;

Замечание. Динамические массивы позволяют задавать свой размер во время выполнения программы.

Выделение памяти для динамических массивов

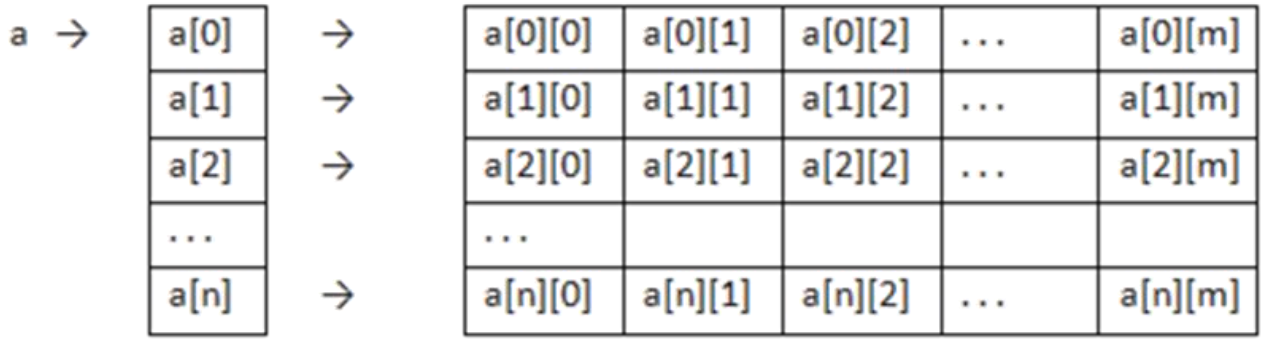
```
a = new int*[m];

for(int i = 0; i < m; i++)
  a[i] = new int[n];

// Теперь можно обращаться
a[1][2] ~ *(*(a+1)+2)
```



Освобождение памяти

Освобождение памяти, занятой динамическим массивом, нужно проводить в обратной последовательности:

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



Передача двумерного «динамического» массива в функции

```
void print(const int** a, int m, int n) {
  for(int i = 0; i < m; i++)   {
    for(int j = 0; i < n; j++)
      cout << a[i][j] << ' ';
    cout << endl;
  }
}
```

Двумерный статический массив в эту функцию передать нельзя.

Организация двумерного массива в динамической памяти позволяет передавать его в качестве параметра функции как **указатель на указатель**.

```
void create(int **& a, int n=5, int m=5);
void input(int ** a, int n=5, int m=5);
void print(int ** a, int n=5, int m=5);
void free(int ** &a, int n=5);
```

Пример. Создать и распечатать верхнетреугольную матрицу

У динамического двумерного массива **не обязательно** все **строки** должны быть **одинаковой длины**

$$\begin{pmatrix} n & n & \dots & n & n \\ 0 & n-1 & \dots & n-1 & n-1 \\ & & \dots & & \\ 0 & 0 & \dots & 2 & 2 \\ 0 & 0 & \dots & 0 & 1 \end{pmatrix}$$

Особенность данной задачи состоит в том, что в памяти формируется **нерегулярный массив**.
Каждая следующая строка короче предыдущей на один элемент.

```
matr = new int *[n];
for ( i=0; i<n; ++i) {
  m=n-i;
  matr[i]= new int[m];
  for (j=0; j<m; ++j)
     matr[i][j]=m;
}
```

```
for (i=0; i<n; ++i) {
   for (j=0; j<i; ++j)
     cout<<"0 ";
   m=n-i;
   for (j=0; j<m; ++j)
     cout<<matr[i][j]<<" ";
   cout<<endl;
 }
```

Вспомним указатели на функцию

```
void func() {    cout<<"func() called... "<<endl;  }

int main() {

  void (*fp)();  //Определение указателя на функцию
  fp=func;       //Инициализация

  (*fp)();  //Разыменование означает вызов
  fp();       //И это вызов

  void (*fp2)() = func;  //Определение и инициализация

  (*fp2)();
  fp2();

}
```

Пример. Описать функции вычисления суммы и максимума значений тех элементов целочисленного массива, которые удовлетворяют заданному условию

```
//Одноместный предикат
typedef bool (* pred1)(int);

bool pos (int a) {
  return a>0;
}

bool isSimple (int x) {
  bool f=true;
  for (int i=2;  i<sqrt((double)abs(x)) && f; ++i)
    f= x%i != 0;
  return f;
}
```

```
//Двуместный предикат
typedef bool (* pred2)(int,int);

bool count_dig(int a, int b) {
  int c = a==0 ? 1 : 0;
  while (a) {
    c++;
    a/=10;
  }
  return c==b;
}

bool last_dig(int a, int b) {
  return abs(a%10)==b;
}
```

```
int sum (int *a, int n, pred1 f) {
  int s=0;
  for (int i=0;i<n;i++)
    if (f(a[i]))
      s+=a[i];
  return s;
}

int sum (int *a, int n, pred2 f, int b) {
  int s=0;
  for (int i=0;i<n;i++)
    if (f(a[i],b))
      s+=a[i];
  return s;
}
```

```
int max(int *a, int n, pred1 f) {
  int m=INT_MIN;
  for (int i=0;i<n;i++)
    if (f(a[i]) && a[i]>m)
      m=a[i];
  return m;
}
```

```
int main() {
  int a[]={-5,-95,-75,1,4,2,-5,-9,17,-15,24,2};

  cout<<"sum last_dig "<<sum(a,12,last_dig,5)<<endl;

  cout<<"sum count_dig "<<sum(a,12,count_dig,1)<<endl;

  cout<<"sum isSimple "<<sum(a,12,isSimple)<<endl;

  cout<<"sum pos "<<sum(a,12,pos)<<endl;

  cout<<"max isSimple "<<max(a,12,isSimple)<<endl;

  cout<<"max pos "<<max(a,12,pos)<<endl;

  return 0;
}
```