

Лекция 5. Статические массивы. Указатели

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

- 1 Статические одномерные массивы: объявление, инициализация, передача в функции.
- 2 Указатели: определение, разыменование, адресная арифметика.
- 3 `nullptr`, `void*`, `reinterpret_cast`.
- 4 Ошибки при работе с указателями.
- 5 Передача параметров через указатели.
- 6 Указатели на структуры, оператор `->`.
- 7 Связь указателей и массивов. Идиома `*p++`.
- 8 `const` и указатели. Ссылки на константы.
- 9 Статические двумерные массивы (матрицы) и указатели.
- 10 Итоги и практические рекомендации.

Что такое массив?

- Массив — это коллекция элементов одного типа, расположенных в памяти последовательно.
- Элементы индексируются с нуля.
- Размер фиксирован на этапе компиляции.
- Массив не хранит свой размер (в отличие от `std::vector`).
- Доступ к элементу: `arr[i]`.

Объявление

тип имя[размер];

```
int arr[5];           // массив из 5 целых чисел
double values[10];    // массив из 10 вещественных чисел
char symbols[256];    // массив из 256 символов
```

Зачем нужны массивы?

- Хранение большого количества однотипных данных.
- Удобный доступ по индексу.
- Эффективная работа с памятью (последовательное хранение).
- Основа для многих алгоритмов (сортировка, поиск, обработка данных).

Пример

Примеры использования:

- Список студентов в группе.
- Таблица умножения (матрица).
- Буфер для ввода/вывода данных.

Особенности массивов в C++

- **Индексация с нуля:** первый элемент — `arr[0]`, последний — `arr[N-1]`.
- **Нет контроля выхода за границы:** компилятор не проверяет индекс.
- **Высокая скорость:** прямой доступ к памяти.
- **Не хранят свой размер:** нужно передавать размер отдельно.

Опасность

```
int arr[5] = {1, 2, 3, 4, 5};  
arr[10] = 100; // Ошибка! Выход за границы (UB)
```

Размер массива — константа времени компиляции

- Размер статического массива должен быть известен на этапе компиляции.
- Способы задания размера:
 - ❶ Числовая константа: `int arr[10];`
 - ❷ Именованная константа: `const int N = 10; int arr[N];`
 - ❸ constexpr: `constexpr int N = 10; int arr[N];`
 - ❹ Макрос: `#define N 10 int arr[N];` (не рекомендуется)
- **Нельзя** использовать переменную, значение которой определяется во время выполнения:

Ошибка

```
int n;  
cin >> n;  
int arr[n]; // НЕДОПУСТИМО в стандартном C++!
```

Объявление и инициализация

```
int arr1[5];           // значения не определены (мусор)
int arr2[5] = {1, 2, 3, 4, 5}; // полная инициализация
int arr3[5] = {1, 2, 3};   // остальные элементы = 0
int arr4[] = {1, 2, 3, 4, 5}; // размер вычисляется автоматически
int arr5[5] = {0};        // все элементы = 0
int arr6[5] = {};         // все элементы = 0 (C++11)
```

- При частичной инициализации остальные элементы заполняются нулями.
- Размер массива должен быть константой времени компиляции.

```
int a[3] = {1, 2, 3};
int b[3];

b = a; // ошибка: встроенные массивы так не присваиваются
```

Элементы можно скопировать циклом, но присвоить один встроенный массив другому через = нельзя.

Определение размера массива

```
int arr[] = {1, 2, 3, 4, 5};  
int size = sizeof(arr) / sizeof(arr[0]); // size = 5  
auto n = std::size(arr); // n = 5, начиная с C++17, надо подключить <iterator>
```

- `sizeof(arr)` — размер массива в байтах.
- `sizeof(arr[0])` — размер одного элемента.
- Размер должен быть известен на этапе компиляции.
- **Важно:** для параметра функции `int arr[]` выражение `sizeof(arr)` даёт размер указателя. Для массива, объявленного внутри функции, размер вычисляется нормально.
- `std::size` работает со встроенным массивом, но не восстанавливает его длину из обычного указателя.

Ошибка

```
void f(int arr[]) {  
    int size = sizeof(arr) / sizeof(arr[0]); // НЕ РАБОТАЕТ!  
}
```

Циклы по массиву

```
int arr[] = {1, 2, 3, 4, 5};
int n = sizeof(arr) / sizeof(arr[0]);

// 1) По индексу
for (int i = 0; i < n; ++i) {
    cout << arr[i] << " ";
}

// 2) Range-based for (C++11)
for (int x : arr) {
    cout << x << " ";
}

// 3) Range-based for с ссылкой (для изменения)
for (int& x : arr) {
    x *= 2;
}
```

Передача массива в функцию

```
void printArray(int arr[], int size) {  
    for (int i = 0; i < size; ++i)  
        cout << arr[i] << " ";  
}  
  
int main() {  
    int data[] = {1, 2, 3, 4, 5};  
    int n = sizeof(data) / sizeof(data[0]);  
    printArray(data, n);  
    return 0;  
}
```

- В функцию передаётся указатель на первый элемент. Сам массив не копируется, поэтому изменения его элементов видны после вызова функции.
- Размер нужно передавать отдельно.
- Изменения в функции влияют на оригинальный массив.

const-параметр для защиты массива

```
void printArray(const int arr[], int size) {  
    // arr[0] = 10; // ОШИБКА! нельзя изменять  
    for (int i = 0; i < size; ++i)  
        cout << arr[i] << " ";  
}
```

- const защищает массив от изменений в функции.
- Это полезно, когда функция только читает данные.
- Соблюдение принципа наименьших привилегий.

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
void change(int arr[]) {
    arr[0] = 100;
}
int main() {
    int a[] = {1, 2, 3};
    change(a);
    cout << a[0];
    return 0;
}
```

Вопрос: что выведет этот код?

```
#include <iostream>
using namespace std;
void change(int arr[]) {
    arr[0] = 100;
}
int main() {
    int a[] = {1, 2, 3};
    change(a);
    cout << a[0];
    return 0;
}
```

Ответ: 100 (массив передаётся через указатель, изменения видны снаружи).

Практика: сумма элементов

```
int sumArray(const int arr[], int size) {  
    int sum = 0;  
    for (int i = 0; i < size; ++i)  
        sum += arr[i];  
    return sum;  
}  
  
int main() {  
    int data[] = {1, 2, 3, 4, 5};  
    int n = sizeof(data) / sizeof(data[0]);  
    cout << sumArray(data, n); // 15  
    return 0;  
}
```

Практика: поиск максимума

```
int maxArray(const int arr[], int size) {  
    int max = arr[0];  
    for (int i = 1; i < size; ++i)  
        if (arr[i] > max) max = arr[i];  
    return max;  
}  
  
int main() {  
    int data[] = {3, 7, 1, 9, 2};  
    int n = sizeof(data) / sizeof(data[0]);  
    cout << maxArray(data, n); // 9  
    return 0;  
}
```

Что такое указатель?

- Указатель — это переменная, которая хранит адрес другой переменной.
- Позволяет косвенно обращаться к данным.
- Объявление: `тип* имя;`

Аналогия

Указатель — как номер ячейки в камере хранения. Зная номер, можно найти вещь.

```
int x = 5;  
int* p = &x; // p хранит адрес x
```

Объявление и разыменование

```
int x = 5;
int* p = &x;    // p - указатель на x
cout << p;      // адрес x (например, 0x7ffd)
cout << *p;     // значение x (5)
*p = 10;        // изменяем x через указатель
cout << x;      // 10
```

- & — оператор взятия адреса.
- * — оператор разыменования (доступ к значению по адресу).
- Тип указателя должен соответствовать типу переменной.

nullptr (C++11)

```
int* p = nullptr; // указатель не указывает ни на что
if (p == nullptr) {
    cout << "p is null";
}
```

- nullptr — безопасная замена NULL и 0.
- nullptr задаёт нулевой указатель: он не указывает на объект или функцию.
- Разыменование nullptr — ошибка (UB).
- Используйте nullptr всегда вместо NULL или 0.

Ошибка

```
int* p = nullptr;
*p = 5; // ОШИБКА! разыменование нулевого указателя
```

void* — указатель на неизвестный тип

```
int x = 5;
double d = 3.14;
void* p = &x;    // OK
p = &d;          // OK
// *p = 10;      // ОШИБКА! нельзя разыменовать void*
double* pd = static_cast<double*>(p); // явное приведение
cout << *pd;    // 3.14
```

- void* может хранить адрес любого типа.
- Нельзя разыменовывать напрямую.
- Требуется явное приведение к нужному типу.
- Используется в низкоуровневом программировании.

reinterpret_cast

```
int x = 42;  
int* p = &x;  
long long addr = reinterpret_cast<long long>(p);  
cout << hex << addr; // адрес в шестнадцатеричном виде
```

- `reinterpret_cast` — опасное приведение типов.
- Интерпретирует биты одного типа как другой.
- Почти всегда указывает на проблему дизайна.
- Используйте только при крайней необходимости.

Осторожно

Компилятор проверяет допустимость самого преобразования, но не гарантирует безопасность последующего использования результата.

Указатели и ссылки: сравнение

Характеристика	Указатель	Ссылка
Инициализация	необязательна	обязательна
Перенаправление	можно	нельзя
Нулевое значение	можно (nullptr)	нельзя
Арифметика	есть	нет
Синтаксис	*p, ->	r, .

- Ссылка — это другое имя для уже существующего объекта.
- Ссылка безопаснее, указатель гибче.
- Для передачи параметров в функции обычно используют ссылки.

Ошибки при работе с указателями

1 Неинициализированный указатель:

```
int* p; // мусор
*p = 5; // UB
```

2 Разыменование nullptr:

```
int* p = nullptr;
*p = 5; // UB
```

3 Висячий указатель (после delete):

```
int* p = new int(5);
delete p;
*p = 10; // UB
```

Важно

Всегда инициализируйте указатели. Перед разыменованием указатель должен указывать на **существующий** объект.

Вопрос: что выведет этот код?

```
int a = 5;  
int* p = &a;  
*p = 10;  
cout << a;
```

Вопрос: что выведет этот код?

```
int a = 5;  
int* p = &a;  
*p = 10;  
cout << a;
```

Ответ: 10 (изменение через указатель влияет на переменную).

Передача параметров через указатели

```
void swap(int* a, int* b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
}  
  
int main() {  
    int x = 5, y = 10;  
    swap(&x, &y);  
    cout << x << " " << y; // 10 5  
    return 0;  
}
```

- Указатели позволяют изменять переданные переменные.
- Нужно передавать адреса (&x, &y).
- Альтернатива: ссылки (удобнее, безопаснее).

Указатели на структуры

```
struct Point { double x, y; };  
Point p{1.0, 2.0};  
Point* pp = &p;  
cout << (*pp).x; // 1.0  
cout << pp->x;   // 1.0 (оператор ->)  
pp->y = 3.0;      // изменяем поле
```

- Оператор `->` — сокращение для `(*pp).field`.
- Удобен для доступа к полям структуры через указатель.

Вопрос: что выведет этот код?

```
struct Point { int x, y; };  
Point p{3, 4};  
Point* pp = &p;  
pp->x = 10;  
cout << p.x;
```

Вопрос: что выведет этот код?

```
struct Point { int x, y; };  
Point p{3, 4};  
Point* pp = &p;  
pp->x = 10;  
cout << p.x;
```

Ответ: 10 (изменение через указатель влияет на структуру).

Массивы указателей

```
int a = 10, b = 20, c = 30;
int* arr[3] = {&a, &b, &c};
for (int i = 0; i < 3; ++i)
    cout << *arr[i] << " "; // 10 20 30
```

- Массив может хранить указатели.
- Доступ к данным через разыменование.
- Можно использовать для хранения С-строк (массив char*).

Указатели на указатели (char**)

```
const char* strings[] = {"Hello", "World", "C++"};
const char** p = strings; // указатель на первый указатель
cout << *p;           // Hello
cout << *(p + 1);    // World
```

- char** — указатель на указатель на char.
- Используется для работы с массивами строк (например, argv).

Преобразование массива в указатель на первый элемент

```
int arr[] = {1, 2, 3, 4, 5};  
int* p = arr; // p - указатель на первый элемент массива  
cout << *p;   // 1  
cout << *(p + 1); // 2  
cout << *(p + 2); // 3
```

- Во многих выражениях массив **неявно** преобразуется в указатель на свой первый элемент.
- `arr` эквивалентно `&arr[0]`.
- `arr[i]` эквивалентно `*(arr + i)`.

Доступ к элементам через указатель

```
int arr[] = {10, 20, 30, 40, 50};  
int* p = arr;  
for (int i = 0; i < 5; ++i) {  
    cout << *(p + i) << " "; // 10 20 30 40 50  
}  
// Или так:  
for (int* q = arr; q < arr + 5; ++q) {  
    cout << *q << " ";  
}
```

- Указатели позволяют обходить массив без индексов.
- $p + i$ — адрес i -го элемента.
- $*(p + i)$ — значение i -го элемента.

```
int arr[] = {1, 2, 3, 4, 5};  
int* p = arr;  
p++;      // теперь p указывает на arr[1]  
p += 2;   // теперь p указывает на arr[3]  
int* q = arr + 5; // указатель за последний элемент  
int n = q - p;    // n = 2 (количество элементов между p и q)
```

- $p++$ — переход к следующему элементу.
- $p += n$ — переход на n элементов вперёд.
- $p - q$ — расстояние между указателями (в элементах).
- **Нельзя складывать указатели!**

- *p++ означает *(p++) (более высокий приоритет постфиксного ++ по сравнению с разыменованием *).
- p++ возвращает прежнее значение указателя и увеличивает сам указатель. Поэтому разыменование выполняется по прежнему адресу.
- Широко используется при работе с массивами.

```
int arr[5] = {1, 2, 3, 4, 5};  
int* p = arr;  
int x = *p++; // x = 1, p указывает на arr[1]  
int y = *p++; // y = 2, p указывает на arr[2]
```

Пример 1: заполнение нулями

```
int arr[10];  
int* p = arr;  
for (int i = 0; i < 10; ++i)  
    *p++ = 0; // заполняем нулями
```

Пример 2: сумма двух массивов

```
int a[5]{ 1, 2, 3, 4, 5 }, b[5]{6, 7, 8, 9, 10}, c[5];  
int* pa = a;  
int* pb = b;  
int* pc = c;  
for (int i = 0; i < 5; ++i)  
    *pc++ = *pa++ + *pb++;
```

Пример 3: проверка симметричности

```
int arr[] = {1, 2, 3, 2, 1};
int* left = arr;
int* right = arr + 4;
bool sym = true;
while (left < right) {
    if (*left != *right) { sym = false; break; }
    left++;
    right--;
}
cout << (sym ? "yes" : "no"); // yes
```

Вопрос: что выведет этот код?

```
int arr[] = {10, 20, 30, 40};  
int* p = arr;  
cout << *(p + 2) << " " << *(arr + 1);
```

Вопрос: что выведет этот код?

```
int arr[] = {10, 20, 30, 40};  
int* p = arr;  
cout << *(p + 2) << " " << *(arr + 1);
```

Ответ: 30 20 ($*(p+2) = arr[2] = 30$, $*(arr+1) = arr[1] = 20$).

Вопрос: что выведет этот код?

```
int arr[] = {1, 2, 3};  
int* p = arr;  
int x = *p++;  
cout << x << " " << *p;
```

Вопрос: что выведет этот код?

```
int arr[] = {1, 2, 3};  
int* p = arr;  
int x = *p++;  
cout << x << " " << *p;
```

Ответ: 1 2 (сначала взяли значение 1, затем p перешёл на arr[1]).

Указатель на константу

```
int x = 5;
const int* p = &x; // указатель на константу
cout << *p; // OK
// *p = 10; // ОШИБКА! нельзя изменять данные
x = 10;      // можно изменить через оригинал
```

- `const int* p` — указатель на константу.
- Нельзя изменять данные через указатель.
- Данные могут быть изменены через другую ссылку.

Константный указатель

```
int x = 5, y = 10;  
int* const p = &x; // константный указатель  
*p = 10; // ОК  
// p = &y; // ОШИБКА! нельзя менять адрес
```

- `int* const p` — константный указатель.
- Нельзя менять адрес, но можно менять данные.

Константный указатель на константу

```
int x = 5, y = 10;  
const int* const p = &x;  
// *p = 10; // ОШИБКА!  
// p = &y; // ОШИБКА!
```

- `const int* const p` — константный указатель на константу.
- Нельзя менять ни данные, ни адрес через `p`.

```
int x = 5;  
const int& r = x;  
cout << r; // OK  
// r = 10; // ОШИБКА! нельзя менять через ссылку  
x = 10;    // можно через оригинал
```

- `const int& r` — ссылка на константу.
- Нельзя изменять данные через ссылку.
- Защита от изменений.

Вопрос: что выведет этот код?

```
int a = 5;  
const int* p = &a;  
a = 10;  
cout << *p;
```

Вопрос: что выведет этот код?

```
int a = 5;  
const int* p = &a;  
a = 10;  
cout << *p;
```

Ответ: 10 (указатель на константу не запрещает изменять данные через оригинал).

Двумерные массивы (матрицы)

- Массив массивов: `int matrix[3][4];`
- Первый индекс — строка, второй — столбец.
- Хранение в памяти: по строкам (row-major order).
- Строки матрицы расположены в памяти подряд, но каждая строка остаётся **отдельным массивом**. Указатель на элемент строки позволяет обходить эту строку; для перехода к другой строке нужно обратиться к ней отдельно.

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

Организация двумерного массива

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Как таблица

	Столбец 0	Столбец 1	Столбец 2
Строка 0	1	2	3
Строка 1	4	5	6

В памяти — строки подряд, без промежутков

matrix[0][0] 1	matrix[0][1] 2	matrix[0][2] 3	matrix[1][0] 4	matrix[1][1] 5	matrix[1][2] 6
Строка 0 — matrix[0]			Строка 1 — matrix[1]		

Направление увеличения адресов

Каждая ячейка — один int. Отдельного массива указателей на строки нет.

Инициализация матрицы

```
int m1[2][3] = {{1,2,3}, {4,5,6}}; // полная
int m2[2][3] = {{1,2}, {4,5}};      // остальные = 0
int m3[2][3] = {1,2,3,4,5,6};       // тоже работает, но не наглядно
int m4[][3] = {{1,2,3}, {4,5,6}};    // количество строк вычисляется
```

- При инициализации можно не указывать количество строк: оно определяется по инициализатору. Количество столбцов всегда необходимо указать.
- Вложенные фигурные скобки улучшают читаемость.

Передача матрицы в функцию

```
void printMatrix(int matrix[][3], int rows) {  
    for (int i = 0; i < rows; ++i) {  
        for (int j = 0; j < 3; ++j)  
            cout << matrix[i][j] << " ";  
        cout << endl;  
    }  
}  
  
int main() {  
    int m[2][3] = {{1,2,3}, {4,5,6}};  
    printMatrix(m, 2);  
    return 0;  
}
```

- Второй размер (количество столбцов) должен быть указан.
- Первый размер можно не указывать.
- Причина: компилятор должен знать, сколько байтов в строке.

Связь матриц и указателей

```
int matrix[2][3] = {{1,2,3}, {4,5,6}};  
// matrix[i][j] эквивалентно (*(matrix + i) + j)  
int* p = matrix[0]; // указатель на первый элемент первой строки  
cout << *(p + 2);    // 3 (matrix[0][2])  
cout << (*(matrix + 1) + 1); // 5 (matrix[1][1])
```

- `matrix[i]` — *i*-я строка, то есть массив. В данном выражении она преобразуется в указатель на свой первый элемент.
- `*(matrix[i] + j)` — элемент в строке *i*, столбце *j*.
- Двойное разыменованное.

Вопрос: что выведет этот код?

```
int m[2][3] = {{1,2,3}, {4,5,6}};  
cout << m[1][2] << " " << (*(m + 1) + 2);
```

Вопрос: что выведет этот код?

```
int m[2][3] = {{1,2,3}, {4,5,6}};  
cout << m[1][2] << " " << (*(m + 1) + 2);
```

Ответ: 6 6 (оба выражения обращаются к элементу `m[1][2]`).

Вопрос: что выведет этот код?

```
int m[2][3] = {{1,2,3}, {4,5,6}};  
int (*p)[3] = m; // указатель на строку из трёх int  
cout << *(*p + 1); // 4
```

Вопрос: что выведет этот код?

```
int m[2][3] = {{1,2,3}, {4,5,6}};  
int (*p)[3] = m; // указатель на строку из трёх int  
cout << *(*p + 1); // 4
```

Ответ: 4 (*p+1) указывает на первый элемент второй строки).
Что следовало поместить в cout, если хотели бы вывести 5?

- Статические одномерные массивы: объявление, инициализация, передача в функции.
- Указатели: объявление, разыменование, `nullptr`.
- `void*`, `reinterpret_cast`.
- Ошибки при работе с указателями.
- Передача параметров через указатели.
- Указатели на структуры, оператор `->`.
- Связь указателей и массивов.
- Адресная арифметика и идиома `*p++`.
- `const` и указатели. Ссылки на константы.
- Статические двумерные массивы (матрицы) и их связь с указателями.

Стандарт	Нововведение
C++11	nullptr, range-based for
C++17	std::size

- `nullptr` — безопасная замена `NULL`.
- `range-based for` — упрощает перебор массивов.
- `std::size` — позволяет определять размер массива (не внутри функции) вместо `sizeof`.

- Всегда инициализируйте массивы при объявлении.
- Используйте `const` для защиты данных в функциях.
- Для размера массива используйте `constexpr` или константы.
- Избегайте выхода за границы массива.
- Используйте `nullptr` вместо `NULL`.
- Для передачи больших массивов используйте указатели или ссылки.
- Всегда проверяйте указатели на `nullptr` перед разыменованием.

- ❶ Выход за границы массива.
- ❷ Разыменование `nullptr`.
- ❸ Использование неинициализированного указателя.
- ❹ Путаница между `*p++` и `(*p)++`.
- ❺ Забыли передать размер массива в функцию.
- ❻ Смешение указателей и ссылок.

Что дальше?

- Следующая лекция: углубление по указателям, `std::array`, чтение сложных объявлений.
- На практике: закрепление работы с массивами и указателями.

Вопросы?