

Лекция 6. Указатели (продолжение). `std::array`. Указатели на функции

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

- 1 Повторение: связь указателей и массивов, адресная арифметика.
- 2 Указатели на указатели.
- 3 Массивы указателей.
- 4 `std::array` (C++11): создание, методы, итераторы, примеры.
- 5 `std::span` (C++20): невладеющее представление, методы, нюансы.
- 6 Указатели на функции: синтаксис, вызов, `using`.
- 7 Callback-вызовы: передача функции как параметра.
- 8 Чтение сложных объявлений: правило «изнутри наружу».
- 9 Итоги.

Повторение: указатели и массивы

- В большинстве выражений массив `arr` неявно преобразуется в указатель на первый элемент `&arr[0]`.
- Доступ через указатель: `arr[i]  $\equiv$  *(arr + i)`.
- Адресная арифметика: `p++`, `p += n`, разность указателей.
- Идиома `*p++`: сначала взять значение, затем сдвинуть указатель.

```
int arr[] = {10, 20, 30};  
int* p = arr;  
cout << *p++;    // 10, теперь p указывает на 20  
cout << *(arr + 2); // 30
```

Указатели на указатели (int**)

- Указатель может хранить адрес другого указателя.
- Объявление: `int** pp;`.
- Используется для работы с массивами указателей на строки и для изменения указателя через передачу его адреса.

```
int a = 5;
int* p = &a;    // указатель на int
int** pp = &p;  // указатель на указатель на int
cout << **pp;   // 5 (двойное разыменование)
```

Массивы указателей

- Массив может хранить указатели.
- Пример: массив указателей на `int`.

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

- Можно использовать для массивов С-строк `char**` (будет позже).

Пример: массив указателей на double

```
double a = 1.1, b = 2.2, c = 3.3;
double* ptrs[3] = {&a, &b, &c};
for (int i = 0; i < 3; ++i)
    *ptrs[i] *= 2; // удваиваем значения
cout << a << " " << b << " " << c; // 2.2 4.4 6.6
```

Вопрос: что выведет этот код?

```
int a = 1;  
int* p = &a;  
int** pp = &p;  
**pp = 10;  
cout << a;
```

Вопрос: что выведет этот код?

```
int a = 1;  
int* p = &a;  
int** pp = &p;  
**pp = 10;  
cout << a;
```

Ответ: 10 (двойное разыменование меняет исходную переменную).

Передача указателя на указатель

```
void setNull(int** pp) {  
    *pp = nullptr; // перенаправляем указатель  
}  
  
int main() {  
    int a = 5;  
    int* p = &a;  
    setNull(&p);  
    cout << a << endl; // p теперь nullptr, а по-прежнему равно 5  
    return 0;  
}
```

- Позволяет изменить значение указателя внутри функции.
- Альтернатива: ссылка на указатель (`int* &`).

Зачем нужен `std::array`?

- Статический массив с фиксированным размером, известным на этапе компиляции.
- Преимущества перед C-массивом:
 - 1 Знает свой размер (`size()`).
 - 2 Поддерживает итераторы (совместим с STL).
 - 3 Безопасный доступ через `at()` (с проверкой границ).
 - 4 Можно копировать и присваивать.
- `std::array` хранит элементы непосредственно внутри своего объекта.

Синтаксис

```
std::array<тип, размер> имя;
```

```
#include <array>
std::array<int, 5> a1;           // не инициализирован (содержит мусор)
std::array<int, 5> a2 = {1,2,3,4,5}; // полная инициализация
std::array<int, 5> a3 = {1,2};   // остальные нули
std::array<int, 5> a4{};         // все нули (C++11)
std::array<int, 5> a5 = {0};     // все нули
```

- Можно использовать универсальную инициализацию.
- Размер должен быть константой времени компиляции.

Методы: size(), empty(), fill()

```
std::array<int, 5> a{1,2,3,4,5};  
cout << a.size();    // 5  
cout << a.empty();   // 0 (false)  
a.fill(0);           // заполняет все элементы нулями  
for (int x : a) cout << x << " "; // 0 0 0 0 0
```

- size() возвращает size_t.
- empty() возвращает bool (всегда false для std::array, если размер > 0).
- fill() заполняет все элементы заданным значением.

Методы: front() и back()

```
std::array<int, 5> a{10, 20, 30, 40, 50};  
cout << a.front(); // 10 (первый элемент)  
cout << a.back();  // 50 (последний элемент)  
a.front() = 100;    // изменяем первый элемент  
cout << a[0];       // 100
```

- Возвращают ссылку на элемент.
- Удобны для быстрого доступа к концам.

Безопасный доступ: at() vs operator[]

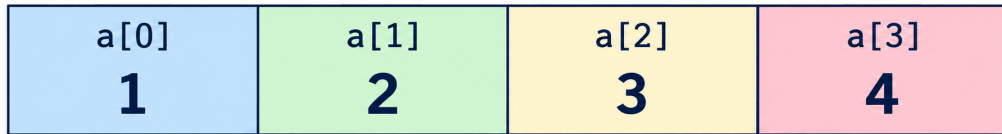
```
std::array<int, 3> a{1,2,3};  
// cout << a[5]; // UB (выход за границы, проверка границ не предусмотрена)  
try {  
    cout << a.at(5); // выбрасывает std::out_of_range  
}  
catch (const std::out_of_range& e) {  
    cout << "Ошибка: " << e.what();  
}
```

- operator[] — быстрый, но без проверки.
- at() — медленнее, но безопаснее (с проверкой).

Память `std::array`

```
std::array<int, 4> a{1, 2, 3, 4};
```

Объект `a`



Адреса увеличиваются

Элементы расположены подряд, без промежутков.
Все четыре элемента находятся внутри объекта `a`.

`std::array<int, 4> a{1,2,3,4};` — элементы лежат в памяти последовательно, как C-массив.

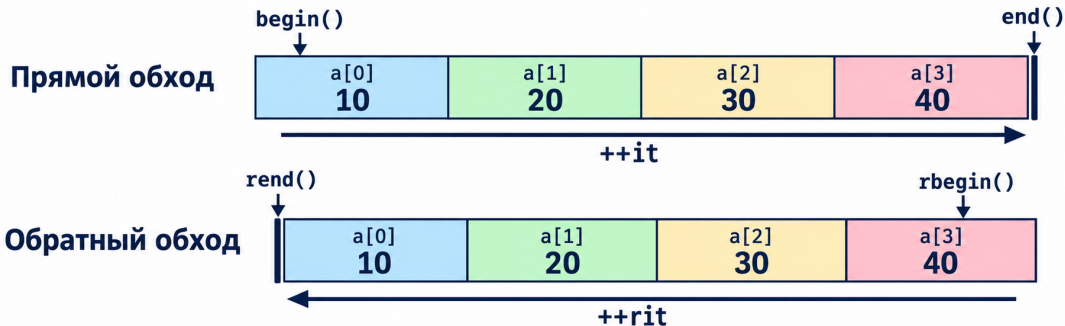
Итераторы `std::array`

```
std::array<int, 4> a{10, 20, 30, 40};  
auto it = a.begin(); // итератор на первый элемент  
auto it_end = a.end(); // итератор за последним  
auto rit = a.rbegin(); // обратный итератор на последний  
auto rit_end = a.rend(); // конец обратного обхода  
cout << *it;          // 10  
cout << *rit;         // 40
```

- Итераторы позволяют использовать алгоритмы STL.
- `rbegin()` и `rend()` для обхода в обратном порядке.

Итераторы `std::array`

```
std::array<int, 4> a{10, 20, 30, 40};
```



`end()` и `rend()` обозначают конец обхода. Их нельзя разыменовывать.

Позиция `rend()` показана условно: это не указатель `a.data() - 1`.

`begin()` → первый, `end()` → за последним, `rbegin()` → последний, `rend()` → перед первым.

Пример: range-based for с std::array

```
std::array<int, 4> a{1, 2, 3, 4};  
for (int x : a) {  
    cout << x * x << " "; // 1 4 9 16  
}  
for (int& x : a) {  
    x *= 2; // изменяем элементы  
}  
for (auto x : a) {  
    cout << x << " "; // 2 4 6 8  
}
```

Пример: сортировка `std::array`

```
#include <algorithm>
std::array<int, 5> a{5, 2, 8, 1, 3};
std::sort(a.begin(), a.end());
for (int x : a) cout << x << " "; // 1 2 3 5 8
```

- `std::sort` работает с итераторами `std::array`.
- Можно сортировать по убыванию с компаратором.

Вставка и удаление в `std::array`

- Размер фиксирован, поэтому вставка/удаление требует ручного сдвига элементов.

```
std::array<int, 5> a{1,2,3,4,5};  
// удаляем элемент с индексом 2 (значение 3)  
for (int i = 2; i < 4; ++i)  
    a[i] = a[i+1]; // сдвиг влево на 1 позицию  
a[4] = 0; // затираем последний  
// теперь a: 1 2 4 5 0
```

- Это логическое удаление со сдвигом. Размер `std::array` не изменяется; количество используемых элементов при необходимости учитывают отдельно.
- Если требуется изменение количества элементов, используйте `std::vector` или динамические массивы.

Передача `std::array` в функцию

```
void byValue(std::array<int, 3> a) { a[0] = 100; }
void byRef(std::array<int, 3>& a) { a[0] = 100; }
int main() {
    std::array<int, 3> a{1,2,3};
    byValue(a);
    cout << a[0]; // 1 (копия, оригинал не изменился)
    byRef(a);
    cout << a[0]; // 100 (оригинал изменён)
    return 0;
}
```

- По значению — копирование всех элементов (дорого).
- По ссылке — эффективно, изменения видны снаружи.

Сравнение `std::array` и C-массива

Характеристика	<code>std::array</code>	C-массив
Получение размера	<code>a.size()</code>	<code>std::size(a)</code> или <code>sizeof(a) / sizeof(type)</code>
Итераторы	да	указатели выполняют роль итераторов
Копирование	да (оператор <code>=</code>)	нет (только поэлементно)
Безопасный доступ	<code>at()</code>	нет
Совместимость с STL	да	да, через указатели на элементы
Накладные расходы	нет (такой же размер)	нет

Рекомендация: использовать `std::array` вместо C-массивов везде, где размер известен на этапе компиляции.

У встроенного массива нет методов, но для него работают отдельные функции `std::begin(arr)` и `std::end(arr)`. Они возвращают указатели, которые используются как итераторы.

Вопрос: что выведет этот код?

```
std::array<int, 3> a{1,2,3};  
cout << a.at(3);
```

Вопрос: что выведет этот код?

```
std::array<int, 3> a{1,2,3};  
cout << a.at(3);
```

Ответ: Будет выброшено исключение `std::out_of_range` (индекс 3 вне диапазона 0..2). Если исключение не обработано, программа аварийно завершится.

Вопрос: что выведет этот код?

```
void f(std::array<int, 2> a) { a[0] = 99; }  
int main() {  
    std::array<int, 2> a{1,2};  
    f(a);  
    cout << a[0];  
    return 0;  
}
```

Вопрос: что выведет этот код?

```
void f(std::array<int, 2> a) { a[0] = 99; }  
int main() {  
    std::array<int, 2> a{1,2};  
    f(a);  
    cout << a[0];  
    return 0;  
}
```

Ответ: 1 (передача по значению создаёт копию, оригинал не меняется).

Когда использовать `std::array`?

- Если размер массива известен на этапе компиляции.
- Если нужны методы STL (итераторы, алгоритмы).
- Если нужна безопасность (проверка границ через `at()`).
- Если нужно копирование массива.
- Если не требуется динамическое изменение размера.

Не используйте

Если размер меняется во время выполнения — используйте `std::vector`.

Что такое `std::span`?

- `std::span` (C++20) — невладеющее представление непрерывной последовательности.
- Это «окно» в существующий массив, `std::array` или `std::vector`.
- Не владеет данными, только ссылается на них.
- Позволяет передавать часть последовательности без копирования.

Аналогия

`std::span` — как указатель с размером, но с удобными методами.

Зачем нужен `std::span`?

- Передача подмассива в функцию без копирования.
- Единый интерфейс для работы с разными типами (C-массив, `std::array`, `std::vector`).
- Отличие от `std::array`:
 - `std::array` хранит элементы непосредственно внутри своего объекта.
 - `std::span` не владеет — это просто ссылка.
- Отличие от C-массива: знает размер, имеет методы.

Основные методы `std::span`

```
#include <span>
int arr[] = {1, 2, 3, 4, 5};
std::span<int> sp(arr); // или sp(arr, 5)
cout << sp.size();      // 5
cout << sp.empty();     // false
cout << sp.front();     // 1
cout << sp.back();      // 5
sp[2] = 100;            // изменяет arr[2]
cout << sp.data();      // указатель на arr
```

- `data()` — сырой указатель.
- `size()` — количество элементов.
- `front()`, `back()` — доступ к концам.
- `operator[]` — доступ по индексу без проверки границ.
- Метод `at()` для `std::span` появился только в C++26.

Пример: создание span

```
#include <span>
#include <array>
int arr[] = {1, 2, 3};
std::span<int> sp1(arr);

std::array<int, 4> a{10, 20, 30, 40};
std::span<int> sp2(a); // или sp2(a.begin(), a.size())

for (int x : sp2) cout << x << " "; // 10 20 30 40
```

- `std::span` можно создать из любого непрерывного контейнера.

Пример: передача части массива

```
void print(std::span<int> sp) {  
    for (int x : sp) cout << x << " ";  
}  
  
int main() {  
    int arr[] = {1, 2, 3, 4, 5};  
    print(arr);           // весь массив  
    print(std::span(arr+1, 3)); // подмассив {2,3,4}  
    return 0;  
}
```

- `std::span(arr+1, 3)` — создаёт span из 3 элементов, начиная со второго.

Нюансы и ограничения `std::span`

- **Непрерывность данных:** работает только с непрерывными последовательностями (C-массив, `std::array`, `std::vector`). Не работает с `std::list`.
- **Время жизни данных:** `std::span` не продлевает жизнь. Если источник данных уничтожен, `span` становится висячим.
- **Модификация:** можно изменять элементы через `span`, если они не константные.
- `std::span` — лёгкий объект (обычно два указателя или указатель + размер).

Примеры UB с `std::span`

```
// ОПАСНО: span указывает на локальный массив
std::span<int> createSpan() {
    int arr[] = { 1,2,3 };
    return std::span(arr); // arr уничтожится при выходе
}

auto sp = createSpan();
sp[1] = 100; // UB: arr уже не существует (вывод может быть правильным, а может нет)

int arr[] = { 1, 2, 3 };
std::span<int> sp2(arr);
sp2[5] = 100; // UB: индекс 5 вне [0, 3)
```

- Всегда проверяйте время жизни источника.
- В C++23 используйте индексацию через `operator[]`; `std::span::at()` появился в C++26.

Что такое указатель на функцию?

- Указатель на функцию хранит адрес функции.
- Позволяет вызывать функцию динамически, передавать как параметр.
- Синтаксис: тип_возврата (*имя)(параметры);

```
int add(int a, int b) { return a + b; }  
int (*pf)(int, int) = add; // pf указывает на add  
cout << pf(2, 3); // 5
```

Объявление и инициализация

```
int add(int, int);  
int sub(int, int);  
  
int (*pf)(int, int); // объявление  
pf = add;           // присваивание адреса add  
cout << pf(5, 3);   // 8  
pf = sub;  
cout << pf(5, 3);   // 2
```

- Имя функции неявно преобразуется к указателю.
- Можно взять адрес явно: `pf = &add;`.

Вызов через указатель

```
int add(int a, int b) { return a + b; }  
int (*pf)(int, int) = add;  
  
int result1 = pf(2, 3);    // эквивалентно add(2,3)  
int result2 = (*pf)(2, 3); // тоже работает (явное разыменование)
```

- Оба способа допустимы.
- Предпочтительнее: `pf(2,3)` — короче и читаемее.

Использование using для упрощения

```
using Func = int (*)(int, int); // псевдоним типа
int add(int a, int b) { return a + b; }
int sub(int a, int b) { return a - b; }

Func pf = add;
cout << pf(10, 5); // 15
pf = sub;
cout << pf(10, 5); // 5
```

- using делает код читаемее.
- Можно передавать Func как тип параметра.

Пример: функция с параметром-указателем

```
using Operation = int (*)(int, int);

int apply(int a, int b, Operation op) {
    return op(a, b);
}

int add(int a, int b) { return a + b; }
int mul(int a, int b) { return a * b; }

int main() {
    cout << apply(3, 4, add); // 7
    cout << apply(3, 4, mul); // 12
    return 0;
}
```

Пример: массив указателей на функции

```
void menu1() { cout << "Option 1\n"; }  
void menu2() { cout << "Option 2\n"; }  
void menu3() { cout << "Option 3\n"; }  
  
using MenuFn = void(*)();  
MenuFn menu[] = {menu1, menu2, menu3};  
  
int choice = 2;  
menu[choice](); // вызов menu3 (индексация с 0)
```

- Используется для реализации меню, командных обработчиков.

Вопрос: что выведет этот код?

```
int f(int x) { return x * 2; }
int g(int x) { return x + 10; }
int h(int x, int (*op)(int)) { return op(x); }
int main() {
    cout << h(5, f) << " " << h(5, g);
    return 0;
}
```

Вопрос: что выведет этот код?

```
int f(int x) { return x * 2; }
int g(int x) { return x + 10; }
int h(int x, int (*op)(int)) { return op(x); }
int main() {
    cout << h(5, f) << " " << h(5, g);
    return 0;
}
```

Ответ: 10 15 (f(5)=10, g(5)=15).

- Callback — это функция, переданная как параметр и вызываемая внутри другой функции.
- Применение:
 - Обработка событий (например, нажатие кнопки).
 - Сортировка с компаратором.
 - Фильтрация данных.
 - Алгоритмы обхода контейнеров.
- В C++ callback реализуется через указатели на функции, функторы или лямбды (будет позже).

Пример: сортировка с компаратором

```
#include <algorithm>
bool cmp(int a, int b) { return a > b; } // по убыванию
int arr[] = {5, 2, 8, 1, 3};
std::sort(arr, arr+5, cmp);
// std::sort(std::begin(arr), std::end(arr), cmp); // то же самое
for (int x : arr) cout << x << " "; // 8 5 3 2 1
```

- `std::sort` принимает компаратор — указатель на функцию.
- В этом примере компаратор передаётся как указатель на функцию.
- Компаратор задаёт строгий порядок: для равных значений возвращает `false`; результат сравнения должен быть согласованным.

Пример: подсчет элементов с предикатом

```
bool isEven(int x) { return x % 2 == 0; }
int filter(int* arr, int size, bool (*pred)(int)) {
    int count = 0;
    for (int i = 0; i < size; ++i)
        if (pred(arr[i])) count++;
    return count;
}
int main() {
    int data[] = {1,2,3,4,5};
    cout << filter(data, 5, isEven); // 2 (2 и 4)
    return 0;
}
```

Вопрос: что выведет этот код?

```
bool isPositive(int x) { return x > 0; }
int countIf(int* arr, int n, bool (*pred)(int)) {
    int cnt = 0;
    for (int i = 0; i < n; ++i) if (pred(arr[i])) cnt++;
    return cnt;
}
int main() {
    int a[] = {-1, 2, -3, 4};
    cout << countIf(a, 4, isPositive);
    return 0;
}
```

Вопрос: что выведет этот код?

```
bool isPositive(int x) { return x > 0; }
int countIf(int* arr, int n, bool (*pred)(int)) {
    int cnt = 0;
    for (int i = 0; i < n; ++i) if (pred(arr[i])) cnt++;
    return cnt;
}
int main() {
    int a[] = {-1, 2, -3, 4};
    cout << countIf(a, 4, isPositive);
    return 0;
}
```

Ответ: 2 (числа 2 и 4 положительные).

Указатели на функции и перегрузка

```
int max(int a, int b) { return (a > b) ? a : b; }  
double max(double a, double b) { return (a > b) ? a : b; }  
  
int (*pf_int)(int, int) = max; // выбирает версию для int  
double (*pf_double)(double, double) = max; // версию для double
```

- При инициализации указателя компилятор выбирает нужную перегрузку по типам.
- Если неоднозначность — ошибка компиляции.

std::function (кратко)

- std::function — обёртка для функций и других совместимых вызываемых объектов (функции, лямбды, функторы).
- Более гибкий, чем указатель на функцию.

```
#include <functional>
std::function<int(int,int)> f = add;
cout << f(2,3); // 5
```

- Подробно изучим в весеннем семестре (лямбды).

Итог по указателям на функции

- Позволяют динамически выбирать поведение.
- Используются для callback-вызовов.
- Синтаксис: тип (*имя) (параметры).
- `using` упрощает объявление.
- `std::function` — более мощная альтернатива (будет позже).
- Применение: сортировка, фильтрация, обработка событий.

Чтение сложных объявлений: правило

Правило «изнутри наружу»

- 1 Найдите идентификатор (имя переменной).
- 2 Читайте объявление, двигаясь от идентификатора наружу, чередуя направо и налево.
- 3 Приоритет: скобки, [] (массив), () (функция), * (указатель). [] и () связываются с именем сильнее, чем *; группирующие скобки меняют порядок чтения.

Пример

`int (*pf)(int, int);` — начните с `pf`, затем `*` → указатель, затем `(int, int)` → функция, принимающая два `int`, затем `int` → возвращает `int`.

Пример 1: `int (*pf)(int, int)`

- ❶ Идентификатор: `pf`.
- ❷ Смотрим вправо: `)` — нет.
- ❸ Смотрим влево: `*` — указатель.
- ❹ Выходим за скобки: направо `(int, int)` — функция с двумя параметрами `int`.
- ❺ Слева `int` — возвращает `int`.

Итог: `pf` — указатель на функцию, возвращающую `int` и принимающую два `int`.

Пример 2: `int (*daytab)[13]`

- ❶ Идентификатор: `daytab`.
- ❷ Вправо: `)` — нет.
- ❸ Влево: `*` — указатель.
- ❹ Выходим за скобки: направо `[13]` — массив из 13 элементов.
- ❺ Слева `int` — тип элементов массива.

Итог: `daytab` — указатель на массив из 13 элементов типа `int`.

Пример 3: `void (*comp)()`

- ❶ Идентификатор: `comp`.
- ❷ Вправо: `)` — нет.
- ❸ Влево: `*` — указатель.
- ❹ Выходим за скобки: направо `()` — функция без параметров.
- ❺ Слева `void` — возвращает `void`.

Итог: `comp` — указатель на функцию без параметров, возвращающую `void`.

Различие: `int *a[10]` vs `int (*a)[10]`

```
int *a[10];    // массив из 10 указателей на int
int (*a)[10]; // указатель на массив из 10 int
```

- Скобки меняют смысл.
- `int *a[10]` — сначала `[]` (массив), затем `*` (указатель).
- `int (*a)[10]` — сначала `*` (указатель), затем `[]` (массив).

Вопрос: что означает `int (*(*fp)(int))[10]`?

- Идентификатор: `fp`.
- Влево: `*` — указатель.
- Выходим за скобки: направо `(int)` — функция с параметром `int`.
- Возвращаемое значение: `*`... — указатель.
- Далее: `[10]` — массив из 10 элементов.
- Тип элементов: `int`.

Вопрос: что означает `int (*(*fp)(int))[10]`?

- Идентификатор: `fp`.
- Влево: `*` — указатель.
- Выходим за скобки: направо `(int)` — функция с параметром `int`.
- Возвращаемое значение: `*`... — указатель.
- Далее: `[10]` — массив из 10 элементов.
- Тип элементов: `int`.

Ответ: `fp` — указатель на функцию, принимающую `int`, возвращающую указатель на массив из 10 элементов типа `int`.

Использование `using` для сложных объявлений

```
// Без using:  
int ((*fp)(int))[10];  
  
// С using:  
using Arr = int[10];          // массив из 10 int  
using Func = Arr* (*)(int); // указатель на функцию, возвращающую Arr*  
Func fp = ...; // объявление становится читаемым
```

- `using` разбивает сложное объявление на части.
- Улучшает читаемость и поддерживаемость кода.

- Углубили знания об указателях: указатели на указатели, массивы указателей.
- Познакомились с `std::array` — безопасной альтернативой C-массивам.
- Изучили `std::span` — невладеющее представление для непрерывных последовательностей.
- Научились использовать указатели на функции для callback-вызовов.
- Освоили правило чтения сложных объявлений.

Ключевые нововведения и рекомендации

Стандарт	Нововведение
C++11	<code>std::array</code> , <code>nullptr</code> , range-based for
C++20	<code>std::span</code>

- Предпочитайте `std::array` C-массивам.
- Используйте `std::span` для передачи подмассивов без копирования.
- Указатели на функции — мощный инструмент для динамического выбора поведения.

Что дальше?

- Лекция №7: Алгоритмы сортировки и O-нотация.
- На практике: закрепление работы с `std::array`, `std::span`, указателями на функции.

Вопросы?