Программирование на C++

Массивы, указатели, ссылки.

Лекция 3

ПМИ 2 курс

Демяненко Я.М.

ЮФУ 2026

Перечисления: зачем они нужны

Перечисление — это тип, значения которого — именованные целые константы.

Зачем:

- Код становится читаемым: DayOfWeek::MON понятнее, чем 0.
- Компилятор проверяет, что вы не присвоили «не то» значение.
- Удобно использовать в switch.

```
enum DayOfWeek { MON, TUE, WED, THU, FRI, SAT, SUN };
// MON = 0, TUE = 1, WED = 2, ...

enum DayOfWeek { MON, TUE = 3, WED, THU, FRI, SAT, SUN };
// MON = 0, TUE = 3, WED = 4, ...
```

Это старый стиль (C++98). В **современном C++** (начиная с C++ 11) используется **enum class**

enum class: современный стиль (C++11)

Проблема старого enum: имена экспортируются во внешнюю область видимости → коллизии.

```
enum Color { RED, GREEN };
enum TrafficLight { RED, YELLOW, GREEN }; // ошибка: RED уже определён
```

Решение — enum class

```
enum class DayOfWeek { MON, TUE, WED, THU, FRI, SAT = 6, SUN };

// Ошибка: WED нет в области видимости
int wday = WED;

// Ошибка: нет неявного преобразования к int
int day = DayOfWeek::WED;

// Правильно:
DayOfWeek dow = DayOfWeek::FRI;
int day = static_cast<int>(DayOfWeek::WED); // явно
DayOfWeek sat = static_cast<DayOfWeek>(6);  // явно
```

В современном C++ enum class — выбор по умолчанию.

Обычный enum — только для совместимости с C.

enum class в switch

```
DayOfWeek day = DayOfWeek::MON;

switch (day) {
    case DayOfWeek::MON: std::cout << "Monday\n";    break;
    case DayOfWeek::TUE: std::cout << "Tuesday\n";   break;
    case DayOfWeek::WED: std::cout << "Wednesday\n"; break;
    case DayOfWeek::THU: std::cout << "Thursday\n";  break;
    case DayOfWeek::FRI: std::cout << "Friday\n";    break;
    case DayOfWeek::SAT: std::cout << "Saturday\n";  break;
    case DayOfWeek::SUN: std::cout << "Sunday\n";    break;
}
```

Полезно знать:
Если в switch не обработать какое-то значение enum class,
компилятор может выдать предупреждение (-Wswitch).

Это защита:
добавили новый элемент — компилятор напомнит обновить все switch.

Дополнение (C++20)

```
void f() {
    using enum DayOfWeek; // можно писать MON вместо DayOfWeek::MON
    DayOfWeek d = MON;
}
```

Python vs C++: главное отличие — Как хранятся данные

	Python	C++
Переменная	Имя, ссылающееся на объект	Сам объект в памяти
Присваивание a = b	Оба имени → один объект	Копирование значения (но не для массивов)
Список / массив	Объект в куче, знает размер	Непрерывная область памяти, не знает размер
Адрес	id(x)	&x
Указатели	Неявно внутри интерпретатора	Явные: int* p
Ссылки	Все имена — ссылки	Явные: int& r

Ключевая мысль:
В Python переменная — это имя.
В C++ переменная — это сам объект, а указатель и ссылка — отдельные сущности.

Сравните

Вспомните что будет в Python после

```
a = [1,2]
b = a
b.append(3)
```

```
[1, 2, 3]
[1, 2, 3]
```

В C++

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

ошибка компиляции

Массивы в C++: C-массивы и современные контейнеры

C-массивы (низкоуровневые)	**Современные контейнеры**
• Элементы индексируются с нуля • **Нет контроля выхода за пределы массива** • Высокая скорость (как следствие предыдущего) • Не хранят свой размер • Не копируются через присваивание	• Элементы индексируются с нуля • Знают свой размер (.size()) • std::array<T, N> — размер известен на этапе компиляции (C++11) • std::vector<T> — размер динамический • Копируются через присваивание • Не деградируют до указателя

С-массивы: синтаксис

```
// Создание массива из 5 элементов типа int
int b[5];

// Создание и инициализация массива из 4 элементов
int a[] = {2, 3, 5, 7};

// Запись в последний элемент
a[3] = 1;

int c[4];

// Так нельзя: массивы не копируются через =
c = a; // ошибка компиляции

// Только поэлементно:
for (int i = 0; i < 4; ++i) c[i] = a[i];
```

Для std::array и std::vector
копирование через = работает

Размеры массивов

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

// **Старый способ**:
Так как массивы в C/C++ не помнят своего размера,
то его необходимо определять вручную

```
int size = sizeof(a) / sizeof(a[0]);
```
или
```
int size = sizeof(arr2) / sizeof(int);

// sizeof(a) возвращает размер массива в байтах
// sizeof(a[0]) возвращает размер элемента массива
// sizeof(int) возвращает размер элемента массива
```

// **C++17**:
```
#include <iterator>
std::cout << std::size(a); // 5
```

sizeof работает только там, где массив ещё не превратился в указатель.
Внутри функции sizeof(a) даст размер указателя, а не массива.

Цикл по массиву (for)

```
int a[10];
int n = 10;

// Классический for
for (int i = 0; i < n; ++i) {
   std::cout << i << " array element ";
   std::cin >> a[i];
}

// Range-based for или foreach (C++11)
for (int x : a)         std::cout << x; // копия, только чтение
for (int& x : a)        x += 1;         // ссылка, можно менять
for (const auto& x : a)  std::cout << x; // ссылка, только чтение
```

Рекомендация, если **не нужно менять**:
const auto& —не копирует и не даёт случайно изменить.

Передача массива в функцию

Массив в функции: **проблема размера**

```
void f(int a[], int len) { // len нужно передавать явно
    for (int i = 0; i < len; ++i)
        std::cout << a[i] << ' ';
}

int arr[] = {1, 3, 5, 6, 1};
f(arr, 5);
```

Проблема:
В C/C++ массив передаётся по указателю,
sizeof внутри функции бесполезен.

Современное решение (C++20)

```
#include <span>

void f(std::span<int> a) {
    for (auto x : a) std::cout << x << ' ';
    std::cout << a.size(); // размер сохраняется
}

int arr[] = {1, 2, 3, 4, 5};
f(arr); // размер выводится автоматически
```

std::span не владеет данными — это «вид» на непрерывную область памяти

const-массивы в функциях

Цель: Защита массива от изменений

```
// Функция не должна менять массив:
int check_null(const int a[], int n);
```

Принцип наименьших привилегий:
Если функция не должна менять данные — пометьте данные как const.
Это защита от ошибок и подсказка читателю.

Двумерные массивы

Двумерные массивы определяются как массив массивов.
Вот как будет выглядеть двумерный массив int a[3][4].

…	**a[ ][0]**	**a[ ][1]**	**a[ ][2]**	**a[ ][3]**
a[0][ ]	a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][ ]	a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][ ]	a[2][0]	a[2][1]	a[2][2]	a[2][3]

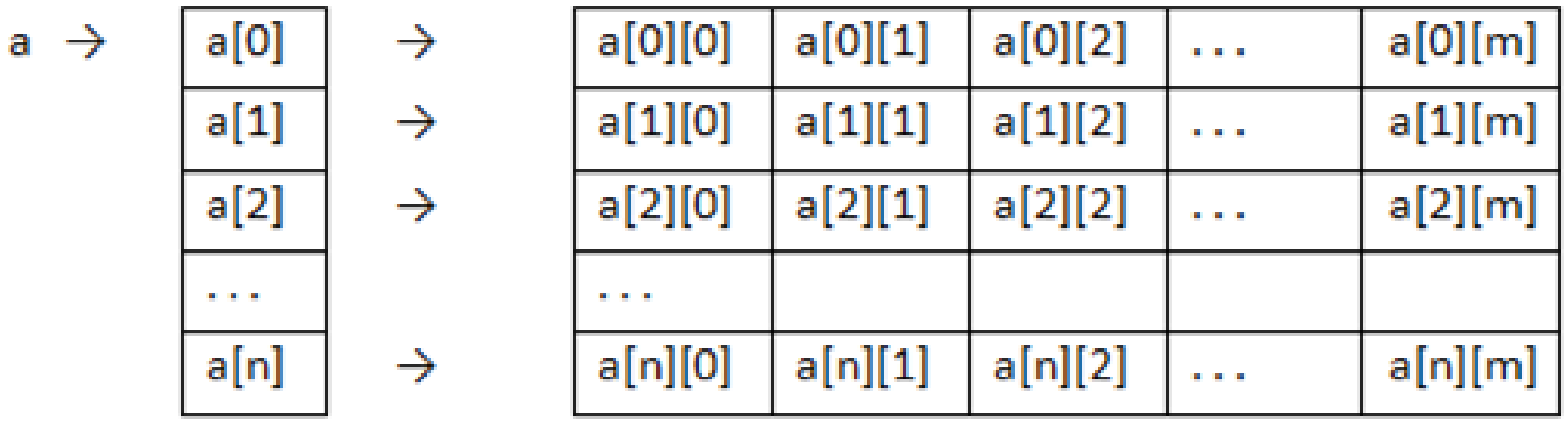
Передача в функции двумерных массивов

```
void print(int a[3][4], int m, int n) {
   for (int i = 0; i < n; i++)
      ...
}
```

При передаче массива в функцию сохранится размер массива **a[][4]**: размер внешнего массива потеряется, а размер внутренних массивов сохранится.

Современное (C++23):
std::mdspan — многомерный span. Пока можно не использовать, но знать полезно.

Организация двумерного массива



Двумерный массив — это непрерывная область памяти, где строки идут одна за другой.

a[i][j] эквивалентно *(*(a + i) + j) — но использовать такую запись постоянно не нужно

Указатель

C++ унаследовал от C возможность работы на низком уровне, например, с адресами памяти

Пусть мы имеем ячейку памяти int i = 5;

Объявление
int *p;
вводит **указатель**, то есть переменную, которая **может хранить адрес ячейки** памяти с любым int, например, i

```
p = &i;     // В указателе p хранится адрес ячейки памяти i
*p=6;       //разыменование указателя p
cout<<i;  //будет выведено 6
```



Сравнение с Python

Python	C++
id(x) — идентификатор объекта	&x — адрес в памяти
Все переменные — ссылки на объекты	Обычная переменная — сам объект
Указателей нет	int* p — явный указатель

Нулевой указатель

Можно использовать **нулевой указатель**, чтобы показать, что переменная-указатель пока не хранит никакого адреса

В современном C++ (C++11 и новее) используется только nullptr

Способ	Статус
p = NULL;	Устаревший макрос из C
p = 0;	Устаревшая практика C++98
p = nullptr;	Правильно (C++11)

```
void f(int);
void f(char*);

f(NULL);    // вызовет f(int) или ошибка — плохо
f(nullptr); // вызовет f(char*) — правильно
```

Объявление vs разыменование

Два смысла символа *

- Если * после типа — это объявление указателя
- Если * перед именем — это разыменование

```
int* p;   // объявление: p — указатель на int
*p = 6;   // разыменование: записать 6 по адресу p
```



Указатели и ссылки

```
int i = 5;

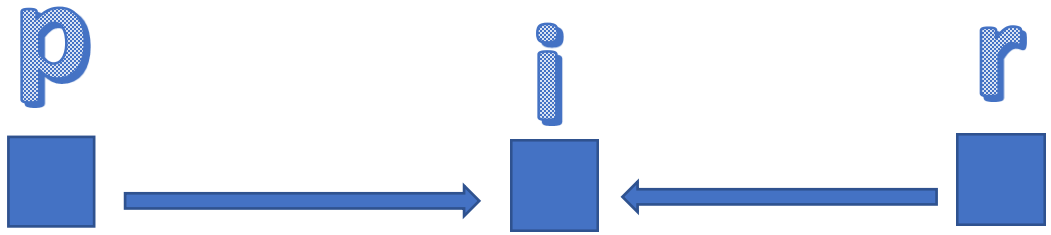
// Указатель
int* p = &i;
*p = 6;

// Ссылка
int& r = i; // r — второе имя i
r = 7;      // i тоже стало 7
```

Если **&** пишется **после** названия типа, то это **ссылка**.
Если **&** пишется **перед** именем переменной, то это **адрес** этой переменной.
Ссылку можно трактовать, как **указатель**, который **постоянно** находится в **разыменованном** состоянии

Ссылка (int&)	Указатель (int*)
Всегда инициализируется	Может быть неинициализирован
Не может быть nullptr	Может быть nullptr
Нельзя переприсвоить	Можно переприсвоить
Синтаксис как у обычной переменной	Требует * и &

Ключевая мысль:
Ссылка — это **второе имя** (псевдоним) объекта.
Указатель — это «**адрес** объекта».



Передача параметров в функции

```
void byValue(int x)    { x++; }          // копия, оригинал не меняется
void byRef(int& x)     { x++; }          // ссылка, оригинал меняется
void byPtr(int* x)     { (*x)++; }       // указатель, оригинал меняется
void byConstRef(const int& x) { }  // ссылка, менять нельзя
```

```
int i = 5;
byValue(i);  // i == 5
byRef(i);    // i == 6
byPtr(&i);   // i == 7
```

Производительность при передаче по ссылке и по указателю **одинаковая**.

Рекомендации:
Маленькие типы (int, double) — по значению.
Большие объекты не меняем (std::string, std::vector) — по const&
Большие объекты меняем— по & или *.

Указатель void*

```
void *p;    // указатель на область памяти

int i = 5;
p = &i;

double d = 3.14;
p = &d;
```
т.е. **p может хранить адрес любого объекта**

Ограничение:
Нельзя разыменовать void* напрямую.

```
*p = 6; // ошибка компиляции

// Нужно явное приведение:

// Явное приведение к типу int* в стиле C
*(int*)p = 6;

// Использование представляет опасность, если i не является int
// Стиль C++ более явно заявляет об этой опасности:

*static_cast<int*>(p) = 6;

// но работает точно так же, как и выше
```

Современный взгляд:
void* теряет информацию о типе.
В современном C++ старайтесь использовать шаблоны или std::variant.

Приведение типов

Невозможно выполнить:

```
int *pa;
double *pb;
pa = pb;
pb = pa;
```

Но можно с помощью явного приведения типов
в **стиле С** (int*)p // в С++ желательно не использовать
или
в **стиле С++**, но **не static_cast, а reinterpret_cast**

```
pa = reinterpret_cast< int * > (pb); // явно и опасно
```

Приведение	Для чего
static_cast	Типа int↔double
dynamic_cast	Полиморфные типы (будет позже)
const_cast	Снятие/добавление const
reinterpret_cast	Переинтерпретация битов (опасно)

Определение типов: using

```
using Uint = unsigned int;
using byte = unsigned char;

using fType = double (*)(double x);  // указатель на функцию
using Arr = int[3];                  // массив из 3 int
using Matr = int[3][4];              // матрица 3×4
```

Сравнение:

```
typedef double (*fType)(double x); // старый стиль
using fType = double (*)(double x); // новый стиль (C++11)
```

Рекомендация:
В современном C++ предпочитайте using. Он читается слева направо и работает с шаблонами.

Указатели на структуры

```
struct Person {
    std::string name;
    int age;
};

Person p {"Иванов", 19};
Person* pp = &p;

(*pp).age = 20; // так можно
pp->age = 20;   // так удобнее
```

Современный взгляд:

Для доступа к полям обычно используют **ссылки** или **умные указатели** (будут позже).

Голые **указатели** на структуры — **низкоуровневый** стиль.

Указатели и константность

```
int i = 5;
int *p = &i;

const int *cp = &i;   // Указатель на константу
cout << *cp;
(*cp)++;                        // Ошибка компиляции
```

Это используется при передаче аргументов в функции.
Объявление const int *p заведомо не позволяет написать функцию, которая изменяет переменную, переданную через указатель.

```
void q(const int *p) {
  (*p)++;      // Ошибка компиляции
  cout << *p;
}
```

Константные указатели

```
int i = 5;
int j = 7;
const int n = 10;       // Обычная константа
int* const pc = &i;     // Константный указатель
pc = &j;                // Ошибка компиляции
(*pc)++;                // А здесь ошибки не будет
```

Нельзя обычному указателю присваивать адрес константы:

```
const int n = 10;
int* pn = &n;           // Ошибка компиляции
```

Нельзя константному указателю присваивать адрес переменной

```
const int *pn = &n;
*pn = 11;               // Ошибка компиляции
cout << *pn;
```

Однако возможно заставить компилятор **снять константность**:

***const_cast<int*>(pn) = 11;**

Шпаргалка: const и указатели

Запись	Что нельзя менять
int* p	ничего
const int* p	*p
int* const p	p
const int* const p	*p и p

Правило чтения:
Читайте справа налево:
const int* p —
«p — указатель на int, который const».

```
int a=100;       //два обычных объекта типа int
int b=222;

int *const P2=&a;   //Константный указатель
*P2=987;            //Менять значение разрешено
//P2=&b;            //Но изменять адрес не разрешается

const int *P1=&a; //Указатель на константу
//*P1=110;          //Менять значение нельзя
P1=&b;              //Но менять адрес разрешено

const int *const P3=&a;  //Константный указатель на константу
//*P3=155;          //Изменять нельзя ни значение
//P3=&b;            //Ни адрес к которому такой указатель привязан
```

Ссылки на константы

```
int i = 5;
int& ci = i;

const int& cci = i;        // здесь все будет нормально

const int n = 10;
int& cn = n;               // такое компилятор запретит
int& cn = const_cast<int&>(n); //осторожно
const int& ccn = n;        // правильно
```

Применение:
const T& — предпочтительный способ передачи больших объектов в функцию без копирования.

```
void print(const std::string& s); // не копирует и не меняет
```

Указатели и массивы

В C++ указатели и массивы тесно связаны

```
int a[10];

int* p = &a[0]; // адрес первого элемента
*p = 5;             // a[0] == 5
```




```
a[0]  a[1]  a[2]  a[3]  a[4]  a[5]  a[6]  a[7]  a[8]  a[9]
5
*p

p=a   // то же самое: имя массива → указатель на первый элемент
```

Важно:
Имя массива неявно преобразуется к указателю на первый элемент.
Но массив — это не указатель: sizeof(a) и sizeof(p) дадут разные результаты.

Операции при работе с указателями





Выполним следующую **операцию p++**;
Теперь указатель p указывает на следующий (второй) элемент массива.

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
5
*p

Операция p++ увеличивает адрес в зависимости от типа указателя т.е.
p++ равносильно тому, что p передвинется на sizeof(int) байт

Арифметика указателей

```
int a[10];
int* p = a;

p++;              // переход к следующему элементу
p += 1;           // то же самое
p += n;           // сдвиг на n элементов
p1 = p + n;       // адрес n-го элемента
ptrdiff_t d = p1 - p; // количество элементов между указателями
```

Важно:

p++ увеличивает адрес на sizeof(T) байт, а не на 1

Складывать указатели нельзя — это не имеет смысла

Разность указателей даёт количество элементов, а не байт

Операция p+i

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
5
*(p+0)

a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8] a[9]
5
*(p+1)

*(p+0) и a[0] - являются ссылками на первый элемент массива
*(p+1) и a[1] - являются ссылками на второй элемент массива
*(p+2) и a[2] - являются ссылками на третий элемент массива

Эквивалентность a[n] и *(a+n)

Почему массивы индексируются с нуля?

Вообще говоря, более строгая связь массивов и указателей выглядит следующим образом:
a[n] == *(a+n)

Отсюда следует вывод (крамольная истина):
В языке C/C++ массивов нет - есть только указатели!!!

Следствие 1. Понятно, почему нет контроля выхода за границы массива.

Следствие 2. Понятно, почему массивы индексируются с нуля. Это самое эффективное по этой формуле.

Следствие 3. **a[n] == *(a+n) == *(n+a) == n[a]** **но n[a] никогда не пишите**

```
int a[]={1,2,3,4,5};
int n = 2;
cout<<a[n]<<' '<< *(a+n)<<endl;
cout<< *(n+a)<<' '<< n[a]<<endl;
```

Современный взгляд:
В реальном коде используйте std::vector и .at(i) —
он проверяет границы.

Имя массива — константный указатель

Почему массив нельзя присвоить?

```
int a[10];
int b[10];

a = b; // ошибка компиляции
```

Объяснение: имя массива ведёт себя как константный указатель на первый элемент

```
int* const a; // концептуально
```

Поэтому нельзя переприсвоить имя массива

Уточнение:
Это не совсем указатель, а отдельный тип «массив из N элементов», который ведёт себя как константный указатель в большинстве выражений.
Но sizeof и &a для массивов и указателей работают по-разному.

Идиома *p++

Идиома - устойчивое выражение, которое воспринимается как единое целое.

```
int a[10];
int* p = a;

Теперь допустим, нам необходимо сделать следующее:
*p = 3;
p++;

*p++=3 // Эквивалент

Как читать?

В C/C++ унарные операции ассоциируются справа налево,
поэтому в данном случае ++ относится к указателю,
// *p++ ~ *(p++) верно!!!
// *p++ ~ (*p)++ неверно!!!
```

Современный взгляд:
Такие идиомы встречаются реже: range-based for и алгоритмы STL делают код понятнее.

Пример 1. Заполнить массив a нулями

```
int a[10];

for(int* p = a; p != a+10; *p++ = 0);       // Вариант 1: идиома *p++

for(int* p = a; p != a+10; p++)             // Вариант 2а: классический
 *p = 0;

for(int* p = a; p != a+10; ++p)             // Вариант 2б: не очень классический
 *p = 0;

std::fill(std::begin(a), std::end(a), 0);   // Вариант 3: современный (C++11)

std::ranges::fill(a, 0);                    // Вариант 4: C++20
```

Пример 2. Сумма двух массивов

Даны 2 массива int a[10], b[10].
Необходимо заполнить 3-й массив c[10] суммой элементов массивов a[10] и b[10].

```
int *pa = a, *pb = b, *pc = c;

// Ручной цикл с указателями:
//for(; pa != a + 10;)
while(pa != a + 10)
    *pc++ = *pa++ + *pb++;

// Современный вариант:
std::transform(std::begin(a), std::end(a),  std::begin(b), std::begin(c),  std::plus{});
```

Передача массива в функцию с помощью указателя

```
void InitZero(int* a, int n) {
  for(int* p = a; p != a+n;)
      *p++ = 0;
}

// Эквивалентно: int* a ~ int a[]
void InitZero(int a[], int n) { ... }
```

Современное решение:

```
void InitZero(std::span<int> a) {
   std::ranges::fill(a, 0);
}
```

Как читать сложные объявления

Правило «изнутри наружу»:
Используйте приём «**движения изнутри наружу, чередуя направо и налево**».

Примеры:

```
int (*pf)();            // pf — указатель на функцию, возвращающую int

char** argv;            // argv — указатель на указатель на char

void (*comp)();         // comp — указатель на функцию, возвращающую void

int (*daytab)[13];      // daytab — указатель на массив из 13 int
```

Современный взгляд: В C++11+ такие объявления заменяются на using и auto:

```
using FuncPtr = int(*)();

using ArrayOfFuncPtrs = FuncPtr[10];

auto x = ...; // пусть компилятор выведет тип
```

Перенасыщенные скобками объявления

```
int (*pf)();      // указатель на функцию, возвращающую int

char** argv;      // указатель на указатель на char

void (*comp)();   // указатель на функцию, возвращающую void
```

Разбор

```
void *(*(*fp1)(int))[10];

// fp1 — указатель на функцию, которая получает int

// и возвращает указатель на массив из 10 указателей на void
```

Тренировка

char (*(*x[3])())[5];

char (*(*pArrPChar ())[])();

void *(*(*fp1)(int))[10];

fp1 – указатель на функцию, которая получает аргумент типа int и возвращает указатель на массив из 10 указателей типа void;

float (*(*fp2)(int,int,float))(int);

fp2 – указатель на функцию, которая получает три аргумента (int, int и float) и возвращает указатель на функцию, получающую аргумент int и возвращающую float;

int (*(*fp3())[10])();

fp3 – функция, которая возвращает указатель на массив из 10 указателей на функции, возвращающие значения типа int

using для сложных типов

using вместо typedef

```
// Старый стиль:
typedef double (*fType)(double x);
typedef int Arr[3];
typedef int Matr[3][4];

// Новый стиль (C++11):
using fType = double (*)(double x);
using Arr = int[3];
using Matr = int[3][4];
```

Пример использования

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

FuncPtr f = add; // указатель на функцию add
```

Преимущество:
using читается слева направо и работает с шаблонами

Итоги

1. С-массивы — низкоуровневая конструкция. В реальном коде: std::vector, std::array
2. Указатели — мощный, но опасный инструмент. В современном C++ старайтесь использовать ссылки и умные указатели
3. nullptr вместо NULL и 0
4. enum class вместо enum
5. using вместо typedef
6. std::span (C++20) — безопасная замена «указатель + размер»
7. Алгоритмы STL (std::fill, std::transform) вместо ручных циклов с указателями