

Лекция 3. Структуры, перечисления, кортежи

Программирование на C++

Мехмат ЮФУ

Осенний семестр 2026

- 1 Структуры: объявление, поля, инициализация.
- 2 Передача структур в функции.
- 3 Структурное связывание (C++17).
- 4 Вложенные структуры и массивы структур.
- 5 Перечисления: `enum` и `enum class` (C++11).
- 6 Кортежи: `std::tuple`, `std::pair`.
- 7 `std::apply` (C++17).
- 8 `std::initializer_list` (C++11).
- 9 Определение типов: `using` vs `typedef`.

Зачем нужны структуры?

- Группировка связанных данных под одним именем.
- Пример: точка на плоскости имеет координаты x и y .
- Без структуры: две отдельные переменные x и y — неудобно.
- Со структурой: один объект `Point` содержит обе координаты.
- Упрощает передачу данных в функции.
- Позволяет создавать более сложные типы данных.

Аналогия

Структура — как анкета: имя, возраст, адрес — всё в одном месте.

Объявление структуры

```
struct Point {  
    double x;  
    double y;  
};
```

- `struct` — ключевое слово.
- `Point` — имя нового типа.
- Поля `x` и `y` — члены структуры.
- Точка с запятой после закрывающей скобки **обязательна!**
- Поля по умолчанию `public` (в отличие от `class`). Подробнее — весной.

Инициализация структуры

```
Point p; // значения не определены (мусор)
Point p2{}; // 0 0
Point p3{1.5}; // 1.5 0
Point p4 = {1.5, 2.3}; // агрегатная инициализация
Point p5{3.2, 4.3}; // универсальная инициализация (C++11)
Point p6{.x = 5.1, .y = 6.2}; // именованные инициализаторы (C++20)
```

- Агрегатная инициализация работает, если все поля `public`.
- Универсальная инициализация `{}` — предпочтительна.
- C++20: именованные инициализаторы (designated initializers).

```
Point p{1.0, 2.0};  
p.x = 3.0;      // изменение поля  
double y = p.y; // чтение поля  
cout << "x = " << p.x << ", y = " << p.y << endl;
```

- Доступ через оператор . (точка).
- Поля можно читать и изменять.
- Если структура передана как const, поля нельзя изменять.

Передача структуры по значению

```
void printPoint(Point p) {  
    cout << "(" << p.x << ", " << p.y << ")" << endl;  
}  
  
int main() {  
    Point p{1.0, 2.0};  
    printPoint(p); // создаётся копия  
    return 0;  
}
```

- Создаётся копия структуры.
- Для маленьких структур — ОК.
- Для больших структур — дорого.

Передача структуры по ссылке

```
void move(Point& p, double dx, double dy) {  
    p.x += dx;  
    p.y += dy;  
}  
  
int main() {  
    Point p{1.0, 2.0};  
    move(p, 0.5, 1.5);  
    cout << p.x << " " << p.y; // 1.5 3.5  
    return 0;  
}
```

- Изменения в функции влияют на оригинал.
- Нет копирования — эффективно.
- Если не нужно изменять — используйте const ссылку.

const-ссылка на структуру

```
void printPoint(const Point& p) {  
    cout << "(" << p.x << ", " << p.y << ")" << endl;  
    // p.x = 5; // ОШИБКА!  
}  
  
int main() {  
    Point p{1.0, 2.0};  
    printPoint(p);  
    return 0;  
}
```

- const защищает от изменений.
- Эффективно, безопасно.
- Рекомендуемый способ передачи больших структур.

Возврат структуры из функции

```
Point createPoint(double x, double y) {  
    return {x, y}; // возврат структуры  
}  
  
int main() {  
    Point p = createPoint(5.0, 3.0);  
    cout << p.x << " " << p.y; // 5 3  
    return 0;  
}
```

- Структуру можно возвращать из функции.
- Используется копирование или move-семантика (C++11).

Возврат ссылки на структуру

```
Point& getPoint(Point& p) {  
    return p; // возврат ссылки на существующий объект  
}  
  
int main() {  
    Point p{1.0, 2.0};  
    Point& ref = getPoint(p);  
    ref.x = 10.0;  
    cout << p.x << '\n'; // 10.0  
  
    Point copy = getPoint(p);  
    copy.x = 20.0;  
    cout << p.x; // 10.0  
    return 0;  
}
```

Осторожно

Не возвращайте ссылку на локальный объект — он будет уничтожен!

Структурное связывание (C++17)

```
struct Point { double x, y; };  
Point p{1.0, 2.0};  
auto [x, y] = p; // распаковка полей в переменные  
cout << x << " " << y; // 1 2
```

- Позволяет распаковать структуру в отдельные переменные.
- Работает с `struct`, `class`, `std::tuple`, `std::pair`.
- Переменные получают копии значений.
- Доступно с C++17.

Структурное связывание с модификаторами

```
struct Point { double x, y; };  
Point p{1.0, 2.0};  
const auto [x, y] = p; // константные копии  
auto& [rx, ry] = p;     // ссылки на поля (изменяемые)  
const auto& [crx, cry] = p; // константные ссылки  
rx = 5.0; // изменяет p.x  
cout << p.x; // 5.0
```

- Можно использовать `const`, `&`, `const&`.
- `auto&` позволяет изменять оригинал.

Структурное связывание для массивов (C++17)

```
int arr[] = {1, 2, 3, 4, 5};  
auto [a, b, c, d, e] = arr;  
cout << a << " " << c << " " << e; // 1 3 5
```

- Работает с массивами фиксированного размера.
- Количество переменных должно совпадать с размером массива.

Вложенные структуры

```
struct Point { double x, y; };  
struct Rectangle {  
    Point topLeft;  
    Point bottomRight;  
};  
int main() {  
    Rectangle rect{{0, 0}, {10, 10}};  
    cout << rect.topLeft.x; // 0  
    return 0;  
}
```

- Структура может содержать другие структуры.
- Инициализация: вложенные фигурные скобки.

Массив структур

```
struct Point { double x, y; };
int main() {
    Point points[3] = {{1, 2}, {3, 4}, {5, 6}};
    for (int i = 0; i < 3; ++i)
        cout << points[i].x << " " << points[i].y << endl;

    for (auto p : points) // работаем с копиями
        cout << p.x << " " << p.y << endl;

    return 0;
}
```

- Массив структур — как массив любого другого типа.
- Доступ к полям через индекс и точку.

Вопрос: что выведет этот код?

```
struct Point { int x; int y; };  
int main() {  
    Point p{3, 4};  
    auto [a, b] = p;  
    a = 10;  
    cout << p.x << " " << a;  
    return 0;  
}
```

Вопрос: что выведет этот код?

```
struct Point { int x; int y; };  
int main() {  
    Point p{3, 4};  
    auto [a, b] = p;  
    a = 10;  
    cout << p.x << " " << a;  
    return 0;  
}
```

Ответ: 3 10 (переменные a и b — копии, изменение a не влияет на p.x)

Перечисления (enum) — C-стиль

```
enum Color { RED, GREEN, BLUE };  
Color c = RED;  
int value = GREEN; // неявное преобразование к int  
cout << value; // 1
```

- Элементы — целые числа (по умолчанию с 0).
- Можно задать значения: `enum Color { RED = 1, GREEN = 2, BLUE = 4 };`.
- Элементы экспортируются во внешнюю область видимости.
- Неявное преобразование к `int` — может привести к ошибкам.

- **Конфликт имён:** два разных enum не могут иметь одинаковые имена элементов.
- **Неявное преобразование:** `Color c = RED; int i = c;` — работает, даже если это не нужно.
- **Отсутствие области видимости:** `RED` доступен везде.

Пример

```
enum A { X, Y };  
enum B { X, Z }; // ОШИБКА! X уже определён
```

enum class (C++11)

```
enum class Color { RED, GREEN, BLUE };  
Color c = Color::RED;  
// int value = Color::RED; // ОШИБКА! нет неявного преобразования  
int value = static_cast<int>(Color::RED); // явно
```

- Элементы находятся в своей области видимости (Color::).
- Нет неявного преобразования к int.
- Безопаснее и удобнее.
- Можно указать базовый тип: enum class Color : char { RED, GREEN };

Сравнение enum и enum class

Характеристика	enum	enum class
Область видимости	глобальная	внутри enum class
Неявное приведение к int	да	нет
Безопасность	низкая	высокая
Поддержка базового типа	да (можно указать)	да (можно указать)

Рекомендуется всегда использовать enum class.

enum class с базовым типом

```
enum class Color : char { RED, GREEN, BLUE };  
enum class Flags : unsigned int { FLAG1 = 1, FLAG2 = 2, FLAG3 = 4 };  
// Размер Color = 1 байт (char)  
// Размер Flags = 4 байта (unsigned int)
```

- Экономия памяти.
- Удобно для битовых флагов.

Перечисления и switch

```
enum class Color { RED, GREEN, BLUE };  
Color c = Color::RED;  
switch (c) {  
    case Color::RED:    cout << "Red"; break;  
    case Color::GREEN:  cout << "Green"; break;  
    case Color::BLUE:   cout << "Blue"; break;  
}
```

- switch отлично работает с enum class.
- Компилятор может предупредить о пропущенных вариантах.

Вопрос: что выведет этот код?

```
enum class Season { WINTER, SPRING, SUMMER, AUTUMN };  
int main() {  
    Season s = Season::WINTER;  
    cout << (int)s;  
    return 0;  
}
```

Вопрос: что выведет этот код?

```
enum class Season { WINTER, SPRING, SUMMER, AUTUMN };  
int main() {  
    Season s = Season::WINTER;  
    cout << (int)s;  
    return 0;  
}
```

Ответ: 0 (по умолчанию первый элемент равен 0, явное приведение)

Вопрос: что выведет этот код?

```
enum Color { RED, GREEN, BLUE };  
int main() {  
    Color c = GREEN;  
    int x = c + 1;  
    cout << x;  
    return 0;  
}
```

Вопрос: что выведет этот код?

```
enum Color { RED, GREEN, BLUE };  
int main() {  
    Color c = GREEN;  
    int x = c + 1;  
    cout << x;  
    return 0;  
}
```

Ответ: 2 (GREEN = 1, c + 1 = 2)

C++23: std::to_underlying

```
#include <utility>
enum class Color { RED, GREEN, BLUE };
int main() {
    Color c = Color::GREEN;
    int value = std::to_underlying(c);
    cout << value; // 1
    return 0;
}
```

- Преобразует перечисление в его базовый тип.
- Альтернатива `static_cast<int>(c)`.
- Доступно с C++23.

Кортежи (std::tuple)

```
#include <tuple>
std::tuple<int, double, std::string> t(1, 3.14, "Hello");
int a = std::get<0>(t);
double b = std::get<1>(t);
std::string s = std::get<2>(t);
```

- Гетерогенная коллекция фиксированного размера.
- Доступ по индексу через std::get<I>.
- Удобен для возврата нескольких значений из функции.

Создание кортежа

```
#include <tuple>
using namespace std;
tuple<int, double, string> t1(1, 3.14, "Hello");
auto t2 = make_tuple(2, 2.71, "World");
tuple<int, double, string> t3{3, 4.2, "C++"}; // универсальная инициализация
```

Доступ к элементам кортежа

```
tuple<int, double, string> t(1, 3.14, "Hello");  
cout << get<0>(t); // 1  
cout << get<1>(t); // 3.14  
cout << get<2>(t); // Hello  
get<0>(t) = 10;    // изменение
```

- Индексы известны на этапе компиляции.
- Нельзя использовать переменную в качестве индекса.

Доступ по типу (C++14)

```
tuple<int, double, string> t(1, 3.14, "Hello");  
cout << get<int>(t);    // 1 (если тип уникален)  
cout << get<double>(t); // 3.14  
cout << get<string>(t); // Hello
```

- Можно обращаться по типу, если тип уникален в кортеже.
- Доступно с C++14.
- Если тип встречается несколько раз — ошибка компиляции.

Распаковка кортежа с `std::tie` (C++11)

```
tuple<int, double, string> t(1, 3.14, "Hello");
int a;
double b;
string s;
std::tie(a, b, s) = t;
cout << a << " " << b << " " << s; // 1 3.14 Hello

a = 10;
cout << get<0>(t); // 1
```

- `std::tie` создаёт кортеж ссылок.
- Позволяет распаковать кортеж в существующие переменные.
- Можно игнорировать элементы с `std::ignore`.

Игнорирование элементов

```
tuple<int, double, string> t(1, 3.14, "Hello");  
int a;  
string s;  
std::tie(a, std::ignore, s) = t;  
cout << a << " " << s; // 1 Hello
```

- `std::ignore` позволяет игнорировать ненужные элементы.

Структурное связывание для кортежей (C++17)

```
tuple<int, double, string> t(1, 3.14, "Hello");  
auto [a, b, s] = t; // распаковка в новые переменные  
cout << a << " " << b << " " << s; // 1 3.14 Hello
```

- Более компактный синтаксис, чем `tie`.
- Создаёт новые переменные с копиями значений.
- Работает с C++17.

Вопрос: что выведет этот код?

```
auto t = make_tuple(2, 3.14, "Hi");  
auto [x, y, z] = t;  
x = 10;  
cout << get<0>(t) << " " << x;
```

Вопрос: что выведет этот код?

```
auto t = make_tuple(2, 3.14, "Hi");  
auto [x, y, z] = t;  
x = 10;  
cout << get<0>(t) << " " << x;
```

Ответ: 2 10 (структурное связывание создаёт копии)

std::pair — частный случай кортежа

```
#include <utility>
pair<int, double> p(1, 3.14);
cout << p.first << " " << p.second;
p.first = 10;
```

- Кортеж из двух элементов.
- Доступ через поля `first` и `second`.
- Удобен для возврата двух значений.

```
auto p1 = make_pair(5, "Hello");  
auto p2 = make_pair(3.14, 42);  
cout << p1.first << " " << p1.second; // 5 Hello  
cout << p2.first << " " << p2.second; // 3.14 42  
  
std::pair x{ 1, "Hello" };  
cout << x.first << ' ' << x.second << endl;
```

- Автоматический вывод типов.
- Удобно для быстрого создания пар.

Вопрос: что выведет этот код?

```
auto p = make_pair(5, "Hello");  
auto [a, b] = p;  
cout << a << " " << b;
```

Вопрос: что выведет этот код?

```
auto p = make_pair(5, "Hello");  
auto [a, b] = p;  
cout << a << " " << b;
```

Ответ: 5 Hello (структурное связывание для pair)

std::apply (C++17)

```
#include <tuple>
int add(int a, int b) { return a + b; }
int main() {
    auto t = make_tuple(3, 5);
    int result = std::apply(add, t);
    cout << result; // 8
    return 0;
}
```

- Применяет функцию к аргументам из кортежа.
- Удобно для вызова функций с параметрами из кортежа.
- Доступно с C++17.

apply с лямбдой

```
auto t = make_tuple(4, 7);  
auto product = [](int a, int b) { return a * b; };  
int result = std::apply(product, t);  
cout << result; // 28
```

Вопрос: что выведет этот код?

```
auto t = make_tuple(2, 3);  
auto sum = [](int a, int b) { return a + b; };  
int res = std::apply(sum, t);  
cout << res;
```

Вопрос: что выведет этот код?

```
auto t = make_tuple(2, 3);  
auto sum = [](int a, int b) { return a + b; };  
int res = std::apply(sum, t);  
cout << res;
```

Ответ: 5 (apply вызывает лямбду с элементами кортежа)

std::initializer_list (C++11)

```
#include <initializer_list>
void print(std::initializer_list<int> values) {
    for (int x : values)
        cout << x << " ";
}
int main() {
    print({1, 2, 3, 4}); // 1 2 3 4
    return 0;
}
```

- Позволяет передавать список значений в функцию.
- Используется для инициализации контейнеров.
- Только для чтения.

initializer_list и циклы

```
int sum(std::initializer_list<int> values) {  
    int s = 0;  
    for (int x : values)  
        s += x;  
    return s;  
}  
  
int main() {  
    cout << sum({ 1, 2, 3, 4, 5 }); // 15  
    return 0;  
}
```

- Можно использовать range-based for.
- Удобно для функций с переменным числом аргументов.

initializer_list с разными типами

```
#include <string>
void printAll(std::initializer_list<std::string> words) {
    for (const auto& w : words)
        cout << w << " ";
}
int main() {
    printAll({"C++", "Python", "Java"}); // C++ Python Java
    return 0;
}
```

using vs typedef

typedef (старый стиль):

```
typedef unsigned char byte;  
typedef std::vector<int> IntVec;
```

using (современный, C++11):

```
using byte = unsigned char;  
using IntVec = std::vector<int>;
```

- using читается лучше.
- using работает с шаблонами.

using с шаблонами

```
template<typename T>
using Vec = std::vector<T>;

int main() {
    Vec<int> v; // std::vector<int>
    Vec<double> w; // std::vector<double>
    return 0;
}
```

- typedef не поддерживает шаблоны.
- using позволяет создавать шаблонные псевдонимы.

using для указателей на функции

```
using Func = int (*)(int, int);  
int add(int a, int b) { return a + b; }  
int main() {  
    Func f = add;  
    cout << f(2, 3); // 5  
    return 0;  
}
```

- using упрощает объявление сложных типов.
- Указатели на функции будут подробно изучены позже.

Вопрос: что выведет этот код?

```
using IntPair = std::pair<int, int>;  
IntPair p{1, 2};  
cout << p.first << " " << p.second;
```

Вопрос: что выведет этот код?

```
using IntPair = std::pair<int, int>;  
IntPair p{1, 2};  
cout << p.first << " " << p.second;
```

Ответ: 1 2

Структуры vs классы

Характеристика	struct	class
Доступ по умолчанию	public	private
Наследование по умолчанию	public	private
Использование	пассивные типы данных	активные объекты

- В C++ struct и class почти одинаковы.
- Разница только в доступе по умолчанию.
- struct используют для простых данных, class — для сложных.

Когда использовать структуры?

- Для простых типов-носителей данных.
- Когда все поля должны быть открыты.
- Для совместимости с C (взаимодействие с C-кодом).
- Для агрегатной инициализации (инициализация с помощью фигурных скобок).

Совет

Если у структуры есть методы (поведение), рассмотрите использование `class`.

- Структуры: группировка данных, передача в функции, структурное связывание (C++17).
- Перечисления: `enum` (устаревший) и `enum class` (безопасный).
- Кортежи: `std::tuple`, `std::pair`, распаковка, `std::apply`.
- `std::initializer_list` для работы со списками значений.
- Определение типов: `using` предпочтительнее `typedef`.

Стандарт	Нововведение
C++11	enum class, std::tuple, std::initializer_list, using
C++14	доступ к кортежу по типу (get<T>)
C++17	структурное связывание, std::apply
C++20	designated initializers для структур
C++23	std::to_underlying

Что дальше?

- Следующая лекция: представление данных и побитовые операции.
- На практике: структуры, перечисления, работа с векторами структур.

Вопросы?