

**C++**

# Лекция V

Classes

# Classes

Классы

# Class

*Задаёт пользовательский тип*

```
#include <iostream>

struct MyType {
    // BODY
};

int main() {
    MyType myObject;
    std::cout
        << "sizeof(MyType): "
        << sizeof(MyType)
        << std::endl;
}
```

# Syntax

```
class-key class-head-name { member-spec }
```

`class-key` — это `class` или `struct`; влияет на дефолтный модификатор доступа к членам класса

`class-head-name` — имя класса

`member-spec` — содержит спецификаторы доступа, поля, псевдонимы типов, методы...

# Syntax

Пример 1 (struct - есть доступ к членам класса)

*class-key class-head-name { member-spec }*

```
struct Car { // class-key: struct
    // члены класса
    int fuel; // поле класса
    void start() { ... } // метод класса
    using Weight = float; // вложенный псевдоним
};

int main() {
    Car car;
    car.start(); // вызов метода класса
    int fuel = car.fuel; // доступ к полю класса
}
```

# Syntax

Пример 2 (class - отсутствует доступ к членам класса)

*class-key class-head-name { member-spec }*

```
class Car { // class-key: class
    // члены класса
    int fuel; // поле класса
    void start() { ... } // метод класса
    using Weight = float; // вложенный псевдоним
};

int main() {
    Car car;
    // car.start(); - нет доступа к методу
    // int fuel = car.fuel; - нет доступа к полю
}
```

# Спецификаторы доступа

public, private

---

```
class MyType {  
    // private members by default  
public:  
    // public members...  
private:  
    // private members...  
public:  
    // public members...  
private:  
    // private members...  
};
```



# Спецификаторы доступа

## Пример

---

```
class C {  
public:  
    int getI() const { return i_; } // public  
    int getJ() const { return j_; } // public  
private:  
    int i_ = 10; // private  
    int j_ = 11; // private  
};  
  
int main() {  
    C c;  
    c.getI(); // OK  
    c.getJ(); // OK  
    c.i_; // error  
    c.j_; // error  
}
```

# Class members

Члены класса

# Члены класса

- Поля (data members)
- Функции-члены (member functions)
- Вложенные типы (nested types):
  - псевдонимы (aliases)
  - вложенные классы и [\*перечисления]
  - [\*injected-class-name]
- [\*Элементы перечисления]
- [\*Шаблоны]

# Члены класса

```
class T {  
    int field; // data member  
    int method() { ... }; // member function  
  
    // nested types:  
    using Type = int; // alias  
    struct S { }; // nested class  
};
```

# Data members

Поля

```
struct MyType {  
    int field;  
};  
  
int main() {  
    MyType obj{10};  
    std::cout << obj.field << std::endl;  
}
```

# Data members

## Поля

```
struct MyType {  
    int field; // data member  
    int& ref;  // data member of reference type  
    int* ptr;  // data member of pointer type  
  
    int arr[2]; // array  
};  
  
int main() {  
    int i = 0;  
    MyType obj{10, i, &i, {1,2}};  
}
```

# Static data members

Статические поля (*не связаны с объектом класса*)

---

```
struct T {  
    static int n;    // declaration  
};  
  
int T::n = 10;        // definition  
  
int main()  
{  
    // forms of access:  
    int n1 = T::n; // form1  
    T t;  
    int n2 = t.n;  // form2  
}
```

# Static data members

## Проблема инициализации в теле класса

---

```
struct T {  
    static int n = 10;  
    // error: non-const static data member  
    // must be initialized out of line  
};
```

## Решения:

```
struct T {  
    inline int n = 10; // OK  
  
    // const:  
    const int m = 10; // OK only for integral types and enums  
    constexpr int p = 10; // OK, inline  
};
```



# Const static data members

## Проблема

```
struct T {  
    const static int i = 32; // OK only for integral types and enums  
};  
  
int main()  
{  
    std::cout << T::i << std::endl; // OK  
    // but...  
    const int* ptr = &T::i;  
}
```

```
error: linker command failed with exit code 1  
<source>:8: undefined reference to `T::i'
```

# Const static data members

Решение:

- use constexpr (or inline)

```
struct T {  
    constexpr static int i = 32;  
    inline static const int j = 32  
};
```

- denition in cpp-file

```
// t.hpp  
struct T {  
    static const int i;  
};  
  
// t.cpp  
const int T::i = 32;
```

# Member functions

Функции-члены класса, объявление и определение

---

```
struct Vec {  
    double x;  
    double y;  
    double norm2() { return std::sqrt(x*x + y*y); }  
    double norm1();  
};  
  
double Vec::norm1() // not inline  
{  
    return std::abs(x) + std::abs(y);  
}  
  
int main() {  
    Vec vec{3.0, 4.0};  
    assert(vec.norm2() == 5.0);  
    assert(vec.norm1() == 7.0);  
}
```

- Методы, определенные в теле класса, считаются inline

# Static member functions

Статические функции класса (*не связаны с объектом класса*)

---

```
struct T {  
    static int n() { return 10; }  
    static int m();  
};  
  
int T::m() { return 10; }  
  
int main()  
{  
    // forms of access:  
    int n1 = T::n(); // form1  
    T t;  
    int n2 = t.m();  // form2  
}
```

# Non-static member functions

## КОНСТАНТНОСТЬ

```
struct Vec {  
    double x;  
    double y;  
    double norm2() { return std::sqrt(x*x + y*y); }  
};  
  
int main() {  
    const Vec vec{3.0, 4.0}; // const object  
    assert(vec.norm2() == 5.0); // error: function not marked const  
}
```

# Non-static member functions

## КОНСТАНТНОСТЬ

```
struct Vec {  
    double x;  
    double y;  
    double norm2() const { return std::sqrt(x*x + y*y); } // mark const  
};  
  
int main() {  
    const Vec vec{3.0, 4.0}; // const object  
    assert(vec.norm2() == 5.0); // OK  
}
```

# Non-static member functions

Пример 1. Реализовать метод value()

```
struct ValueHolder {  
    int value_;  
    ...  
};  
  
int main() {  
    ValueHolder holder{10};  
    holder.value() = 11;           // OK  
    assert(holder.value() == 11);  
    assert(holder.value_ == 11);  
  
    const ValueHolder& holderRef = holder;  
    assert(holderRef.value() == 11);  
    // holderRef.value() = 12;      // Error;  
}
```

# Non-static member functions

Пример 1. Реализовать метод value()

```
struct ValueHolder {  
    int value_;  
    int& value() { return value_; }  
    int value() const { return value_; }  
};  
  
int main() {  
    ValueHolder holder{10};  
    holder.value() = 11;           // OK  
    assert(holder.value() == 11);  
    assert(holder.value_ == 11);  
  
    const ValueHolder& holderRef = holder;  
    assert(holderRef.value() == 11);  
    // holderRef.value() = 12;      // Error;  
}
```



# Non-static member functions

Что еще дает const specifier

```
struct ValueHolder {  
    int value_;  
    int value() const {  
        // Error: cannot assign within const member func  
        value_ = 1;  
        return value_;  
    }  
};  
  
int main() {  
    ValueHolder holder{10};  
    assert(holder.value() == 11);  
    const ValueHolder& holderRef = holder;  
    assert(holderRef.value() == 11);  
}
```

# Non-static member functions

## Константные методы

- Используются для константных объектов
- Используются для неконстантных объектов, если неконстантная перегрузка метода отсутствует
- Защищают поля класса от изменений
- **Не применимо к статическим функциям-членам класса**

# Pointers to members

## function members

---

```
struct T
{
    void f() const { std::cout << "f()\n"; }
    static void g() { std::cout << "g()\n"; }
};

int main()
{
    void (T::*ptrToMethod)() const = &T::f;
    void (*ptrToStaticFunc)() = T::g;

    T t;
    (t.*ptrToMethod)();
    ptrToStaticFunc();
}
```

# Pointers to members

Поля

---

```
struct T
{
    static int s;
    int n;
};
int T::s = 10;

int main()
{
    int* ptrToStaticMember = &T::s;
    int T::* ptrToMember = &T::n;

    T t { 11 };
    std::cout << *ptrToStaticMember << "\n"
               << (t.*ptrToMember);
}
```

# Pointer to class

```
struct Pair {  
    int first;  
    int second;  
};  
  
int main()  
{  
    Pair pair{1, 2};  
    Pair* ptr = &pair;  
    std::cout << ptr->first + ptr->second;  
}
```

Для доступа к членам класса используется →

# this pointer

- в теле **нестатической** функции указывает на объект, для которого этот метод был вызван

```
struct T {  
    void f() const {  
        std::cout << "f: " << this << std::endl;  
    }  
};  
  
int main()  
{  
    T t;  
    std::cout << "main: " << &t << std::endl;  
    t.f();  
}
```

```
main: 0x7ffee001da18  
f: 0x7ffee001da18
```

# this pointer

ИСПОЛЬЗОВАНИЕ → для доступа к членам класса

---

```
struct T {  
    int f() { return this->g() + this->i; }  
private:  
    int g() { return 42; }  
    int i;  
};  
  
int main()  
{  
    T t;  
    std::cout << t.f() << std::endl;  
}
```

# Constructor

Конструктор — специальный метод, используемые для инициализации объекта

---

Declaration syntax:

```
class-name ([parameter-list])
```



# Constructor

## Example

---

```
class T {  
public:  
    T() { std::cout << "T()\n"; }  
    T(int) { std::cout << "T(i)\n"; }  
    T(int, int) { std::cout << "T(i,i)\n"; }  
};  
  
int main()  
{  
    T t1();  
    T t2(1);  
    T t3(1, 1);  
}
```

# Constructor

## Example

---

```
class T {  
public:  
    T() { std::cout << "T()\n"; }  
    T(int) { std::cout << "T(i)\n"; }  
    T(int, int) { std::cout << "T(i,i)\n"; }  
};  
  
int main()  
{  
    T t1(); // function declaration  
    T t2(1);  
    T t3(1, 1);  
}
```

# Data member initialization

**Все** поля класса инициализируются до выполнения тела конструктора.

Два способа инициализации:

1. member initializer list (*список инициализации*)
2. default member initializer

# Data member initialization

```
struct D {  
    D() { std::cout << "D()\n" << std::endl; }  
};  
  
class T {  
public:  
    T(int i, int j) : i_(i), j_(j) { } // member initializer list  
  
private:  
    int i_;  
    int j_ = 3; // default member initializer  
    int k_ = 4; // default member initializer  
  
    std::vector<int> items_;  
    D d_;  
};
```

# Data member initialization

## Порядок инициализации

---

Поля инициализируются в порядке появления их объявлений в теле класса.

Изменения в порядке инициализаторов в списке инициализации не имеют никакого эффекта.

# Data member initialization

Порядок инициализации. Пример.

---

```
struct T {  
    // field 'j' is uninitialized when used here  
    T(int val) : j(2*val), i(j) { }  
    int i;  
    int j;  
};  
  
int main()  
{  
    T t(3);  
    std::cout << t.i;  
}
```

# Default constructor

Конструктор, который может быть вызван без аргументов.

```
struct T {  
    T() : i(0) {}  
    int i;  
};
```

```
struct T {  
    T(int i = 10) : i(i) {}  
    int i;  
};
```

# Default constructor

Implicitly-declared default constructor

Если пользователь не задал других конструкторов, то компилятор может создать неявно конструктор по умолчанию.

При наличии других конструкторов, можно явно дать указание компилятору сгенерировать конструктор по умолчанию:

```
struct T {  
    T() = default;  
    T(int) { }  
};
```

Либо явно дать указание не генерировать его с помощью

```
T() = delete;
```



# Copy constructor

Конструктор, у которого:

- первый параметр имеет тип `T&` или `const T&`
- либо больше параметров нет, либо все они имеют значение по умолчанию

Этот конструктор вызывается всякий раз, когда нужно инициализировать новый объект, используя объект того же типа.

Компилятор может сгенерировать конструктор копирования, если отсутствуют пользовательские конструкторы копирования.

# Destructor

Специальный метод, который вызывается, когда время жизни объекта подходит к концу.

```
struct T {  
    T(size_t n) : data_(new int[n]{}) {}  
    ~T() { delete[] data_; }  
  
    Obj obj;  
    int* data_;  
};
```

При уничтожении объекта, деструкторы полей вызываются в порядке обратном порядку инициализации.

**specifiers**

# explicit specifier

```
struct T {  
    T(int) { }  
};  
  
void f(T t) {}  
  
int main()  
{  
    f(10); // implicit conversion  
}
```

# explicit specifier

```
struct T {  
    explicit T(int) { }  
};  
  
void f(T t) {}  
  
int main()  
{  
    f(10); // error  
}
```

# friend specifier

Позволяет получить доступ к приватным членам класса из:

- функций, не являющихся членами этого класса
- других классов

# friend functions

Syntax:

*friend function-declaration*

```
struct X {  
    friend int getI(const X&);  
private:  
    int i_ = 1;  
};  
  
int getI(const X& x) { return x.i_; }
```

# friend functions

```
struct X {  
    friend void setI(X& x, int i) { x.i_ = i; }  
private:  
    int i_ = 1;  
};  
  
int main() {  
    X x;  
    setI(x, 3);  
}
```

- setI — не член класса, inline функция, external linkage



# friend functions

```
struct X {  
    friend int f() { return 32; };  
    friend int g(const X& x) { return 32; }  
private:  
    int i_ = 1;  
};  
  
int main() {  
    f(); // Error  
    X x;  
    g(x); // OK  
}
```

Компилятор находит функцию g, так как он использует знания о типах аргументов для поиска (ADL)

# friend class

Syntax:

*friend elaborated-class-specifier ;*

```
struct X {  
    friend class Y;  
private:  
    int i_ = 1;  
};  
  
class Y {  
public:  
    int getI(const X& x) { return x.i_; }  
    void setI(X& x, int i) { x.i_ = i; }  
};
```

# friends

- дружественность не транзитивна: из (А друг В, В друг С) не следует (А друг С)
- часто применяется при перегрузке операторов

# operator overloading

Позволяет использовать операторы с пользовательскими типами

# operator overloading

Function name syntax:

*operator op*

- возможно переопределить **почти все** операторы, за исключением:
  - `::` — scope resolution
  - `.` — доступ к члену класса
  - `.*` — доступ к члену класса по указателю
  - `?:` — тернарный условный оператор

# overloaded operator

Нельзя:

- ввести свой оператор: `<>`, `**`, etc.
- изменить приоритет и количество операндов
- переопределить оператор если операнды имеют фундаментальные типы

# overloaded operator

## Prefix operator

Form:

```
@a  
a.operator@() // as member  
operator@(a)  // as non-member
```

# overloaded operator

## Prefix operator

Пример:

```
class IntHolder {  
public:  
    IntHolder& operator++() { ++x_; return *this; }  
    int value() const { return x_; }  
private:  
    int x_ = 0;  
};  
  
int main() {  
    IntHolder holder;  
    std::cout << (++holder).value();  
    std::cout << (holder.operator++().value()); // alternative  
}
```



# overloaded operator

## Postfix operator

Form:

```
a@  
a.operator@(0)    // as member  
operator@(a, 0)   // as not member
```

# overloaded operator

## Postfix operator

Пример:

```
class IntHolder {  
public:  
    IntHolder operator++(int) {  
        IntHolder res = *this; ++x_; return res;  
    }  
    int value() const { return x_; }  
private:  
    int x_ = 0;  
};  
int main() {  
    IntHolder holder;  
    std::cout << (holder++).value();  
    std::cout << (holder.operator++(0).value());  
}
```

# overloaded operator

## Assignment operator

Syntax:

```
a = b  
a.operator=(b) // !only member allowed!
```

# overloaded operator

## Assignment operator

Пример:

```
class IntHolder {
public:
    IntHolder(int x) : x_(x) {}
    IntHolder& operator=(const IntHolder& other) {
        if (this == &other) return *this;
        x_ = other.x_;
        return *this;
    }
private:
    int x_ = 0;
};

int main() {
    IntHolder a{1}, b{2};
    b = a;
    b.operator=(a);
}
```

*move-assignment* пока не рассматриваем

# overloaded operator

## Function call operator

Syntax:

```
a(args...)  
a.operator()(args...) // !only member allowed!
```

# overloaded operator

## Function call operator

```
struct FunctionObject {  
    int operator()(int arg) {  
        return 2*arg;  
    }  
};  
  
int main() {  
    FunctionObject f;  
    f(10); // call like function  
    f.operator()(10);  
}
```

# overloaded operator

## Array subscript operator

Syntax:

```
a[b]  
a.operator[](b) // !only member allowed!
```

# overloaded operator

## Array subscript operator

```
struct Vector10t {  
    T array[10] = {};  
  
    T& operator[](size_t idx) { return array[idx]; }  
    const T& operator[](size_t idx) const { return array[idx]; }  
};  
  
int main() {  
    Vector10t v;  
    v[0] = T{1};  
    v[9] = T{2};  
}
```



# overloaded operator

infix operator

Syntax:

```
a @ b  
a.operator@(b)    // as member  
operator@(a, b)   // as non-member
```

# overloaded infix operator

ВВОД/ВЫВОД

```
struct Complex { int i; int j; };
std::ostream& operator<<(std::ostream& os, const Complex& c) {
    os << c.i << " " << c.j;
    return os;
}
std::istream& operator>>(std::istream& is, Complex& c) {
    is >> c.i >> c.j;
    return is;
}
int main() {
    std::stringstream ss{"1 2"};
    Complex c{};
    ss >> c; // operator>>(ss, c);
    std::cout << c; // operator<<(std::cout, c)
}
```

# overloaded infix operator

## Ввод/вывод

```
...  
std::ostream& operator<<(std::ostream& os, const Complex& c) {  
    os << c.i << " " << c.j;  
    return os;  
}  
std::istream& operator>>(std::istream& is, Complex& c) {  
    is >> c.i >> c.j;  
    return is;  
}  
...
```

- non-member функция
- добавить friend объявление, если нужен доступ к приватным полям
- возвращаемый тип позволяет стоит цепочки последовательных операций ввода/вывода

# overloaded infix operator

## Арифметика

```
Complex operator+(const Complex& lhs, int rhs) {  
    return {lhs.i + rhs, lhs.j};  
}  
Complex operator+(int lhs, const Complex& rhs) {  
    return {lhs + rhs.i, rhs.j};  
}  
  
int main() {  
    Complex c{1, 2};  
    Complex c1 = c + 1; // operator+(c, 1)  
    Complex c2 = 1 + c; // operator+(1, c)  
}
```

- Обычно non-member функции, чтобы достигнуть симметричности  $(1 + c)$  и  $(c + 1)$

# overloaded infix operator

## Операции сравнения

```
struct Complex { int i; int j; };  
  
bool operator<(const Complex& lhs, const Complex& rhs) {  
    return std::tie(lhs.i, lhs.j) < std::tie(rhs.i, rhs.j);  
}  
  
bool operator==(const Complex& lhs, const Complex& rhs) {  
    return lhs.i == rhs.i && lhs.j == rhs.j;  
}
```

- Алгоритмы стандартной библиотеки ожидают operator< и operator==
- >, >=, <= реализуются через operator<
- != реализуется через operator==

# user-defined conversion function

Function name syntax:

```
operator type // implicit and explicit conversion  
explicit operator type // explicit conversion
```

# conversion to bool

Использование объектов нашего класса в условном выражении

```
struct Complex {  
    int i;  
    int j;  
};  
  
Complex randComplex() { return {rand(), rand()}; };  
  
int main() {  
    Complex c = randComplex();  
    if (c) { // Error  
        std::cout << "not null complex";  
    }  
}
```

# conversion to bool

Использование объектов нашего класса в условном выражении

Попытка 1.

```
struct Complex {  
    int i;  
    int j;  
  
    operator bool() const { return i != 0 || j != 0; }  
};  
  
// if (Complex(1, 1)) { ... } - works!
```



# conversion to bool

## Проблемы текущей реализации

```
struct Complex {  
    int i;  
    int j;  
  
    operator bool() const { return i != 0 || j != 0; }  
};  
  
Complex randComplex() { return {rand(), rand()}; }  
  
int main() {  
    Complex c = randComplex();  
    if (c == 0) { // OK, why?  
        // ...  
    }  
}
```

# conversion to bool

Оставим только явное приведение типа

Попытка 2.

```
struct Complex {
    int i;
    int j;

    explicit operator bool() const { return i == 0 && j == 0; }
};

Complex randComplex() { return {rand(), rand()}; };

void process(bool flag);

int main() {
    Complex c = randComplex();
    if (c) { /* ... */ } // Ok
    if (c == 0) { /* ... */ } // Error as expected
    process(static_cast<bool>(c)); // Ok with static_cast – explicit conversion
}
```

# user-defined conversion function

Function name syntax:

```
operator type // implicit and explicit conversion  
explicit operator type // explicit conversion
```

Ограничения:

- В type не могут встречаться ( ) и [ ]
- type не может быть функцией или массивом

# user-defined conversion function

```
struct T {  
    operator int(*)[3]() const { /* ... */ } // Error  
  
    using arr_type = int[10];  
    operator arr_type() const { /* ... */ } // Error  
    operator arr_type*() const { /* ... */ } // OK  
  
    using func = void(int);  
    operator func() const { /* ... */ } // Error  
    operator func*() const { /* ... */ } // OK  
};
```

