C++

Fall 2022

compscicenter.ru

Башарин Егор

https://t.me/egorbasharin

Лекция V

Classes, [Enumerations]

Classes

Классы

Class

Задает пользовательский тип

```cpp
#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`; влиет 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 {
public:
    // members...
private:
    // members...
public:
    // members...
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 int m = 10;    // OK only for integral types and
  constexpr int p = 10; // OK, inline
};
```

Const static data members

Проблема

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

int main()
{
   const int* ptr = &T::i;
}
```

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

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 m
}
```

Non-static member functions

Константность

```
struct Vec {
  double x;
  double y;
  double norm2() const { return std::sqrt(x*x + y*y); }
};

int main() {
    const Vec vec{3.0, 4.0}; // const object
    assert(vec.norm2() == 5.0); // OK
}
```

Non-static member functions

Пример 1

```
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

```
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

- Используюся 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();
}
```

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

Поля класса инициализируются до выполнения тела конструктора.

Два способа инициализации:

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

Data member initialization

```
class T {
public:
    T(int i, int j) : i_(i), j_(j) { } // member initial

private:
    int i_;
    int j_ = 3; // default member initializer
    int k_ = 4; // default member initializer
};
```

Data member initialization

Порядок инициализации

Поля инициализируются в порядке появлениях их объявлений в теле класса.

Изменения порядке инициализаторов в списке инициализации не имеют никакого эффекта.

Data member initialization

Порядок инициализации. Пример.

```
struct T {
    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;
};
```

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) { }
};
```

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 вызываетя, когда время жизни объекта подходит к концу.

```
struct T {
    ~T() { }

    Obj obj;
    Vec vec;
};
```

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 (A друг B, B друг C) не следует (A друг C)
- часто применяется при перегрузке операторов

operator overloading

Позволяет использовать операторы с пользовательскими типами

operator overloading

overloaded operator

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 опрератор 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()); // alter
}
```

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[i
};

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
    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
    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 фукнции, 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, lhs.
}

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 convers
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; }
};
```

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

Использование объектов нашего класса в условном выражении

Попытка 2.

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

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

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

int main() {
    Complex c = randComplex();
    if (c == 0) { // Error
        // ...
    }
}
```

user-defined conversion function

Function name syntax:

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

Ограничения:

- В `type` не могут встречаться `()` и `[]`
- `type` не может быть фукнцией 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
};
```

Enumerations

Перечисления

Motivation

```
const int RED = 0;     // we could use #define instead of
const int GREEN = 1;   // to get the same behavior
const int BLUE = 2;

uint32_t uintColor(int color);
```

нетипобезопасно: `uintColor` может принять любой объект, который неявно преобразуется к `int`

Что хотим получить:

```
uintColor(1);         // Error
uintColor(BLUE);      // OK
```

Enumeration

```
enum Color {
    RED,
    GREEN,
    BLUE
};

uint32_t uintColor(Color color);

uintColor(1);     // Error: No matching function for call
uintColor(RED);   // OK
```

Enumeration

```
enum Color { RED, GREEN, BLUE };
```

Syntax:

```
enum-key [enum-name] [enum-base] { [enumerator-list] };
```

Перечисление:

- Задает отдельный тип
- Значения ограничены некоторым диапазоном
- Может содержать именованные константы, значения который имеют целочисленный тип. Этот тип известен как базовый тип (`underlying type`) перечисления

Enumeration

Syntax:

```
enum-key [enum-name] [enum-base] { [enumerator-list] };
```

- `enum-key`
  - unscoped: `enum`
  - scoped: `enum class`, `enum struct`
- `enum-name` — имя перечисления; может отсутствовать у unscoped-перечислений
- `enum-base` — двоеточие с целочисленным типом; указывает на базовый тип перечисления

Enumeration

Syntax:

```
enum-key [enum-name] [enum-base] { [enumerator-list] };
```

- `enumerator-list` — список идентификаторов (enumerators), перечисленных через запятую.

Enumerator syntax:

```
identifier [ = constexpr]
```

Unscoped enumeration

Form:

```
enum name { [enumerator-list] }         (1)
enum name : type { [enumerator-list] }  (2)
```

(1) Базовый тип не фиксирован и зависит от реализации:

- все элементы перечисления представимы в этом типе
- не более `int`/`unsigned int`, если соответсвующего типа достаточно

(2) Базовый тип фиксирован

Unscoped enumeration

Элементы перечисления становятся именованными константами

```
enum ShapeType {
  TRIANGLE,
  SQUARE,
};

ShapeType getShapeType();

int main() {
    switch (getShapeType()) {
    case TRIANGLE: std::cout << "Triangle\n"; break;
    case SQUARE:   std::cout << "Square\n";   break;
    }
}
```

Unscoped enumeration

Enumerator values

- Значения элементов перечисления имеют тип совпадающий с базовым
- Значение явно задается инициализатором
- Если инициализатора нет, то значение равняется значению предыдущего элемента плюс 1
- Если у первого элемента нет инициализатора, то его значение 0

Unscoped enumeration

Значения неявно приводятся к целочисленным типам

```
enum ShapeType {
  TRIANGLE,
  SQUARE,
};

int main() {
    int t = TRIANGLE; // implicitly-convertible
}
```

Unscoped enumeration

Значение целочисленного типа, типа с плавающей точкой или типа перечисления может быть преобразовано к любому типу перечисления с помощью `static_cast`

- unfixed underlying type:

```
enum AccessType { read = 1, write = 2 }; // range: 0..3

int main() {
    // OK, value in range
    AccessType ac1 = static_cast<AccessType>(3);

    // UB (since c++17), value out of range
    AccessType ac2 = static_cast<AccessType>(4);
}
```

Unscoped enumeration

Значение целочисленного типа, типа с плавающей точкой или типа перечисления может быть преобразовано к любому типу перечисления с помощью `static_cast`

- fixed underlying type:

```
// range: all values of int
enum AccessType : int { read = 1, write = 2 };

int main() {
    AccessType ac1 = static_cast<AccessType>(3); // OK
    AccessType ac2 = static_cast<AccessType>(4); // OK
}
```

Unscoped enumerations

Проблемы

- Если **базовый тип** не указан явно, то он **не фиксирован стандартом**. Поэтому знаковость и размеры типа могут меняться от компилятора к компилятору, что ведет к проблемам с **переносимостью**.
- **Неявное преобразование** к целочисленным типам.
- **Область видимости**. Имена констант из разных перечислений могут конфликтовать.

Scoped enumerations

Syntax:

```
enum struct|class name { [enumerator-list] }         (1)
enum struct|class name : type { [enumerator-list] }  (2)
```

- (1) — перечисление с базовым типом `int`
- `struct` и `class` эквивалентны
- элементы перечисления — именованные константы, которые доступны, только с указанием имени перечисления
- отсутствует неявное преобразование к интегральным типам (можно преобрзовать используя `static_cast`)

Scoped enumerations

Syntax:

```
enum struct|class name { [enumerator-list] }         (1)
enum struct|class name : type { [enumerator-list] }  (2)
```

```
enum class Color { red, green, blue };

int main() {
    int red = Color::red; // error
    int blue = static_cast<int>(Color::blue); // OK
}
```