

# С/С++: Лекция 3

Воробьев Д.В

18.09.2020

# ООП

## Принципы

- Incapsulation - свойство сокрытия данных и части функциональности  $entity_1$  от  $entity_2$ , оперирующую  $entity_1$
- Inheritance - свойство задания иерархичности между  $entity$
- Polymorphism - свойство переопределения общего интерфейса каждой иерархии entities общим интерфейсом

## class

- Пользовательский тип
- Описывает проектируемую *entity*
- Доступ по умолчанию *private*

## struct

- Пользовательский тип
- Описывает проектируемую *entity*
- Доступ по умолчанию *public*

# class, struct

```
#include <iostream>

class A {
    int x = 10;
};

int main() {
    A a;
    // CE
    std::cout << a.x;
    return 0;
}
```

```
#include <iostream>

struct A {
    int x = 10;
};

int main() {
    A a;
    // 10
    std::cout << a.x;
    return 0;
}
```

# Access specifiers

- `public` - к полю, методу можно получить доступ из вне
- `private` - к полю, методу нельзя получить запрещает доступ из вне
- `protected` - проявляется себя при наследовании (пока отложим)

# class, struct

## Incapsulation

Введение access specifiers позволяет реализовать Incapsulation

```
#include <iostream>

class A {
    private:
        int x = 10;
};

int main() {
    A a;
    // x скрыт от внешнего доступа
    std::cout << a.x;
    return 0;
}
```

# Конструктор

## Default constructor

Parameter list: пуст

При отсутствии генерируется компилятором с спецификатором *public*

```
#include <iostream>

class A {};

int main() {
    //CE: нет
    A a;
    return 0;
}
```



# Конструктор

## Copy constructor

- Parameter list: T& для класса T
- Вызывается при инициализации *entity<sub>1</sub>* *entity<sub>2</sub>*
  - ▶ инициализация
  - ▶ при передаче в функцию
  - ▶ при возврате из функции

```
#include <iostream>

class A {
public:
    A(const A& value) {}
};
```

# Copy constructor: Инициализация

```
#include <iostream>

class A {
public:
    A(const A& value) {
        std::cout << "copy";
    }
};

int main() {
    A x;
    // "copy"
    A y = x;
    return 0;
}
```

# Copy constructor: Передача

Вызывается

```
#include <iostream>

class A {
public:
    A() = default;
    A(const A& value) {
        std::cout << "copy";
    }
};

void foo(const A val) {};

int main() {
    A x;
    foo(x);
    return 0;
}
```

Не вызывается

```
#include <iostream>

class A {
public:
    A() = default;
    A(const A& value) {
        std::cout << "copy";
    }
};

void foo(const A& val) {};

int main() {
    A x;
    foo(x);
    return 0;
}
```

# Перегрузка конструкторов

```
#include <iostream>

struct A {
    A() {std::cout << 0;}
    A(int x) {std::cout << 1;}
    A(int x, int y) {std::cout << 2;}
    A(const A& x) {std::cout << 3;}
};

int main() {
    A a;
    A b(1);
    A c(1, 2);
    A d = a;
    // 0123
    return 0;
}
```

# Shadowing

```
#include <iostream>

struct A {
    int x = 10;
    void foo() {
        int x = 30;
        // 30
        std::cout << x;
    }
};

int main() {
    A a;
    a.foo();
    return 0;
}
```

```
#include <iostream>

struct A {
    int x = 10;
    void foo() {
        // 10
        std::cout << x;
    }
};

int main() {
    A a;
    a.foo();

    return 0;
}
```

# this

## this

Указатель на экземпляр класса

```
#include <iostream>

struct A {
    int x = 10;
    A() { this->foo(this->x); }
    void foo(int x) {std::cout << x;}
};

int main() {
    //10
    A a;
    return 0;
}
```

# Списки инициализации

## Проблема

```
#include <iostream>

struct A {
    const int x;
};

int main() {
    // СЕ: требуется иниц.
    A a;
    return 0;
}
```

## Решение

```
#include <iostream>

struct A {
    A(int x_) : x(x_) {}
    const int x;
};

int main() {
    A a(10);
    return 0;
}
```

## Проблема

```
#include <iostream>

class Graph {
public:
    Graph(int n_) :
        n(n_) {}

private:
    int n;
};

int main() {
    Graph a = 1 + 3 + 4;
    return 0;
}
```

## Решение

```
#include <iostream>

class Graph {
public:
    explicit Graph(int n_) :
        n(n_) {}

private:
    int n;
};

int main() {
    // CE
    Graph a = 1 + 3 + 4;
    return 0;
}
```



# Деструктор

- Вызывается по окончании *lifetime* объекта
- Нужен для освобождения ресурсов

# Деструктор

```
#include <iostream>

class A {
public:
    A() { x = new int[10]; }
    ~A() {
        std::cout << "~A";
        delete x;
    }
private:
    int* x;
};

int main() {
    A* a = new A();
    delete a;
    return 0;
}
```

# Константные методы

## Требование

Предоставить гарантию на сценарий:  
"при выполнении метода поля не изменятся"

# Константные методы

```
#include <iostream>

class A {
public:
    // CE
    void foo() const {value = 10;};
private:
    int value;
};

int main() {
    A a;
    return 0;
}
```

# Статическая переменная

```
#include <iostream>

void foo() {
    static int x = 0;
    x++;
    std::cout << x;
}

int main() {
    // 1
    foo();
    foo();
    // 2
    return 0;
}
```

# Статическое поле

```
#include <iostream>

struct A {
    static int count;
};

int A::count = 0;

int main() {
    A a;
    a.count++;
    A b;
    // 1
    std::cout << b.count;
    return 0;
}
```

# Статический метод

```
#include <iostream>

class A {
public:
    static void foo() {
        count++;
        std::cout << count;
    }
private:
    static int count;
};

int A::count = 0;

int main() {
    A::foo();
    A::foo();
    return 0;
}
```

# Статический метод: this нет

```
#include <iostream>
class A {
public:
    static void foo() {
        // CE
        std::cout << this;
    }
};

int main() {
    A a;
    a.foo();
    return 0;
}
```



# Rule of three

## Проверить

Класс содержит *entity* non-class типа:

- указатель на выделенную память
- файловый дескриптор

## Сделать

Определите для класса все три:

- assignment оператор
- default конструктор
- copy конструктор

# Rule of three: Проблема

```
struct A {  
    A() { ptr = new int(1); };  
    ~A() { delete ptr; }  
  
    int* ptr;  
};  
  
int main() {  
    A a;  
    A b = a;  
    // 2-ой free  
    return 0;  
}
```

# Rule of three:Решение

```
struct A {  
    A() { ptr = new int(1); };  
    A(const A& val) {  
        ptr = new int(*val.ptr);  
    }  
    A& operator=(const A& right) {  
        if(this == &right) { return *this; }  
        ptr = new int(*right.ptr);  
    }  
    ~A() { delete ptr; }  
  
    int* ptr;  
};  
  
int main() {  
    A a;  
    A b = a;  
    // 2-ой free отсутствует  
    return 0;  
}
```

## Перегрузка операторов

## Подчеркиваем симметрию

- Определяем вне класса
- Объявляем как *friend*

## + : как friend

```
#include <iostream>

struct A {
    A& operator +=(const A& right) {
        x += right.x;
        return *this;
    }
    friend A operator+(const A& left, const A& right);

    int x;
};

A operator+(const A& left, const A& right) {
    left += right;
    return left;
}
```

Но как метод тоже можно

## + : как метод

```
#include <iostream>

struct A {
    A& operator += (const A& right) {
        x += right.x;
        return *this;
    }
    A operator+(const A& right) {
    }

    int x;
};
```



# Сравнения

```
bool operator< (const A& lhs, const A& rhs) {  
    // реализация сравнения  
}  
bool operator> (const A& lhs, const A& rhs) {  
    return rhs < lhs;  
}  
bool operator<=(const A& lhs, const A& rhs) {  
    return !(lhs > rhs);  
}  
bool operator>=(const A& lhs, const A& rhs) {  
    return !(lhs < rhs);  
}
```

# Сравнения

```
bool operator==(const A& lhs, const A& rhs) {  
    // реализация сравнения  
}  
bool operator!=(const A& lhs, const A& rhs) {  
    return !(lhs == rhs);  
}
```

# Инкремент / Декремент

```
struct X {  
    X& operator++() {  
        // реализация  
        return *this;  
    }  
    X operator++(int) {  
        X tmp(*this);  
        operator++();  
        return tmp;  
    }  
};
```

# Вызов функции

## Функтор

Пользовательский тип, переопределивший ()

```
#include <iostream>

struct PrintNum {
    void operator()(int x) {
        std::cout << x;
    }
};

int main() {
    PrintNum x;
    x(10);
    return 0;
}
```

# Вызов функции

## Компаратор

Функтор, переопределяющий сравнения

# Вызов функции

```
#include <algorithm>
#include <iostream>
#include <vector>

class NaiveCompare {
public:
    bool operator()(int x, int y) {
        return x < y;
    }
};

int main() {
    NaiveCompare a;
    std::vector<int> v = {0, 5, 4};
    std::sort(v.begin(), v.end(), NaiveCompare() );
    for (auto x : v) {
        std::cout << x;
    }
    return 0;
}
```

# Ограничения

- Не перегружаются:
  - ▶ "::" (scope resolution)
  - ▶ "." (member access)
  - ▶ ".\*" (member access through pointer to member)
  - ▶ "?:"
- Precedence и аридность оператора не изменяются

# Наследование



# Объявление наследования

```
#include <iostream>

class Base {
public:
    void foo() {
        std::cout << "Base";
    }
};

class Derived : public Base {
};

int main() {
    Derived d;
    d.foo();
    return 0;
}
```

# Передача по ссылке на Base

```
#include <iostream>

struct Base {
    void bar() {
        std::cout << "Base";
    }
};

struct Derived : Base {
};

void foo(const Base& val) {
    val.bar();
}

int main() {
    foo(Derived());
    return 0;
}
```

# По умолчанию

## Private

```
#include <iostream>

class Base {
public:
    void foo() {
        std::cout << "Base";
    }
};

class Derived : Base {
};

int main() {
    Derived d;
    // CE
    d.foo();
    return 0;
}
```

## Public

```
#include <iostream>

struct Base {
    void foo() {
        std::cout << "Base";
    }
};

struct Derived : Base {
};

int main() {
    Derived d;

    d.foo();
    return 0;
}
```

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Figure: Взаимоотношение спецификаторов

# Not accessible

```
#include <iostream>

class Base {
    void foo() {
        std::cout << "Base";
    }
};

class Derived : public Base {
};

int main() {
    Derived d;
    d.foo();
    return 0;
}
```

Позволяет пробрасывать поля вниз по иерархии сохраняя запрет доступа извне.

# Protected

```
#include <iostream>

class A {
    protected:
        int x = 10;
};

class B : private A {
    // x -> private
};

// x-> не доступен в C
class C : public B {
    public:
        void foo() {
            std::cout << x;
        }
};
```

```
#include <iostream>

class A {
    protected:
        int x = 10;
};

class B : public A {
    // x -> protected
};

// x-> доступен в C
class C : private B {
    public:
        void foo() {
            std::cout << x;
        }
};
```

# Protected

```
#include <iostream>

class A {
    protected:
        int x = 10;
};

class B : protected A {
    // x -> protected
};

// x-> доступен в C
class C : private B {
    public:
        void foo() {
            std::cout << x;
        }
};
```



# Name hiding

```
struct Base {  
    void foo() {  
        std::cout << "Base";  
    }  
};  
  
struct Derived : Base {  
    void foo() {  
        std::cout << "Derived";  
    }  
};  
  
int main() {  
    Derived d;  
    // Derived  
    d.foo();  
    return 0;  
}
```

# Name hiding: access Base

```
struct Base {  
    void foo() {  
        std::cout << "Base";  
    }  
};  
  
struct Derived : Base {  
    public:  
    void foo() {  
        Base::foo();  
        std::cout << "Derived";  
    }  
};  
  
int main() {  
    Derived d;  
    d.foo(); // Base Derived  
    return 0;  
}
```

# Name hiding: access Base

## Проблема

```
struct A {  
    int x = 10;  
};  
  
struct B {  
    int x = 20;  
};  
  
struct C : A, B {  
    C() {  
        std::cout << x; // CE  
    }  
};  
  
int main() {  
    C c;  
    return 0;  
}
```

## Решение

```
struct A {  
    int x = 10;  
};  
  
struct B {  
    int x = 20;  
};  
  
struct C : A, B {  
    C() {  
        std::cout << A::x;  
    }  
};  
  
int main() {  
    C c;  
    return 0;  
}
```

# using

```
#include <iostream>

struct B {
    void g(char x) { std::cout << x; }
};

struct D : B {
    using B::g;
    void g(int x) { std::cout << x; }
};

int main() {
    D d;
    d.g(1); // 1
    d.g('a'); // a
    return 0;
}
```

# Конструкторы и деструкторы

## Конструкторы

Слева направо и сверху вниз

## Деструкторы

В обратном порядке

# Конструкторы и деструкторы

```
#include <iostream>

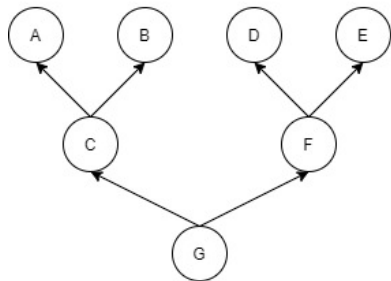
struct A {
    A() { std::cout << "A"; }
    ~A() { std::cout << "~A"; }
};

struct B {
    B() { std::cout << "B"; }
    ~B() { std::cout << "~B"; }
};

struct C : A, B {
    C() { std::cout << "C"; }
    ~C() { std::cout << "~C"; }
};

int main() {
    C c;
    return 0;
}
```

Иерархия



Вызов

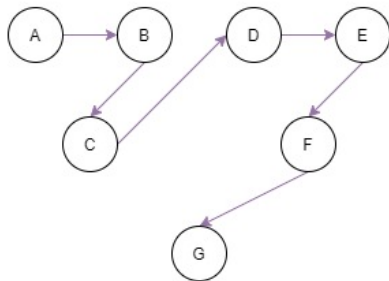


Figure: Вызов конструкторов

```
#include <iostream>

struct A {
    int x;
};

struct B : A {};

struct C : A {};

struct D : B, C {};

int main() {
    D d;
    d.x; // CE
    return 0;
}
```



# Разрешение неоднозначности с ::

```
#include <iostream>

struct A {
    int x;
};

struct B : A {};

struct C : A {};

struct D : B, C {};

int main() {
    D d;
    d.B::x;
    return 0;
}
```