

# ООП

## Классы в C++

Кафедра ИВТ и ПМ

2018

# План

Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Outline

## Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

## Прошлые темы

- ▶ Опишите парадигму ООП?
- ▶ Чем она отличается от парадигмы процедурного и модульного программирования?
- ▶ Из каких элементов строится программа написанная в объектно-ориентированном стиле?
- ▶ Что такое класс?
- ▶ Что такое объект?
- ▶ Что такое поле класса?
- ▶ Что такое метод класса?

## Прошлые темы

- ▶ Для чего нужен `this`?
- ▶ Какие модификаторы доступа могут применяться к атрибутам класса?

# Outline

Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Описание класса в C++

```
class ClassName {  
private:  
    // закрытые члены класса  
    // рекомендуется для описания полей  
public:  
    // открытые (доступные из вне) члены класса  
    // рекомендуется для описания интерфейса  
protected:  
    // защищенных члены класса  
    // доступны только наследникам  
  
    // дружественные функции и классы  
    // модификатор доступа не важен  
friend заголовок-функции;  
friend имя_класса;  
};
```

# Объекты и обращение к методам

```
class MyClass {  
    float _x;  
    public:  
        void foo() const { cout << "foo" << endl;}  
        float x() const {return _x;}  
};  
int main(){  
    MyClass c; // статическое создание объекта  
    MyClass *c1 = new MyClass(); // динамическое создание объекта  
    const unsigned n = 10;  
    MyClass cc[n]; // массив из объектов  
    float s = 0;  
    for (unsigned i = 0; i<n; i++){  
        cc[i].foo();  
        s = s + cc[i].x();  
    }  
    vector<MyClass*> v(n); // вектор из указателей на MyClass  
    for (unsigned i=0; i<n; i++)  
        v[i] = new MyClass();  
    s = 0;  
    for (unsigned i=0; i<n; i++)  
        s = s + v[i].x();  
}
```



# Outline

Прошлые темы

Классы в C++

**Конструкторы, деструкторы и операторы присваивания**

Конструкторы

Операторы присваивания

Перегрузка операторов

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Конструктор

**Конструктор** — это особый метод, инициализирующий экземпляр своего класса.

```
class MyClass{  
    float x, y;  
    public:  
  
    // Это конструктор  
    MyClass(){  
        x = 0;  
        y = 42;  
        cout << "new object";}  
};
```

...

```
MyClass o1; // new object  
MyClass o2 = new MyClass(); // new object
```

# Конструктор

- ▶ Имя конструктора совпадает с именем класса<sup>1</sup>.
- ▶ Тип возвращаемого значения не указывается - конструктор ничего не возвращает
- ▶ У конструктора может быть любое число параметров.
- ▶ У класса может быть любое число конструкторов.
- ▶ Конструкторы могут доступными (public), защищенными (protected) или закрытыми (private).
- ▶ Если не определено ни одного конструктора, компилятор создаст конструктор по умолчанию, не имеющий параметров (а также некоторые другие к. и оператор присваивания)

---

<sup>1</sup>конструктор в python называется `__init__`

# Деструктор

**Деструктор** — специальный метод класса, служащий для деинициализации объекта (например освобождения памяти).

```
class MyClass{  
    float x, y;  
    public:  
  
        // Конструктор  
        MyClass();  
  
        // Деструктор  
        ~MyClass();  
};
```

# Деструктор

- ▶ Деструктор - метод класса
- ▶ Объявление деструктора начинается с ~
- ▶ У деструкторов нет параметров и возвращаемого значения.
- ▶ В отличие от конструкторов деструктор в классе может быть только один.
- ▶ Деструктор вызывается *автоматически* при удалении объекта
- ▶ Если деструктор не определён, то он будет создан компилятором
- ▶ Такой деструктор не будет выполнять никакой работы
- ▶ Деструкторы как правило нужны если объекту необходимо освободить ресурсы, например закрыть файл; освободить память, выделенную вручную и т.п.

## Деструктор. Пример

```
class MyClass{
    float x, y;
public:
    // Конструктор
    MyClass();
    // Деструктор
    ~MyClass() {cout << "I'm Finished"; }
};

int main(){
    MyClass c1, c2;

    if ( 1 ){
        MyClass c3;}
    // вызов деструктора c3;

    cout << "End.";
    // вызов деструкторов c1 и c2
}
```

# Outline

Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Конструкторы

- ▶ конструктор умолчания (default constructor)
- ▶ конструктор преобразования (conversion constructor)
- ▶ конструктор с двумя и более параметрами (parameterized constructors)
- ▶ конструктор копирования (copy constructor)
- ▶ конструктор перемещения (move constructor)



# Конструкторы

Компилятор выбирает тот конструктор, который удовлетворяет ситуации по количеству и типам параметров.

В классе не может быть двух конструкторов с одинаковым набором параметров.

## Конструктор по умолчанию (Default constructor)

MyClass()

- ▶ Не имеет параметров.
- ▶ Может быть только один.
- ▶ Может отсутствовать.
- ▶ Создаётся компилятором, если отсутствует

Когда вызывается

```
class MyClass {...};  
...
```

```
MyClass c0 = MyClass();  
MyClass c1;  
MyClass cv[16];           // к. будет вызван 16 раз  
list<MyClass> cl(10)      // к. будет вызван 10 раз
```

# Конструктор с параметрами (Parametrized constructor)

- ▶ Принимает несколько параметров

Общий вид:

```
MyClass(T1 t1, T2 t2, T3 t3, .... )
```

T1, T2, T3, ... - некоторые типы

# Конструктор преобразования (Conversion constructor)

Общий вид:

```
MyClass(T t)
```

T - некоторый тип

- ▶ Принимает один параметр
- ▶ Тип параметра должен отличаться от самого класса
- ▶ Такой конструктор как бы преобразует один тип данных в экземпляр данного класса
- ▶ Может вызываться при инициализации объекта значением принимаемого типа

```
MyClass c = t
```

## Конструктор с параметрами (parametrized constructor). Пример

```
class Point{
    float _x, _y;
public:
    Point() { _x = 0; _y = 0; }

    Point(float x) { _x = x; }

    Point(float x, float y){
        _x = x;
        _y = y;}

    Point(const vector<float> &v){// v - вектор из двух значений
        if (v.size() == 2){_x = v[0]; _y = v[1];}
        else throw "Vector Size Error";}
    // ...
};
```

## Пример

```
int main(){
    Point p1;                // к. по умолчанию
    Point *pp1 = new Point(); // к. по умолчанию

    Point p2(2);              // к. преобразования
    Point *pp2 = new Point(42); // к. преобразования

    Point p3(1.5, -1);        // к. с параметрами
    Point *pp3 = new Point(-10.7, 127.2); // к. с параметрами

    vector<float> v = {1,2};
    Point p4(v);              // к. преобразования
    Point *pp4 = new Point(v); // к. преобразования
}
```

## Пример

```
int main(){  
    Point *pp5;           // к. не вызывается  
    Point *pp6 = &p1;     // к. не вызывается  
    Point *pp7 = &p3;     // к. не вызывается  
    vector<Point* > vp0; // к. не вызывается  
  
    Point p5 = 5;         // к. преобразования  
    Point p6 = v;         // к. преобразования  
  
    Point pp[3] = {0.2, 3, -4.2}; // к. преобразования  
    vector<Point> vp3 = {0.2, 3, -4.2}; // к. преобразования  
  
    vector<Point> vp;      // к. не вызывается  
    vector<Point> vp2(10); // к. по умолчанию  
}
```

# Конструктор копирования (copy constructor)

- ▶ Один параметр: ссылка на экземпляр данного класса
- ▶ Необходим, если копировать

```
MyClass(MyClass &c)
```



## Конструктор перемещений (move constructor)

```
MyClass(MyClass &&c)
```

# Outline

Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Оператор присваивания копированием (assignment operator)

```
MyClass& operator=(MyClass& data)
```

- ▶ используется для присваивания одного объекта текущему (существующему)
- ▶ генерируется автоматически компилятором если не объявлен
- ▶ сгенерированный компилятором, выполняет побитовое копирование
- ▶ должен очищать поля цели присваивания (и правильно обрабатывать самоприсваивание)

# Оператор присваивания перемещением (move assignment operator)

```
MyClass& operator = (const MyClass &c)
```

- ▶ используется для присваивания *временного* объекта существующему
- ▶ "забирает" временный объект "в себя"; временный объект перестаёт существовать
- ▶ генерируется автоматически компилятором если не объявлен
- ▶ сгенерированный компилятором, выполняет побитовое копирование
- ▶ должен очищать поля цели присваивания (и правильно обрабатывать самоприсваивание)

Когда вызывается?

Когда существующему объекту присваиваю значение временного объекта.

# Правило пяти

Если класс или структура определяет один из следующих методов, то они должны явным образом определить все методы:

- ▶ Конструктор копирования
- ▶ Конструктор перемещения
- ▶ Оператор присваивания копированием
- ▶ Оператор присваивания перемещением
- ▶ Деструктор

# Спецификаторы default и delete

Спецификаторы **default** и **delete** заменяют тело метода.

Спецификатор **default** означает реализацию по умолчанию (компилятором). Может быть применён только к конструкторам, деструктору и операторам присваивания.

Спецификатором **delete** помечают те методы, работать с которыми нельзя.

## Спецификаторы default и delete

```
class Foo{  
public:  
    Foo() = default;  
    Foo(const Foo&) = delete;  
    Foo operator = (const Foo& f) = delete;  
};
```

...

```
Foo o1, o2; // вызов констр. созданного компилятором  
o1 = o2; // Ошибка компиляции! Оп-р присваивания запрещён.  
Foo o3(o1); // Ошибка компиляции! Констр. копирования запрещён.
```

# Вопросы

- ▶ Зачем нужны конструкторы?
- ▶ Как запретить создание объекта на основе уже существующего?
- ▶ Как запретить любой другой способ создания объекта?
- ▶ Зачем нужны конструкторы перемещения? В чём их отличие от к. копирования?
- ▶ Когда вызывается конструктор, а когда оператор присваивания?
- ▶ Что если не описать ни одного конструктора?
- ▶ Что если не описать ни одного оператора присваивания?



# Outline

Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

**Перегрузка операторов**

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Перегрузка операторов

```
Type operator opr ( parameters );
```

type - тип возвращаемого значения

opr - обозначение оператора, например \* или =

parameters - параметры, описываются также как и для метода

Число параметров функции должно соответствовать аности оператора. Например для бинарных операторов параметра два.

Когда перегруженный оператор является методом класса, тип первого операнда должен быть указателем на данный класс (всегда \*this), а второй должен быть объявлен в списке параметров.

Операторы в C и C++

## Перегрузка операторов overloading

```
class T{  
    ...  
    public:  
        T operator+ (const T&) const { ... }  
};
```

...

```
T a,b,c;
```

```
// этот код будет транслирован компилятором в  
c = a + b;
```

```
// этот  
c = a.operator+(b);
```

# Перегрузка операторов (overloading)

Когда оператор делать методом, а когда дружественной функцией?

Унарные операторы и бинарные операторы типа “X=” рекомендуется реализовывать в виде методов класса, а прочие бинарные операторы — в виде дружественных функций. Так стоит делать потому, что оператор-метод всегда вызывается для левого операнда.

# Outline

Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

**Наследование (Inheritance)**

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Outline

Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Наследование (Inheritance)

**Наследование** - построение новых классов на основе уже существующих.

**Базовый класс (предок)** — класс на основе которого строится определение нового класса - **производного класса (потомка)**.

# Наследование. Пример

```
class B{
    int x_;
protected:
    int y;
public:
    B() { cout << "Base constructor";
        x_ = 42; y = 9000;}
    void setX(int x_) {x = x_;}
    int x() const {return x_;}
    int getY() {return y;}
    void foo() const {cout << "Base";}
};
```

```
class D : public B{
    // поле x_ унаследовано,
    // но к нему нет прямого доступа
    // к полю y есть прямой доступ
    // только внутри этого класса
public:
    // в списке инициализации возможен
    // вызов конструктора базового класса
    D() : B() { setX(1729); }
    void bar() const
        {cout << "Delivered";}
};
```

```
B b;
D d;
a.foo();    // Base
            // вызов унаследованного метода
b.foo();    // Base
b.bar();    // Delivered
b.getX();   // 1729
b.getY();   // 9000
```



# Outline

Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Перекрытие имен (overriding)

Перекрытие имён в классе = переопределение имен

```
class B {  
    public:  
        void foo(){cout << "base";}  
};
```

```
class D: public B{  
    public:  
        void foo(){cout << "delivered";}  
};
```

```
B b;  
D d;  
b.foo(); // base  
d.foo(); // delivered  
// Если нужно вызывать метод из базового класса,  
// то явно указывается имя этого класса  
d.B::foo(); // base
```

# Множественное наследование

Множественное наследование - наследование от нескольких классов одновременно.

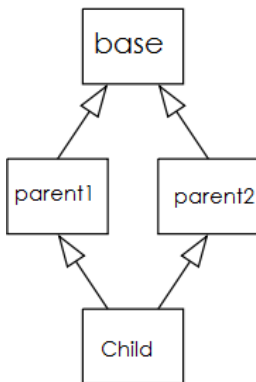
```
class Z: public X, public Y { . . . };
```

При множественном наследовании возникает проблема неоднозначности из-за совпадающих имен в базовых классах.

Поэтому лучше наследоваться от интерфейсов и классов-контейнеров.

# Deadly Diamond of Death

Проблема ромба [ [wiki](#) ]



если метод класса Child вызывает метод, определенный в классе A, а классы B и C по-своему переопределили этот метод, то от какого класса его наследовать: B или C ?

# Outline

Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Outline

Прошлые темы

Классы в C++

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Наследование (Inheritance)

Простое наследование

Перегрузка и перекрытие методов

Динамический полиморфизм

Виртуальные методы

# Виртуальные методы

**Виртуальный метод** - метод, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.

**Чистый виртуальный (абстрактный) метод** - виртуальный метод для которого не приведена реализация.

# Виртуальные методы

Зачем нужны?

- ▶ Реализует динамический полиморфизм
- ▶ Упрощает интерфейс

У целого набора классов может быть метод с одним именем и набором параметров (или несколько таких), который решает одну и ту же задачу, но специфичным для каждого класса способом. Какая конкретно реализация метода должна быть вызвана определяется во время выполнения программы.



# Раннее и позднее связывание

**Статическая типизация (раннее связывание)** —  
определение типа на этапе компиляции.

**Динамическая типизация (позднее связывание)** —  
определения типа во время выполнения программы.

# Динамический полиморфизм

Реализуется с помощью виртуальных методов.

# Вопросы

- ▶ Чем отличается переопределение виртуальных методов от переопределения виртуальных?
- ▶ Для чего нужен динамический полиморфизм? Приведите примеры.
- ▶ Как задействовать позднее связывание?

## Ссылки и литература

1. Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений. 720 с. 2010 г. 700 страниц. Теория. Примеры на C++. Картинки! Вторая половина книги - примеры OOA и OOD с UML диаграммами.
2. MSDN - Microsoft Developer Network
3. Qt 5.X. Профессиональное программирование на C++. Макс Шлее. 2015 и более поздние издания г. 928 с. Книга периодически обновляется с выходом новых версий фреймворка Qt.
4. [www.stackoverflow.com](http://www.stackoverflow.com) - система вопросов и ответов
5. [draw.io](http://draw.io) — создание диаграмм.

# Материалы курса

Слайды, вопросы к экзамену, задания, примеры

[github.com/VetrovSV/OOP](https://github.com/VetrovSV/OOP)

