

# ООП

## ООП в C++.

Черновик

Кафедра ИВТ и ПМ

2019

# План

Прошлые темы

Классы в C++

Инкапсуляция

Конструкторы, деструкторы и операторы присваивания

    Конструкторы

    Операторы присваивания

Перегрузка операторов

Статические члены классов

Отношения между классами

Другие темы

# Outline

## Прошлые темы

Классы в C++

Инкапсуляция

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Статические члены классов

Отношения между классами

Другие темы

## Прошлые темы

- ▶ Опишите парадигму ООП
- ▶ Чем она отличается от парадигмы процедурного и модульного программирования?
- ▶ Из каких элементов строится программа написанная согласно парадигме объектно-ориентированного программирования ?
- ▶ Что такое класс?
- ▶ Что такое объект?
- ▶ Чем отличается класс от объекта?
- ▶ 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++

Инкапсуляция

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Статические члены классов

Отношения между классами

Другие темы

# Описание класса в C++

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

# Объекты и обращение к методам

## Пример

```
class MyClass {
    float _x;
public:
    int n;
    void foo() const { cout << "foo" << endl;}
    float bar() const {return 42;}
};

int main(){
    MyClass c; // статическое создание объекта
    MyClass *c1 = new MyClass(); // динамическое создание объекта
    // обращение к полям
    c.n = 42;
    c1->n = 43;
    // c._x и c1->_x не доступны
    // вызов метода
    c.foo();
    c1->foo();
    // вызов метода и запись возвращаемого значения в переменную
    float a = c.x();
    float b = c1->x();
}
```



# Объекты и обращение к методам

## Пример. Массивы объектов

```
unsigned n = 20;
MyClass cc[n]; // массив из объектов

for (unsigned i = 0; i<n; i++){
    cc[i].n = rand();
    cc[i].foo();
}

// Запись отдельного объекта из массива в отдельную переменную
MyClass mc = cc[2];
mc.foo();

// динамический массив из объектов с заранее заданным количеством n
vector<MyClass> v(n);
for (unsigned i =0; i<v.size(); i++) {
    v[i].n = rand();
    v[i].foo();
}
```

# Объекты и обращение к методам

Пример. Массивы объектов

```
vector<MyClass> v;  
unsigned n = 10;  
  
// добавление объектов в динамический массив  
for (unsigned i =0; i<n; i++) {  
    MyClass mc;  
    mc.n = rand();  
    v.push_back(mc);  
}  
  
for (unsigned i =0; i<v.size(); i++) {  
    v[i].foo();  
}
```

# Объекты и обращение к методам

Пример. Массивы из указателей

```
// динамический массив из указателей на MyClass
vector<MyClass*> v;
unsigned n = 10;

// добавление объектов в динамический массив
for (unsigned i = 0; i<n; i++) {
    MyClass *mc = new MyClass();
    mc->n = rand();
    v.push_back(mc);
}

for (unsigned i=0; i<v.size(); i++) {
    v[i]->foo();
}
```

# Outline

Прошлые темы

Классы в C++

**Инкапсуляция**

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

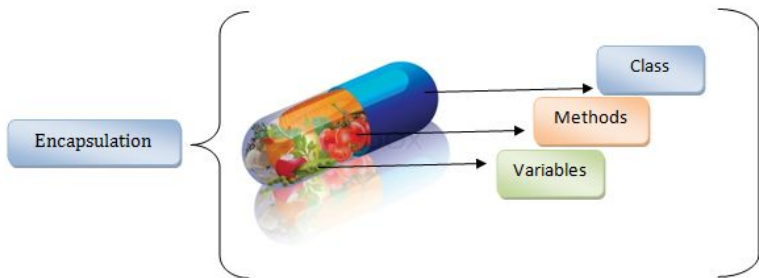
Статические члены классов

Отношения между классами

Другие темы

# Инкапсуляция

**Инкапсуляция (Encapsulation)** - это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных. Доступ к коду и данным жестко контролируется интерфейсом.



# Инкапсуляция

- ▶ Если класс содержит данные (поля), то он должен содержать и методы для работы с этими данными
- ▶ Методы должны исключать некорректную работу с данными
- ▶ Методы задающие значение полей должны проверять входные данные - *проверка предусловий*

# Инкапсуляция

- ▶ В C++ возможно объявить поля класса сделав их недоступными извне класса (область `private` и `protected`) - *принцип сокрытия*
- ▶ Для доступа к полям тогда нужно будет создать методы, объявив их в открытой области класса
- ▶ Как правило для одного поля приходится создавать два метода:
  - ▶ метод чтения – геттер (`getter`) – для получения значения поля класса
  - ▶ модифицирующий метод – сеттер (`setter`) – для задания значения полю класса. сеттер как раз включает проверку предусловий

# Инкапсуляция

## Пример

*// класс для хранения времени в 24-часовом формате*

```
class Time{
```

```
private:
```

```
    uint8_t _hours;
```

```
    uint8_t _minutes;
```

```
public:
```

```
    Time(){_hours=0; _minutes=0;}
```

```
    uint8_t hours() {return _hours;}           // геттер: возвращает часы
```

```
    uint8_t minutes() {return _minutes;}       // геттер: возвращает минуты
```

```
    void setHours(uint8_t h){                  // сеттер: задаёт часы
```

```
        if (h >= 0 && h < 24)                  // проверка предусловий
```

```
            _hours = h;        }
```

```
    void setMinutes(uint8_t m){                // сеттер: задаёт минуты
```

```
        if (m >= 0 && m < 60)                  // проверка предусловий
```

```
            _minutes = m;        }
```

```
};
```



# Outline

Прошлые темы

Классы в C++

Инкапсуляция

**Конструкторы, деструкторы и операторы присваивания**

Конструкторы

Операторы присваивания

Перегрузка операторов

Статические члены классов

Отношения между классами

Другие темы

# 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 быть 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++

Инкапсуляция

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Статические члены классов

Отношения между классами

Другие темы

# 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 p11 = Point();     // к. по умолчанию (явный вызов)
    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

    // Явный вызов конструкторов
    Point pa[3] = {Point(), Point(), Point(2.3)};
}
```

## 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++

Инкапсуляция

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Статические члены классов

Отношения между классами

Другие темы

# 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 запре
```

# 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++

Инкапсуляция

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

Статические члены классов

Отношения между классами

Другие темы

# Перегрузка операторов (operator overloading)

Код может выглядеть логичнее и читаться лучше если использовать операторы.

Без использования операторов:

```
// комплексное число
class Complex {
    float im, re;  // мнимая и действительная часть

public:
    // ...
    // Метод не изменяет текущий объект, а создаёт новый
    // поля суммируются и записываются в новый объект
    Complex plus (const Complex& b){
        Complex result;
        result.im = this->im + b.im;
        result.re = this->re + b.re;
        return result; }
};

Complex a, b;
Complex c = a.plus(b);
```

## Перегрузка операторов (operator overloading)

```
Complex a, b;  
Complex c = a.plus(b);
```

*// если бы оператор сложения (+) был перегружен для класса Complex  
// то сумма выглядела бы лаконичнее:*

```
Complex a, b;  
Complex c = a + b;
```

# Перегрузка операторов (operator overloading)

Общий вид перегружаемого оператора:

```
ReturnType operator opr ( parameters );
```

type - тип возвращаемого значения

opr - обозначение оператора (например +, \*, = и др.)

parameters - параметры, описываются также как и в функции

Такое объявление похоже на объявление функции за исключением того, что используется ключевое слово `operator` и вместо имени функции указывается обозначение оператора.

Список доступных операторов: [Операторы в С и С++](#)

# Перегрузка операторов (operator overloading)

Оператор может быть перегружен как отдельная функция и как метод класса.

Если оператор определяется как отдельная функция (часто такие операторы определяются дружественными функциями Число параметров функции должно соответствовать арности оператора. Например для бинарных операторов (+, -, \* и др) 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.

# Перегрузка операторов

## оператор как функция

- ▶ Перегрузим оператор сложения как функцию для класса `Complex`.
- ▶ Это бинарный оператор, поэтому у оператора будет два аргумента - комплексные числа.
- ▶ После сложения двух чисел должно получиться тоже комплексное число, поэтому и возвращаемый тип данных тоже `Complex`.

Общий вид оператора:

```
Complex operator + (const Complex &a, const Complex &b);
```

# Перегрузка операторов

оператор как функция

```
class Complex {  
    public:    float im, re;  
    // ...  
  
friend Complex operator + (const Complex &a, const Complex &b);  
};  
  
Complex operator + (const Complex &a, const Complex &b){  
    Complex result;  
    // этот оператор дружественный для класса Complex  
    // поэтому имеет доступ к его закрытым членам  
    result.im = a.im + b.im;  
    result.re = a.re + b.re;  
    return result;    }  
  
Complex a, b;  
Complex c = a + b;  
// аналогично вызов оператора можно записать:  
Complex d = operator+(a, b);
```

# Перегрузка операторов

## оператор как метод

- ▶ Каждому методу любого класса неявно передаётся параметр - `this`
- ▶ Поэтому первым параметром оператора определяемого внутри класса *всегда* будет объект данного класса
- ▶ тело оператора будет таким же как и в методе `plus` (на предыдущих слайдах)

```
class Complex {  
    // ...  
    Complex operator + (const Complex &b);  
    // ...  
};
```



# Перегрузка операторов

## оператор как метод

```
class Complex {  
    Complex operator + (const Complex &b){  
        Complex result;  
        // этот оператор дружелюбный для класса Complex  
        // поэтому имеет доступ к его закрытым членам  
        result.im = a.im + b.im;  
        result.re = a.re + b.re;  
        return result;    }  
};
```

```
Complex a, b;  
Complex c = a + b;  
// аналогично вызов оператора можно записать:  
Complex d = 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.

# Перегрузка операторов (overloading)

## Вопросы

- ▶ Будет ли компилироваться следующий код? Почему?

```
Complex a, b;
```

```
Complex c = a + 42;
```

# Перегрузка операторов (overloading)

## Вопросы

- ▶ Будет ли компилироваться следующий код? Почему?

```
Complex a, b;  
Complex c = a + 42;
```

- ▶ Будет ли компилироваться следующий код? Почему?

```
class Complex {  
    // ...  
    Complex operator + (double b);  
    // ...  
};  
  
Complex a;  
Complex c = a + 42;  
Complex e = 42 + a;
```

# Перегрузка операторов (overloading)

## Вопросы

- ▶ Будет ли компилироваться следующий код? Почему?

```
Complex a, b;  
Complex c = a + 42;
```

- ▶ Будет ли компилироваться следующий код? Почему?

```
class Complex {  
    // ...  
    Complex operator + (double b);  
    // ...  
};  
  
Complex a;  
Complex c = a + 42;  
Complex e = 42 + a;
```

- ▶ В чём разница между вызовами оператора в двух последних строчках?

# Перегрузка операторов (overloading)

## Вопросы

- ▶ Будет ли компилироваться следующий код? Почему?

```
Complex a, b;  
Complex c = a + 42;
```

- ▶ Будет ли компилироваться следующий код? Почему?

```
class Complex {  
    // ...  
    Complex operator + (double b);  
    // ...  
};  
  
Complex a;  
Complex c = a + 42;  
Complex e = 42 + a;
```

- ▶ В чём разница между вызовами оператора в двух последних строках?
- ▶ Как определить оператор чтобы последний вариант вызова оператора работал?

# Outline

Прошлые темы

Классы в C++

Инкапсуляция

Конструкторы, деструкторы и операторы присваивания

Конструкторы

Операторы присваивания

Перегрузка операторов

**Статические члены классов**

Отношения между классами

Другие темы

# Статические члены классов



# Outline

Прошлые темы

Классы в C++

Инкапсуляция

Конструкторы, деструкторы и операторы присваивания

    Конструкторы

    Операторы присваивания

Перегрузка операторов

Статические члены классов

**Отношения между классами**

Другие темы

# Отношения между классами

- ▶ Ассоциация  
Поле одного класса может быть указателем на другой класс
- ▶ Агрегация  
Поле одного класса может быть указателем на другой класс, при этом первый класс может управлять временем жизни агрегируемого объекта.
- ▶ Композиция Один класс является полем другого класса

Наследование рассматривается в отдельной лекции.

# Композиция

```
class Part{  
    // ...  
};  
  
class Big{  
    Part filed1;  
}
```

Big включает в себя Part. Причем при создании экземпляра класса Big будет обязательно создан экземпляр класса Part;

Если того требует мощность отношения в Big можно включить несколько экземпляров класса Part, например использовав массив.

# Агрегация

```
class Part{  
    // ...  
};  
  
class Big{  
    Part *filed1;  
}
```

Класс Big может хранить в себе указатель на Part, но экземпляр класса Big можно создать и не создавая экземпляра Part;

Если того требует мощность отношения в Big можно включить несколько экземпляров класса Part, например использовав массив:

```
vector<Part*> filed1;
```

# Агрегация

```
class Doctor{  
    // ...  
};  
  
class Patient{  
    Doctor *filed1;  
}
```

Класс Patient знает про ассоциированный с ним объект типа Doctor. Может менять ссылку на Doctor, но не может удалять экземпляр класса Doctor. Ассоциация не потеряет своего смысла, если в классе Doctor добавить указатель (или массив указателей) на Patient.

Синтаксически, эта форма ассоциации, может быть похожа на агрегацию. Однако объекты здесь более независимы.

В зависимости от своего характера ассоциация, может быть представлена и иначе.

# Outline

Прошлые темы

Классы в C++

Инкапсуляция

Конструкторы, деструкторы и операторы присваивания

    Конструкторы

    Операторы присваивания

Перегрузка операторов

Статические члены классов

Отношения между классами

Другие темы

## Не освещенные темы

- ▶ Структуры vs объединения vs классы
- ▶ inline методы
- ▶ RAII
- ▶ Почему в обработчике исключений C++ нет раздела finalize?
- ▶ Статические члены класса
- ▶ Ссылки на методы класса
- ▶ ...

## Ссылки и литература

1. [stepik.org/course/7](http://stepik.org/course/7) - Программирование на языке C++ курс по C++ и ООП от Computer Science Center (CS центр)
2. 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.
3. MSDN - Microsoft Developer Network
4. 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.
5. [www.stackoverflow.com](http://www.stackoverflow.com) - система вопросов и ответов
6. [draw.io](http://draw.io) — создание диаграмм.



# Материалы курса

Слайды, вопросы к экзамену, задания, примеры

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

