Алгоритмизация и программирование

Лекция 5 (C++)

Код до ООП (инкапсуляции)

```cpp
#include <iostream>
#include <cmath>

void move(int& this_x, int& this_y, int x, int y){
    this_x += x;
    this_y += y;
}

double dist(const int this_x, const int this_y, const int other_x, const int other_y){
    return sqrt(pow(other_x - this_x, 2) + pow(other_y - this_y, 2))
}

int main(){
    int x1 = 1, y1 = 1;
    int x2 = 2, y2 = 2;

    move(x1, y1, -1, -1);
    std::cout << dist(x1, y1, x2, y2);
}
```

Код с ООП (с инкапсуляцией)

```cpp
#include <iostream>
#include <cmath>

struct Point{    // Класс Point
    int x, y;

    void move(int x, int y){
        this->x += x;
        this->y += y;
    }

    double dist(const Point other){
        return sqrt(pow(other.x - this->x, 2) + pow(other.y - this->y, 2));
    }
};

int main(){
    Point p1{1, 1}, p2{2, 2};    // Экземпляры класса Point - объекты

    p1.move(-1, -1);
    std::cout << p1.dist(p2);
}
```

Какие бывают поля – поля объекта

```
struct Point{
    int x1, x2;       // Поле-значение
    const int x3;     // Константное поле
    int* x4;          // Поле-указатель
    int& x5;          // Поле-ссылка

    mutable int x6;
};
```

Тип данных может быть любым, не только `int`

Какие бывают поля – поля объекта

```cpp
#include <iostream>

struct Point{
    int x, y;
};

int main(){
    Point p1{1, 1}, p2{2, 2};

    std::cout << "p1: [" << p1.x << ", " << p1.y << "]\n";
    std::cout << "p2: ["  << p2.x << ", " << p2.y << "]";
}
```

Каждый **объект** имеет собственный, не зависящий от других, набор полей объекта;

Доступ к полям объекта происходит через точечную нотацию с указанием нужного **объекта**.

Какие бывают поля – поля класса

```
struct Point{
    static int x1;           // Поле-значение
    static const int x2;     // Константное поле
    static int* x3;          // Поле-указатель
    static int& x4;          // Поле-ссылка

    mutable static int x5;   // НЕ БЫВАЕТ
};
```

Тип данных может быть любым, не только `int`

Какие бывают поля – поля класса

```
#include <iostream>

struct Point{
    static int count;
};

int Point::count = 0;

int main(){
    std::cout << "Point: " << Point::count << "\n";

    Point p1, p2;
    p1.count += 1;
    std::cout << "p1: " << p1.count << "\n";
    p2.count += 1;
    std::cout << "p2: " << p2.count << "\n";
}
```

Каждый **класс** имеет единственный, набор полей класса;

Все объекты имеют доступ к полям класса;

Доступ к полям класса происходит через точечную нотацию с указанием любого **объекта** данного класса или через полное квалифицированное имя поля класса. В последнем случае нет необходимости создавать объект.

Поля класса – особенности определения

```
struct Point{
    static int x1;
    static int x2;
    static const int x3;
    static const int x4 = 10;
    // static const double x5 = 10; // Ошибка
    static const double x5;
};

int Point::x1 = 0;
int Point::x2;          // Гарантируется, что x2 == 0
const int Point::x3 = 10;
// const int Point::x4 = 10; // Можно не писать, т.к. x4 определено в
const double Point::x5 = 10; // Обязательно
```

Статические поля объявляются в теле класса, но **определяются** и **инициализируются** за его пределами;

Исключение – константы интегральных (целых) типов. Их разрешено определять и инициализировать в точке объявления;

Определение и инициализация статических полей синтаксически похожа на создание глобальной переменной, но имя поля должно быть дополнено квалификатором (именем класса);

Ключевое слово static не указывается;

Если не указать значение, то гарантируется, что оно будет нулевым, а не мусорным;

Статические поля инициализируются при старте программы в том порядке, который указан в коде.

Какие бывают методы – методы объекта

```
struct Point{
    void simple_method(){}                  // Обычный метод
    void some_method(){}                    // Не константная версия
    void some_method() const{}              // Константная версия

    virtual void virtual_method(){}         // Виртуальный метод
    virtual void abstract_method() = 0;     // Чистый виртуальный метод
};
```

Класс содержащий хотя бы один абстрактный метод называют абстрактным классом. Экземпляр абстрактного класса создать нельзя;
Специального ключевого слова для обозначения таких классов в C++ нет;

Класс содержащий только абстрактные методы и не содержащий полей называют интерфейсным классом;
Специального ключевого слова для обозначения таких классов в C++ нет;

Метод объекта – вызов метода

```
#include <iostream>

struct Point{
    void simple_method(){
        std::cout << "simple_method" << std::endl;
    }
};

int main(){
    Point p;

    p.simple_method();
}
```

Доступ к любым методам объекта происходит через точечную нотацию с указанием **объекта** у которого вызывается метод;

Методы объекта – доступ к другим членам класса из тела метода

```cpp
#include <iostream>

struct Point{
    int x, y;

    void simple_method(){
        std::cout << "--- simple_method ---" << std::endl;
        std::cout << "x: " << x << "; y: " << y << std::endl;
        other_method();
    }

    void other_method(){
        std::cout << "--- other_method ---" << std::endl;
    }
};

int main(){
    Point p1{1, 1}, p2{2, 2};

    p1.simple_method();
    p2.simple_method();
}
```

В теле **метода объекта** можно получить доступ к любым полям и метода **объекта и класса;**

Доступ к полям и методам осуществляется просто по имени;

Метод объекта работает с полями того объекта у которого он вызван;

Изменение значения поля в теле метода приводит к изменению значения этого поля в объекте;

Область видимости полей и методов – весь класс, поэтому можно обращаться к членам класса объявленным как выше, так и ниже текущего.

Методы объекта – параметры

```
#include <iostream>

struct Point{
    int x;

    void set_x(int x){
        x = x;          // x - слева от = это параметр
        this->x = x;    // x - слева от = это поле объекта
    }
};


int main(){
    Point p;

    p.set_x(10);
    std::cout << p.x << std::endl;
}
```

Методы, как и обычные функции, могут принимать и возвращать значения;

Методы объекта получают неявный первый параметр `this`:

```
void set_x(/* Point* this, */int x)
```

Указатель `this` получает адрес объекта у которого вызван метод;

При помощи `this` можно получить доступ к полям и методам текущего объекта;

Как и в случае структур для доступа можно использовать стрелочную нотацию.

Константные методы объекта

```cpp
#include <iostream>

struct Point{
    void simple_method(){
        std::cout << "simple_method" << std::endl;
    }

    void some_method(){
        std::cout << "some_method" << std::endl;
    }

    void some_method() const{
        std::cout << "const some_method" << std::endl;
    }

    void only_const_method() const{
        std::cout << "only_const_method" << std::endl;
    }
};
```

```cpp
int main(){
    Point p;
    const Point cp;

    p.simple_method();
    p.some_method();
    p.only_const_method();

    // cp.simple_method(); // Ошибка
    cp.some_method();
    cp.only_const_method();
}
```

Для любого метода объекта можно написать константную и не константную версию. У константной версии в конце прототипа должно быть ключевое слово const;

У не константного объекта можно вызывать как константный, так и не константный методы;
Если объявлены оба, то предпочтение отдаётся не константной версии;

У константного объекта можно вызывать только константный метод;

Методы объекта – доступ к другим членам класса из тела константного метода

```cpp
#include <iostream>

struct Point{
    int x, y;

    void simple_method() const{
        std::cout << "--- simple_method ---" << std::endl;
        std::cout << "x: " << x << "; y: " << y << std::endl;
        // x = 100;             // Ошибка
        const_method();
        // not_const_method(); // Ошибка
    }

    void const_method() const{
        std::cout << "--- const_method ---" << std::endl;
    }

    void not_const_method(){
        std::cout << "--- not_const_method ---" << std::endl;
    }

};

int main(){
    const Point p{1, 1};

    p.simple_method();
}
```

В теле **константного метода объекта** можно получить доступ к любым полям объекта, но только на чтение, и к любым **константным** метода **объекта;**
К полям **класса** можно получить доступ как на чтение так и на запись;
К методам **класса** тоже можно получить доступ;

В теле **не константного** метода объекта можно получить доступ к полям объекта как на чтения, так и на запись, а так же вызывать как константные, так и не константные методы объекта;
К полям **класса** можно получить доступ как на чтение так и на запись;
К методам **класса** тоже можно получить доступ.

НЕ константные методы объекта получают неявный первый параметр `this`, а константные методы - константный `this:`

```cpp
void const_method(/* const Point* this */) const;
void not_const_method(/* Point* this */);
```

Константные объекты - Семантическая константность и синтаксическая константность

```
struct Unit{
    // Состояние
    int shield
    int HP;
    int armor;

private:
    // Статистика
    int kills;
};


struct Data{
    // Состояние
    int get_data();
    void set_data(int value);

private:
    // Статистика
    int cache;
};
```

Поля класса предназначены для описания состояния объекта в текущий момент времени. Но бывает необходимость в таких полях, которые не относятся к состоянию объекта, а выполняют другую задачу, например наблюдение за объектом или оптимизация;

С точки зрения ООП объект считается константным, если не изменяется его состояние, т.е. то, что можно наблюдать снаружи. Изменение полей, значение которых невозможно получить снаружи допустимо. Это семантическая константность.

С точки зрения компилятора объект считается константным, если невозможно изменить ни одно из его полей. Это синтаксическая константность.

mutable

```
struct Unit{
    void add_kill(){
        kills++;
    }

    void add_kill() const{
        kills++;
    }

    void add_HP() {
        HP++;
    }

    void add_HP() const{
        // HP++; // Ошибка HP - константа
    }

private:
    int HP = 0;
    mutable int kills = 0;
};


int main(){
    Unit unit1;
    unit1.add_kill();
    unit1.add_HP();

    const Unit unit2;
    unit2.add_kill();
    unit2.add_HP();
}
```

Иногда бывает, что в объекте, который отмечен в коде как константа требуется изменить некоторые поля, т.е. нарушить синтаксическую константность, но при этом сохранив семантическую;

Это можно сделать, если отметить поле которое нужно менять как **mutable.** Такие поля компилятор разрешает изменять даже в константных методах.

Методы объекта – специальные методы

- Конструктор – отвечает за создание и инициализацию объекта;
- Деструктор – отвечает за корректное освобождение ресурсов и уничтожение объектов;
- Конструктор копирования – вызывается во время копирования объекта;
- Конструктор перемещения – вызывается во время копирования временного объекта, чтобы не копировать ресурсы, а забрать их;
- Оператор присваивания (копированием) – вызывается во время присваивания одного объекта другому;
- Оператор присваивания (перемещением) – вызывается во время присваивания объекту значения временного объекта, чтобы не копировать его ресурсы, а забрать;

Какие бывают методы – методы класса

```
struct Point{
    static void simple_class_method(){}   // Обычный метод класса
    // Константных и виртуальных нет
};
```

Методы класса – вызов метода

```
#include <iostream>

struct Point{
    static void simple_class_method(){
        std::cout << "simple_class_method" << std::endl;
    }
};

int main(){
    Point p;

    p.simple_class_method();
    Point::simple_class_method();
}
```

Доступ к методам класс можно получить через точечную нотацию с указанием любого объекта требуемого класса;

Но для вызова метода класса нет необходимости создавать объект, его можно вызвать через имя класса, указав полное квалифицированное имя метода.

Доступ к другим членам класса из тела статического метода

```
struct Point{
    int x;
    static const int y = 0;

    static void simple_class_method(){
        // std::cout << x << std::endl;  // Ошибка
        std::cout << y << std::endl;

        // simple_object_method();       // Ошибка
        other_class_method();
    }

    static void other_class_method(){
        std::cout << "other_class_method" << std::endl;
    }

    void simple_object_method(){
        std::cout << "simple_object_method" << std::endl;
    }
};
```

В теле **метода класса** можно получить доступ только к полям класса и методам класса, но не к полям объекта или методам объекта**;**

Доступ к другим полям и методам класса осуществляется просто по имени;

Изменение значения поля класса в теле метода приводит к изменению значения этого поля всамом классе;

Область видимости полей и методов – весь класс, поэтому можно обращаться к членам класса объявленным как выше, так и ниже текущего.

Методы класса – параметры

```
#include <iostream>

struct Point{
    static int x;
    static void set_x(int x){
        x = x;          // x - слева от = это параметр
        Point::x = x;   // x - слева от = это поле класса
    }
};

int Point::x = 0;

int main(){
    Point::set_x(10);
    std::cout << Point::x << std::endl;
}
```

Методы класса, как и обычные функции, могут принимать и возвращать значения;

Методы класса, по своей сути – обычные функции, поэтому, они НЕ получают никаких дополнительных параметров

Методы класса – способы определения

```
struct Point{
    void one(){  // Объявление и определение
        std::cout << "one" << std::endl;
    }

    void two();  // Объявление
};


// Определение
void Point::two(){
    std::cout << "two" << std::endl;
}
```

Методы могут быть объявлены и определены в месте объявления класса;

Методы могут быть объявлены в месте объявления класса, а определены за пределами объявления, далее по коду;
В этом случае имя метода должно быть дополнено квалификатором – именем класса;

Методы определённые внутри класса автоматически назначаются встраиваемыми (inline);
Если метод только объявлен внутри класса, то можно сделать его встраиваемым добавил ключевое слов inline к объявлению метода руками.

Специальные методы

Конструктор

Конструктор

```
struct Dot{
    // Присутствует реализация конструктора от компилятора
};

struct Point{
    Point(){}                    // Конструктор по умолчанию
    Point(int i){}
    Point(int i, double j){}
};

int main(){
    Dot dot;

    Point p0();  // Объекта p0 не создаётся. Это объявление функции
    Point p1;
    Point p2(10);
    Point p3(10, 10.0);
}
```

Конструктор – отвечает за создание и инициализацию объекта;

У класса может быть любое количество конструкторов;

Конструктор объявляется как метод с именем совпадающем с названием класса и без возвращаемого значения;

Конструктор автоматически вызывается во время создания объекта;

Если в теле класса не объявлен конструктор, то компилятор предоставляет реализацию по умолчанию. Этот конструктор ничего не делает;
Если есть хоть один пользовательский конструктор, то реализация от компилятора не предоставляется;

Конструктор по умолчанию

```
#include <iostream>

struct Dot{
    Dot(){
        std::cout << "Dot по умолчанию" << std::endl;
    }
};

struct Point{
    Point(int i=0){
        std::cout << "Point по умолчанию" << std::endl;
    }
};

int main(){
    Dot dot;
    Point point;
}
```

Конструктор без параметров называется конструктором по умолчанию;

Конструктор который может быть использован без параметров тоже называется конструктором по умолчанию;

Конструктор с одним параметром

```
#include <iostream>

struct Dot{
    Dot(int i){
        std::cout << "Dot по умолчанию" << std::endl;
    }
};

struct Point{
    explicit Point(int i){
        std::cout << "Point по умолчанию" << std::endl;
    }
};

void dot_func(Dot dot){}
void point_func(Point point){}

int main(){
    Dot dot = 1;          // Неявный вызов конструктора
    // Point point = 1;   // Ошибка
    Point point(1);       // Явный вызов конструктора

    dot_func(1);          // Неявное создание объекта
    // point_func(1);     // Ошибка
}
```

Конструктор с одним параметром может быть вызван компилятором неявно, если с левой стороны от знака равенства указан тип равный или преобразуемый в тип параметра конструктора;
Тоже поведение возможно во время вызова функции которая принимает объект по значению, но вместо объекта ей передают другой тип данных;

Чтобы предотвратить такое поведение нужно добавить к конструктору ключевое слово `explicit.` В этом случае вызов конструктора неявно будет запрещено;

Конструкторы с несколькими параметрами не обладают каким поведением, если только для параметров не установлено значение по умолчанию

Делегирующий конструктор

```cpp
#include <iostream>

struct Dot{
    // Делегирующий конструктор
    Dot(): Dot(10){
        std::cout << "Dot по умолчанию" << std::endl;
    }

    Dot(int i){
        std::cout << "Dot с параметрами" << std::endl;
        value = i;
    }

    int value;
};
```

Многие классы имеют несколько конструкторов, которые выполняют аналогичные действия. Делегирование работы другому конструктору позволяет уменьшить дублирование кода.

```cpp
struct Point{
    Point(){
        Point(10); // Создаётся временный объект, не делегирование
        std::cout << "Point по умолчанию" << std::endl;
    }

    Point(int i){
        std::cout << "Point с параметрами" << std::endl;
        value = i;
    }

    int value;
};


int main(){
    Dot dot;
    std::cout << "Dot: " << dot.value << std::endl;

    Point point;
    std::cout << "Point: " << point.value << std::endl;
}
```

Инициализация полей

```cpp
#include <iostream>

struct Point{
    Point(int value)
    {
        x = value;
        y = value;
        value = value;       // так не получится
        this->value = value; // нужно так
    }

    int x;
    int y;
    int value;
};


int main(){
    Point point(10);
}
```

```cpp
#include <iostream>

struct Point{
    Point(int value):
    x(value), y(value), value(value)  // Список инициализаторов
    {
        // Тело пустое
    }

    int x;
    int y;
    int value;
};


int main(){
    Point point(10);
}
```

Поля объекта инициализируются в конструкторе, чтобы привести объект к начальному допустимому состоянию;

Поля можно инициализировать в теле конструктора через обычное присваивание или в списке инициализаторов;

В списке инициализаторов можно использовать одно и тоже имя для поля объекта и параметра конструктора.

Порядок инициализации полей определяется НЕ порядком в котором указаны поля в списке инициализаторов, а порядком их объявления в теле класса;

Можно комбинировать инициализацию в теле конструктора и списки инициализаторов, но часто тело оставляю пустым.

Инициализация полей-констант и ссылок

```cpp
#include <iostream>

struct Point{
    Point(int& value){
        x = value;
        // y = value;  // Ошибка
        z = &value;
        // k = value;  // Ошибка
    }

    int x;       // Обычное значение
    const int y; // Константа
    int* z;      // Указатель
    int& k;      // Ссылка
};


int main(){
    int value = 10;
    Point point(value);
}
```

```cpp
#include <iostream>

struct Point{
    Point(int& value): x(value), y(value), z(&value), k(value)
    {
    }

    int x;       // Обычное значение
    const int y; // Константа
    int* z;      // Указатель
    int& k;      // Ссылка
};


int main(){
    int value = 10;
    Point point(value);
}
```

Тело конструктора исполняется уже после создания объекта, поэтому инициализация поля-константы будет воспринято как попытка его изменить, что недопустимо;

Инициализация поля-ссылки также не допускается в теле конструктора, по той же причине что указана выше. Т.е. такое поле в момент выполнения тела конструктора считается неинициализированной ссылкой, что тоже не допустимо;

Инициализация полей-констант и ссылок в конструкторе возможна только в списке инициализаторов.

Инициализация полей в точке объявления

```
#include <iostream>

int global = 10;

struct Point{
    Point(){}

    int x       = global;  // Обычное значение
    const int y = global;  // Константа
    int* z      = &global; // Указатель
    int& k      = global;  // Ссылка
};

int main(){
    Point point;
}
```

Поля объекта можно инициализировать в точке объявления, в том числе ссылки и константы;
Этот способ пригодится, когда начальное значение поля должно быть одинаковым для всех объектов и известно на этапе компиляции;

Код показанный на примере, хоть и работает, но нарушает принцип инкапсуляции, т.к. часть данных о классе находится за пределами самого класса.

Деструктор

Деструктор

```
#include <iostream>

struct Point{
    Point(int value):i(value){
        std::cout << "Created Point(" << i << ")" << std::endl;
    }

    ~Point(){
        std::cout << "Destroyed Point(" << i << ")" << std::endl;
    }

    int i;
};

int main(){
    Point p1(1);
    Point p2(2);
}
```

Деструктор – отвечает за корректное удаление объекта и освобождения выделенных ресурсов;

Деструктор автоматически вызывается в момент удаления объекта;

Если в теле класса не объявлен деструктор, то компилятор предоставляет реализацию по умолчанию. Этот деструктор ничего не делает;

В классе может быть только один деструктор – деструктор с пустым списком параметров;

Деструктор объявляется таким же образом как конструктор, но перед именем добавляется символ тильда (~);
Если есть хоть пользовательский деструктор, то реализация от компилятора не предоставляется;

В деструкторе нельзя бросать исключения.

Перед удалением объекта гарантируется вызов деструкторов (если они есть) для всех его полей.

Конструктор копирования

Конструктор копирования

```
#include <iostream>

struct Point{
    Point(int value):i(value){
        std::cout << "Created Point(" << i << ")" << std::endl;
    }

    Point(const Point& other){
        this->i = other.i;
        std::cout << "Copying Point(" << i << ")" << std::endl;
    }

    ~Point(){
        std::cout << "Destroyed Point(" << i << ")" << std::endl;
    }

    int i;
};

int main(){
    Point p1(1);
    Point p2(p1);
    Point p3 = p1;
}
```

Конструктор копирования – отвечает за корректное копирование объекта;

Конструктор копирования автоматически вызывается в тех местах, где требуется создать копию объекта: при инициализации одного объекта другим, передача в функцию по значению, возврат из функции по значению;

Если в теле класса не объявлен конструктор копирования, то компилятор предоставляет реализацию по умолчанию. Этот конструктор копирования выполняет побайтовое копирование для простых типов и вызов кондукторов копирования для классов;

В классе может быть 2 конструктора копирования – с константным параметром и с не константным параметром, но обычно объявляют только версию с константным параметром;

Конструктор копирования – это обычный конструктор во всех отношениях, но в качестве параметра этот конструктор должен принимать ссылку на объект текущего класса (константную или не константную);

Конструктор копирования

```
#include <iostream>

struct Point{
    Point(int value){
        i = new int(value); // Выделяем память
        std::cout << "Created Point(" << *i << ")" << std::endl;
    }
    /*
    Point(const Point& other){
        this->i = new int(*other.i); // Выделяем память для копии
        std::cout << "Copying Point(" << *i << ")" << std::endl;
    }
    */
    ~Point(){
        std::cout << "Destroyed Point(" << *i << ")" << std::endl;
        delete i;           // Освобождаем память
    }

    int* i;
};

int main(){
    Point p1(1);
    Point p2(p1);
    Point p3 = p1;
}
```

В этом примере без конструктора копирования объекты будут работать не правильно;

Как говорилось выше, стандартная реализация конструктора копирования выполняет побайтовое копирование полей простых типов.
В данном случае все три копии объекта будут указывать на один и тот же участок памяти выделенный в момент создания первого объекта.
На этапе удаления (в деструкторе), первый удаляемый объект освободит память выделенную под `i`, а остальные две копии попробуют освободить уже освобождённую память поворотно, что недопустимо;

Если раскомментировать конструктор копирования, то для каждой копии будет выделен свой собственный блок памяти, который затем корректно будет освобождён деструктором.

Передача объектов в функцию

```
#include <iostream>

struct Point{
    Point(){ std::cout << "Created Point" << std::endl; }
    Point(const Point& other){ std::cout << "Copying Point" << std::endl; }
    ~Point(){ std::cout << "Destroyed Point" << std::endl; }
};

void func_value(Point p){
    std::cout << "--- By value ---" << std::endl;
}

void func_pointer(Point* p){
    std::cout << "--- By pointer ---" << std::endl;
}

void func_reference(Point& p){
    std::cout << "--- By reference ---" << std::endl;
}

int main(){
    Point p;

    func_value(p);
    func_pointer(&p);
    func_reference(p);
}
```

Если передать объект в функцию по значению, то перед входом в тело функции будет создана копия оригинального объекта, т.е. вызван конструктор копирования;

Передача объекта в функцию по ссылке и указателю, как и ожидалось, не приводит к вызову конструктора копирования, т.к. функция получает доступ к оригинальному объекту.

Возврат объекта из функции

```cpp
#include <iostream>

struct Point{
    Point(){ std::cout << "Created Point" << std::endl; }
    Point(const Point& other){ std::cout << "Copying Point" << std::endl; }
    ~Point(){ std::cout << "Destroyed Point" << std::endl; }
};

Point global;

Point func_value(){
    std::cout << "--- By value ---" << std::endl;
    return global;
}

Point* func_pointer(){
    std::cout << "--- By pointer ---" << std::endl;
    return &global;
}

Point& func_reference(){
    std::cout << "--- By reference ---" << std::endl;
    return global;
}

int main(){
    func_value();
    func_pointer();
    func_reference();
}
```

Если вернуть объект из функции по значению, то в результате будет создана копия оригинального объекта, т.е. вызван конструктор копирования;

Возврат объекта из функции по ссылке или указателю, как и ожидалось, не приводит к вызову конструктора копирования, т.к. возвращается адрес оригинального объекта;

Есть особенности, если возвращается локальный объект.

Возврат локального объекта из функции

```cpp
#include <iostream>

struct Point{
    Point(int value){ std::cout << "Created Point" << std::endl; }
    Point(const Point& other){ std::cout << "Copying Point" << std::endl; }
    ~Point(){ std::cout << "Destroyed Point" << std::endl; }
};

Point func_copy_elision(int i){
    std::cout << "--- Copy elision ---" << std::endl;
    Point p(i);
    return p;
}

Point func_copy(int i){
    std::cout << "--- Copy ---" << std::endl;
    Point p(i);
    if (i == 0) {
        return p;
    }
    return Point(i);
}

int main(){
    func_copy_elision(0);
    func_copy(0);
}
```

Возвращать указатель или ссылку на локальный объект нельзя, т.к. он уничтожается перед выходом из функции;

В случае возврата локального объекта из функции по значению может быть применена оптимизация - пропуск копирования (copy elision);
В этом случае объект создаётся сразу же за пределами функции и происходит экономия на операциях: копирование локального и удаление локального;

Пропуск копирования может быть применён, даже если копирование/перемещение объекта имеет побочные эффекты (в этом примере побочный эффект это печать сообщения в терминал);

Применение этой оптимизации зависит от кода и настроек компилятора

Начиная с C++17, пропуск копирования гарантируется, когда объект возвращается напрямую, и в этом случае конструктор копирования или перемещения необязательно должен быть доступен или присутствовать.

Конструктор перемещения

Конструктор перемещения

Конструктор перемещения – отвечает за корректное копирование объекта который вот-вот должен быть уничтожен (т.е. временного объекта);

Если в теле класса не объявлен конструктор перемежения, то компилятор предоставляет реализацию по умолчанию. Если объявлен конструктор копирования, то конструктор перемещения компилятором не создаётся.

Подробнее данный конструктор будет рассмотрен на следующих лекциях.

```
class Point{
public:
    Point(const Point& src);      // копирующий конструктор
    Point(Point&& src) noexcept; // перемещающий конструктор
};
```

default и delete

default

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

struct Point{
    // Point() = default;
    Point(int value){
        std::cout << "Created Point" << std::endl;
    }
    ~Point(){
        std::cout << "Destroyed Point" << std::endl;
    }
};

int main()
{
    Point p(0);  // Нормально
    std::vector<Point> points(10); // Ошибка
}
```

Если, по каким либо причинам, нужно вернуть стандартную реализацию метода предоставляемую компилятором, то можно воспользоваться ключевым словом **default.**

В этом примере добавление конструктора с параметром приводит к тому, что компилятор больше не предоставляет реализацию конструктора по умолчанию. Но конструктор по умолчанию требуется для создания вектора точек в теле функции **main**, поэтому, чтобы код скомпилировался без ошибок можно добавить свою собственную реализацию конструктора по умолчанию или воспользоваться ключевым словом default, чтобы попросить компилятор вернуть его реализацию.

delete

```cpp
#include <iostream>

struct Dot{
    void print(){
        std::cout << "I am Dot" << std::endl;
    }
};

struct Point: Dot{
    Point() = default;

    Point(int x, int y) = delete;
    void print() = delete;
private:
    // Point(int x, int y); // Или так
    // void print();        // вместо delete
};

int main()
{
    Dot d;
    Point p;

    d.print();
    // p.print(); // Ошибка
}
```

Иногда бывает необходимо запретить использование некоторых методов у которых уже существует реализация. Например, реализация предоставляемая компилятором или попавшая в наш класс из класса предка;

Раньше в качестве стандартного решения применяли перенос ненужного метода в **private** секцию класса, что позволяло лишить пользователя доступа к нему. С появлением ключевого слова **delete** можно просто отметить метод как удалённый. Это касается любого метода класса;

Обратите внимание, что:
`Point(int x, int y) = delete;`

тоже считается реализацией конструктора которую предоставил пользователь, поэтому компилятор больше НЕ будет предоставлять свою и следовательно невозможно будет создать экземпляр такого класса.
Чтобы исправить ситуацию здесь была возвращена реализация конструктора по умолчанию:
`Point() = default;`

Доступ к членам класса

Спецификаторы доступа

```
struct Point{
    // тут объявления
private:
    // тут объявления
protected:
    // тут объявления
public:
    // тут объявления
private:
    // тут объявления
};
```

В C++ существует три спецификатора доступа:

private:
Доступ к членам класса есть только у методов этого же класса и друзей;

protected:
Доступ к членам класса есть у методов этого же класса, друзей и методов класса наследника;

public:
Доступ к членам класса есть у всех, в том числе и у обычных функций;

Каждый спецификатор создаёт секцию, которая действует до следующего спецификатора или конца объявления класса;
Ко всем объявлениям попавшим в секцию применяется соответствующий спецификатор доступа;
Количество секций и порядок может быть любым.

Спецификаторы доступа

```
struct Point{
    // тут всё public
};


class Point{
    // тут всё private
};

class Point: /* private */ Dot{
    // Всё, что пришло от Dot здесь стало private
};

struct Point: /* public */ Dot{
    // Всё, что пришло от Dot здесь не поменяло модификаторы доступа
};
```

Спецификатор доступа который действует по умолчанию у структуры – public;

Спецификатор доступа который действует по умолчанию у класса – private;

Если объявляется структура, то умолчанию она наследует все члены предка public способом;

Если объявляется класс, то умолчанию он наследует все члены предка private способом;

При наследовании не играет роль как объявлен предок, через class или struct.

Ключевое слово friend - функция

```
#include <iostream>

struct Point{
private:
    int val = 10;

    friend void foo(Point p);
};

void foo(Point p){
    std::cout << p.val << std::endl;
}

void bar(Point p){
    // std::cout << p.val << std::endl; // Ошибка
}

int main(){
    Point p;

    foo(p);
    bar(p);
}
```

Ключевое слово **friend** позволяет функции или классу получить доступ к членам другого класса игнорируя спецификаторы доступа;

Чтобы функция стала дружественной вашему классу, нужно написать её прототип в теле объявления вашего класса и перед прототипом написать friend. К объявлению самой функции ничего добавлять не нужно;

Чтобы класс стал дружественным вашему классу, нужно написать его неполное объявление в теле вашего класса и перед объявление написать friend. К объявлению самого класса-друга ничего добавлять не нужно;

Использование ключевого слова friend нарушает принцип инкапсуляции, т.к. кроме игнорирования спецификаторов доступа, сама функция не является частью класса, но тем не менее способна **напрямую (не через сообщение)** изменить состояние объекта.

Ключевое слово friend - класс

```
#include <iostream>

struct Dot;  // Чтобы Point знал, что Dot-класс

struct Point{
private:
    int val = 10;

    friend struct Dot; // Dot - друг
};

struct Dot{
    void foo(Point p){
        std::cout << p.val << std::endl;
    }
};

int main(){
    Point p;
    Dot d;

    d.foo(p);
}
```

Все методы дружественного класса могут получить доступ к членам вашего класса игнорирую спецификаторы доступа;

Ваш класс не получает приоритетного права доступа к членам класса-друга и действует в соответствии с его спецификаторами доступа.

Вложенные (nested) классы

```
struct Point{
private:
    class X{
        // Вложенные класс
    };

public:
    class Y{
        // Вложенные класс
    };

    X field_x; // поле типа X
};

int main(){
    Point p;

    // Point::X local1; // Ошибка X - приватный
    Point::Y local2;    // Можно Y - публичный

    // Point::X local3 = p.field_x; // Нельзя
    auto local4 = p.field_x;        // Можно
}
```

Объявление вложенного класса должно находиться внутри вашего класса;
Объявление класса – это объявление **типа данных**, поэтому автоматически не происходит создание никаких объектов;

На объявление вложенного класса действуют спецификаторы доступа вашего класса. Т.е. если класс объявлен в private секции, то **идентификатор** класса нельзя будет **писать** за пределами тела и методов класса и т.д.;

Во внешнем коде обращение к идентификатору вложенного класса происходит по полному квалифицированному имени, т.е. перед именем вложенного класса добавляется имя вашего класса и два двоеточия;

Вложенный класс, во всех отношениях является полноценным классом, т.е. он может иметь свои спецификаторы доступа, свои вложенные классы, друзей и т.д.

Вложенные (nested) классы

```cpp
#include <iostream>

struct Point{
    struct X{
        void print_static();
        void print_not_static(Point obj);
    };

    Point(int i):value(i){
    }

private:
    static int s_value;
    int value;
};

int Point::s_value = 10;

void Point::X::print_static(){
    std::cout << s_value << std::endl;
}

void Point::X::print_not_static(Point obj){
    std::cout << obj.value << std::endl;
}

int main(){
    Point p(50);
    Point::X printer;

    printer.print_static();
    printer.print_not_static(p);
}
```

Методы вложенного класса игнорируют модификаторы доступа вашего класса, т.е. он действует как класс-друг;

Из методов вложенного класса можно обращаться напрямую (по не квалифицированному имени) к статическим полям и методов вашего класса;

Неполный тип

Неполный тип

Неполные типы в C++ — это типы данных, которые еще не полностью определены на момент их использования. Они могут быть полезны во многих случаях, например, для обработки рекурсивных структур данных или для работы с классами определение которых по каким-либо причинам не доступно для нашего класса.

Объявление неполного типа, сообщает компилятору, что существует тип с таким именем, но ничего не говорит ему о том, как он реализован: какие есть методы, поля и т. д. Обычно эти типы полностью определяются позже, поэтому такое объявление часто называют опережающим (forward declaration).

Неполным типом может быть:
- тип void;
- структуры, классы, объединения, для которых ещё не определена реализация;
- массивы неопределённого размера;
- массивы элементов неполного типа.

Чтобы тип стал полным (полностью определенным) необходимо указать недостающую информацию. Отдельно стоит отметить тип void, который не может быть полным в принципе.

Так как неполный тип ничего не говорит компилятору о своём внутреннем устройстве и размере, то не получится скомпилировать код, который пытается получить доступ к его содержимому или создать переменную такого типа. Переменную-указатель или переменную-ссылку на неполный тип создать можно.

Класс использующий себя как поле

```
// Нельзя
struct Node {
    int data;
    Node next; // Следующая нода
};


// Так можно
struct Node {
    int data;
    Node* next; // Следующая нода
};
```

Класс не может использовать себя в качестве своего же поля, потому что это приведёт к бесконечному рекурсивному выделению памяти под объект такого класса;

Класс не может использовать себя в качестве своего же поля, потому что на момент использования объявления класса ещё не закончено – т.к. это неполный тип;

Класс может использовать ссылку или указатель на себя как поле, как возвращаемый тип или параметр метода;

Перекрёстное использование

```
struct A{
    B b_field;
};

struct B{
    A a_field;
};

/*********************************/
struct B;

struct A{
    B* b_field; // Здесь B - неполный тип
};

struct B{
    A* a_field; // Звёздочка не обязательна чисто для симметрии
};
```

Бывает, что два класса используют друг друга в качестве полей, как возвращаемые значения или как параметры методов. Без использования неполных типов такое сделать невозможно, т.к. один из классов в любом случае будет находится за пределами области видимости другого.

Использование в методах

```
#include <iostream>

struct B;

struct A{
    void method(B param); // Здесь B неполный тип
};

struct B{
    int value = 10;
};


int main(){
    A a;
    B b;

    a.method(b);
}

void A::method(B param){ // А здесь B уже полный
    std::cout << param.value << std::endl;
}
```

Тело метода можно определить уже после того, как класс станет полным типом, в этом случае можно будет использовать его методы и принимать/возвращать объекты по значению;

Если в этом примере определить тело внутри класса A, то использование методов класса B будет недоступно и передать объект класса B в методы можно будет только по указателю или ссылке.

Удаление переменных неполного типа

```cpp
#include <iostream>

struct A;

A* create_A();

int main(){
    A* a = create_A(); // Создаём объект
    delete a;          // Удаляем объект
}

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

A* create_A(){
    return new A();
}
```

Удаление объектов неполного типа может сопровождаться проблемами в том случае, если на момент удаления тип всё ещё остаётся неполным;

Обычно компилятор покажет предупреждение, но не ошибку, т.е. код будет работать, но с непредсказуемыми последствиями;

В этом примере видно, удаление объекта происходит ещё до того, как объявится тело класса A. Это приведёт к тому, что деструктор класса A не будет вызван (хотя память будет освобождена). Здесь мы просто не увидим сообщение на экране, но если деструктор делает что-то более важное, то он не будет выполнено и может сломать всю логику работы кода.

Устройство объекта в памяти

Объект в памяти

```cpp
#include <iostream>

struct A{};

struct B{
    int value = 0;
};

struct C{
    int value = 0;
    void print(int value){
        std::cout << value << std::endl;
    }
};

struct D{
    int value = 0;
    virtual void print(int value){
        std::cout << value << std::endl;
    }
};
```

```cpp
struct E{
    int value = 0;
    static int svalue;
};
int E::svalue = 0;

int main(){
    std::cout << "A: " << sizeof(A) << std::endl; // A: 1
    std::cout << "B: " << sizeof(B) << std::endl; // B: 4
    std::cout << "C: " << sizeof(C) << std::endl; // C: 4
    std::cout << "D: " << sizeof(D) << std::endl; // D: 16
    std::cout << "E: " << sizeof(E) << std::endl; // E: 4
}
```

Всё, что было сказано про устройство в памяти структур справедливо и для классов;
- Объект пустого класса занимает в памяти 1 байт, только для того, чтобы его можно было адресовать (меньше не получится);
- Место в экземпляре класса занимают только поля;
- Статические поля не занимают места в экземплярах класса;
- Методы не занимают в памяти объекта никакого места;
- Если в классе появляется ходьбы один виртуальный метод, то в объекты такого класса добавляется скрытое поле – указатель на таблицу виртуальных методов;

Разделение кода класса на файлы

Разделение кода класса на файлы

В языке C++ используется раздельная компиляция кода. Т.е. каждый .cpp файл компилируется в объектный файл отдельно от остальных. Основная причина такого разделение – **долгая компиляция** кода и попытка ускорить её за счёт разделения на несвязанные блоки.

Если мы размещаем весь класс в .h файле, то каждый .cpp файл который будет использовать наш класс нужно будет перекомпилировать при внесении любых изменений в наш класс.
Чтобы уменьшить количество перекомпиляций код класса обычно разделяют на 2 файла:

- интерфейс класса. Объявляет поля и методы без реализации. Размещается в .h файле;
- реализация класса. Здесь прописывается реализация всех методов и инициализация статических полей. Размещается в .cpp файле;

Таким образом можно свободно вносить изменения в реализацию класса и перекомпилировать придётся только один .cpp файл. Внесение изменений в интерфейс класса приведёт к необходимости перекомпилировать все файлы.

В качестве второй причины такого разделения может быть желание разработчика **скрыть исходный код** реализации своего класса. Т.е. разработчик может предоставлять потребителю пачку .h файлов с исходным кодом интерфейса класса и уже скомпилированный объектный файл, а не исходный .cpp файл.

Третьей причиной разделения кода на несколько файлов можно назвать желание повысить **структурированность** своего кода. Общепринятой практикой является выделения под каждый класс отдельной пары .h и .cpp файлов.

Разделение кода класса на файлы

```cpp
#include <iostream>

struct A{};

struct B{
    int value = 0;
};

struct C{
    int value = 0;
    void print(int value){
        std::cout << value << std::endl;
    }
};

struct D{
    int value = 0;
    virtual void print(int value){
        std::cout << value << std::endl;
    }
};
```

```cpp
struct E{
    int value = 0;
    static int svalue;
};
int E::svalue = 0;

int main(){
    std::cout << "A: " << sizeof(A) << std::endl; // A: 1
    std::cout << "B: " << sizeof(B) << std::endl; // B: 4
    std::cout << "C: " << sizeof(C) << std::endl; // C: 4
    std::cout << "D: " << sizeof(D) << std::endl; // D: 16
    std::cout << "E: " << sizeof(E) << std::endl; // E: 4
}
```

Всё, что было сказано про устройство в памяти структур справедливо и для классов;
- Объект пустого класса занимает в памяти 1 байт, только для того, чтобы его можно было адресовать (меньше не получится);
- Место в экземпляре класса занимают только поля;
- Статические поля не занимают места в экземплярах класса;
- Методы не занимают в памяти объекта никакого места;
- Если в классе появляется ходьбы один виртуальный метод, то в объекты такого класса добавляется скрытое поле – указатель на таблицу виртуальных методов;

Разделение кода класса на файлы

point.h

```
struct Point{
    Point(int value);
    Point(const Point& other);
    ~Point();

private:
    int i;
    static int j;
};
```


point.cpp

```
#include <iostream>
#include "point.h"

Point::Point(int value):i(value){
    std::cout << "Created Point(" << i << ")" << std::endl;
}

Point::Point(const Point& other){
    this->i = other.i;
    std::cout << "Copying Point(" << i << ")" << std::endl;
}

Point::~Point(){
    std::cout << "Destroyed Point(" << i << ")" << std::endl;
}

int Point::j = 10;
```


main.cpp

```
#include "point.h"

int main(){
    Point p(50);
}
```

Идиома Pimpl (Pointer to implementation)

```cpp
#include <iostream>

struct Point{
    Point(int value):i(value){

    }
    void print(){
        printer.print(i);
    }

private:
    struct Printer{
        void print(int value){
            std::cout << value << std::endl;
        }
    };

    int i;
    Printer printer;
};


int main(){
    Point p(50);
    p.print();
}
```

Есть класс, который использует другой класс как часть своего объявления (не обязательно вложенный).
В таком виде любое изменение в используемом классе приведёт к необходимости перекомпилировать **все** файлы использующие данный класс, даже если мы разделим код класса на 2 файла как ранее.

Чтобы избавится от зависимости от реализации вспомогательного класса нам поможет идиома pimpl. Суть идиомы заключается в том, чтобы заменить в интерфейсе (.h файл) вспомогательные классы на указатели (ссылки) на эти классы, а всё что касается реализации перенести в .cpp файлы.

После такого разделения код использующий наш класс больше не будет требовать перекомпиляции при изменении вспомогательного класса, т.к. он не будет знать о деталях его реализации.

Такой подход часть используют при разработке кроссплатформенных библиотек, или библиотек которые хотят скрыть исходный код своих классов.

Идиома Pimpl (Pointer to implementation)

point.h

```
struct Point{
    Point(int value);
    void print();
    ~Point();

private:
    class Printer;  // Неполный тип

    int i;
    Printer* printer;
};
```

main.cpp

```
#include "point.h"

int main(){
    Point p(50);
    p.print();
}
```

point.cpp

```
#include <iostream>
#include "point.h"

struct Point::Printer{
    void print(int value){
        std::cout << value << std::endl;
    }
};

Point::Point(int value)
:i(value), printer(new Printer){
}

void Point::print(){
    printer->print(i);
}

Point::~Point(){
    delete printer;
}
```

Перегрузка операторов

Перегрузка операторов

В C++17 стандарт разрешает перегружать следующие операторы:

`+, -, *, /, %, ^, &, |, ~, !, ,, =, <, >, <=, >=, ++, --, <<, >>, ==, !=, &&, ||, +=, -=, /=, %=, ^=, &=, |=, *=, <<=, >>=, [], (), ->, ->*, new, new[], delete, delete[]`

- Оператор преобразования к заданному типу (user-defined conversion function)
- Оператор литерала (user-defined literal)
- Перегруженный оператор `co_await` для использования в выражениях `co_await` (с C++11)
- Оператор трёхстороннего сравнения, названный «космическим кораблем»: <=> (с C++20)

Следующие операторы перегрузить нельзя:

`::` (разрешение области видимости)

`.` (оператор доступа к членам класса)

`.*` (оператор доступа к членам класса через указатель на объект класса)

`?:` (тернарный условный оператор)

`sizeof` (оператор получения размера типа)

`typeid` (оператор получения информации о типе)

Перегрузка операторов – ограничения

- Новые операторы, такие как **, <> , &| и т.д. не могут быть созданы;
- Арность, приоритет и ассоциативность операторов, перегрузке не меняются (нельзя изменить в принципе);
- Перегрузить оператор для встроенного типа данных нельзя, можно только для пользовательских, т.е. у перегрузки оператора хотя бы один операндов должен быть пользовательского типа;
- Операторы могут быть перегружены и как отдельные функции (не все), и как функции-члены класса;
- Перегрузки операторов && и || теряют ленивое поведение, т.е. перегруженные версии вычисляют оба своих операнда;
- Перегрузка оператора -> должна возвращать либо указатель на класс (структуру, объединение), либо тип, для которого перегружен оператор ->
- При перегрузке оператора рекомендуется учитывать ожидания пользователя класса. Например если перегрузить оператор +, но внутри прописать код вычитания, то для пользователя будет сюрприз.

Как можно перегрузить оператор

Операторы можно перегружать в двух вариантах: как функцию-член и как свободную (не-член) функцию. Четыре оператора можно перегрузить только как функцию-член — это `=, ->, [], ()`. Для перечислений операторы можно перегружать только как свободные функции.

- Для того, чтобы перегрузить оператор как функцию-член необходимо объявить нестатическую функцию-член с именем operator@, где @ символ(ы) оператора. В случае перегрузки унарного оператора эта функция не должна иметь параметров, а в случае бинарного должна иметь ровно один параметр. В случае перегрузки оператора () эта функция может иметь произвольное число параметров.

- Для того, чтобы перегрузить оператор как свободную (не-член) функцию, необходимо объявить функцию с именем operator@, где @ символ(ы) оператора. В случае перегрузки унарного оператора, эта функция должна иметь один параметр, а в случае бинарного должна иметь два параметра.

Перегрузка как свободная функция

```cpp
#include <iostream>

struct Integer{
    Integer(int value):value(value){
    }

    friend Integer operator+(const Integer& lhs, const Integer& rhs);
    friend std::ostream& operator<<(std::ostream& out, const Integer& obj);
private:
    int value;
};


Integer operator+(const Integer& lhs, const Integer& rhs){
    return Integer(lhs.value + rhs.value);
}

std::ostream& operator<<(std::ostream& out, const Integer& obj){
    out << obj.value;
    return out;
}


int main(){
    Integer a(10);
    Integer b(20);

    auto c = a + b;
    std::cout << c << std::endl;
}
```

В этом примере операторы + и << перегружены как свободные функции, но для доступа с приватному полю они были добавлены в объявление класса как друзья (хотя это и не обязательное условие перегрузки);

Параметры операторов и возвращаемы типы могут отличаться. Здесь они выбраны такими, чтобы работать таким же образом как и операторы для встроенных типов;

Перегрузка как функция-член

```
#include <iostream>

struct Integer{
    Integer(int value):value(value){
    }

    Integer operator+(const Integer& other) const{
        return Integer(this->value + other.value);
    }

    friend std::ostream& operator<<(std::ostream& out, const Integer& obj);
private:
    int value;
};

std::ostream& operator<<(std::ostream& out, const Integer& obj){
    out << obj.value;
    return out;
}


int main(){
    Integer a(10);
    Integer b(20);

    auto c = a + b;
    std::cout << c << std::endl;
}
```

Здесь оператор + перегружен как метод класса;
Т.к. оператор + бинарный, у него один параметр – правый относительно оператора. В качестве правого оператора используется this;

По этой причине оператор << здесь перегрузить как метод класса невозможно, т.к. он ждёт в качестве правого оператора std::ostream&.

Другие перегрузки оператора << возможно реализовать как методы класса.

Перегрузка как функция-член

```
class Matrix{
public:
    Matrix(int rowCount, int colCount);

    class RowProxy;
    RowProxy operator[](int i) const;

    class RowProxy {
    public:
        T& operator[](int j);
        const T& operator[](int j) const;
    };
};


int main(){
    Matrix mtx(5, 6);
    double s = mtx[1][2];
    mtx[2][3] = 3.14;
}
```

Это классический пример реализации двумерного массива.
У класса `Matrix` перегружен оператор [ ] который возвращает прокси объект вложенного класса `RowProxy`;
У вложенного класса `RowProxy` тоже перегружен оператор [ ] который возвращает ссылку на элемент матрицы.

Способы вызова перегруженного оператора

```
#include <iostream>

struct Integer{
    Integer(int value):value(value){
    }

    Integer operator+(const Integer& other) const{
        return Integer(this->value + other.value);
    }

    friend std::ostream& operator<<(std::ostream& out, const Integer& obj);
private:
    int value;
};

std::ostream& operator<<(std::ostream& out, const Integer& obj){
    out << obj.value;
    return out;
}


int main(){
    Integer a(10);
    Integer b(20);

    auto c = a.operator+(b);
    operator<<(std::cout, c).operator<<(std::endl);
}
```

Оператор может быть вызван как обычный оператор и как функция или методы класса с передачей ей аргументов.

Оператор присваивания

```
class Point{
public:
    // копирующий конструктор
    Point(const Point& src);
    // перемещающий конструктор
    Point(Point&& src) noexcept;

    // оператор копирующего присваивания
    Point& operator=(const Point& src);
    // оператор перемещающего присваивания
    Point& operator=(Point&& src) noexcept;
};
```

Оператор присваивания можно реализовать только, как функцию-член, которая должна иметь ровно один параметр. Тип этого параметра произвольный, соответственно, перегрузок может быть несколько, для разных типов параметра.

Если описать семантику присваивания «на пальцах», то присваивание должно полностью освободить все текущие ресурсы, которыми владеет объект (левый операнд), и на его месте создать новый объект, определяемый правым операндом.

Среди операторов присваивания выделяются два стандартных — оператор копирующего присваивания и оператор перемещающего присваивания, которые соответствуют копирующему конструктору и перемещающему конструктору.

Оператор копирующего присваивания

```cpp
#include <iostream>

struct Point{
    Point(int value){
        i = new int(value); // Выделяем память
        std::cout << "Created Point(" << *i << ")" << std::endl;
    }

    Point(const Point& other){
        this->i = new int(*other.i); // Выделяем память для копии
        std::cout << "Copying Point(" << *i << ")" << std::endl;
    }

    /*
    Point& operator=(const Point& other){
        if (this != &other){
            delete this->i;              // Освобождаем память
            this->i = new int(*other.i); // Выделяем память для копии
            std::cout << "operator= Point(" << *i << ")" << std::endl;
        }

        return *this;
    }*/

    ~Point(){
        std::cout << "Destroyed Point(" << *i << ")" << std::endl;
        delete i;    // Освобождаем память
    }

    int* i;
};

int main(){
    Point p1(1);
    Point p2(p1);
    p2 = p1;
}
```

В этом примере без оператора присваивания объекты будут работать не правильно;

Как говорилось выше, стандартная реализация оператора присваивания выполняет побайтовое копирование полей простых типов.
В данном случае оба объекта будут указывать на один и тот же участок памяти выделенный в момент создания первого объекта.
На этапе удаления (в деструкторе), первый удаляемый объект освободит память выделенную под i, а второй попробуют освободить уже освобождённую память поворотно, что недопустимо;

Если раскомментировать оператор присваивания, то для каждой копии будет выделен свой собственный блок памяти, который затем корректно будет освобождён деструктором;

Если в операторе присваивания убрать проверку на присваивания себе, то память с данными будет освобождена, а на его место придёт случайный фрагмент памяти с мусорным значением.

Идиома copy-and-swap

```
struct Point{
    Point(int value){
        i = new int(value); // Выделяем память
        std::cout << "Created Point(" << *i << ")" << std::endl;
    }

    Point(const Point& other){
        this->i = new int(*other.i); // Выделяем память для копии
        std::cout << "Copying Point(" << *i << ")" << std::endl;
    }

    Point& operator=(const Point& other){
        Point temp(other);
        swap(*this, temp);

        return *this;
    }

    ~Point(){
        std::cout << "Destroyed Point(" << *i << ")" << std::endl;
        delete i;    // Освобождаем память
    }

    friend void swap(Point& first, Point& second);

    int* i;
};

void swap(Point& first, Point& second){
    using std::swap;
    swap(first.i, second.i);
}
```

Ручная реализация оператора присваивания страдает от нескольких проблем, в том числе от лишнего копирования кода;

Идиома copy-and-swap определяет способ реализации оператора присваивания лишённого этих проблем, но в том случае, если у вас реализованы конструктор копирования и функция/метод swap(последний не всегда обязателен);

В данном примере основное тело оператора присваивания можно заменить на два действия: создание временной копии объекта который присваиваем и затем обмен текущего объекта с временной копией;

Если поменять параметр оператора присваивания с ссылки на значение, то от копирования тоже можно будет избавится (параметр-значение и так копия):

```
Point& operator=(Point other){
    swap(*this, other);

    return *this;
}
```

Оператор преобразования типа

```cpp
#include <iostream>

class Counter{
public:
    Counter(int number){
        value = number;
    }
    operator int() const { return value; }
private:
    int value;
};

int main(){
    Counter counter{25};
    int n = counter;    // преобразуем от Counter в int
    std::cout << n << std::endl;              // 25
    // или так
    int m {counter};
    std::cout << m << std::endl;              // 25
}
```

C++ позволяет определить функцию оператора преобразования из типа текущего класса в другой тип;

Тип возвращаемого значения у оператора не указывается, поскольку целевой тип всегда подразумевается в имени функции;

В отличие от большинства операторов, операторы преобразования должны быть определены только как функции-члены класса. Их нельзя определить как обычные функции;

Оператор преобразования типа будет применятся как в явных преобразованиях static_cast или в стиле Си, так и в неявных.
Чтобы отключить неявные преобразования можно воспользоваться ключевым словом explicit;

Оператор круглые скобки

```
#include <iostream>

class Print{
public:
    void operator()(const std::string& message) const {
        std::cout << message << std::endl;
    }

    void operator()(int val) const {
        std::cout << val << std::endl;
    }

    void operator()(int val1, int val2) const {
        std::cout << val1 << ", " << val2 << std::endl;
    }
};

int main(){
    Print print;     // определяем объект функции
    print("Hello"); // выполняем как функцию
    print(1);       // выполняем как функцию
    print(1, 3);    // выполняем как функцию
}
```

Этот оператор можно реализовать только как функцию-член. Он может иметь любое число параметров любого типа, тип возвращаемого значения также произвольный. Классы, с перегруженным оператором (), называются функциональными, их экземпляры называются функциональными объектами или функторами.

Лямбда-функции это синтаксический сахар вокруг функторов, т.е. лямбды == функторы.

Функциональный класс может иметь несколько вариантов перегрузки оператора (), с разными параметрами. Такие классы и соответствующие объекты можно назвать мультифункциональными.

Подробнее о перегрузке операторов

- Operator overloading: https://en.cppreference.com/w/cpp/language/operators
- Перегрузка в C++. Часть II. Перегрузка операторов: https://habr.com/ru/articles/489666/
- What is the copy-and-swap idiom? (ответ): https://stackoverflow.com/a/3279550