

Курс лекций: Язык программирования C++

Лекция 4: Классы в C++ (ч. 2) Конструктор, деструктор, перегрузка операторов

Преподаватель: Митричев Иван Игоревич,
ассистент кафедры ИКТ, к.т.н.

Конструктор класса

Для инициализации объектов (присвоение значений полям объекта) используется специальный метод класса, имеющий одно имя с классом — **конструктор**.

Если вы не объявите хотя бы один свой конструктор, то всегда будет создан конструктор по умолчанию (default constructor).

Запомните: конструктор по умолчанию вызовет конструкторы всех полей данных. Если эти поля - объекты, для них будут вызваны конструкторы по умолчанию. Если это - указатели на объекты, то указатели будут проинициализированы значением 0, сами же объекты созданы не будут (!), память выделена не будет.

Запомните: если среди полей класса есть хотя бы один указатель, почти всегда вам нужно писать свой конструктор (потому что инициализировать объекты лучше всегда в конструкторе, а не в методах).

Типы конструкторов

```
class Car {  
private:  
    double weight;  
public:  
    Car(){ }// конструктор без параметров  
    Car(double _d){weight=_d;}// параметризованный конструктор  
    Car(const Car& c):weight(c.weight)  
        { //пустое тело конструктора } // конструктор копирования  
};
```

:weight(c.weight) – запись означает *список инициализации* вместо присваивания в теле конструктора. Можно присваивать ряд переменных : weight(c.weight), length(c.length) {...}

Если явно не объявлен ни один конструктор с параметрами или без, то создается конструктор по умолчанию. Если не объявлен ваш собственный конструктор копирования, то создается конструктор копирования по умолчанию.

Конструктор копирования по умолчанию

```
class Car {  
private:  
    Engine* engine;  
... (конструктор копирования далее не объявлен)  
};
```

```
Car mashina();  
Car ford(mashina);
```

Теперь у ford и mashina
одинаковый указатель на мотор
engine. При изменении мотора у
одной машины, у второй
произойдут те же изменения!

Конструктор копирования по умолчанию создает «shadow copy» - теньюую копию объекта, то есть не копирует память, выделенную с помощью new, new[], а только копирует указатель (поле-указатель указывает на ту же область памяти, что и у старого! При удалении старого объекта новый объект указатель на область памяти, которая может быть перезаписана. Отсюда странные «баги» — «гейзенбаги»)

Конструктор копирования и передача по значению
Пусть в программе реализована функции проверки веса машины,
которая принимает параметр Car по значению.

```
void checkWeight(Car some_car)
{
    if (some_car.weight .....)
        ...
}
```

```
Car mashina;
checkWeight(mashina);
```

Запомните: при передаче по значению создается копия переменной (объекта класса Car), т. е., вызывается конструктор копирования! Some_car – это уже копия объекта mashina в другой области оперативной памяти! А значит, для корректной работы функции checkWeight() конструктор копирования должен отработать правильно.

Конструктор копирования и временная переменная
Пусть в программе реализована функции проверки веса машины,
которая принимает параметр Car по значению.

```
void checkWeight(Car some_car)
{
    if (some_car.weight .....)
        ...
}
class Car
{ double weight;
  public: Car(double _d):weight(_d){ }
}
checkWeight(Car(1800));
```

Car(1800) – это временный объект (temporary). В данном коде будет
вызван конструктор с параметром класса Car, а затем **конструктор
копирования! Получается две переменных, лишние накладные
расходы как памяти, так и времени на вызов и работу
конструкторов.**

Перегрузка операторов

Можно перегружать операторы C++

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

Нельзя перегружать :: . .* ?:

Пример: перегрузка операции инкремента

```
struct Car{  
    Car& operator++() // префиксный инкремент  
    {  
        ...  
        здесь реальная имплементация (например: weight+=1.0;)  
        ...  
        return *this; // вернуть новое значение  
    }  
    Car operator++(int) // постфиксный инкремент  
    {  
        Car tmp(*this); // copy  
        operator++(); // вызвать префиксный инкремент  
        return tmp; // вернуть старое значение – объясните почему  
    }  
};
```

Правило «Трёх» (Rule of Three)

Если вам пришлось объявить что-либо из собственного конструктора копирования, собственного оператора присваивания и собственного деструктора, то, **скорее всего, вам нужно объявить все вышеперечисленное.**

Упрощенное объяснение:

Если среди полей класса есть хотя бы один указатель, вам нужно писать свой конструктор копирования, оператор присваивания и деструктор класса.

См. ресурсы

<https://stackoverflow.com/questions/4172722/what-is-the-rule-of-three/4172724#4172724>

Правило трёх (C++) — Википедия

Пример использования конструктора копирования

```
class Engine{                                oop3_6.cpp                                }
    int VIN;                                // 2. copy assignment operator
    public:                                (оператор присваивания)
        Engine(){}                        Car& operator=(const Car& that)
        Engine(int _v):VIN(_v){}        {
        void setVIN(int _v){VIN=_v;}    if (this != &that)
        int getVIN(){return VIN;}        {
};                                         delete engine;
                                         engine = new Engine();
                                         *engine = *(that.engine);
                                         }
                                         return *this;
                                         }
                                         ~Car() {delete engine;}
                                         void printInfo(){cout<<engine<<"
                                         "<<engine->getVIN()<<endl;}
                                         void setVIN(int _v){engine-
                                         >setVIN(_v);}
                                         };

class Car{                                oop3_5.cpp - без
    double fuel;                            собственного конструктора
    Engine* engine;                        копирования, оператора
    public:                                присваивания и деструктора
        Car(int _v){engine = new
Engine(_v);}
// 1. copy constructor (к. копирования)
    Car(const Car& that)
    {
        engine = new Engine();
        *engine = *(that.engine);
```

Пример использования конструктора копирования

```
Car* mashinka [2];
```

```
mashinka[0] = new Car(111111); // устанавливает номер  
двигателя
```

```
mashinka[1] = new Car(222222); // устанавливает номер  
двигателя
```

```
Car ford(*mashinka[1]);
```

```
mashinka[1]->setVIN(33); // меняет номер у mashinka[1] на 33
```

```
mashinka[0]->printInfo(); //111111
```

```
mashinka[1]->printInfo(); //33
```

```
ford.printInfo(); //222222
```

```
delete mashinka[0];
```

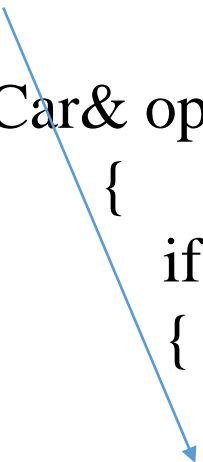
```
delete mashinka[1];
```

```
ford.printInfo(); //222222
```

Операция присваивания: идиома “Copy and swap”

Стандартный, опасный путь – недостаток – надо выделить память и удалить старый объект. Если посередине произойдет исключение – утечка памяти!

```
Car& operator=(const Car& that)
{
    if (this != &that)
    {
        delete engine;
        engine = new Engine();
        *engine = *(that.engine);
    }
    return *this;
}
```



«Copy and swap» (**передача по значению**, а значит, вызывается конструктор копирования для создания временной переменной)
Car& operator=(Car that)

```
{
    swap(*this, that);
    return *this;
}
```

Используется своя функция или std::swap, которая гарантированно не выбросит исключение. Ненужное старое значение остается во временной переменной, которая автоматически удалится

Объект, который копировать нельзя

Иногда требуется создать объект, который нельзя скопировать (паттерн «синглтон», или мьютекс – средство синхронизации переменной в многопоточном приложении, мьютекс должен существовать один на одну переменную, и при его использовании в нескольких единицах трансляции нельзя допустить его копирование и создание нового мьютекса).

Для этого просто объявите к-р копирования и оп-р присваивания приватными:

private:

```
CarNonCopyable(const CarNonCopyable& that);
```

```
CarNonCopyable& operator=(const CarNonCopyable& that);
```

В C++11 для этого появилось новое синтаксис со словом «delete»:

```
CarNonCopyable(const CarNonCopyable& that) = delete;
```

```
CarNonCopyable& operator=(const CarNonCopyable& that) = delete;
```

CarNonCopyable – скопировать объект данного класса нельзя.

Когда использовать move-семантику

На слайде 6 говорилось, что иногда нам требуется использовать временные объекты. В коде на слайде 6 два раза будет вызван конструктор (обычный + копирования), есть две временных переменных – одна, безымянная, создается при вызове конструктора класса Car в строке

```
checkWeight(Car(1800));
```

Другая – `some_car`, создается при вызове функции `void checkWeight(Car some_car)`.

Чтобы избежать накладных расходов и сократить время работы программы, удобно отказаться от конструктора копирования в таких случаях. C++11 предлагает новое понятие - перемещение (`move`). В случае перемещения данные повторно не копируются, а та же самая временная переменная используется в другом контексте (например, функции).

Когда использовать move-семантику - 2

Рассмотрим пример создания строки (используется конструктор).

`string a(x);` // `x` имеет имя. Такую переменную называют в C++ левосторонним значением (в нее можно что-то присвоить) - lvalue.

Для таких переменных следует использовать к-р копирования (если вдруг при создании строки а переменная `x` изменится /потеряется – это нонсенс!)

`string b(x + y);` // `x + y` – временная переменная (rvalue – ее значение можно куда-то передать или присвоить, но нельзя к ней обратиться или написать `x+y = ...`). Она создается где-то в памяти при вычислении суммы, и все равно вскоре будет автоматически удалена, не может быть использована вне (). Такую переменную можно перемещать - Move!

`string c(some_function());` // функция `some_function()` вернет временную переменную. Move!

Как использовать move-семантику

В C++11 появился новый тип – ссылка на rvalue. Обозначается &&rvalue все равно будет уничтожена, поэтому можно написать move constructor (конструктор перемещения), если внутри скобок rvalue:

```
string(string&& that) {
```

```
    data = that.data;
```

```
    that.data = nullptr;
```

```
}
```

```
string b(x + y);
```

<https://stackoverflow.com/a/3109981>

Конструктор перемещения устанавливает некоторый указатель data строки b на область памяти, где хранятся данные временной переменной без имени (x+y). Указатель x+y обнуляется. Т.е., вместо копирования (копирование всегда выделяет какую-либо память) содержимого временной переменной в b (*в случае конструктора копирования с параметром-ссылкой*) и удаления временной переменной, мы «украли» указатель на содержимое временной переменной, не выделяя новую память! Ускорение!

Превращаем lvalue в rvalue

Данный конструктор перемещения вызовется только если на вход подано правостороннее значение rvalue. А если мы хотим применить его к lvalue? (то есть, переместить содержимое lvalue в новую переменную).

```
string(string&& that) {  
    data = that.data;  
    that.data = nullptr;  
}
```

Для этого существует `std::move`

```
string d(std::move(x));
```

После выполнения данного утверждения (statement), старая переменная lvalue `x` будет содержать что угодно, мусор. Мы *переместили* ее содержимое в `d`.

`std::move(x)` возвращает rvalue, но реально не создает временной переменной. Такой вид rvalue называют xvalue (expiring value).

«Правило пяти»

В случае, если ваш класс использует move-семантику, то «правило трех» превращается в «правило пяти»: нужно перегружать конструктор перемещения и оператор присваивания с перемещением

```
X(const X&);           // к-р копирования
```

```
X& operator=(const X&); // оператор присваивания с копированием
```

```
~X();                 // деструктор
```

```
X(X&&);               // конструктор перемещения
```

```
X& operator=(X&&); // оператор присваивания с перемещением
```

default

В C++11 появилось новое ключевое слово `default`. Позволяет сгенерировать конструктор по умолчанию, даже если есть другие конструкторы.

```
class Foo
{
public:
    Foo() {} = default;
    Foo & (const &Foo that) = default;
    Foo(int Bar) {...}
};
```

override

В C++11 появилось новое ключевое слово `override`. Перегрузка виртуальных функций в классах наследниках выполняется автоматически при наличии `virtual` в базовом классе. Одна при написании/изменении параметров функций может произойти ошибка, что программист произведет изменения одновременно не во всех классах. Это приведет к тому, что при использовании указателя на базовый класс вызываться методы дочерних классов не будут. Предлагается указывать для всех перегружаемых методов `override` после `()`. Следующий код не скомпилируется, так как сигнатуры функций различны:

```
class Base
{
public:
    virtual void Foo(int Bar);
};
```

```
class Derived : public Base
{
public:
    virtual void Foo(long Bar) override;
};
```

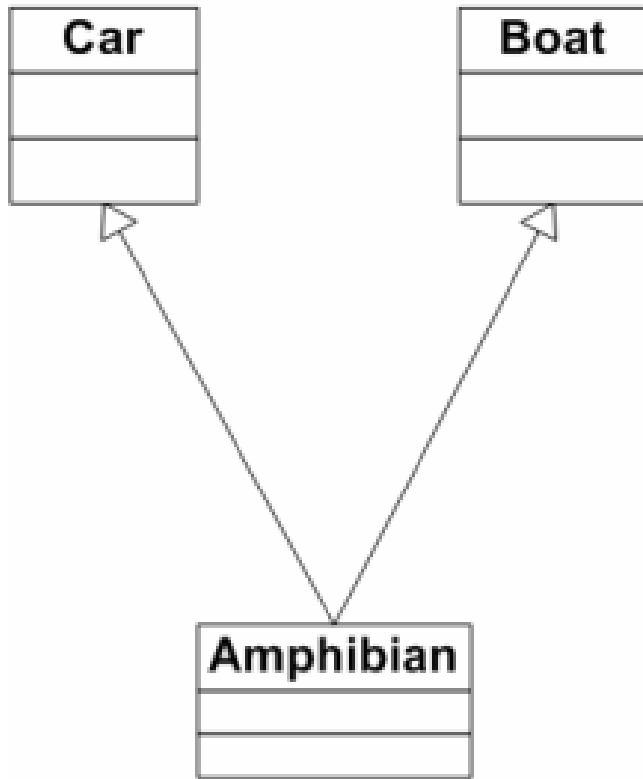
final

В C++11 появилось новое ключевое слово `final`. Запрещает перегрузку функции, помечая ее как «окончательную» (`final`).
Следующий код не скомпилируется:

```
class Base
{
public:
    virtual void Foo(int Bar) final;
};
// ...
```

```
class Derived : public Base
{
public:
    virtual void Foo(int Bar); // ошибка!
};
```

Множественное наследование



Класс «амфибия» одновременно обладает чертами лодки Boat и машины Car. В таких случаях используют множественное наследование (диаграмма классов UML приведена слева).

```
class Amphibian: public Car, public Boat
{
    ...
}
```

Вызов конструкторов при множественном наследовании

```
#include<iostream>
using namespace std;

class A {
public:
    A() { cout << "A's constructor
called" << endl; }
};

class B {
public:
    B() { cout << "B's constructor
called" << endl; }
};

class C: public B, public A {
public:
    C() { cout << "C's constructor
called" << endl; }
};

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

Порядок вызова конструкторов соответствует порядку объявления классов-родителей при наследовании. Сначала вызываются конструкторы родительских классов, потом – дочерних.

Ромбовидное наследование

Проблема: класс В и С переопределяют один и тот же метод класса А. Какой метод следует наследовать классу D?

В C++ наследуются оба. Также, реально в памяти будет выделено две области, соответствующие классу А, то есть класс D, объявленный так:

```
class A {...};  
class B: public A {...};  
class C: public A {...};  
class D: public B, public C {...};
```

будет храниться в памяти как:

участок, соответствующий наследуемой части класса А

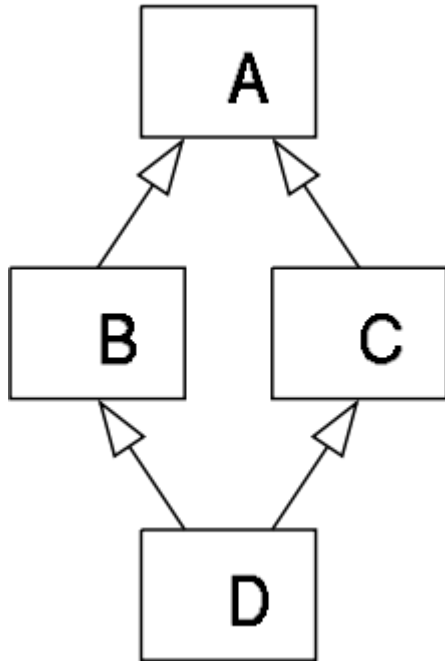
участок, содержащие дополнительные свойства класса В по сравнению с А

участок, соответствующий классу А

участок, содержащие дополнительные свойства класса С по сравнению с А

участок, содержащие дополнительные свойства класса D по сравнению с В и С.

Для вызова методов А пишут В::A.method() или С::A.method().



Виртуальное наследование

Чтобы оставить только метод класса А (например), и не писать каждый раз `B::A::method()` или `C::A::method()` при ромбовидном наследовании, можно использовать виртуальное наследование.

```
class A {...};  
class B: public virtual A {...};  
class C: public virtual A {...};  
class D: public B, public C {...};
```

будет храниться в памяти как:

участок, соответствующий наследуемой части класса А
участок, содержащие дополнительные свойства класса В по сравнению с А
участок, содержащие дополнительные свойства класса С по сравнению с А
участок, содержащие дополнительные свойства класса D по сравнению с В и С.

*См. также:
Википедия.
Виртуальное
наследование.*

Для вызова методов А пишут `method()`. Перегрузить виртуальные методы класса А в виртуальных классах нельзя.

А где бы попроще и на примерах...

Вот тут

<https://www.intuit.ru/studies/courses/17/17/info>

Лекции 12, 13, также 10 (виртуальное и
множественное наследование)