**1 запрет на копирование объектов класса в C++**

**1.1 зачем нужен этот прием?**

*- что делает компилятор если программа пытается скопировать объект класса?*

*- что делают автоматически сгенерированные copy c-tor & operator=*

**1.2 Что происходит в результате работы кода:**

*- что будет вызвано*

*- последовательность вызовов деструкторов и суть происходящего*

*- привести вариант решения*

class DoubleDelete {

public:

DoubleDelete() {

ptr = new char[100];

}

~DoubleDelete() {

delete[] ptr;

}

private:

char\* ptr;

};

\* \* \* \*

{

DoubleDelete first;

DoubleDelete second( first );

}

*- все ли объекты по своей сути подлежат копированию? 2-а примера*

**1.3 Попытки изобразить решение:**

*- Что делают если копирование объекта не имеет смысла?*

*- Привести три примера реализации запрета на копирование и объяснение*

*минусов каждого из вариантов:*

*модификатор private в любом ли случае не даст вызвать копирование?*

*привести примеры кода и 2 проблемных случая*

*- дать вариант с оптимальным решением*

*- почему возвращаемый тип operator= возвращает void,*

*а не ссылку на тот же клас*

**1 Запрет на копирование объектов класса в C++**

У одного или нескольких классов конструктор копирования и оператор присваивания объявлены private и написан комментарий вида «копирование запрещено».  
  
При его использовании возможны подводные камни, приводящие к ошибкам.  
  
***1.1 зачем нужен этот прием?***  
  
Если программа пытается скопировать объект класса, компилятор C++ по умолчанию автоматически генерирует конструктор копирования или оператор присваивания, если они не объявлены в классе явно. Автоматически сгенерированный конструктор выполняет почленное копирование, а автоматически сгенерированный оператор присваивания – почленное присваивание.

class DoubleDelete

{

public:

DoubleDelete()

{

ptr = new char[100];

}

~DoubleDelete()

{

delete[] ptr;

}

private:

char\* ptr;

};

В этом коде:

{

DoubleDelete first;

DoubleDelete second( first );

}

возникнет неопределенное поведение. Будет вызван сгенерированный компилятором конструктор копирования, который скопирует указатель. В результате оба объекта будут хранить указатели с равными адресами. Первым отработает деструктор объекта second, он выполнит delete[], затем будет вызван деструктор объекта first, он попытается выполнить delete[] повторно для того же адреса, и это приведет к неопределенному поведению.  
  
Решение вроде бы очевидно – реализовать конструктор копирования и оператор присваивания с правильным поведением. Например, при копировании новый объект создает свой массив и копирует в него данные из старого.  
  
Это не всегда верный путь. Не все объекты по своей сути подлежат копированию.  
  
Например, объект может хранить дескриптор открытого файла в виде целого числа. Деструктор «закрывает файл» с помощью функции операционной системы. Очевидно, почленное копирование не подойдет. А что должно происходить при копировании? Должен ли файл открываться еще раз? Обычно в этом случае копирование не имеет смысла.  
  
Другой пример – класс для захвата критической секции при создании объекта такого класса. Какой смысл копировать объект? Секция уже захвачена.  
  
***1.3 Попытка изобразить решение.***  
  
Если копирование объекта не имеет смысла, нужно сделать так, чтобы компилятор не смог случайно его выполнить. Для этого обычно делают так:  
  
// NOT BAD

*class NonCopyable* {

// blahblahblah public:

private:

// copy and assignment prohibited

*NonCopyable( const NonCopyable& );*

*void NonCopyable operator=( const NonCopyable& );*

};

или так:

// FAIL

*class NonCopyable {*

// blahblahblahpublic:

private:

// copy and assignment prohibited

*NonCopyable( const NonCopyable& ) { assert( false ); }*

*void NonCopyable operator=( const NonCopyable& ) { assert( false ); }*

*};*

или так:

// EPIC FAIL

*class NonCopyable {*

// blahblahblah public:

private:

// copy and assignment prohibited

*NonCopyable( const NonCopyable& ) {}*

*void NonCopyable::operator=( const NonCopyable& ) {}*

};

Все три способа встречаются в реальном коде.  
  
Казалось бы, чем второй и третий варианты отличаются от первого? Модификатор private в любом случае не даст вызвать копирование.  
  
КРАЙНЕ НЕОЖИДАННО…  
  
Функции-члены того же класса могут вызывать конструктор копирования и оператор присваивания, даже если те объявлены private. И «друзья» класса (friend) тоже могут. Никто не мешает написать в коде что-нибудь такое:

*NonCopyable NonCopyable::SomeMethod()*

*{*

*// blahblahblah*

*return \*this;*

*}*

или такое:  
  
*void NonCopyable::SomeOtherMehod()*

*{*

*callSomething( \*this );*

*}*

Теперь налицо разница между первым вариантом и остальными.  
  
Первый вариант (нет реализации) приведет к ошибке во время компоновки программы. Сообщение об ошибке не самое понятное, но, по крайней мере, надежное.  
  
Во втором варианте будет срабатывать assert… при условии, что управление пройдет через этот код. Здесь многое зависит от того, насколько часто этот код вызывается, в частности, от покрытия кода тестами. Может быть, вы заметите проблему при первом же запуске, может быть – очень нескоро.  
  
В третьем варианте еще лучше – оператор присваивания не меняет объект, а конструктор копирования вызывает конструкторы по умолчанию всех членов класса. Широкий простор для ошибок, заметить может быть еще сложнее, чем второй.  
  
Ожидаемое возражение – раз конструктор копирования и оператор присваивания объявлены, но не определены, их можно по ошибке или злонамеренно определить где угодно в коде. Эта проблема решается очень просто.  
  
От ошибки помогает комментарий вида «запрещенные операции»

*В C++0x есть ключевое слово* ***delete****:*

// C++0x OPTIMAL

*class NonCopyable {*

*private:*

// copy and assignment not allowed

*NonCopyable( const NonCopyable& ) = delete;*

*void operator=( const NonCopyable& ) = delete;*

};

В этом случае не только определить, но и вызвать их будет невозможно – при попытке компиляции места вызова будет выдана **ошибка компиляции**.  
  
Вариант «объявить и не определять» доступен и ранее C++0x, его, в частности, использует boost::noncopyable. Вариант наследоваться от boost::noncopyable или аналогичного своего класса тоже достаточно надежен и доступен в любой версии.  
  
Внимательный читатель наверняка обратил внимание, что во всех примерах выше оператор присваивания возвращает void, а не ссылку на тот же класс. Это сделано специально, чтобы конструкции вида

first = second = third;

вызывали **ошибку компиляции** в С++03.

**2 delete, new[] в C++**

***2.1 Рассмотреть 2-а варианта не разумных ответов***

- *каков правильный ответ*

***2.2 Поставить в тупик не разумные ответы с помощью примеров кода***

a) class Class {

public:

~Class() {

printf( "Class::~Class()" );

}

};

int main()

{

delete new Class[1];

return 0;

}

b) int main()

{

delete new char[1];

return 0;

}

***2.3 что программа обязана делать когда в коде встречается «new Type[count]» - подробно описать два пункта.***

*- Что происходит? Почему так происходит?*

*- Почему поведение с виду разное?*

***2.4 что происходит Когда в коде встречается «delete[] pointer»***

***2.5 В конструкции «new Type[count]» число элементов было указано явно,***

***а «delete[]» получает только адрес первого элемента. Откуда программа узнает число элементов?***

***2.6 что происходит Когда выполняется «delete» (без скобок)***

*- Вернемся к первому примеру. Что и как вызывается*

*и что должно произойти?*

*- Почему во втором примере поведение другое?*

***2.7 «operator new[]()» и «new Type[count]»***

***«operator new()» и «new Type».***

*- реализации функций «operator new()» и «operator new[]()»*

***2.8 Какие последствия если память освобождается «не той» функцией?***

*- Какая проблема произошла в первом примере с «operator delete()»*

2 delete, new[] в C++

Если в коде на C++ был создан массив объектов с помощью «new[]», удалять этот массив нужно с помощью «delete[]» и ни в коем случае не с помощью «delete» (без скобок). Не путать «delete» и «delete[]». Разумный вопрос: **а не то что?**  
Неразумные ответы:

1. «будет удален только первый объект, остальные утекут»

2. «будет вызван деструктор только первого объекта».  
  
В соответствии со Стандартом C++, в этой ситуации **поведение не определено**.

Все предположения – не более чем популярные городские легенды. Разберем подробно, почему.   
  
Нам понадобится хитрый план с примером, который бы ставил в тупик сторонников городских легенд. Вот такой безобидный будет ок:

class Class {

public:

~Class()

{

printf( "Class::~Class()" );

}

};

int main()

{

delete new Class[1];

return 0;

}

Здесь объект в массиве всего один. Если верить любой из двух легенд выше, «все будет хорошо» – утекать нечему и некуда, деструкторов будет вызвано ровно сколько нужно.  
  
Идем на codepad.org, вставляем код в форму, получаем выдачу:

memory clobbered before allocated block

Exited: ExitFailure 127

42 75 67 20 61 73 73 61 73 73 69 6E 20 77

61 6E 74 65 64 20 2D 20 77 77 77 2E 61 62

62 79 79 2E 72 75 2F 76 61 63 61 6E 63 79

MEMORY WHAT??? Что это было?

Второй пример:

int main()

{

delete new char[1];

return 0;

}

Выдача:

No errors or program output.

Здесь хотя бы с виду все хорошо. Что происходит? Почему так происходит? Почему поведение с виду разное?  
  
Причина в том, что происходит внутри.  
  
***2.3 Когда в коде встречается «new Type[count]», программа обязана:***

1. Выделить память объема, достаточного для хранения указанного числа объектов.

- *Для этого она использует функцию «operator new[]()» которая выделяет память – обычно внутри просто вызов malloc() и проверка возвращаемого значения (при необходимости – вызов new\_handler() и выброс исключения).*

2. Затем в выделенной памяти конструируются объекты – вызывается нужное число конструкторов. Результатом *«new Type[count]»* является адрес первого элемента массива.

Итак, при выполнении «new Type[count]» программа выделяет чуть больше памяти, записывает число элементов в начало выделенного блока памяти, вызывает конструкторы и возвращает вызывающему коду адрес первого элемента. Адрес первого элемента будет отличаться от адреса, который возвратила функция выделения памяти «operator new[]()».  
  
***2.4 что происходит Когда в коде встречается «delete[] pointer»***

Когда в коде встречается «delete[] pointer», программа должна разрушить все объекты в массиве, вызвав для них деструкторы. Для этого (и только для этого) ей нужно знать число элементов.

При выполнении «delete[]» программа берет адрес первого элемента, переданный в «delete[]», определяет адрес начала блока (вычитая ровно столько же, сколько было прибавлено при выполнении «new[]»), читает число элементов из начала блока, вызывает нужное число деструкторов, затем – вызывает функцию «operator delete[]()», передав ей адрес начала блока.  
  
В обоих случаях вызывающий код работает не с тем адресом, который был возвращен функцией выделения памяти и позже – передан функции освобождения памяти.  
  
***2.5 В конструкции «new Type[count]» число элементов было указано явно, а «delete[]» получает только адрес первого элемента. Откуда программа узнает число элементов?***

Раз у нее есть только адрес первого элемента, она должна вычислить длину массива на основании одного этого адреса. Как это делается, зависит от реализации, обычно используется следующий способ:

*При выполнении «new Type[count]» программа выделяет памяти столько, чтобы в нее поместились не только объекты, но и беззнаковое целое (обычно типа size\_t), обозначающее число объектов. В начало выделенной области пишется это число, дальше размещаются объекты. Компилятор при компиляции «new Type[count]» вставляет в программу код, который реализует эти свистелки*.

***2.6 что происходит Когда выполняется «delete» (без скобок)***  
*Теперь вернемся к первому примеру*. Когда выполняется «delete» (без скобок), вызывающий код понятия не имеет, что нужно проиграть последовательность со смещением адреса. Скорее всего, он вызывает деструктор единственного объекта, затем передает в функцию «operator delete()» адрес, который отличается от ранее возвращенного функцией «operator new[]()».  
  
*Что должно произойти?* В этой реализации программа аварийно завершается. Поскольку Стандарт говорит, что поведение не определено, это допустимо.  
  
*Почему во втором примере поведение другое?* Скорее всего, компилятор учел, что у типа char тривиальный деструктор, т.е. не нужно ничего делать для разрушения объектов, а достаточно просто освободить память, поэтому и число элементов хранить не нужно, а значит, можно сразу вернуть вызывающему коду тот же адрес, который вернула функция «operator new[]()». Никаких смещений адреса – точно так же, как и при вызове «new» (без скобок). Такое поведение компилятора полностью соответствует Стандарту.  
  
***2.7 «operator new[]()» и «new Type[count]»***

***«operator new()» и «new Type».***  
Они могут быть реализованы совершенно по-разному. Даже когда компилятор пытается сэкономить, он всегда вызывает функцию «operator new[]()», когда видит в коде «new Type[count]», и всегда вызывает функцию «operator new()», когда видит в коде «new Type».  
  
Обычно реализации функций «operator new()» и «operator new[]()» одинаковы (обе вызывают malloc()), но их можно заменить – определить свои, причем можно заменить как одну пару, так и обе, также можно заменять эти функции по отдельности для любого выбранного класса. Стандарт позволяет это делать сколько угодно (естественно, нужно адекватно заменить парную функцию освобождения памяти). Это дает богатые возможности для неопределенного поведения.

***2.8 Какие последствия если память освобождается «не той» функцией***

Если ваш код приводит к тому, что память освобождается «не той» функцией, это может приводить к любым последствиям, в частности, к **повреждению кучи, порче памяти или немедленному аварийному завершению программы**.

*В первом примере* реализация функции «operator delete()» не смогла распорядиться переданным ей адресом и программа **аварийно завершилась**.  
  
Вывод: мы неможем утверждать, что использование «delete» вместо «delete[]» (и наоборот – тоже) приводит к какому-то конкретному результату. **Стандарт говорит, что поведение не определено**. Даже полностью соответствующий Стандарту компилятор не обязан выдать вам программу с каким-либо адекватным поведением. Поведение программы, на которое вы будете ссылаться в комментариях и спорах, является только наблюдаемым – внутри может происходить все что угодно. Вы только констатируете наблюдаемое вами поведение.  
  
*Во втором примере* с виду все хорошо… на этой реализации. На другой реализации функции «operator new()» и «operator new[]()» могут быть, например, реализованы на разных кучах (Windows позволяет создавать более одной кучи на процесс). Что произойдет при попытке возвратить блок «не в ту» кучу?  
  
Кстати, рассчитывая на какое-то конкретное поведение в этой ситуации, вы автоматически получаете непереносимый код. Даже если на текущей реализации «все работает», при переходе на другой компилятор, при смене версии компилятора или даже при обновлении C++ runtime вы можете быть крайне неприятно удивлены.

**3 Multiple inheritance**

*Что такое Multiple inheritance?*

**3.1 Зачем оно может использоваться?**

*- что такое Интерфейсы? расскажите на пример классов*

*-* IDrawable, ISerializable, JPGPicture

*- наследование от нескольких полноценных классов на примере*

*-* Cow, Sniper, CowSniper

*В чем суть, что важно, какие отличия от обычного*

***- Пример 2****.* Human *может кататься на любом объекте, который реализовывает интерфейс* IBicycle*. Реализовываем интерфейс* IBicycle *для* SimpleBicycle *и* SportBicycle.

**3.2 Проблемы при множественном наследовании**

*3.2.1 Перекрытие имен функций (Привести пример, проблему и решение)*

*3.2.2 Перекрытие виртуальных функций (Привести пример, проблему и решение)*

*3.2.3 Представление объекта в памяти*

*а) в случае линейного наследования*

*б) простейшее множ наслед*

*в) множ наслед если в графе наследования есть цикл (diamond inheritance) или Ромбовидное).*

*в.1)* **Пример** *с разрешением не однозначности с помощью приведения типов.* Class FlyingCar, Car & Airplane *имеют метод* use(). *Как убрать неоднозначность при вызове* use() *с помощью приведения типов.*

*в.2) Когда нужно ромбовидное наследование.* ***Привести пример*** *(Компоненты к видео плате. Нужно что бы эти данные задваивались потому что по конкретному каждому объекты нужна своя отдельная информация.)*

Class Component, GPU, Memory,

GraphicCard : public GPU, public Memory

*3.2.4 Виртуальное наследование. (Привести пример, проблему и решение)*

**3.3 В чем минус множ наслед? Где его желательно применять и какая альтернатива множественного наследования не от интерфейсов**

*- Агрегация. Пример*

*- Композиция. Пример*

# 3 Множественное наследование

Множественное наследование (*Multiple inheritance*)— наследование от нескольких базовых классов одновременно.

## 3.1 Зачем оно может использоваться

1. **Интерфейсы**   
   *Интерфейс* — специальные классы, описывающие набор методов, но не имеющие данных и реализации.

**Пример 1.** Например интерфейсами являются классы:

* + IDrawable (то, что можно нарисовать)
  + ISerializable (то, что можно записать в файл)

Тогда класс JPGPicture будет наследоваться от этих двух интерфейсов, т. е. методы обоих интерфейсов будут реализованы в классе JPGPicture.   
Интерфейсы — наиболее популярное применение множественного наследования. Возможность использования множественного наследования в виде интерфейсов есть во всех классических ООП-языках. В некоторых языках программирования (Java, C#) есть только такой вид множественного наследования.

*Интерфейс* – это абстрактный класс с чисто виртуальными методами. Это некая договоренность о том, как с чем-то можно взаимодействовать. То есть это те public методы с помощью которых можно взаимодействовать с какой то сущностью. Например *интерфейс* для использования баз данных. Тот же *интерфейс* будет применен для любого типа базы данных если база данных реализовывает общий *интерфейс*.

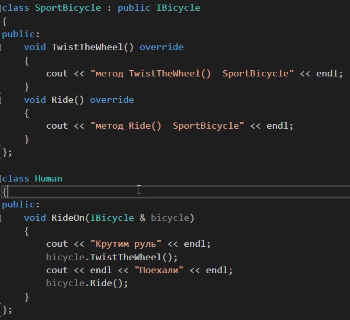
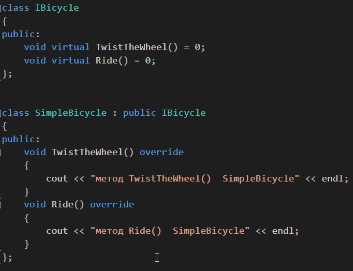
Один объект может реализовывать несколько интерфейсов. (Множ наслед)

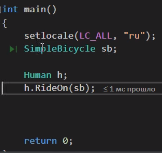
**Пример 2**. *Human* может кататься на любом объекте который реализовывает интерфейс *IBicycle*. Реализовываем интерфейс *IBicycle* для *SimpleBicycle* и *SportBicycle.*

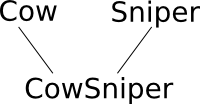
*override- key word для того,* что бы компилятор проконтролировал то, что мы действительно переопределили метод базового класса и не ошиблись

Метод RideOn(IBicycle & bicycle) принимает ссылку на любой тип данных который реализовывает интерфейс *IBicycle*.

Если любой велосипед реализовывает интерфейс *IBicycle,* то *Human* сможет без проблем им пользоваться благодаря методу RideOn(IBicycle & bicycle). Примерно такая логика использ и для баз данных.



1. **Наследование от нескольких полноценных классов**   
   Допустим у нас есть классы Cow и Sniper, а мы хотим получить класс, который обладает данными и методами обоих классов. Назовем его CowSniper. Таким образом у нас есть возможность «скрещивать» классы.   
   
2. class CowSniper: public Cow, public Sniper {

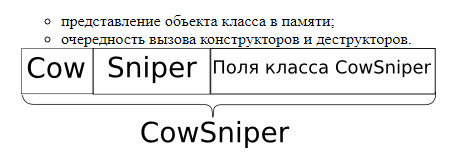
};

CowSniper cs;

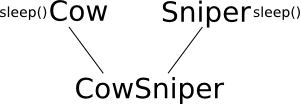
Cow \* c = &cs;

Sniper \* s = &cs;

Таким образом синтаксически множественное наследование почти не отличается от обычного.   
Заметим , что важен порядок перечисления предков, потому как в зависимости от него:



## 3.2 Проблемы при множественном наследовании

1. **Перекрытие имен функций**   
   эта проблема есть как и в обычном наследовании, так и в множественном.  
   Пусть в классах Cow и Sniper были методы sleep().  Тогда код:
2. CowSniper cs;

cs.sleep(); // 2

В строке 2 произойдет ошибка компиляции, т. к. компилятор не может выбрать какой метод sleep() ему вызывать (от Cow или от Sniper). Поэтому необходимо сообщимть ему правильный выбор: cs.Cow::sleep();

1. **Перекрытие виртуальных функций**  
   Теперь рассмотрим тот случай, если метод sleep() виртуальный в классах-предках.

class Cow {

public:

virtual void sleep() = 0;

};

class Sniper {

public:

virtual void sleep() = 0;

};

class CowSniper: public Cow, public Sniper {

public:

virtual void sleep() {}

};

В таком случае будут перегружены сразу оба метода (ведь у них одинаковая сигнатура). Подробнее о том почему это происходит написано в следующем пункте. Тогда в следующих строках кода:

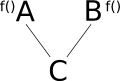
Cow & c = cs;

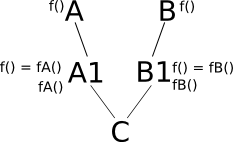
Sniper & s = cs;

c.sleep(); //3

s.sleep(); //4

В строках 3 и 4 вызовется один и тот же метод CowSniper::sleep();   
С одной стороны это удобно, т. к. если методы называются одинаково, то скорее всего они делают что-то похожее. Тогда перегрузив один метод мы перегрузим сразу оба.  
С другой стороны может возникнуть проблема, если класс-потомок должен реализовывать один и тот же виртуальный метод от нескольких базовых классов по-разному. Тогда такая перегрузка будет вредна, но стандартно по другому не поступить. В таких случаях используется следующий способ обойти это ограничение.  
Если есть иерархия классов A,B,C. И в классах A и B есть некоторый виртуальный метод f().



Добавим в эту иерархию еще два класса A1 и B1. В классах A1 и B1 создадим методы fA() и fB() соотвественно. Теперь в C будут методы fA() и fB(), которые необходимо перегрузить, причем код в них разный ;) 

class A{

public:

virtual void f();

};

class B{

public:

virtual void f();

};

class A1: public A{

public:

virtual void f() {fA();}

virtual void fA() = 0;

};

class B1: public B{

public:

virtual void f() {fB();}

virtual void fB() = 0;

};

class C: public A1, public B1 {

public:

virtual void fA();

virtual void fB();

};

Теперь использовать эти классы можно так:

C c;

A & a = c;

B & b = c;

a.f(); //5

b.f(); //6

В строке 5 вызовется C::fA(), а в строке 6 - C::fB(). Обратите внимание на то, как перегружен метод f() в классах A1 и B1, которые ко всему прочему теперь являются абстрактными и невозможно создать их экземпляры. Цель достигнута.

**Представление объекта в памяти**   
**а)** Как мы помним, в случае линейного наследования распределение полей классов для объекта класса наследника в памяти будет такое: img   
При этом если мы создадим три указателя:

C \* = new C();

B \* b = c;

A \* a = c;

То указывать они все будут на начало объекта в памяти:    
и при этом все хорошо.  
  
**б)** Теперь перейдем к рассмотрению простейшего примера множественного наследования:  
   
При этом если создать объект класса C, то в памяти он будет выглядеть так:   
img   
Опять попробуем создать три указателя как и в случае линейного наследования:

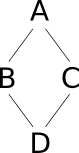
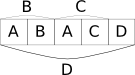
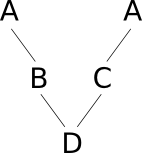
C \* = new C();

B \* b = c;

A \* a = c;

Однако теперь указатели распределятся следующим образом:

   
Мы видим, что классы A и B фактически разделены, но ведь если они имеют виртуальные методы, то должны вызываться перегруженные виртуальные методы класса C. Для того, чтобы это происходило в каждом из них есть ссылка на таблицу виртуальных функций. Тогда получается что в классе C будет сразу два указателя на две различные таблицы виртуальных функций (кол-во указателей = кол-ву полиморфных предков). Получается, что таблиц виртуальных функций две. Это не совсем так, т. к. они просто лежат рядом в памяти, т. е. фактически таблица одна, но в ней могут быть повторения, например, виртуальный деструктор:

img   
  
**в)** Теперь рассмотрим множественного наследования, если в графе наследования есть цикл:  
  
В памяти объект класса D будет представлен так:  
   
Возникает проблема дублирования полей класса A, т.к. фактически будет два разных объекта типа A. Значит при компиляции следующего кода произойдет ошибка, т.к. непонятно к какому полю обращаться.   


D \* d = new D();

A \* a = d;

Для того, чтобы указать, какой из классов A выбрать следует написать так:

D \* d = new D();

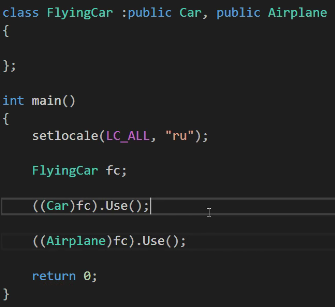
A \* a = static\_cast<B\*> (d);

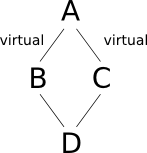
При этом если мы хотим использовать методы класса A, так же необходимо явно указывать которого их них. Вследствие этого даные в двух объектах A в пределах одного D могут стать различными.  
  
Рассмотрим два случая такого наследования и посмотрим на возможные проблемы с наличием двойного объекта.   
1) Такой эффект может быть полезен, если класс A является файловым потоком, а B и C это writer и reader соответственно. А класс D читает из С и пишет в B. Очевидно, что у B и C файлы могут быть различны (скорее всего так и есть).

2) Рассмотрим умный указатель (A = LinkCounter), который внутри себя сдержит счетчик. В таком случае в классе D возникает два счетчика, что может привести к печальным последствием, если в одном месте работать с одним из них, а в другом с другим.  
  
Вообще такое наследование (с циклами в графе родства) называется бриллиантовым (diamond inheritance) или **Ромбовидным**.

**Пример** с разрешением не однозначности с помощью *приведения типов*.

Class FlyingCar, Car & Airplane имеют метод use().



* + 1. **Виртуальное наследование.**  
       Рассмотрим его на примере следующей иерархии  
          
       Синтаксически виртуальное наследование почти не отличается от множественного:

class A{

int k;

};

class B: public virtual A {

};

class C: public virtual A {

};

class D: public B, public C {

};

При этом следующий код будет прекрасно работать:

D \* d = new D();

A \* a = d;

А все потому, что классы будут выглядеть следующим образом:

img

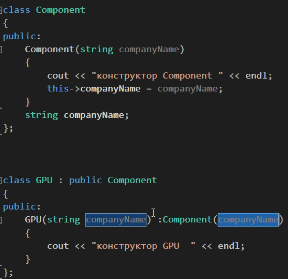
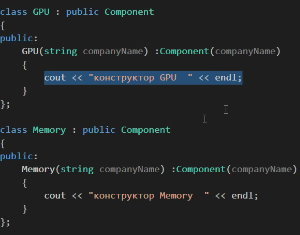
Здась может появиться проблема преобразования указателя на D к указателю на C. Но этой проблемы нет. Необходимо разобраться каким образом это реализуется.   
Дело в том, что при виртуальном наследовании добавляется виртуальная функция, возвращающая указатель на A. Фактически для программиста она не видна. Для пояснения обозначим ее за getA(). Стоит заметить что она будет различна в классах B,C и D.   
Теперь при обращении к полю из A (допустим в нем поле int k) код вида k = 10; будет автоматически преобразован в getA()->k = 10;. 

Теперь рассмотрим очередность вызова конструкторов.   
Логично предположить, что вызов конструкторов пройдет так   img

Но возникает проблема, ведь конструкторы B и C могут вызывать различные конструкторы A и с различными параметрами.   
Для определенности было введено следующее правило: Конструктор A должен быть явно вызван в конструкторе D, при этом в конструкторах B и C вызов конструктора A опустится.   
  
В связи с этим есть замечание: Нужно следить и понимать, что при виртуальном наследовании в конструкторах B и C может не вызваться конструктор A с разными параметрами.

Иногда ромбовидное наследование нужно.

**Пример.** Компоненты к видео плате. Нужно что бы эти данные задваивались потому что по конкретному каждому объекты нужна своя отдельная информация.





***3.3 В чем минус множ наслед? Где его желательно применять и какая альтернатива множественного наследования не от интерфейсов***

Реализации множественного наследования ведут к появлению сильных зависимостей в коде, а следовательно такое наследование желательно использовать только с интерфейсами.

Возможной заменой множественного наследования не от интерфейсов является **агрегация**:

class CowSniper {

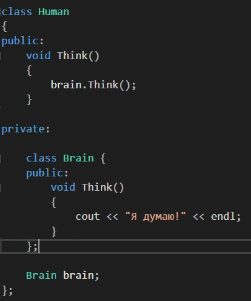
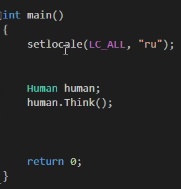
private:

Cow c; };

Sniper s;

**Агрегация** и **композиция** – это включение одного класса в другой, но с отличиями. **Агрегация** позволяет использовать включенный класс еще и в других местах. С другими классами. А **композиция(**жесткая привязка одного объекта к другому**)** – более строгий вариант агрегации, т к класс который включается в другой класс – без этого класса существовать не может. Создается в **private** секции. (нужен для его служебных целей).

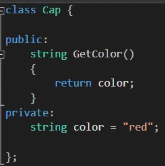
*Пример композиция*. Мозг и человек. Мозг без человека существовать не может – мозг жестко инкапсулирован в человека.

Class Brain является вложенным для class Human и мы нигде его не можем использовать. Если объект Human уничтожается то и объект Brain уничтожается тоже. В примере применяется делегирование(паттерн) в методе think() в Human.

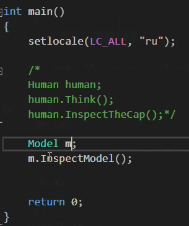
Human.Think(); // мы делегируем метод Think() объекту типа Brain который создан в Human.

*Пример агрегации.*Добавим классCap. Кепка не привязана жестко к человеку и мы ее можем использовать с другими классами (надеть на что то другое). В этом суть агрегации – класс который мы используем с другими классами не привязан жестко к какому то классу (не зависим). Создадим маникен class Model.









Мы используем объект класса кепка и в человеке и в моникене – пример агрегации.

**Наследование**

**Task 1** Начнем с создания фермера, но так как фермеров будет много и все разные, но с одинаковыми параметрами, характеризующими определенного фермера, логично будет создать его класс, а самих фермеров задавать как объекты этого класса:

*Class Unit*

*health*

*void showHealth()*

**Task 2 .**Теперь создадим воинов. Кроме того, что у них есть здоровье, как у фермеров, они могут наносить урон (**damage**). В результате у Геракла теперь 10 healthи 30 damage, а у Ахиллеса 40 healthи 50 damage

*class Soldier : public Unit*

*damage*

*void showDamage()*

**Task 3** Теперь аналогичным образом создадим лошадей, которые в отличии от фермеров, будут двигаться, т.е. будут иметь параметр speed. Все лошади будут иметь 10 healthи 20 speed

*class Horse : public Unit*

*speed*

*void showSpeed()*

**Task 4** Создадим всадников, которые будут иметь характеристики как фермеров(здоровье) и солдат (damage), так и лошадей (speed):

*class Horseman : public Horse, public Soldier { };*

1. Создается стратегическая игра и поставлена задача создания всяческих боевых и не боевых единиц. Начнем с создания фермера, но так как фермеров будет много и все разные, но с одинаковыми параметрами, характеризующими определенного фермера, логично будет создать его класс, а самих фермеров задавать как объекты этого класса:

#include <iostream>   
**using** **namespace** std;   
    
**class** Unit //базовый класс фермера   
{   
**protected**:   
       int health;   
**public**:   
       **void** showHealth() { cout << "Unit health: " << health << endl; };   
       Unit(): health(10) { }       //конструктор по умолчанию   
       Unit(int a): health(a) { } //конструктор с параметром   
};

С этим классом должно быть все понятно. Его объекты будут характеризоваться одним параметром "health" (здоровье) и есть один метод, позволяющий вывести значение этого параметра на экран. Можно создавать фермеров:

int main()   
{   
       Unit bob;   
       bob.showHealth();   
       Unit jack(30);   
       jack.showHealth();   
       **return** 0;   
}

На экран выведется Unit health: 10 и Unit health: 30. Ничего нового пока.

Теперь создадим воинов. Кроме того, что у них есть здоровье, как у фермеров, они могут наносить урон (**damage**). Можно было бы создать отдельный класс для воинов с параметрами **health** и **damage**, но гораздо проще создать класс воинов, имеющий параметр health класса фермеров и свой параметр **damage**. Это и есть наследование:

**class** Soldier : **public** Unit   
{   
**private**:   
       int damage;   
**public**:   
       **void** showDamage(){cout << "Sodier damage: " << damage << endl;}   
       //для базового класса используется конструктор по умолчанию   
       Soldier():damage(20) { }    
       Soldier(int a):damage(a) { } //также health=10   
       //используется конструктор Unit(int a), т.е. health=a   
       Soldier(int a, int b):Unit(a),damage(b) { }    
};

То есть теперь, создавая объект класса Soldier, можно использовать как его поля, так и поля базового класса Unit:

int main()   
{   
       Soldier hercules(30);   
       hercules.showHealth();   
       hercules.showDamage();   
       Soldier achilles(40,50);   
       achilles.showHealth();   
       achilles.showDamage();   
       **return** 0;   
}

В результате у Геракла теперь 10 healthи 30 damage, а у Ахиллеса 40 healthи 50 damage. Сейчас я поясню несколько моментов, которые могли вызвать затруднения, но сначала необходимо изменить метод доступа поля health класса Unit с private на protected, чтобы код скомпилировался. Protected отличается от private тем, что protected не запрещает доступ к полям классам-наследникам.

Теперь о моментах:  
-при объявлении класса-наследника необходимо представить базовый класс:  
class Soldier : public Unit  
-при объявлении базового класса необходимо предоставить модификатор доступа (public, private, protected) перед именем базового класса. Модификатор public позволяет классу-наследнику наследовать поля базового класса так, как они объявлены в базовом классе, protected преобразует public в protected, а модификатор private наследует поля базового класса, как private, вне зависимости от того, как они заданы в базовом классе. Следует отметить, что наследование ни как не изменяет базовый класс  
-в представленном наследнике, для примера, введено несколько конструкторов. При создании объекта класса Soldier, в случае первых двух конструкторов для инициализации полей базового класса Unit будет использоваться его конструктор по умолчанию. Третий же конструктор Soldier вызывает конструктор с параметром базового Unit

Хорошо, теперь аналогичным образом создадим лошадей, которые в отличии от фермеров, будут двигаться, т.е. будут иметь параметр speed:

**class** Horse : **public** Unit   
{   
**private**:   
       int speed;   
**public**:   
       **void** showSpeed(){cout << "Horse speed: " << speed << endl;}   
       Horse():speed(20) { }   
};

Все лошади будут иметь 10 healthи 20 speed.

А теперь создадим всадников, которые будут иметь характеристики как фермеров(здоровье) и солдат (damage), так и лошадей (speed):

**class** Horseman : **public** Horse, **public** Soldier { };

И все бы ничего, но наследуя и Horse и Soldier, Horseman наследует 2 копии Unit. Решить эту проблему можно использовав ключевое слово virtual перед объявлениями о наследовании класса Unit (не путайте с виртуальными функциями):

**class** Soldier : **virtual** **public** Unit   
**class** Horse : **virtual** **public** Unit

Все, класс всадников создан! При создании объекта класса Horseman будут использоваться конструкторы по умолчанию базовых классов.

Допустим, что солдат тоже может двигаться (имеет параметр speed):

**class** Soldier : **virtual** **public** Unit   
{   
**protected**:   
       int speed; //имеет параметр скорости   
**public**:   
       //имеет метод вывода значения   
       //скорости на экран   
       **void** showSpeed(){cout << "Soldier speed: " << speed << endl;}    
       Soldier():speed(3){}                                       
};   
   
**class** Horse : **virtual** **public** Unit   
{   
**protected**:   
       int speed; //имеет параметр скорости как и Soldier   
**public**:   
       //имеет метод вывода значения   
       //скорости на экран как и Soldier   
       **void** showSpeed() { cout << "Horse speed: " << speed << endl; }    
       Horse():speed(20){ }    
};   
   
**class** Horseman : **public** Horse, **public** Soldier{}   
   
int main()   
{   
       Horseman lancelot;   
       //метод какого из базовых классов использовать?   
       //какое значение speed?   
       lancelot.showSpeed();    
   
       **return** 0;   
}

Неоднозначность можно решить дважды использовав оператор разрешения контекста:

lancelot.::Horse::showSpeed();

Но правильнее было бы переназначить поле в классе Horseman (код целиком):

#include <iostream>   
   
**using** **namespace** std;   
   
**class** Unit   
{   
**protected**:   
       int health;   
**public**:   
       **void** setHealth(int a) { health = a; }   
       **void** showHealth(){ cout << "Unit health: " << health << endl; };   
       Unit(): health(10) { }   
       Unit(int a): health(a) { }   
};   
   
**class** Soldier : **virtual** **public** Unit   
{   
**protected**:   
       int damage;   
       int speed;   
**public**:   
       **void** showDamage() { cout << "Sodier damage: " << damage << endl; }   
       **void** showSpeed() { cout << "Soldier speed: " << speed << endl; }   
       Soldier():damage(20), speed(3) { }                                 
};   
   
**class** Horse : **virtual** **public** Unit   
{   
**protected**:   
       int speed;   
**public**:   
       **void** showSpeed(){cout << "Horse speed: " << speed << endl;}   
       Horse():speed(20) { }                                
};   
   
**class** Horseman : **public** Horse, **public** Soldier   
{   
**private**:   
       int speed;   
**public**:   
       **void** showSpeed(){cout << "Horseman speed: " << speed << endl;}   
       Horseman():speed(Horse::speed){}   
};   
   
int main()   
{   
      Horseman lancelot;   
      lancelot.showHealth();   
      lancelot.showDamage();   
      lancelot.showSpeed();        
      **return** 0;   
}

открытое наследование означает Derived - это разновидность Base и все, что верно для объектов класса Base должно быть верным для Derived. В качестве базового класса взят Unit, но ПОЧЕМУ это фермер? Моделируется следующее: воин - это такая разновидность фермера (нонсенс). Фермер у нас строит здания, занимается добычей ресурсов...т.е. воин тоже должен делать это. И лошади должны. И всадники...А какая стратегия без зданий? Здания - тоже сделаем наследником от Unit, потому что Unit для этого подходит?) Замечание критическое. Статья описывает синтаксис, но содержит важные ошибки проектирования.  
  
P.S. Виталий, надеюсь, Вы не обидитесь на критику. Чем её больше - тем лучше, на мой взгляд :)

Думаю, в тексте статьи следует указать, что пример с наследованием Horseman от Horse и Soldier является удобным для демонстрационных целей, но также является неправильным с точки зрения ООП. Причиной тому является то, что наследование выражает отношение "является" ("is a"), которое в данном случае выражено неверно: всадник не является лошадью и солдатом (тем более одновременно).

следует отметить, что лишь public наследование означает "является", смысл private наследования - "реализуется посредством", при этом основная его задача - это не показать взаимосвязь, а использование его как механизма реализации.

**Основные понятия ООП**

1. **ООП** *- какие понятия явл основными концепциями?*

*- в чем отличие от С и С++*

**Пример 1** *Кофемолка*

*Инкапсуляция*

*Наследование*

*Полиморфизм*

**Пример 2** *Автомобиль*

*Инкапсуляция*

*Наследование*

*Полиморфизм*

**Пример 3***. Стратегия (Угольная шахта, Шахта для железной руды, Доменная печь )*

2. *Класс*

3. *Объект*

4. *Экземпляр класса*

5. *Инстанцирование*

6. *Инициализация*

7. *Время жизни объекта*

8. *Метод. Простые и static методы.*

9. *Set/get*

*- Интерфейс класса*

*- Реализация класса*

**ООП** — расширение структурного программирования, в котором основными концепциями являются понятия классов и объектов. Основное отличие языка программирования С++ от С состоит в том, что в С нет классов, а следовательно язык С не поддерживает ООП, в отличие от С++.

*Пример 1* **Кофемолка**. Объект кофимолка есть экземпляром класса Кофимолки. Класс – это как описание некоего шаблона, что есть кофемолка, она имеет размеры, формы, обладает свойствами(скорость вращения вентеля). Предназначение перемалывать зерна и варить кофе. В ООП у объектов тоже есть свойства. Экземпляр класса кофемолки – это конкретно наша кофемолка у которой есть какието свойства созданные по шаблону который гдето описан.и у которой есть какаято функция – молоть кофе. В ООП такие функции называются методами. С помощью методов можно взаимодействовать с объектом.

***Инкапсуляция*** – принцип независимости данных в ООП. Тоесть программист который разрабатывает класс некоего объекта может скрыть его свойства, его некий функционал который нужен для функционирования этого объекта от другого пользователя и оставить ему внешний элемент управления с помощью которого другой пользователь может работать с таким объектом. Если это применить к кофемолке то мы можем насыпать зерно, нажать кнопку перемолоть и достать порошок.(Если у нас простая кофемолка мы не можем влиять на скорость вращения шпинделя, на угол наклона лезвий) То есть скрытый функционал с котрым мы ничего не сможем сделать. Но у нас есть интерфейс с которым мы можем работать – кнопка вкл/выкл. А как реализован функционал мы не знаем и нас это не касается.

Инкапсуляция — это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.

***Наследование*** – это процесс с помощью которого один объект может наследовать свойства и методы другого объекта и дополнять их своим функционалом. К примеру описать класс кофе машины и создать его объект. Наследуем класс кофе машины от кофемолки т . к кофе машина должна уметь молоть кофе, засыпать зерна. Кофе машина получает метод молоть кофе и свойства кофемолки (например скорость вращения шпинделя)

Наследование — это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.

***Полиморфизм*** – это возможность объекта вести себя по разному в зависимости от ситуации и реагировать на одно действие каким то специфическим образом конкретно для этого класса объектов. Кофемолки бывают разные: электрическая, ручная. Реализация разная: механизм перемалует кофе с помощью эл. энергии или с помощью силы человека. Но метод будет тот же – молоть кофе.

Полиморфизм — свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

*Пример 2* **Автомобиль**.

**Инкапсуляция** *–* естьметоды ехать, тормозить, заправиться. Но совсем не обязательно знать как там все устроено. Функционал скрыт.

**Наследование** *–* класс транспортное средство которое может ехать ехать, тормозить, заправиться(абстрактный класс). Мы можем унаследовать легковой автомобиль(ехать, тормозить, заправиться бензином), фуру(ехать, тормозить отличается мощность, отличается топливом), можно наследовать электромобиль.

**Полиморфизм** - методы все одинаковые но выполняться на двигателе внутреннего сгорания и на эл. двигателе они будут по разному. Реализация разная но интерфейс один и тот же.

*Пример 3*. **Стратегия**. Объекты представляем как постройки.

Объекты:

1 Угольная шахта. Свойство: скорость добычи угля.и метод добывать.

2 Шахта для железной руды Свойство: скорость добыч и метод добывать

3 Доменная печь. Свойство: скорость выплавки и в конструкторе принимает два парамеира: уголь и железную руду. Метод добывать: выплавлять руду.

Есть абстрактный класс постройка у которого есть свойства скорость производства и метод добыть. От этого класса наследуем классы Угольная шахта, Шахта для железной руды, Доменная печь. У них у всех есть метод – добывать, но реализация у каждого своя. При этом у каждого класса должны быть свои специфические методы которые мы не видим потому что есть некие особенноси при добыче угля, железа, при выплвке метала. Эти функции скрыты в классах.

***- - -***

**Класс** – это пользовательский тип данных(пользователь – программист), который можно написать самостоятельно. Класс – это шаблон(абстракция) которая описывает методы, свойства, ещё не существующих объектов.

*Термины «***экземпляр класса***» и «***объект***» взаимозаменяемы.*

**Объект** — конкретное представление абстракции. Некоторая сущность, обладающая определённым состоянием и поведением, имеющая заданные значения свойств ([атрибутов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B5_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0)) и операций над ними ([методов](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5))). Объекты принадлежат одному или нескольким [классам](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), которые определяют поведение (являются моделью) объекта.

**Экземпляр класса** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) instance)  – созданный объект на основе одного класса. Класс описывает [свойства](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B5_%D0%BA%D0%BB%D0%B0%D1%81%D1%81%D0%B0) и [методы](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D0%BE%D0%B4_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), которые будут доступны у объекта, построенного по описанию, заложенному в классе. Экземпляры используются для представления (моделирования) конкретных сущностей реального мира. Например, экземпляром класса стиральных машин может быть ваша стиральная машина, имеющая следующие свойства: компания-производитель «Вятка», наименование модели «Вятка-автомат», серийный номер изделия ВЯТ454647, ёмкость 20 л.

**Инстанцирование** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) instantiation) — создание экземпляра класса. В отличие от слова «создание», применяется не к объекту, а к классу. То есть, говорят: (в виртуальной среде) создать экземпляр класса или, другими словами, инстанцировать класс. [Порождающие шаблоны](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D1%80%D0%BE%D0%B6%D0%B4%D0%B0%D1%8E%D1%89%D0%B8%D0%B5_%D1%88%D0%B0%D0%B1%D0%BB%D0%BE%D0%BD%D1%8B_%D0%BF%D1%80%D0%BE%D0%B5%D0%BA%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) используют [полиморфное](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D0%B8%D0%BC%D0%BE%D1%80%D1%84%D0%B8%D0%B7%D0%BC_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) инстанцирование.

**Инициализация** ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) initialization) — присвоение начальных значений полям объекта.

**Время жизни объекта** — время с момента создания объекта (конструкция) до его уничтожения (деструкция).

**Метод** - это [функция](https://ru.wikipedia.org/wiki/%D0%A4%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) или [процедура](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%B0_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)), принадлежащая какому-то [классу](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%81_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) или [объекту](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

Как и процедура в [процедурном программировании](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D0%B4%D1%83%D1%80%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), метод состоит из некоторого количества [операторов](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) для выполнения какого-то действия и имеет набор [входных аргументов](https://ru.wikipedia.org/wiki/%D0%90%D1%80%D0%B3%D1%83%D0%BC%D0%B5%D0%BD%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

Различают ***простые*** *методы* и ***статические*** *методы* (методы класса):

* простые методы имеют доступ к данным объекта (конкретного экземпляра данного класса),
* статические методы не имеют доступа к данным объекта, и для их использования не нужно создавать экземпляры (данного класса).

Методы предоставляют интерфейс, при помощи которого осуществляется доступ к данным объекта некоторого класса, тем самым, обеспечивая [инкапсуляцию данных](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%BA%D0%B0%D0%BF%D1%81%D1%83%D0%BB%D1%8F%D1%86%D0%B8%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)).

В зависимости от того, какой уровень доступа предоставляет тот или иной метод, выделяют:

* открытый (public) интерфейс — общий интерфейс для всех пользователей данного класса;
* защищённый (protected) интерфейс — внутренний интерфейс для всех наследников данного класса;
* закрытый (private) интерфейс — интерфейс, доступный только изнутри данного класса.

Такое разделение интерфейсов позволяет сохранять неизменным открытый интерфейс, но изменять внутреннюю реализацию.

## Set/get

Каждый объект имеет какие-то свои свойства или атрибуты, которые характеризуют его на протяжении всей жизни. Атрибуты объекта хранятся в переменных, объявленных внутри класса, которому принадлежит данный объект. Причём, объявление переменных должно выполняться со спецификатором доступа private. Такие переменные называются элементами данных. Так как элементы данных объявлены в private, то и доступ к ним могут получить только методы класса, внешний доступ к элементам данных запрещён. Поэтому принято объявлять в классах специальные методы — так называемые set и get

set и get функции, с помощью которых можно манипулировать элементами данных. **set-функции**инициализируют элементы данных, **get-функции**позволяют просмотреть значения элементов данных.

## *Отделение интерфейса от реализации*

**Интерфейс класса** — конструкция, определяющая методы и свойства, предоставляемые классом.

**Реализация класса** — это способ осуществления работоспособности класса. До этого мы не отделяли интерфейс класса от его реализации, то есть реализация методов осуществлялась внутри класса. Отделение интерфейса от реализации класса выполняется для того, чтобы скрыть способ осуществления работоспособности класса. Отделение интерфейса от реализации выполняется с помощью .**h** и .**cpp**

.