# [Некоторые вопросы касательно плюсов (не факт, что будет, но лучше быть в теме)](http://neerc.ifmo.ru/wiki/index.php?title=%D0%A1%D1%82%D0%B0%D1%80%D1%8B%D0%B5_%D0%B2%D0%BE%D0%BF%D1%80%D0%BE%D1%81%D1%8B_%D0%BD%D0%B0_%D1%8D%D0%BA%D0%B7%D0%B0%D0%BC%D0%B5%D0%BD_%D0%BF%D0%BE_C%2B%2B)

# Ссылки. Разница между ссылкой и указателем. Зачем в С++ были добавлены ссылки.

## *Что такое ссылка?*

## Ссылки - это особый тип данных, являющийся скрытой формой указателя, который при использовании автоматически разыменовывается.

## При объявлении ссылки на объект перед ее именем ставится амперсанд (&), а также необходимо сразу же ее проинициализировать именем объекта, на который она будет ссылаться:

## **#include <iostream> int main() { int foo = 5; *// создаем и инициализируем объект* int &foo\_ref = foo; *// создаем ссылку* return 0; }**

Сейчас при попытке получить значение ссылки мы получим на выход значение, на которое она ссылается: 5. Любая операция (присваивания/ инкремента/декремента etc.) над ссылкой изменит ее значение, а как следствие - значение объекта, над которой она ссылается.

**std::cout << foo << " " << foo\_ref; *// 5 5* ++foo\_ref;  
 std::cout << foo << " " << foo\_ref; *// 6 6***

Если в ходе работы программы вам не нужно изменять значение по ссылке, то вдобавок к обычной существует и константная ссылка.

Обрати внимание: к неконстантному объекту применима как неконстантная, так и константная ссылки, а к константному объекту - только константная (что очевидно).

***/\*изменить значение foo по такой ссылке нельзя, но вы гарантируете себе то, что данные в foo сохранятся в первозданном виде \*/***

**const int &foo\_const\_ref = foo;**

***/\* к конст. переменной нельзя применить обычную ссылку \*/***

**const int const\_foo = 100500;**

**int &foo\_second\_ref = const\_foo; *// не скомпилируется***

## *Предназначение ссылки в C++ и отличие ее от указателя*.

Объективно обоснованное применение ссылки в C++ - это передача некоторого объекта в функцию по значению с целью разумного использования памяти:

**#include <iostream>**

**#include <vector>  
  
 void foo(std::vector<int> vec);**

**void boo(const std::vector<int>& vec);**

**int main()**

**{**

**std::vector<int> same\_vector;**

**for (int i = 0; i < 100; ++i)**

**same\_vector.push\_back(i);**

**foo(same\_vector); *// скопировали в функцию лишние 400 байт***

**boo(same\_vector); *// работаем с same\_vector по ссылке напрямую***

**return 0;**

**}**

Также можно давать переменным другие имена и псевдонимы.

## *Зачем ссылка, если есть указатель и в чем их отличие?*

Указатель - объект, который, по хорошему, предназначен для организации структур данных (в частности, динамических) и позволяющий менять свой собственный адрес для перемещения по структуре, у него присутствует своя арифметика:

**int \*data = new int(10);**

**data[0] = 0;**

**data[1] = 100500;**

**data[3] = 1337;**

**int \*p = data; *// присваиваем указателю зн. указателя на начало дин. памяти***

**std::cout << \*p << std::endl; // 0**

**++p; *// инкремент указателя перемещает его на 1 int-овую ячейку (т. е. на 4 байта) вперед, т. е. меняет адрес, на который он указывает, а не значение***

**std::cout << \*p << std::endl; // 100500**

**std::cout << \*(p + 2) << std::endl; // 1337**

На примере указателя ссылка является операцией его разыменовывания, а арифметические операции по ссылке - это арифметические операции со значением по адресу, которое хранит указатель:

**int foo = 5;**

**int \*foo\_p = &foo; *// указатель***

**int &foo\_ref = foo; *// ссылка***

**++foo; *// инкремент напрямую по объекту, foo = 6;***

**++foo\_ref; *// инкремент по ссылке, foo = 7;***

**++(\*foo\_p); *// инкремент значения по адресу указателя, foo = 8, эта операция полностью идентична предыдущей***

Ссылка в C++ была придумана с целью упростить такой костыль и позволить работать со значением объекта напрямую, без копирований объекта, на сленге это называется «синтаксическим сахаром» над указателем.

## *Технические отличия ссылки от указателя:*

1. Указатель можно переназначать, ссылка же инициализируется единожды и может ссылаться только на один лишь объект;
2. Указатель имеет свой собственный адрес памяти и размер в стеке (4 или 8 байт, в зависимости от архитектуры), насчет ссылки спорно: с одной стороны, ее адрес совпадает с переменной, на которую она ссылается; с другой - она занимает некоторое пространство в стеке и тоже имеет свой адрес (но его компилятор вам никогда не сообщит по причине одного уровня косвенности ссылок - в пунктах далее);
3. Уровень косвенности указателя может быть любым:

**int a = 5;**

**int \*pa = &a;**

**int \*\*ppa = &p;**

**int \*\*\*pppa = &ppa;**

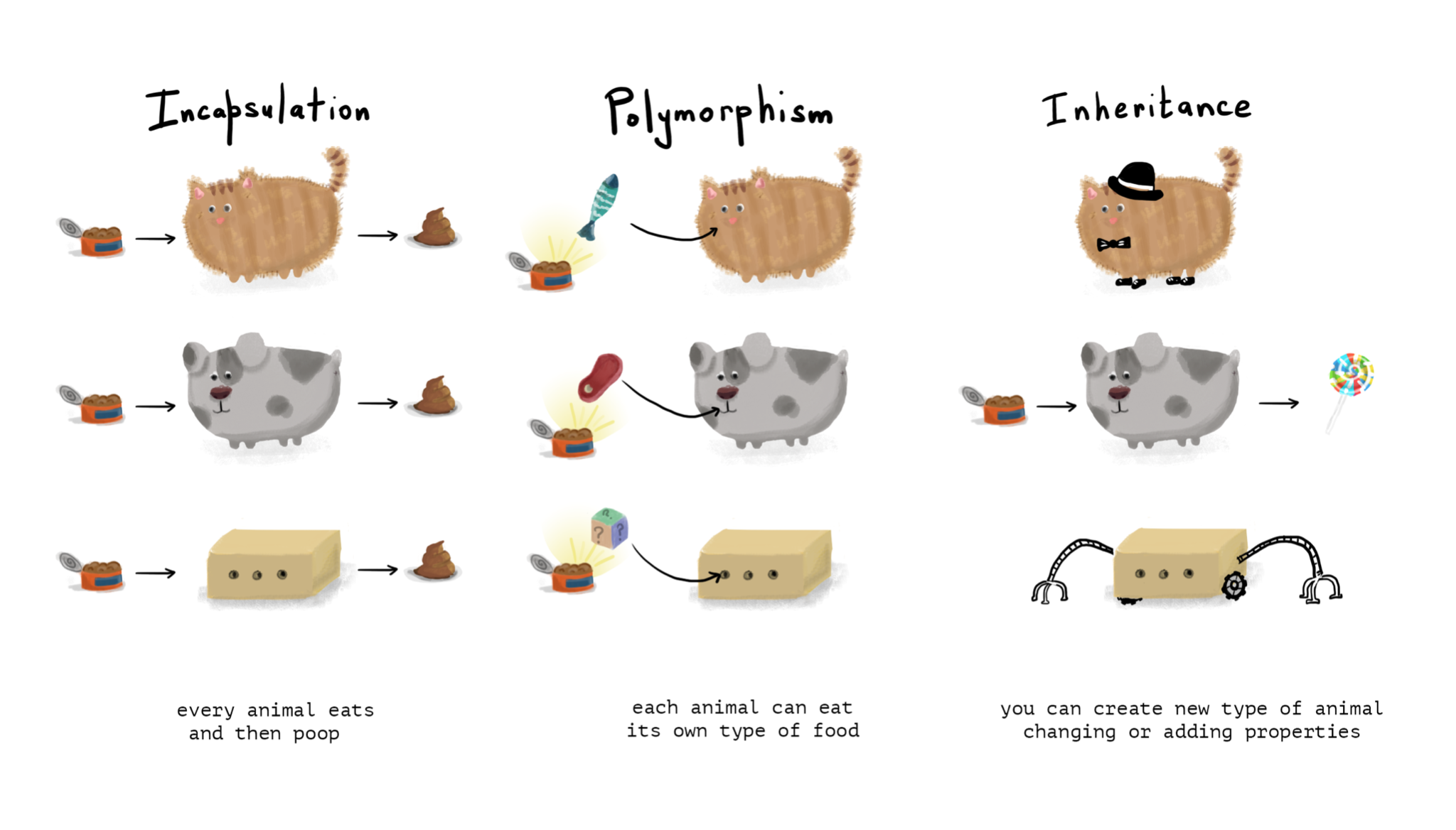
У ссылок же - только один:

**int &a\_ref = a; // всё!**

1. По ссылке нельзя передать nullptr;
2. Указателями можно перемещаться по дин. памяти, по ссылке доступен лишь 1 объект;
3. Ссылку не нужно разыменовывать для доступа к значению, а также доступ к членам класса по ссылке - всегда через . (по указателю - через ->);
4. Из ссылок нельзя сделать массив;
5. Константные ссылки могут быть привязаны к временным объектам, указатели - нет:

# ООП. Базовые принципы. Абстракция, инкапсуляция, наследование, полиморфизм. SOLID.

## **Базовые принципы**



## **Абстракция**

Это придание объекту характеристик, которые чётко определяют его концептуальные границы, отличая от всех других объектов.

Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов

## **Инкапсуляция**

Механизм языка, позволяющий ограничить доступ одних компонентов программы к другим;

Языковая конструкция, позволяющая связать данные с методами, предназначенными для обработки этих данных.

## **Полиморфизм**

Возможность объектов с одинаковой спецификацией иметь различную реализацию.

## **Наследование**

Наследование является одним из важнейших принципов объектно-ориентированного программирования, поскольку оно позволяет создавать иерархические структуры объектов. Используя наследование, можно создать общий класс, который будет определять характеристики и поведение, свойственные какому-то набору связанных объектов. В дальнейшем этот класс может быть унаследован другими, более частными классами, каждый из которых будет добавлять уникальные, свойственные только ему характеристики и дополнять или изменять поведение базового класс

## **SOLID**

[Шикарная статья на эту тему на русском](https://medium.com/webbdev/solid-4ffc018077da)

Вот как расшифровывается акроним SOLID:

* **S: Single Responsibility Principle** (Принцип единственной ответственности).
* **O: Open-Closed Principle** (Принцип открытости-закрытости).
* **L: Liskov Substitution Principle** (Принцип подстановки Барбары Лисков).
* **I: Interface Segregation Principle** (Принцип разделения интерфейса).
* **D: Dependency Inversion Principle** (Принцип инверсии зависимостей).

**Принцип единственной обязанности / ответственности (single responsibility principle / SRP)** означает, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все его сервисы должны быть направлены исключительно на обеспечение этой обязанности.

**Принцип открытости / закрытости (open-closed principle / OCP)** декларирует, что программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения. Это означает, что эти сущности могут менять свое поведение без изменения их исходного кода.

**Принцип подстановки Барбары Лисков (Liskov substitution principle / LSP)** в формулировке Роберта Мартина: «функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа не зная об этом».

**Принцип разделения интерфейса (interface segregation principle / ISP)** в формулировке Роберта Мартина: «клиенты не должны зависеть от методов, которые они не используют». Принцип разделения интерфейсов говорит о том, что слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

**Принцип инверсии зависимостей (dependency inversion principle / DIP)** — модули верхних уровней не должны зависеть от модулей нижних уровней, а оба типа модулей должны зависеть от абстракций; сами абстракции не должны зависеть от деталей, а вот детали должны зависеть от абстракций.

# ООП. Классы. Устройство в памяти. Инкапсуляция. Модификаторы доступа. Наследование. Множественное наследование. Проблемы множественного наследования.

## **Классы**

Класс – это способ описания сущности, определяющий состояние и поведение, зависящее от этого состояния, а также правила для взаимодействия с данной сущностью (контракт).

С точки зрения программирования класс можно рассматривать как набор данных (полей, атрибутов, членов класса) и функций для работы с ними (методов).

## **Устройство в памяти**

## **Инкапсуляция**

Это механизм, который объединяет данные и код, манипулирующий этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В объектно-ориентированном программировании код и данные могут быть объединены вместе; в этом случае говорят, что создаётся так называемый "чёрный ящик". Когда коды и данные объединяются таким способом, создаётся объект (object). Другими словами, объект - это то, что поддерживает инкапсуляцию.

Внутри объекта коды и данные могут быть закрытыми (private). Закрытые коды или данные доступны только для других частей этого объекта. Таким образом, закрытые коды и данные недоступны для тех частей программы, которые существуют вне объекта. Если коды и данные являются открытыми, то, несмотря на то, что они заданы внутри объекта, они доступны и для других частей программы. Характерной является ситуация, когда открытая часть объекта используется для того, чтобы обеспечить контролируемый интерфейс закрытых элементов объекта.

## **Модификаторы доступа**

Уровень доступности позволяет регулировать доступ к данным и методам класса. Одни элементы класса могут быть общедоступны, другие — скрыты. Сокрытие данных и методов в классе называется *инкапсуляцией*.

В языке C++ уровень доступности определяется с помощью специальных модификаторов private, protected, public. Применение этих модификаторов может быть использовано в двух случаях:

* в случае объявления элементов класса. Этот случай рассматривается в данной теме;
* в случае наследования классов.

### **Модификатор доступа private. Особенности использования**

Если в классе объявляется элемент (функция, член данных) в разделе private, то для него действуют следующие правила доступа:

* элемент **недоступен** любым экземплярам методов других классов или методов, которые не являются «дружественными» к классу
* элемент **недоступен** из унаследованных классов
* элемент доступен из методов, которые реализованы в классе
* элемент доступен из дружественных функций
* элемент доступен из методов дружественных классов

### **Модификатор доступа public. Особенности использования**

Если в классе определен элемент с модификатором доступа public, в этом случае справедливо правило:

* элемент доступен всем методам в программе.

Исключение составляет случай, когда класс унаследован как private. Тогда даже public-элементы этого класса будут недоступны в унаследованных классах

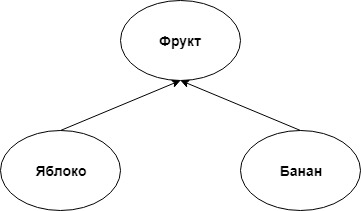
### **Модификатор доступа protected. Особенности применения**

Модификатор доступа protected актуален в случаях, когда классы образуют иерархию наследования. Если в классе объявлен элемент (функция, член данных) с модификатором доступа protected, то для него действуют следующие правила:

* элемент класса недоступен из любого внешнего метода, если этот метод не является дружественным;
* элемент класса доступен из внутренних методов класса. Здесь следует заметить, что protected-элемент класса также доступен из экземпляра класса, если этот экземпляр объявляется во внутреннем методе класса;
* элемент класса доступен из дружественных функций класса
* элемент класса доступен из методов дружественного класса
* элемент класса доступен из методов унаследованного класса
* элемент класса недоступен из экземпляров унаследованного класса. Это правило не касается «дружественных» методов и методов «дружественных» классов.

## **Наследование**

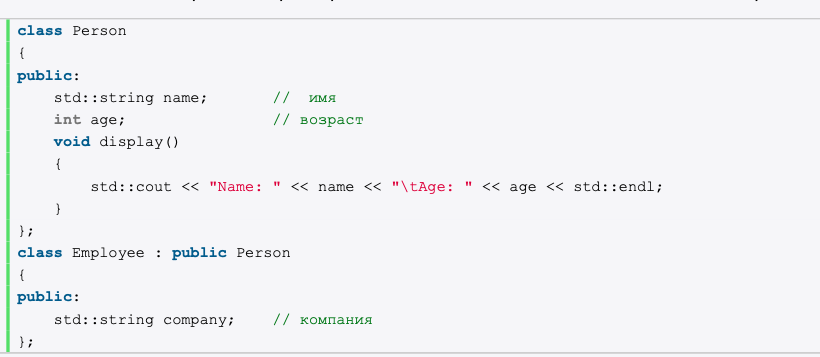
**Наследование** представляет один из ключевых аспектов объектно-ориентированного программирования, который позволяет наследовать функциональность одного класса или базового класса (base class) в другом - производном классе (derived class).



В диаграмме, представленной выше, Фрукт является родительским классом, а Яблоко и Банан — дочерними классами

Для установки отношения наследования после название класса ставится двоеточие, затем идет название класса, от которого мы хотим унаследовать функциональность. В этом отношении класс Person еще будет называться базовым классом, а Employee - производным классом.

Перед названием базового класса также можно указать спецификатор доступа, как в данном случае используется спецификатор **public**, который позволяет использовать в производном классе все открытые члены базового класса. Если мы не используем модификатор доступа, то класс Employee ничего не будет знать о переменных name и age и функции display.



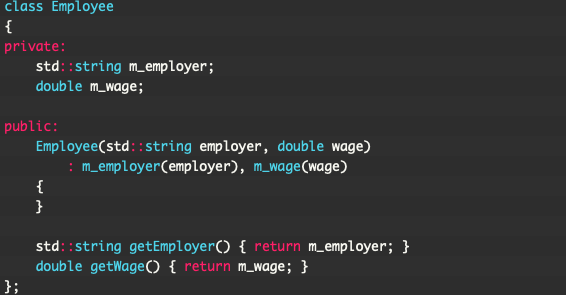
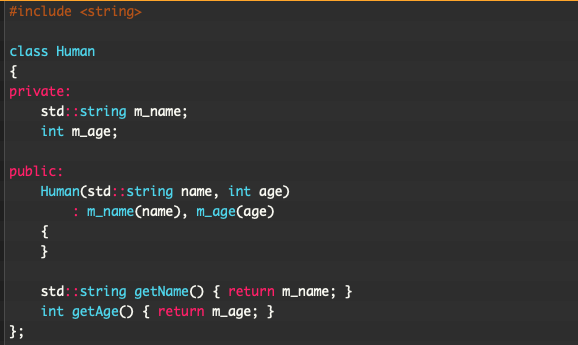
*Конструкторы при наследовании не наследуются. И если базовый класс содержит только конструкторы с параметрами, то производный класс должен вызывать в своем конструкторе один из конструкторов базового класса*

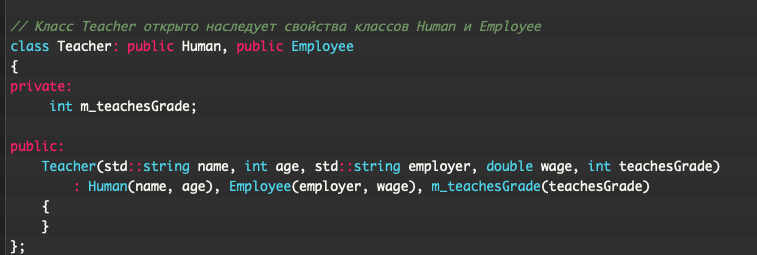
## **Множественное наследование**

Множественное наследование позволяет одному дочернему классу иметь несколько родителей. Предположим, что мы хотим написать программу для отслеживания работы учителей. Учитель — это Human. Тем не менее, он также является Сотрудником (Employee).



Множественное наследование может быть использовано для создания класса Teacher, который будет наследовать свойства как Human, так и Employee. Для использования множественного наследования нужно просто указать через запятую тип наследования и второй родительский класс:



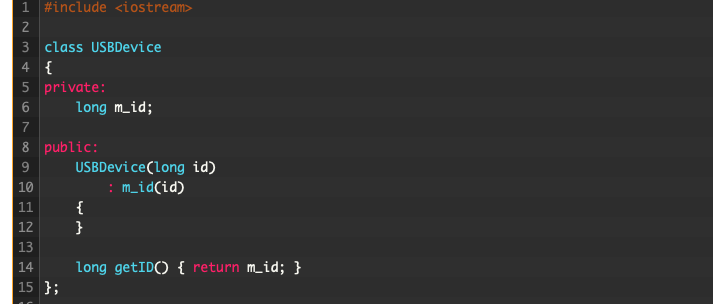


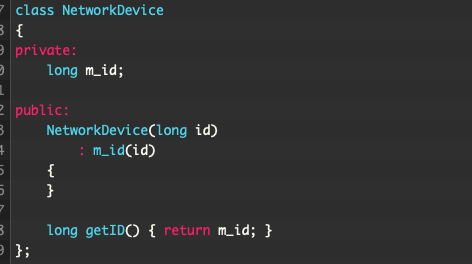
## 

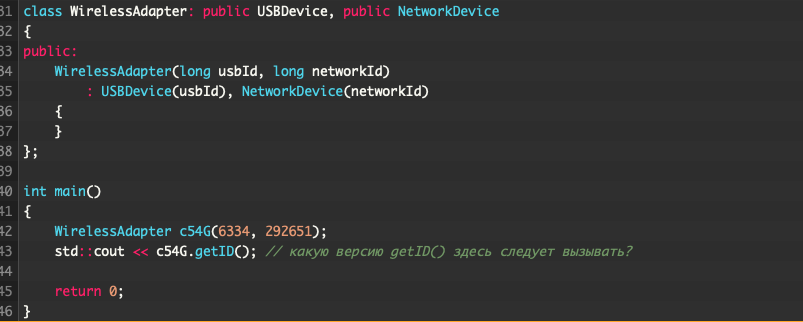
## **Проблемы с множественным наследованием**

Хотя множественное наследование кажется простым расширением одиночного наследования, оно может привести к множеству проблем, которые могут заметно увеличить сложность программ и сделать кошмаром дальнейшую поддержку кода. Рассмотрим некоторые из подобных ситуаций.

Во-первых, может возникнуть неоднозначность, когда несколько родительских классов имеют метод с одним и тем же именем, например:

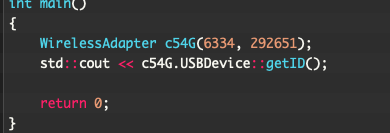






При компиляции c54G.getID() компилятор смотрит, есть ли у WirelessAdapter метод getID(). Этого метода у него нет, поэтому компилятор двигается по цепочке наследования вверх и смотрит, есть ли этот метод в каком-либо из родительских классов. И здесь возникает проблема — getID() есть как у USBDevice, так и у NetworkDevice. Следовательно, вызов этого метода приведет к неоднозначности и мы получим ошибку, так как компилятор не будет знать какую версию getID() ему вызывать.

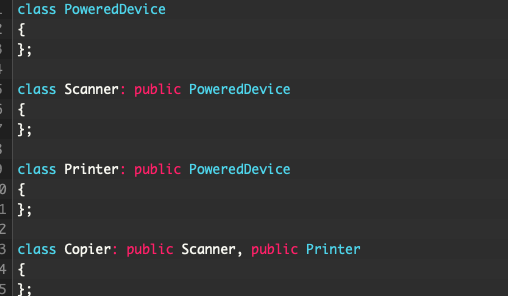
Тем не менее, есть способ обойти эту проблему. Мы можем явно указать, какую версию getID() следует вызывать:



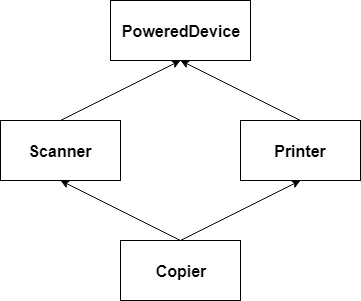
Хотя это решение довольно простое, но всё может стать намного сложнее, если наш класс будет иметь от 4-х родительских классов, которые, в свою очередь, будут иметь свои родительские классы. Возможность возникновения [**конфликтов имен**](https://ravesli.com/urok-24-konflikt-imen-i-std-namespace/) увеличивается экспоненциально с каждым добавленным родительским классом, и в каждом из таких случаев нужно будет явно указывать версии методов, которые следует вызывать, дабы избежать возможности возникновения конфликтов имен.

Во-вторых, более серьезной проблемой является **«алмаз смерти»** (или *«алмаз обреченности»*). Это ситуация, когда один класс имеет 2 родительских класса, каждый из которых, в свою очередь, наследует свойства одного и того же родительского класса. Иллюстративно мы получаем форму алмаза.

Например, рассмотрим следующие классы:



Сканеры и принтеры — это устройства, которые получают питание от розетки, поэтому они наследуют свойства PoweredDevice. Однако ксерокс (Copier) включает в себя функции как сканеров, так и принтеров.



В этом контексте возникает много проблем, включая неоднозначность при вызове методов и копирование данных PoweredDevice в класс Copier дважды. Хотя большинство из этих проблем можно решить с помощью явного указания, поддержка и обслуживание такого кода может привести к непредсказуемым временным затратам

все же остается вероятность, когда множественное наследование будет лучшим решением, нежели придумывание двухуровневых [**«костылей»**](https://ravesli.com/slovar-programmista-sleng-kotoryj-dolzhen-znat-kazhdyj-koder/).

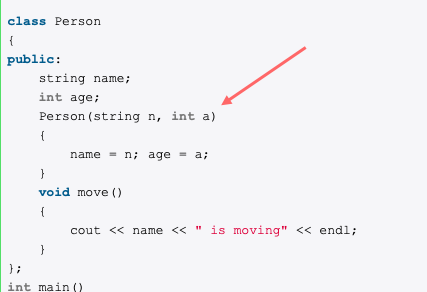
Стоит отметить, что вы сами уже использовали классы, написанные с использованием множественного наследования, даже не подозревая об этом: такие объекты, как std::cin и std::cout библиотеки iostream, реализованы с использованием множественного наследования!

**Правило: Используйте множественное наследование только в крайних случаях, когда задачу нельзя решить одиночным наследованием, либо другим альтернативным способом (без изобретения «велосипеда»).**

# ООП. Классы. Инициализация и уничтожение. Виды конструкторов. Виртуальные методы. Таблица виртуальных функций. Виртуальные деструкторы.

## **Инициализация и уничтожение классов**

***Конструктор*** — функция, предназначенная для инициализации объектов класса. Конструктор всегда имеет то же имя, что и сам класс и никогда не имеет возвращаемого значения. Когда класс имеет конструктор, все объекты этого класса будут проинициализированы.



Если конструктор требует аргументы, их следует указать.

Если необходимо обеспечить несколько способов инициализации объектов класса, задается несколько конструкторов

Конструкторы подчиняются тем же правилам относительно типов параметров, что и [перегруженные функции](https://prog-cpp.ru/cpp-function/#reload). Если конструкторы существенно различаются по типам своих параметров, то компилятор при каждом использовании может выбрать правильный

**Деструкторы**

Определяемый пользователем класс имеет конструктор, который обеспечивает надлежащую инициализацию. Для многих типов также требуется обратное действие. Деструктор обеспечивает соответствующую очистку объектов указанного типа. Имя деструктора представляет собой имя класса с предшествующим ему знаком «тильда» ~. Так, для класса X деструктор будет иметь имя ~X(). Многие классы используют динамическую память, которая выделяется конструктором, а освобождается деструктором.



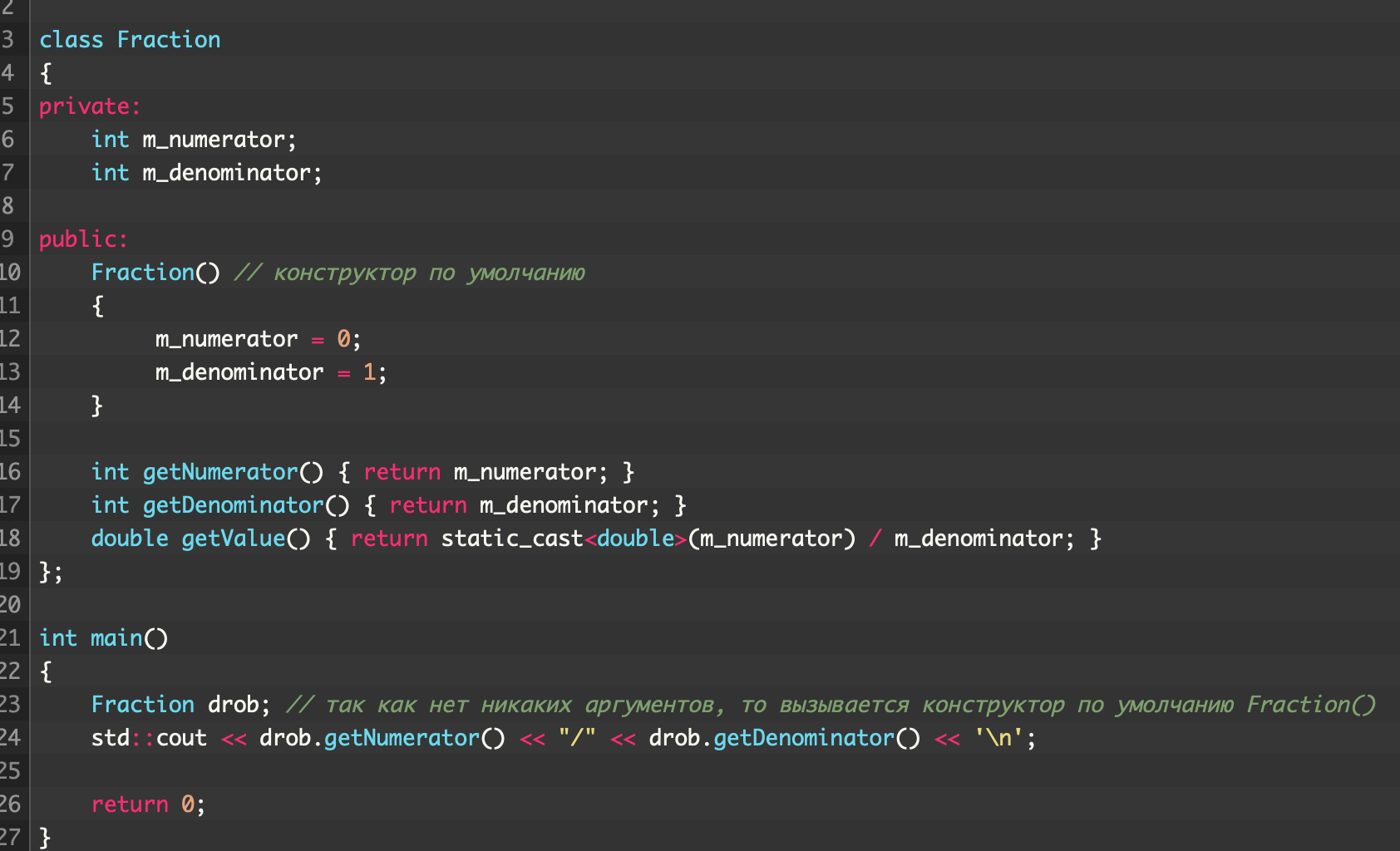
Деструктор выполняет освобождение использованных объектом ресурсов и удаление нестатических переменных объекта. По сути деструктор - это функция, которая называется по имени класса (как и конструктор) и перед которой стоит тильда (~). Деструктор не имеет возвращаемого значения и не принимает параметров. Каждый класс может иметь только один деструктор.

Деструктор автоматически вызывается, когда удаляется объект. Удаление объекта происходит в следующих случаях:

* когда завершается выполнение области видимости, внутри которой определены объекты
* когда удаляется контейнер (например, массив), который содержит объекты
* когда удаляется объект, в котором определены переменные, представляющие другие объекты
* динамически созданные объекты удаляются при применении к указателю на объект оператора **delete**

## **Виды конструкторов**

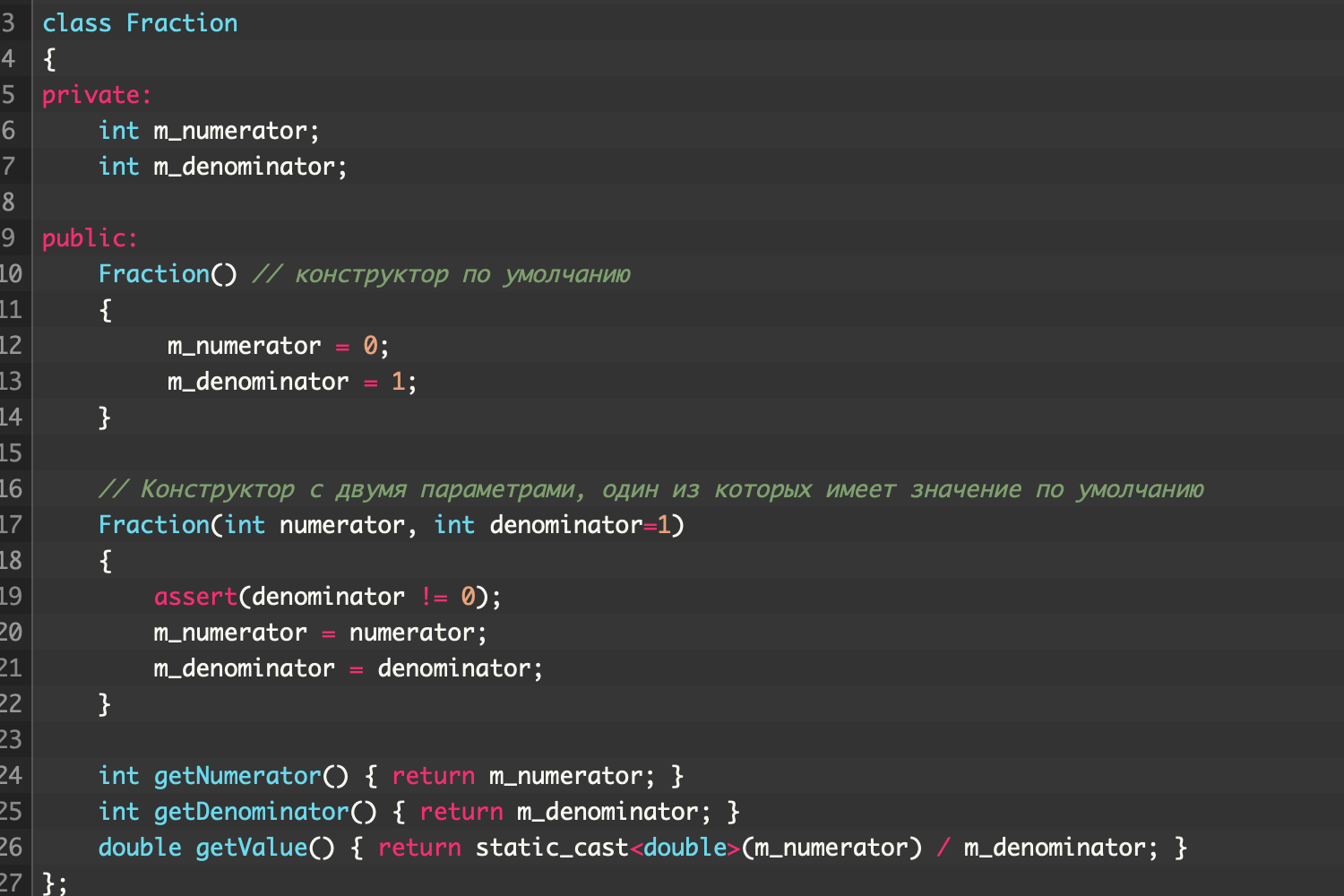
### **Конструктор, который не имеет параметров (или содержит параметры, которые все имеют** [**значения по умолчанию**](https://ravesli.com/urok-103-parametry-argumenty-po-umolchaniyu/)**), называется конструктором по умолчанию.** Он вызывается, если пользователем не указаны значения для инициализации. Например:



Поскольку мы создали объект класса Fraction без аргументов, то конструктор по умолчанию сработал сразу же после выделения памяти для объекта, и инициализировал наш объект.

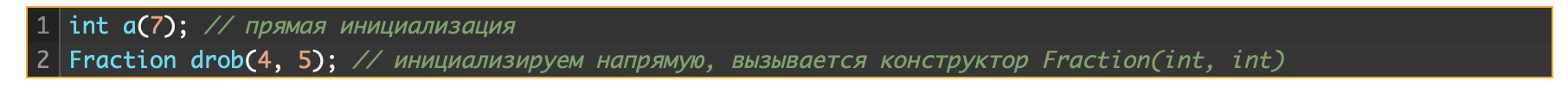
### **Конструкторы с параметрами**

Хотя конструктор по умолчанию отлично подходит для обеспечения инициализации наших классов значениями по умолчанию, часто может быть нужно, чтобы экземпляры нашего класса имели определенные значения, которые мы предоставим позже. К счастью, конструкторы также могут быть объявлены с параметрами. Вот пример конструктора, который имеет два целочисленных параметра, которые используются для инициализации числителя и знаменателя:



Теперь у нас есть два конструктора: конструктор по умолчанию, который будет вызываться, если мы не предоставим значения, и конструктор с параметрами, который будет вызываться, если мы предоставим значения. Эти два конструктора могут мирно сосуществовать в одном классе благодаря [**перегрузке функций**](https://ravesli.com/urok-102-peregruzka-funktsij/). Фактически, вы можете определить любое количество конструкторов до тех пор, пока у них будут уникальные параметры (учитывается их количество и тип).

Как использовать конструктор с параметрами? Всё просто! Прямая инициализация:

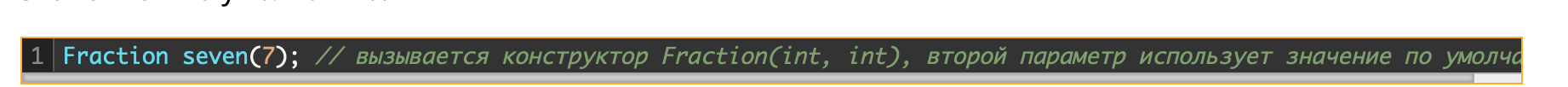


Здесь мы инициализировали нашу дробь числами 4 и 5, результат — 4/5!

В C++11 мы также можем использовать uniform-инициализацию:



Мы также можем указать только один параметр для конструктора с параметрами, а второе значение будет значением по умолчанию:



Значения по умолчанию для конструкторов работают точно так же, как и для любой другой функции, поэтому в вышеприведенном примере, когда мы вызываем seven(7), вызывается Fraction(int, int), второй параметр которого равен 1 (значение по умолчанию).

**Правило: Используйте прямую инициализацию или uniform-инициализацию с объектами ваших классов.**

### **Конструктор копирования**

**Конструктор копирования** — это особый тип конструктора, который используется для создания нового объекта через копирование существующего объекта. И, как в случае с конструктором по умолчанию, если вы не предоставите конструктор копирования для своих классов самостоятельно, то язык C++ создаст [**public**](https://ravesli.com/urok-114-spetsifikatory-dostupa-public-i-private/)-конструктор копирования автоматически. Поскольку компилятор мало знает о вашем классе, то по умолчанию созданный конструктор копирования будет использовать почленную инициализацию.

**Почленная инициализация** означает, что каждый член объекта-копии инициализируется непосредственно из члена объекта-оригинала



Конструктор копирования в вышеприведенном примере использует почленную инициализацию и функционально эквивалентен конструктору по умолчанию, за исключением того, что мы добавили стейтмент вывода, в котором указали текст (сработал конструктор копирования).

## **Виртуальные методы**

***Виртуальные функции*** — специальный вид функций-членов класса. Виртуальная функция отличается от обычной функции тем, что для обычной функции связывание вызова функции с ее определением осуществляется на этапе компиляции. Для виртуальных функций это происходит во время выполнения программы.

Для объявления виртуальной функции используется ключевое слово virtual. Функция-член класса может быть объявлена как виртуальная, если

* класс, содержащий виртуальную функцию, базовый в иерархии порождения;
* реализация функции зависит от класса и будет различной в каждом порожденном классе.

***Виртуальная функция*** — это функция, которая определяется в базовом классе, а любой порожденный класс может ее переопределить. Виртуальная функция вызывается только через указатель или ссылку на базовый класс.

## **Чистая виртуальная функция**

Базовый класс иерархии типа обычно содержит ряд виртуальных функций, которые обеспечивают динамическую типизацию. Часто в самом базовом классе сами виртуальные функции фиктивны и имеют пустое тело. Определенное значение им придается лишь в порожденных классах. Такие функции называются ***чистыми виртуальными функц иями***.

***Чистая виртуальная функция*** — это метод класса, тело которого не определено.

В базовом классе такая функция записывается следующим образом:

**virtual ПрототипФункции = 0;**

***Например***

virtual void func() = 0;

Чистая виртуальные функции используются для того, чтобы отложить решение задачи о реализации функции на более поздний срок. В терминологии ООП это называется ***отсроченным методом***. Класс, имеющий по крайней мере одну чистую виртуальную функцию, называется ***абстрактным базовым классом***. Для иерархии типа полезно иметь абстрактный базовый класс. Он содержит общие свойства иерархии типа, но каждый порожденный класс реализует эти свойства по-своему.

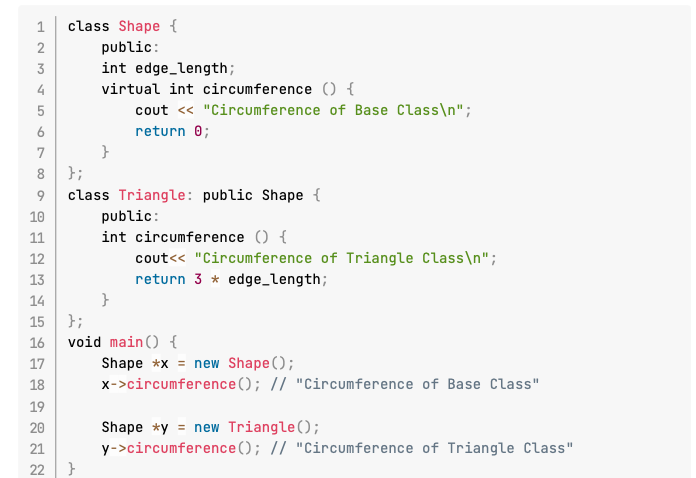
## **Таблица виртуальных функций**

**Виртуальная таблица в языке С++** — это таблица поиска функций для выполнения вызовов функций в режиме позднего (динамического) связывания. Виртуальную таблицу еще называют ***«vtable»***, ***«таблицей виртуальных функций»*** или ***«таблицей виртуальных методов»***.

Если какая-либо функция класса объявлена как виртуальная, создастся vtable, которая хранит адреса виртуальных функций этого класса. Для всех таких классов компилятор добавляет скрытую переменную vptr, которая указывает на vtable. Если виртуальная функция не переопределена в производном классе, vtable производного класса хранит адрес функции в родительском классе. Таблица vtable используется для получения доступа к адресу при вызове виртуальной функции. Механизм vtable позволяет реализовать динамическое связывание в C++.

Когда мы связываем объект производного класса с указателем базового класса, переменная vptr указывает на vtable производного класса. Это присвоение гарантирует, что будет вызвана нужная виртуальная функция.

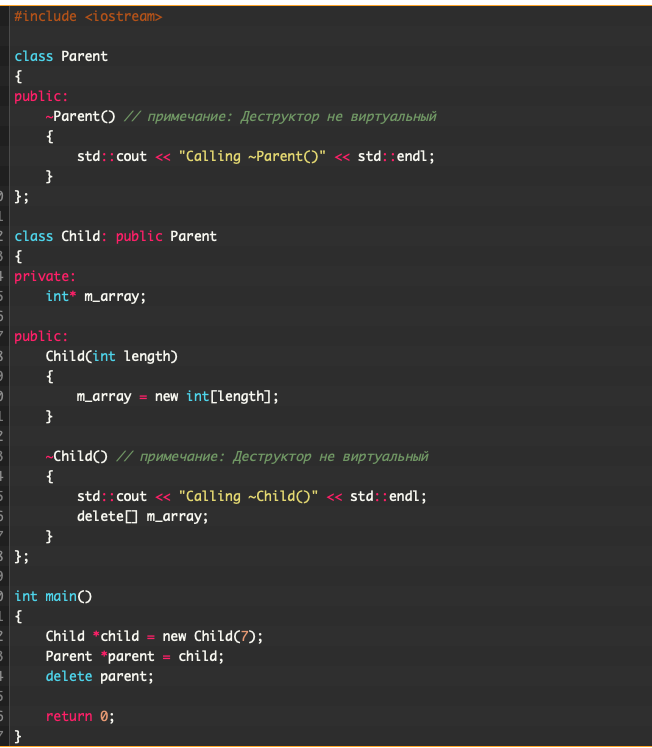
Рассмотрим следующий код:



В предыдущем примере функция circumference — виртуальная функция из класса Shape, значит, она является виртуальной в каждом из произвольных классов (Triangle и т.д.). В C++ разрешены вызовы невиртуальных функций во время компиляции со статическим связыванием, а вызовы виртуальной функции допускаются при динамическом связывании.

## **Виртуальные деструкторы**

**При работе с наследованием ваши деструкторы всегда должны быть виртуальными**. Рассмотрим следующий пример:

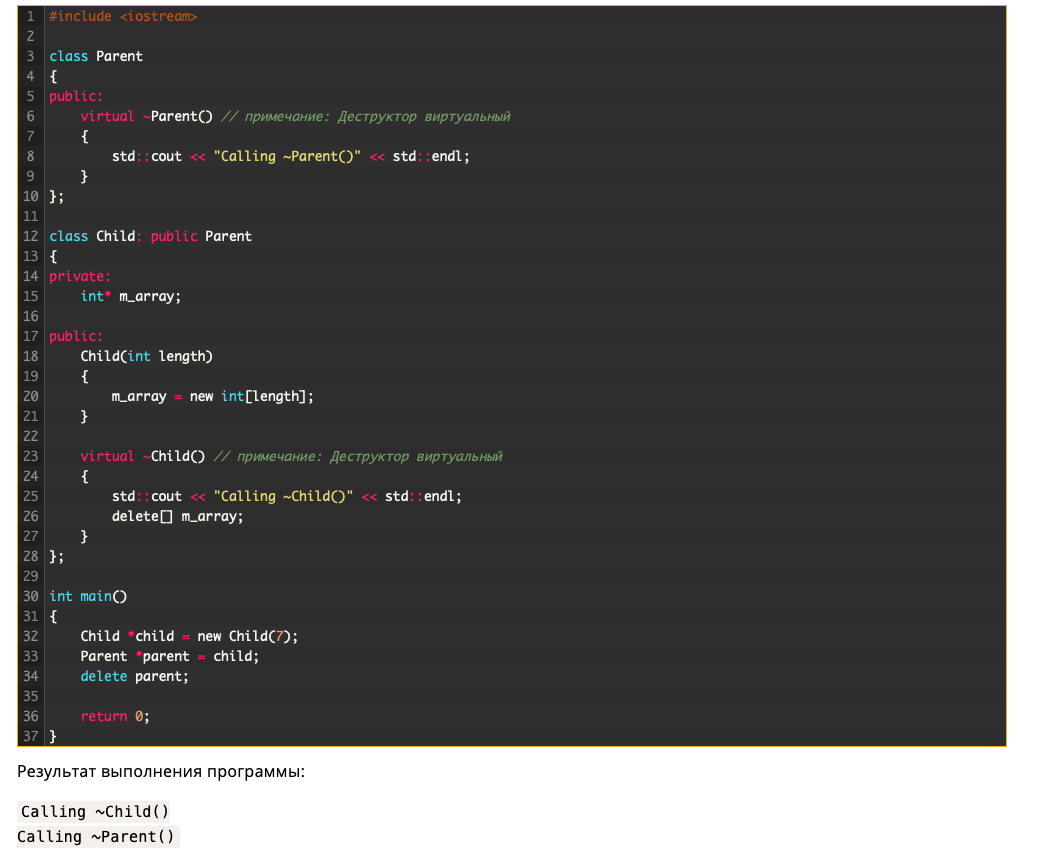


Поскольку parent является [**указателем**](https://ravesli.com/urok-80-ukazateli-vvedenie/) класса Parent, то при его уничтожении компилятор будет смотреть, является ли деструктор класса Parent виртуальным. Поскольку это не так, то компилятор вызовет только деструктор класса Parent.

Результат выполнения программы:

Calling ~Parent()

Тем не менее, нам нужно, чтобы [**delete**](https://ravesli.com/urok-143-konstruktory-preobrazovaniya-klyuchevye-slova-explicit-i-delete/) вызывал деструктор класса Child (который, в свою очередь, будет вызывать деструктор класса Parent), иначе m\_array не будет удален. Это можно выполнить, сделав деструктор класса Parent виртуальным:



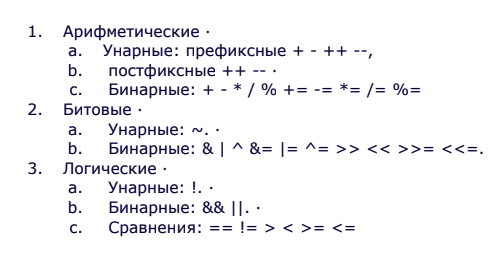
Когда же следует объявлять деструктор виртуальным? Существует правило - *если базовый класс предназначен для полиморфного использования, то его деструктор должен объявляться виртуальным*.

Для реализации механизма виртуальных функций каждый объект класса хранит указатель на таблицу виртуальных функций vptr, что увеличивает его общий размер. Обычно, при объявлении виртуального деструктора такой класс уже имеет виртуальные функции, и увеличения размера соответствующего объекта не происходит.

Если же базовый класс не предназначен для полиморфного использования (не содержит виртуальных функций), то его деструктор не должен объявляться виртуальным.

# ООП. Операторы. Перегрузка операторов.

## **Операторы**



***С лекции***

--------------------------------------------------------

Существуют следующие операторы:

* Арифметические операторы
* Реляционные операторы
* Логические операторы
* Побитовые операторы
* Операторы присваивания
* Другие операторы

По своему типу они делятся еще на несколько:

* Унарные (взаимодействуют только с одним операндом)
* Бинарные (взаимодействуют с двумя операндами)
* Тернарные (по аналогии, с тремя)

## **Арифметические операторы**

| **Оператор** | **Описание** | **Пример** |
| --- | --- | --- |
| + | Добавляет два операнда | A + B даст 30 |
| - | Вычитает второй операнд с первого | A - B даст -10 |
| \* | Умножает оба операнда | A \* B даст 200 |
| / | Делит числитель на де-числитель | B / A даст 2 |
| % | Оператор модуля и остаток после целочисленного деления | B% A даст 0 |
| ++ | Оператор приращения увеличивает целочисленное значение на единицу | A++ даст 11 |
| -- | Уменьшает целочисленное значение на единицу | A-- даст 9 |

## 

## **Реляционные операторы**

| **Оператор** | **Описание** | **Пример** |
| --- | --- | --- |
| == | Проверяет, равны ли значения двух операндов или нет, если да, то условие становится истинным. | (A == B) не соответствует действительности. |
| != | Проверяет, равны ли значения двух операндов или нет, если значения не равны, условие становится истинным. | (A! = B) истинно. |
| > | Проверяет, превышает ли значение левого операнда значение правого операнда, если да, тогда условие становится истинным. | (A> B) неверно. |
| < | Проверяет, является ли значение левого операнда меньше значения правильного операнда, если да, тогда условие становится истинным. | (A <B) истинно. |
| > = | Проверяет, превышает ли значение левого операнда значение правого операнда, если да, тогда условие становится истинным. | (A> = B) неверно. |
| <= | Проверяет, является ли значение левого операнда меньше или равно значению правильного операнда, если да, тогда условие становится истинным. | (A <= B) истинно. |

## 

## **Логические операторы**

| **Оператор** | **Описание** | **Пример** |
| --- | --- | --- |
| && | Вызывается логическим оператором AND. Если оба операнда отличны от нуля, условие становится истинным. | (A && B) является ложным. |
| || | Вызывается логическим оператором ИЛИ. Если любой из двух операндов отличен от нуля, тогда условие становится истинным. | (A || B) истинно. |
| ! | Вызывается логическим оператором NOT. Используется для изменения логического состояния операнда. Если условие истинно, то логический оператор NOT сделает ложным. | ! (A && B) истинно. |

## 

## **Побитовые операторы**

# Побитовый оператор работает с битами и выполняет побитовую операцию. Таблицы истинности для &, |, и ^ заключаются в следующем:

| **p** | **q** | **p & q** | **p | q** | **p ^ q** |
| --- | --- | --- | --- | --- |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

# 

| **Оператор** | **Описание** | **Пример** |
| --- | --- | --- |
| & | Двоичный оператор AND копирует бит в результат, если он существует в обоих операндах. | (A & B) даст 12, что составляет 0000 1100 |
| | | Двоичный оператор OR копирует бит, если он существует в любом из операндов. | (A | B) даст 61, который равен 0011 1101 |
| ^ | Оператор двоичного XOR копирует бит, если он установлен в один операнд, но не тот и другой. | (A ^ B) даст 49, который равен 0011 0001 |
| ~ | Binary Ones Оператор дополнения является унарным и имеет эффект «flipping» бит. | (~ A) даст -61, что составляет 1100 0011 в форме дополнения 2 из-за подписанного двоичного числа. |
| << | Двойной левый оператор сдвига.Значение левых операндов перемещается влево на количество бит, заданных правым операндом. | A << 2 даст 240, что составляет 1111 0000 |
| >> | Двоичный оператор правого сдвига. Значение левых операндов перемещается вправо на количество бит, заданных правым операндом. | A >> 2 даст 15, что составляет 0000 1111 |

## 

## **Операторы присваивания**

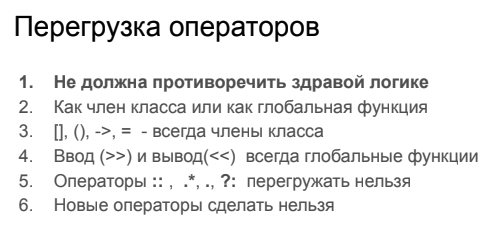
| **Оператор** | **Описание** | **Пример** |
| --- | --- | --- |
| = | Простой оператор присваивания, присваивает значения из правых операндов в левый операнд. | C = A + B присваивает значение A + B в C |
| + = | Оператор Add AND присваивания, Он добавляет правый операнд в левый операнд и присваивает результат левому операнду. | C + = A эквивалентно C = C + A |
| -= | Subtract AND assign operator, вычитает правый операнд из левого операнда и присваивает результат левому операнду. | C - = A эквивалентно C = C - A |
| \*= | Оператор умножения и присваивания, Он умножает правый операнд на левый операнд и присваивает результат левому операнду. | C \* = A эквивалентно C = C \* A |
| /= | Оператор Divide AND assign. Он делит левый операнд на правый операнд и присваивает результат левому операнду. | C / = A эквивалентно C = C / A |
| %= | Модуль и оператор присваивания, он принимает модуль с использованием двух операндов и присваивает результат левому операнду. | C% = A эквивалентно C = C% A |
| << = | Оператор сдвига влево и. | C << = 2 совпадает с C = C << 2 |
| >> = | Оператор правой смещения и назначения. | C >> = 2 совпадает с C = C >> 2 |
| &= | Побитовый И оператор присваивания. | C & = 2 является таким же, как C = C & 2 |
| ^ = | Побитовое исключающее ИЛИ и оператор присваивания. | C ^ = 2 является таким же, как C = C ^ 2 |
| | = | Побитовое включение оператора OR и присваивания. | C | = 2 совпадает с C = C |2 |

## 

## **Другие операторы**

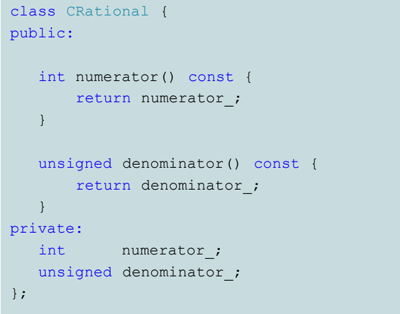
| **Оператор** | **Описание** |
| --- | --- |
| sizeof | Возвращает размер переменной. Например, sizeof (a), где 'a' является целым числом и будет возвращать 4. |
| Condition ? X : Y | Если Условие истинно, то оно возвращает значение X, иначе возвращает значение Y. |
| , | Вызывает последовательность операций. Значение всего выражения запятой - это значение последнего выражения списка, разделенного запятыми. |
| . (dot) and -> (arrow) | Используются для ссылки на отдельных членов классов, структур и союзов. |
| Cast | Преобразуют один тип данных в другой. Например, int (2.2000) вернет 2. |
| & | Возвращает адрес переменной. Например, & a; даст фактический адрес переменной. |
| \* | Является указателем на переменную. Например \* var; будет указывать на переменную var. |

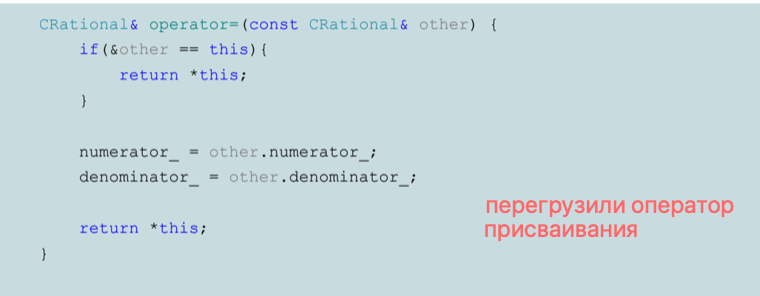
## **Перегрузка операторов**



Перегрузка операторов позволяет определить действия, которые будет выполнять оператор. Перегрузка подразумевает создание функции, название которой содержит слово **operator** и символ перегружаемого оператора. Функция оператора может быть определена как член класса, либо вне класса.

Перегрузить можно только те операторы, которые уже определены в C++. Создать новые операторы нельзя.







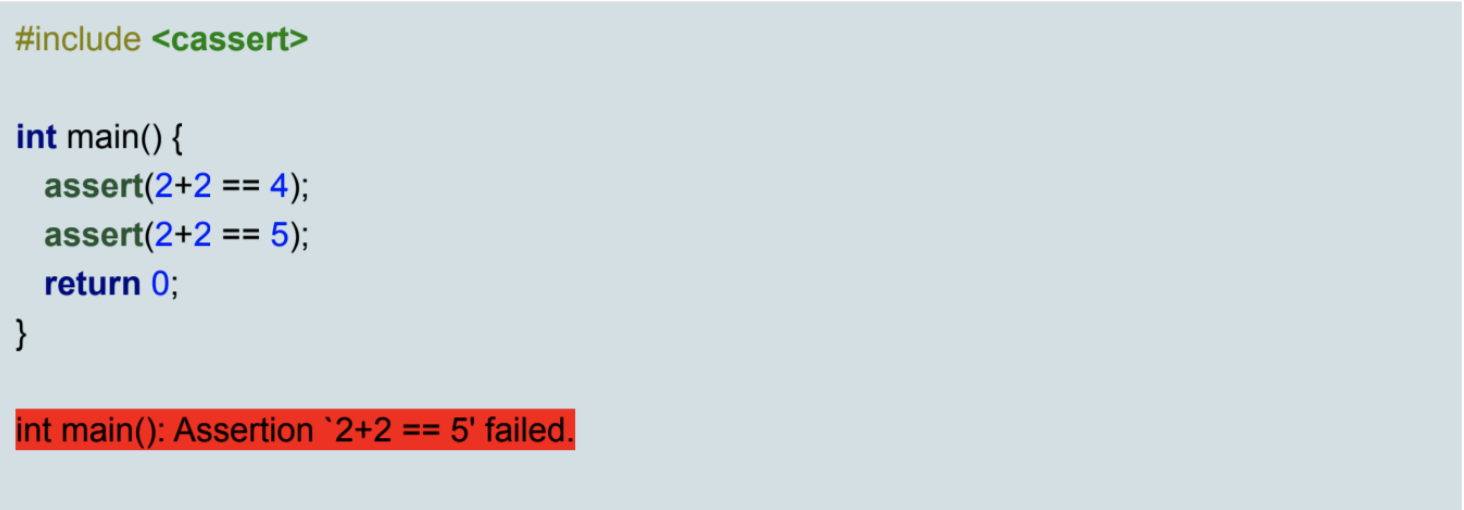
# Исключения. Обработка ошибок в Си. Assert. Исключения.love\_sf

**Исключение (на пальцах)** - это какое-то конкретное действие, которое осуществляется при выполнении критического условия, в случае возникновения которого мы хотим экстренно завершить программу или сделать какие-то конкретные действия (обработать данное исключение)

**Существует несколько подходов к обработке исключений**, одним из них является подход, использующий **assert**, - **он пришёл к нам из языка C**

**Первый подход:**

Суть этого подхода заключается в том, что мы прописываем критические условия внутри assert(), и если хоть один assert() имеет ложное выражение, программа падает, сигнализируя о том, что произошла ошибка



Коротко говоря, в случае с assert мы обрабатываем исключения жёстко - крашим уже запущенную программу, если видим неверное утверждение

**Минусы:**

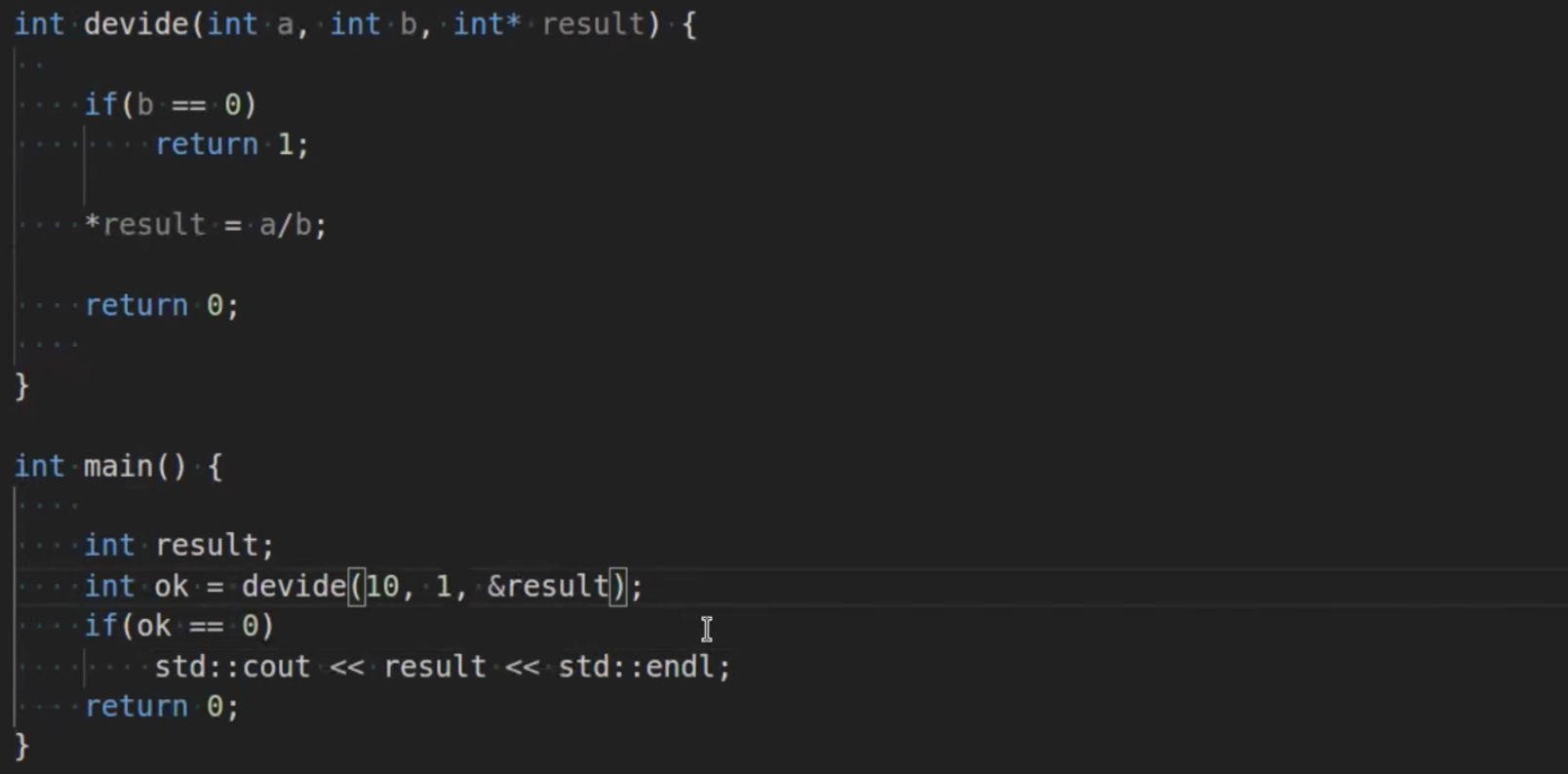
1. Работает уже после компиляции, тем самым давая возможность испортить какие-то данные до падения программы

**Решение:**

1. Использование static\_assert() - работает аналогично assert() за единственным исключением: static\_assert() крашит программу во время компиляции, не давая ей даже запуститься, тем самым предостерегая пользователя от нежеланных ошибок

**Второй подход:**

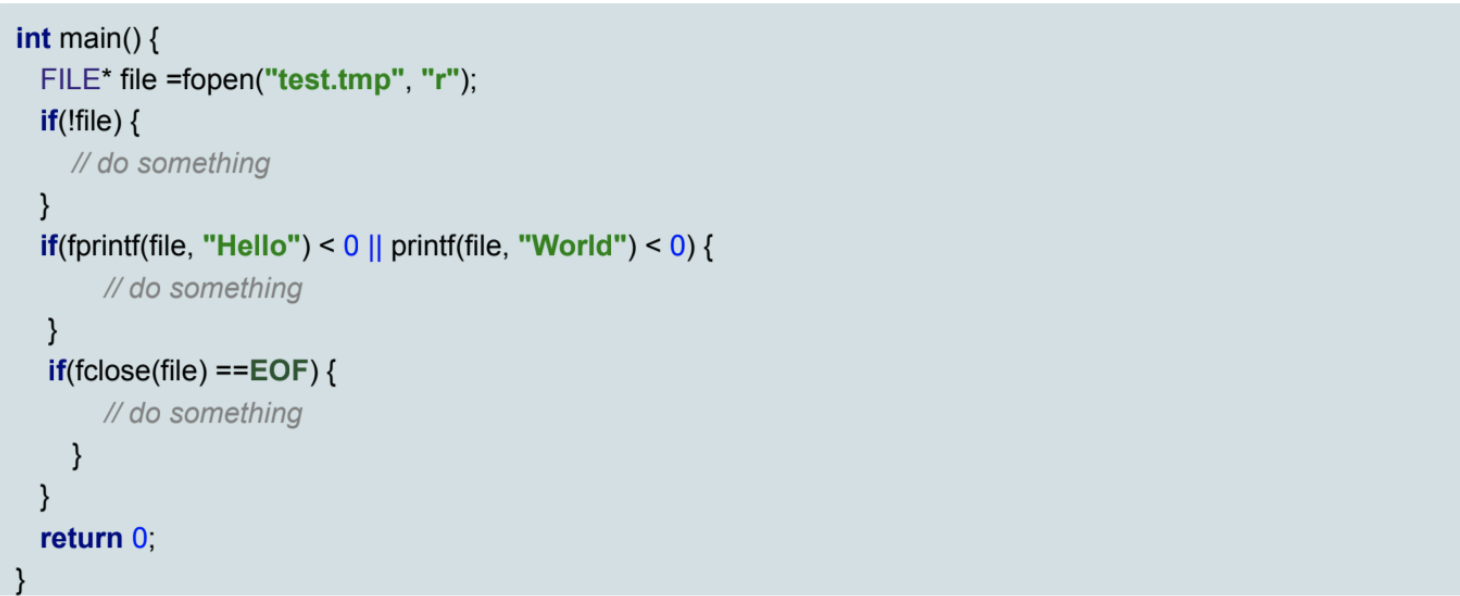
Вторым подходом к обработке ошибок является написание условий с помощью обычного if и возврата конкретных кодов ошибок исходя из конкретных условий - проще говоря подход через “код возврата”.



Такой подход является неудобным, так как коды возврата не всегда будут говорить о том, что пошло не так, а также для человека, впервые видящего подобный код будет очень сложно разобраться в том, почему выбраны те или иные коды возврата.

**Третий подход:**

Третьим и одновременно одним из наиболее распространенных подходов является “обработка в месте возврата” - когда мы просто прописываем на ходу, что делать в той или иной ситуации.



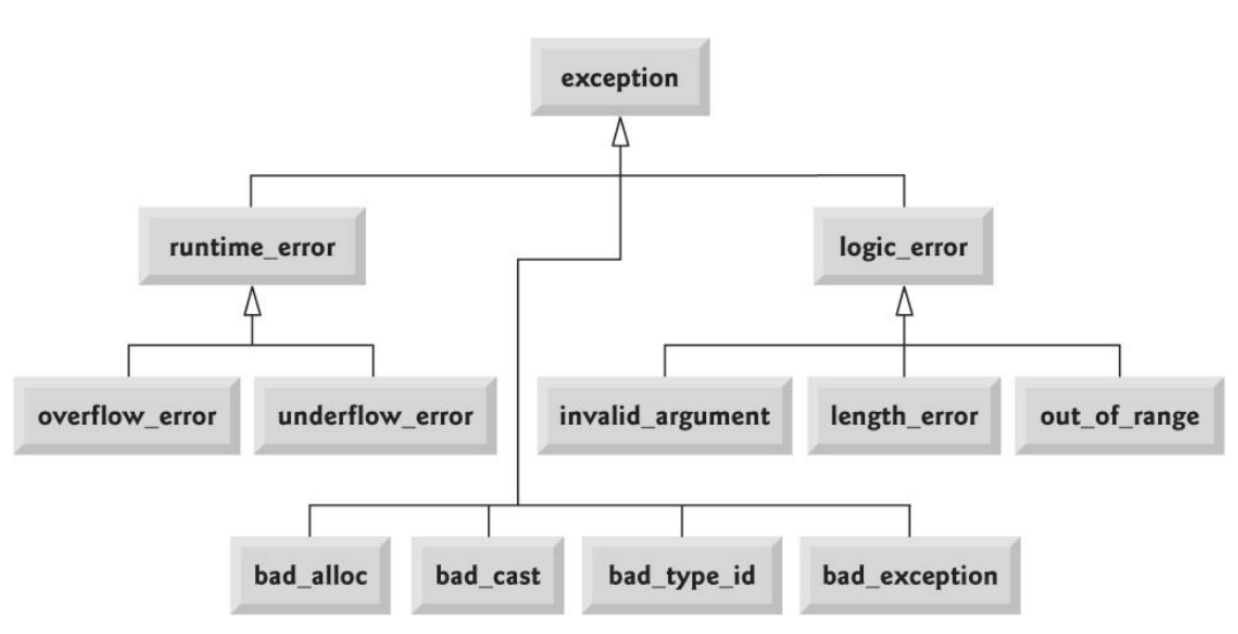
Данный подход также не особо приятен для совместной разработки, так как читабельность кода засчёт большого количества условий сильно снижается.

**Четвертый подход:**

Последний подход, который был добавлен уже в C++ - это исключения (exception)

Exception - класс в C++, элементы которого “пробрасываются” методом throw в коде, а “ловятся” внутри блоков catch конструкции try-catch.

У класса exception есть дерево наследований и довольно большая иерархия, а также определён метод what(), позволяющий узнать, какая конкретно ошибка была выброшена.

Иерархия классов исключений в C++: 

Механизм исключений работает следующим образом: в момент “пробрасывания” исключения (throw) он начинает подниматься вверх по стеку вызова функций, пытаясь найти тот try-catch блок, который ему подойдет - и если он его находит, то вызывает соответствующий обработчик исключения

Если чуть более формально:

Механизм пробрасывания исключений называется Stack Unwinding (раскрутка стека):

1. При выполнении throw конструируется исключение
2. Сгенерированное исключение кладётся в неопределенную область памяти
3. Исключение начинает “пробрасываться” вверх по стеку - если в нем не находится подходящего блока try-catch, то мы снимаем с вершины стека текущий стек-фрейм, параллельно уничтожая все объекты, которые лежали на этом стек-фрейме. После чего оказываемся в следующем стек-фрейме и проделываем там то же самое
4. При любом исходе поиска подходящего блока try-catch программа после начала вызова раскрутки стека завершится **аварийно**

**Резюмируем:**

Есть несколько подходов обработки исключений, но самым эффективным в стиле Си является использование assert() и static\_assert().

Если мы можем проверить какой-то факт (например, размер int) до того, как запустим программу, то используем static\_assert(), который в случае невыполнения условия не позволит компилировать код.

Если же мы хотим проверить инвариант в уже работающей программе, то используем assert, который в случае невыполнения условия экстренно завершит работу программы.

В стиле C++ самым популярным является подход exception, реализованный с помощью блоков try-catch и использованием одноимённого класса exception.

# Преимущества и недостатки исключений.

**Преимущества:**

1. В отличие от, например, так называемых кодов возврата, нам не нужно проверять каждую функцию на факт того, что она отработала верно - всё происходит силами одного блока try-catch.
2. Исключения дают (содержат) информацию касательно того, из-за чего произошло крушение кода, что очень полезно при отладке.
3. Блоки try-catch более читабельны и просты в понимании для стороннего пользователя, чем все остальные подходы и методы, описанные ранее.
4. Можно создавать свои типы исключений, предназначенные под конкретную задачу и более удобные в текущем коде.
5. Исключения обрабатываются единообразно, и **НЕ** в месте их возникновения (не нужно обмазывать каждую функцию проверками корректности её работы)

**Недостатки:**

1. Проблема очистки выделенных ресурсов после генерации исключений (например, если мы открыли файл с какой-то ошибкой, выполнили блок catch, но до закрытия файла так и не дошли)
2. Исключения увеличивают размер исполняемого файла и снижают производительность за счет проверок и раскрутки стека при поиске нужного блока try-catch
3. Когда происходит выброс исключения, и мы начинаем сворачивать стек, то у нас удаляется из стека всё, что на нём лежит, и мы теряем ту память, которая выделена в куче

# Шаблоны. Шаблоны функций, классов.

## **Шаблоны**

Шаблоны - это “не готовые функции”. В момент, когда компилятор видит вызов шаблона, он создает функцию с определёнными (подставленными в шаблон) типами данных и использует ее для обработки результата

Шаблоны определяют семейство функций или классов

Шаблон параметризуется одним или несколькими параметрами, которые

могут являться:

○ Тип

○ Константные выражения (интегральных типов, enum’ов)

○ Указатели (на объект, функцию, член класса)

Реализуют статический полиморфизм (полиморфизм времени компиляции)

Не требуют дополнительных расходов по сравнению с “прямыми” реализациями

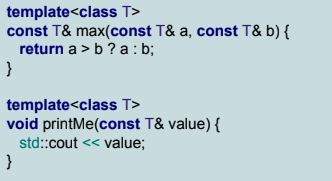
Параметр шаблона представляет произвольный идентификатор, в качестве которого, как правило, применяются заглавные буквы, например, T. Но это необязательно. То есть в данном случае параметр T будет представлять некоторый тип, который становится известным во время компиляции. Это может быть и тип int, и double, и string, и любой другой тип.

## **Шаблоны функций**

Это функции**, которые служат образцом для создания других подобных функций**.

Главная идея — создание функций без указания точного типа(ов) некоторых или всех переменных. Для этого мы определяем функцию, указывая **тип** **параметра шаблона**, который используется вместо любого типа данных.

Синтаксис выглядит так:

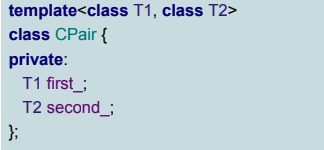


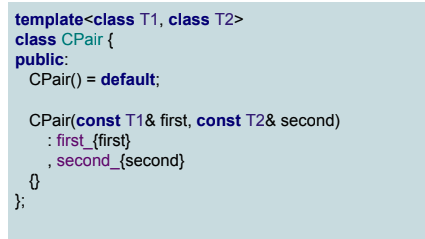
class T (или typename T) шаблонизируют тип данных, который будет подаваться в аргументы метода в процессе работы с ним

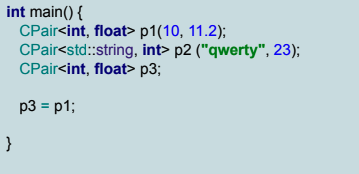
## **Шаблоны классов**

Шаблон класса позволяет задать тип для объектов, используемых в классе

Синтаксис выглядит так:

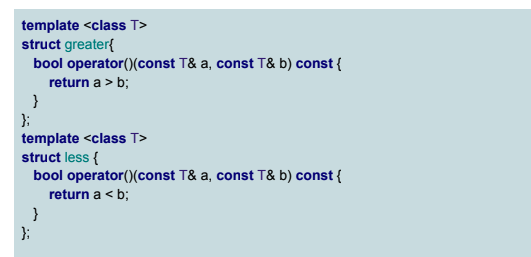




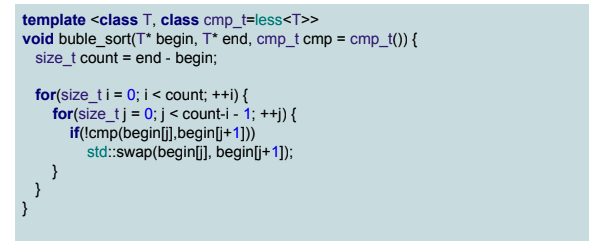


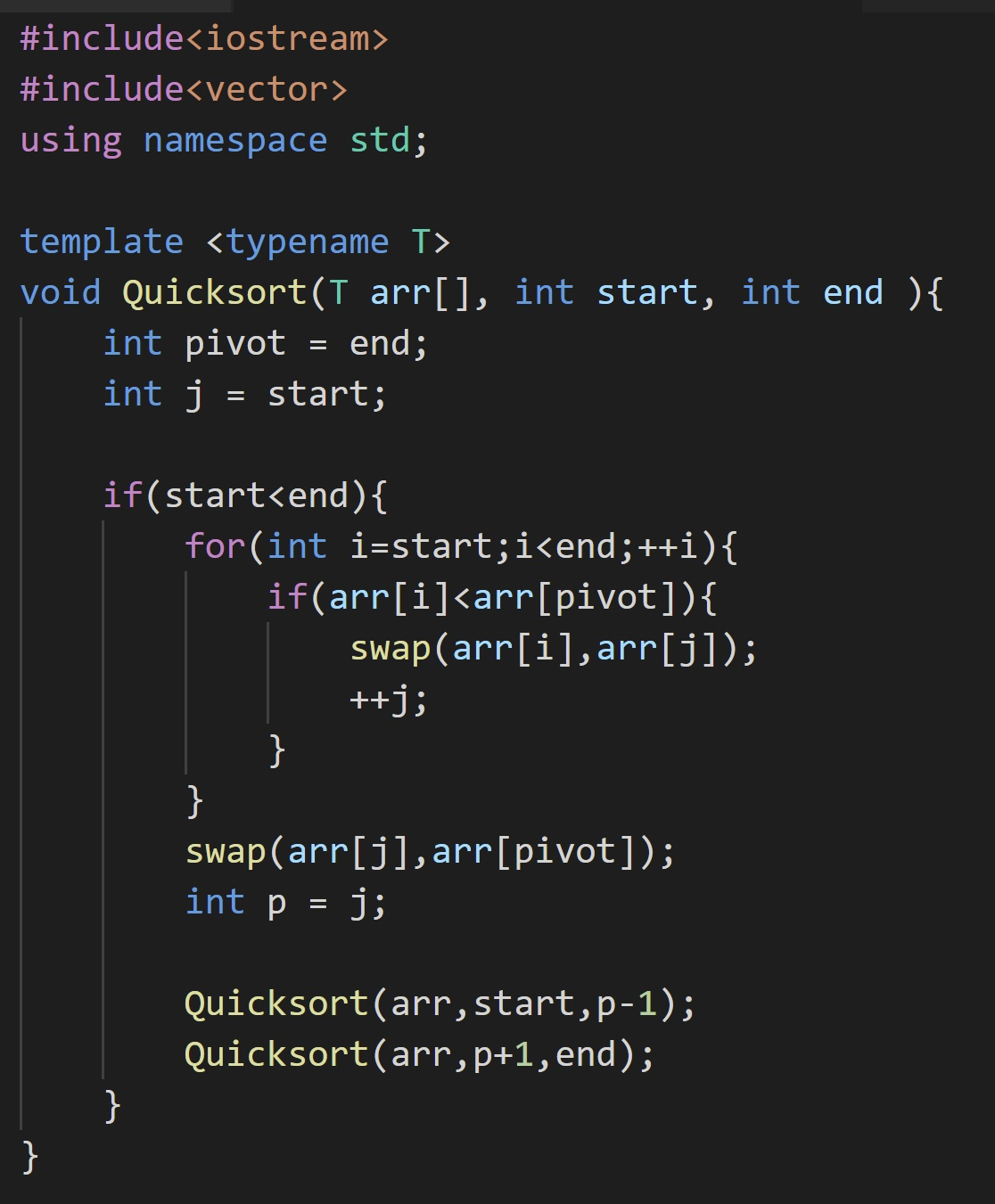
Что еще можно написать на шаблонах (думаю еще попросят написать квиксорт на шаблонах):

* Шаблонные компараторы



* Сортировки





# Шаблоны. Частичная специализация и полная специализация

## При создании [экземпляра шаблона функции](https://ravesli.com/urok-174-ekzemplyary-shablonov-funktsij/) для определенного типа данных компилятор копирует шаблон функции и заменяет параметр типа шаблона функции на фактический (передаваемый) тип данных. Это означает, что все экземпляры функции имеют одну реализацию, но разные типы данных. Хотя в большинстве случаев это именно то, что требуется, иногда может понадобиться, чтобы реализация шаблона функции для одного типа данных отличалась от реализации шаблона функции для другого типа данных.

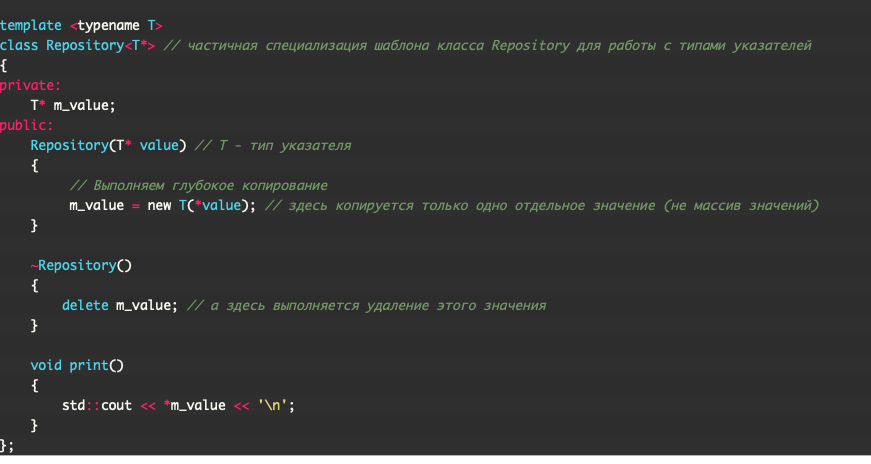
## Специализация шаблонов именно для этого и предназначена.

## **Частичная специализация**

Частичная специализация шаблона позволяет выполнить специализацию шаблона класса **(но не функции!)**, где некоторые (но не все) параметры шаблона явно определены

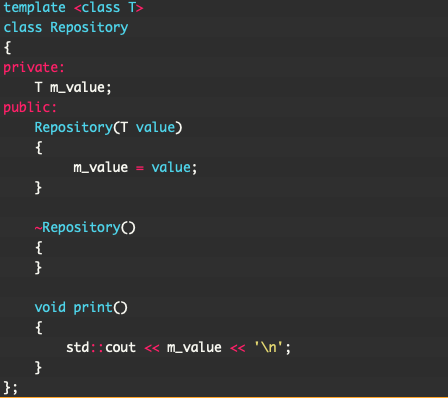
Частичная специализация позволяет частично настроить код шаблона для определенных типов, например:

* Шаблон имеет несколько типов, и только некоторые из них требуют специализации. Результат для остальных типов параметризован шаблоном.
* Шаблон имеет только один тип, но специализация необходима для типов указателя, ссылки, указателя на член или указателя на функцию. Специализация сама по себе по-прежнему является шаблоном с типом, на который задан указатель или ссылка.

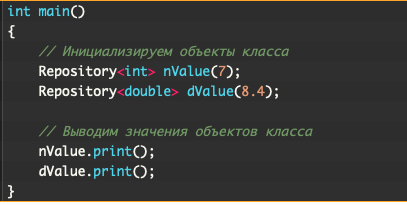


## **Полная (явная) специализация**

Рассмотрим следующий шаблон

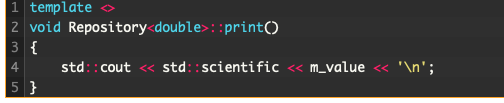


Вышеприведенный код работает со многими типами данных:



Теперь, предположим, что нам нужно, чтобы значения типа double (только типа double) выводились в [**экспоненциальной записи**](https://ravesli.com/urok-33-tip-dannyh-s-plavayushhej-tochkoj-floating-point/#toc-1). Для этого мы можем использовать **специализацию шаблона функции** (или ***«полную/явную специализацию шаблона функции»***) для создания отдельной версии функции print() для вывода значений типа double.

Всё просто: записываем экземпляр шаблона функции (если функция является методом класса, то делаем это за пределами класса), указывая нужный нам тип данных. Например, вот специальный шаблон функции print() для значений типа double:



Когда компилятору нужно будет создать экземпляр Repository<double>::print(), он увидит, что мы уже явно определили эту функцию, и поэтому он будет использовать именно этот экземпляр, а не копировать общую для всех типов данных версию шаблона функции print().

Часть template <> сообщает компилятору, что это шаблон функции, но без параметров (так как в этом случае мы явно указываем нужный нам тип данных).

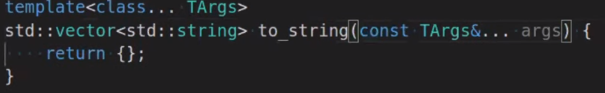
Результат выполнения программы:

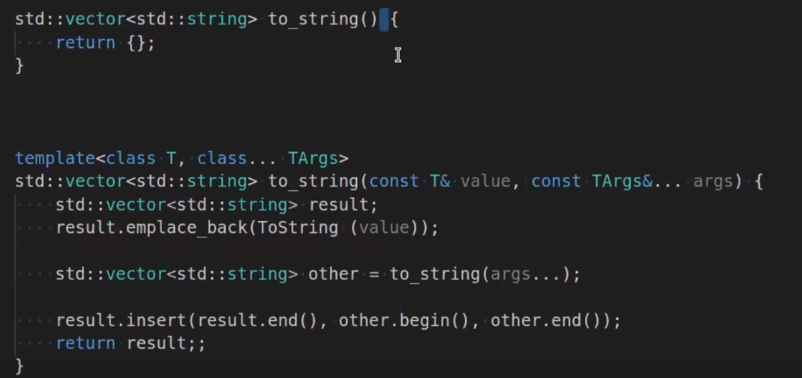
7

8.400000e+00

# Variadic template

**Variadic template** - это шаблон с переменным количеством аргументов





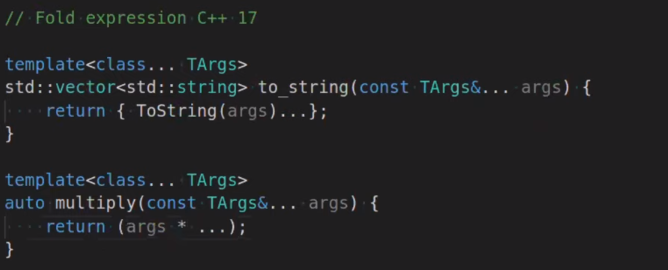
**Минусы:**

1. Очень много функций на стеке -> ухудшается время работы
2. Довольно сложный код
3. Увеличивается размер программы

**Плюсы:**

1. Можем обрабатывать неограниченное (заранее неизвестное) количество аргументов за один раз, что очень хорошо раскрывает все плюсы шаблонного подхода

Также в стандарте C++17 была добавлена возможность сократить написание данный функций, она называется Fold expression



# Преобразования типов. Явные и неявные. С-cast, static\_cast, dynamic\_cast, const\_cast, reinterpret\_cast. CRTP

Преобразование типов – преобразование значения одного типа в значение другого.

Преобразования типов могут быть **явными** и **неявными**, а также «**в стиле Си**» и «**в стиле Си++**»

**Неявное преобразование** – компилятор сам додумывает до нас, что и к какому типу данных ему нужно приводить

int i = 10;

float f = i; // неявное преобразование 10 в 10.0f

**Явное преобразование (в стиле Си)** – мы сами говорим, к какому типу приводить

int i = 10;

float f = (float)i; // явное преобразование

**Плюсы** явного преобразования типов в стиле Си:

1) Всё просто, примитивно и понятно

**Минусы** явного преобразования типов в стиле Си:

1) Есть возможность писать приведение к абсолютно несвязным типам данных

class Foo {};

Foo\* i = new Foo{};

int\* f = (int\*)i;

2) Факт адекватности приведения абсолютно никак не проверяется, в связи с чем могут возникнуть ошибки

**Явное преобразование (в стиле Си++)** – преобразования с использованием специальных кастов, а именно: const\_cast, static\_cast, dynamic\_cast и reinterpret\_cast

**const\_cast**:

1) Умеет убирать const с переменных

2) Умеет преобразовывать указатели на одинаковые типы

**3)** Умеет преобразовывать ссылки на одинаковые типы

**4)** Работает во время компиляции

**static\_cast** – главное преимущество заключается в том, что он происходит в момент компиляции, это эффективно и быстро

1) Производится во время компиляции

2) Преобразование стандартных типов (short -> int)

3) Преобразование указателей в рамках иерархии (родительский класс к дочернему)

4) Конструктор или оператор приведения типов

5) Преобразования из указателя на void

**dynamic\_cast** – он использует информацию о типах (нужны виртуальные методы)

1) В отличие от static\_cast позволяет преобразовывать указатели вверх по иерархии, проверяя, может ли быть выполнено такое преобразование (если нет – кидает exception) **(плюс)**

2) Выполняется в рантайме **(минус)**

3) Если преобразование произвести невозможно, то результатом будет nullptr (кидает исключение)

4) Работает только с полиморфными классами

**reinterpret\_cast**:

1) Позволяет преобразовывать несовместимые типы

2) Использует побитовое представление

3) Ничего не проверяет на совместимость типов

**CRTP = Curiously Recurring Template Pattern**

**В чём заключается суть CRTP:**

Эта идиома проектирования, заключающаяся в том, что класс наследует от базового шаблонного класса с самим собой в качестве параметра шаблона базового класса (создать шаблонный класс, от которого мы наследуемся, при этом в котором мы шаблонизируемся сами от себя)

Такая конструкция делает возможным обращение к производному классу из базового.

\

# Итераторы. Алгоритмы. Основные классы алгоритмов (Стандартная библиотека)

**Итераторы**

● Указателеобразные объекты

● Связь между алгоритмами и контейнерами

● Категории

○ Входные

○ Выходные

○ Однонаправленные

○ Двунаправленные

○ Произвольного доступа

**Входной итератор (InputIterator)**

Требования:

● operator !=

● ++iterator и iterator++

● value = \*iterator

● operator ==

● O(1)

**Выходной итератор (OutputIterator)**

Требования:

● \*iterator = value

● ++iterator и iterator++

● O(1)

**Однонаправленные итераторы (Forward Iterator)**

● Входной итератор

● Выходной итератор

● Сохранение для последующего использования

**Двунаправленные итераторы**

● Однонаправленный

● operator--

**Итераторы с произвольным доступом**

● Двунаправленный итератор

● Достижение любой позиции за O(1)

● Пусть r и s – итераторы с произвольным доступом, n – целое число , тогда:

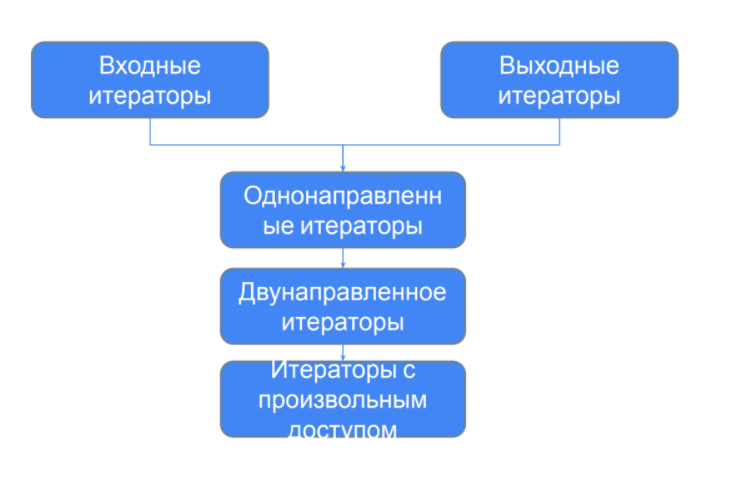
○ r+n, n+r, r-n

○ r[n]=\*(r+n)

○ r+=n, r-=n

○ r-s ->int

○ r<s, r>s, r<=s, r>=s -> bool



Указатель является итератором произвольного доступа!

Итераторы связывают контейнеры и алгоритмы для работы с ними. Для каждого контейнера описаны свои итераторы, поддерживающие эффективность обращения к его элементам.

Описание обобщенных алгоритмов включает описание категорий

итераторов с которыми они работают

const vector<int> v(100,0);

//vector<int>::iterator i = v.begin(); // !!Ошибка

vector<int>::const\_iterator i = v.begin();

**Обобщенные алгоритмы**

●Неизменяющие алгоритмы

●Изменяющие алгоритмы

●Связанные с сортировкой алгоритмы

●Обобщенные числовые алгоритмы

**Неизменяющие алгоритмы**

● find

● adjacent\_find

● count

● for\_each

● mismatch

● equal

● search

**Изменяющие алгоритмы**

●copy

●copy\_backward

●fill

●generate

●partition

●random\_shuffle

●remove

●replace

●remove

●rotate

●swap

●swap\_ranges

●transform

●unique

**Обобщенные числовые алгоритмы**

● accumulate

● partial\_sum

● adjacent\_difference

● inner\_product

Функциональные объекты

● minus;

● multiplies

● ....

● equal\_to

● not\_equal\_to

● greater

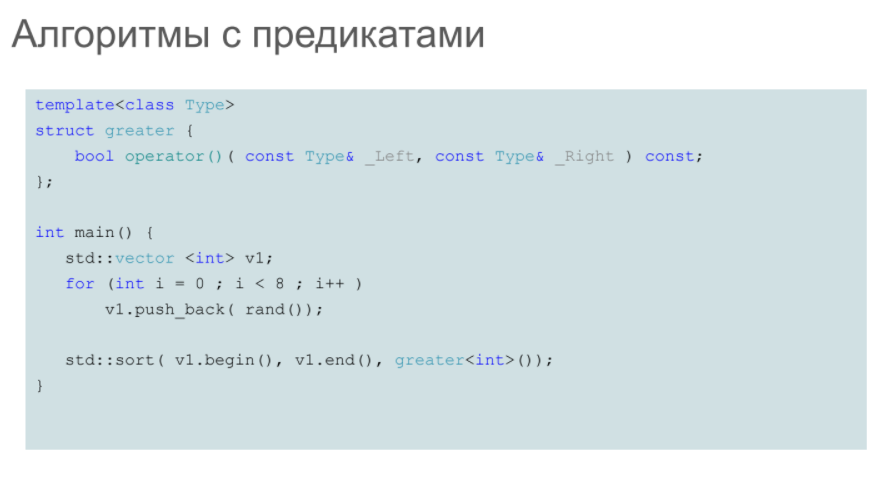
● ...

● logical\_and

● ...

● bit\_and

Все алгоритмы также эффективны.



# Контейнеры. Последовательные контейнеры. (Стандартная библиотека)

**Контейнер** - это объект, который хранит "коллекцию" других объектов.

Контейнер сам заботится о выделении памяти и других низкоуровневых штуках, а нам предоставляет доступ к объектам, которые в нем находятся.

Все основные стандартные классы-контейнеры библиотеки *STL* подразделяются на последовательные и ассоциативные контейнеры.

Ключевое отличие **последовательных контейнеров** от **ассоциативных заключается** в том, что при помещении элемента в **последовательный контейнер** он остаётся там, где его разместили, а в **ассоциативном** он перемещается туда, где ему положено находиться в соответствии со внутренним устройством контейнера

**Контейнеры последовательностей:**

○ vector<T>

○ deque<T>

○ list<T>

○ array<T>

○ forward\_list<T>

**Контейнеры последовательностей** обеспечивают упорядоченный способ хранения своих элементов.

Каждый из элементов такого контейнера имеет определённую позицию, которая зависит от времени и места помещения его в контейнер, но не зависит от значения элемента.

Как правило, к последовательным контейнерам относятся списки, вектора, очереди двустороннего доступа, массивы (начиная со стандарта C++11) и ряд других. Обычно их реализуют на практике в виде связанного списка или массива

# Контейнеры. Ассоциативные контейнеры (Стандартная библиотека)

**Контейнер** - это объект, который хранит "коллекцию" других объектов.

Контейнер сам заботится о выделении памяти и других низкоуровневых штуках, а нам предоставляет доступ к объектам, которые в нем находятся.

Все основные стандартные классы-контейнеры библиотеки *STL* подразделяются на последовательные и ассоциативные контейнеры.

Ключевое отличие **последовательных контейнеров** от **ассоциативных заключается** в том, что при помещении элемента в **последовательный контейнер** он остаётся там, где его разместили, а в **ассоциативном** он перемещается туда, где ему положено находиться в соответствии со внутренним устройством контейнера

**Ассоциативные контейнеры:**

○ set<Key> (multiset)

○ map<Key,T> (multimap)

**Неупорядоченные ассоциативные контейнеры:**

○ unordered\_set<Key> (multiset)

○ unordered\_map<Ket, T> (multimap)

# Адаптеры (Стандартная библиотека)

Адаптеры — это не новые понятия или реализации, а адаптация уже существующих понятий библиотеки под конкретные, часто использующиеся цели. Часто такая адаптация делается посредством ограничения функциональности базового понятия под запросы адаптера. В библиотеке представлены адаптеры контейнеров, итераторов и функций.

Проще всего показать, как появляются адаптеры на примере адаптеров контейнеров: stack (стек), queue (очередь), priority\_queue (очередь с приоритетами). Уже из их перечисления понятно, что:

1. Это очень широко и часто используемые структуры данных;
2. Для них не нужны какие-то отдельные реализации. Для обеспечения их функциональности можно использовать (адаптировать) в качестве базового любой из стандартных контейнеров STL, который обеспечивает операции типа push\_back, pop\_back или pop\_front (в зависимости от типа адаптера);
3. “Лишние” операции из арсенала базового контейнера для адаптера нужно исключить (чтобы не создавать искушения … например, операцию индексации, если vector используется для адаптации стека)

Аналогичный подход используется для адаптеров итераторов. Примером такого адаптера итераторов является std::reverse\_iterator. Он адаптирует оператор operator -- других итераторов таким образом, что когда используется оператор operator ++, то на самом деле вызывается оператор operator --. Это позволяет перебирать элементы контейнера или последовательности в обратном порядке, не меняя код, который основывается на применении оператора operator ++ .

# **Умные указатели. auto\_ptr, unique\_ptr, shared\_ptr, weak\_ptr. Устройство, преимущества и недостатки.**

<https://ravesli.com/urok-189-umnye-ukazateli-i-semantika-peremeshheniya/>

<https://en.cppreference.com/book/intro/smart_pointers>

**RAII (resource acquisition is initialization)** - это идиома, которая считает, что если есть какой-то класс, в котором происходит выделение какой-то динамической памяти, то инициализация должна происходить в конструкторе этого класса, а освобождение памяти - в деструкторе

Основная задача **RAII** - это корректное освобождение ресурсов при некорректной работе программы (при всяких exceptions)

**auto\_ptr** - это одна из разновидностей так называемых умных указателей (smart pointers).

**auto\_ptr** - это фактически просто обёртка над обычным указателем, весь функционал которой ограничивается на том, чтобы следить за удалением выделенной динамический памяти посредством её очистки внутри деструктора

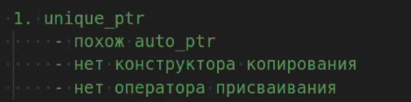
**Плюсы:**

1. Не нужно следить за удалением выделенной динамической памяти

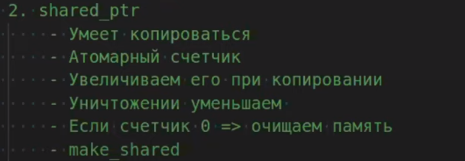
**Минусы:**

1. Не может адекватно взаимодействовать с векторами, так как при копировании auto\_ptr для сохранения логики программы и отсутствия ошибок с удалением памяти, доступ к данным по указателю остаётся только у одного из двух auto\_ptr, следовательно, любые взаимодействия этого класса с векторами - неэффективны

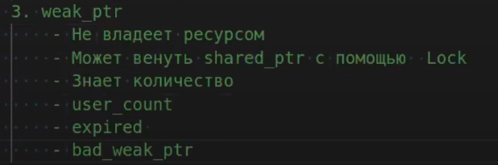
**unique\_ptr** - это урезанный аналог **auto\_ptr (был удалён в С++17)**, в котором отсутствуют операции копирования (удалён конструктор копирования) и присваивания (удалён оператор присваивания).



**shared\_ptr** - это **unique\_ptr** на максималках, у него есть конструкторы копирования, есть оператор равно, а также статический счетчик количества переменных, использующих один и тот же указатель. Очистка памяти происходит в момент, когда счетчик равен нулю



**weak\_ptr**  - это “слабый указатель”, который не владеет ресурсом (самим указателем), но при этом имеет метод, который может вернуть shared\_ptr, а также метод, который сообщает о том, истёк ли указатель, который он хранит или нет



**Плюсы умных указателей в целом:**

С программиста снимается ответственность следить за очищением выделенной памяти

**Минусы умных указателей в целом:**

Производительность понижается

# **Объяснение рубежки**

## *Что выведет программа (ABC)?*





✓ ~~abcCBA~~

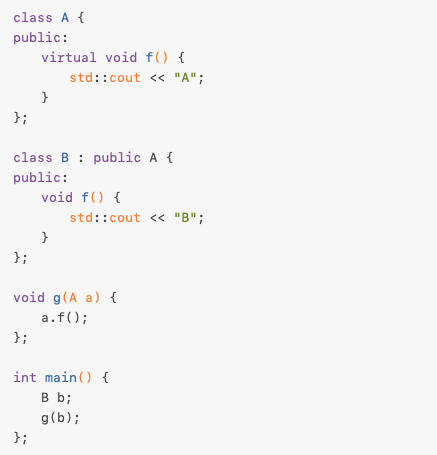
✓ ~~acbABC~~

✓ **acbBCA**

✓ ~~Не определено, зависит от компилятора~~

Сначала выполняется то, что находится на куче (в статической памяти), вне мейна, то есть создаётся элемент класса A, после чего со стека вызываются все конструкторы из мейна (динамической памяти), затем в обратном порядке от порядка создания вызываются деструкторы.

## *Что выведет программа? (С виртуальным методом / virtual)*

 мс

✓ **A**

✓ ~~B~~

✓ ~~AB~~

✓ ~~Ошибка компиляции~~

Сначала создали экземпляр класса В. Вызываем функцию g. Но она у нас определена для класса А, значит у нас будет вызван виртуальный метод, выведет «А».

Так как B является наследником класса A, в то время как функция у нас определена именно для класса A, то при передаче в неё аргумента дочернего класса A функция пропускает эту переменную и работает с ней как с элементом класса A

При вызове функции из элемента класса используется локальная переменная класса А, которая берётся из класса B, а после отработки функции вызывается деструктор этой переменной

## *Что выведет программа? (С map / мап)*



✓ ~~Ошибка компиляции~~

✓ ~~33~~

✓ **13**

✓ ~~Зависит от платформы~~

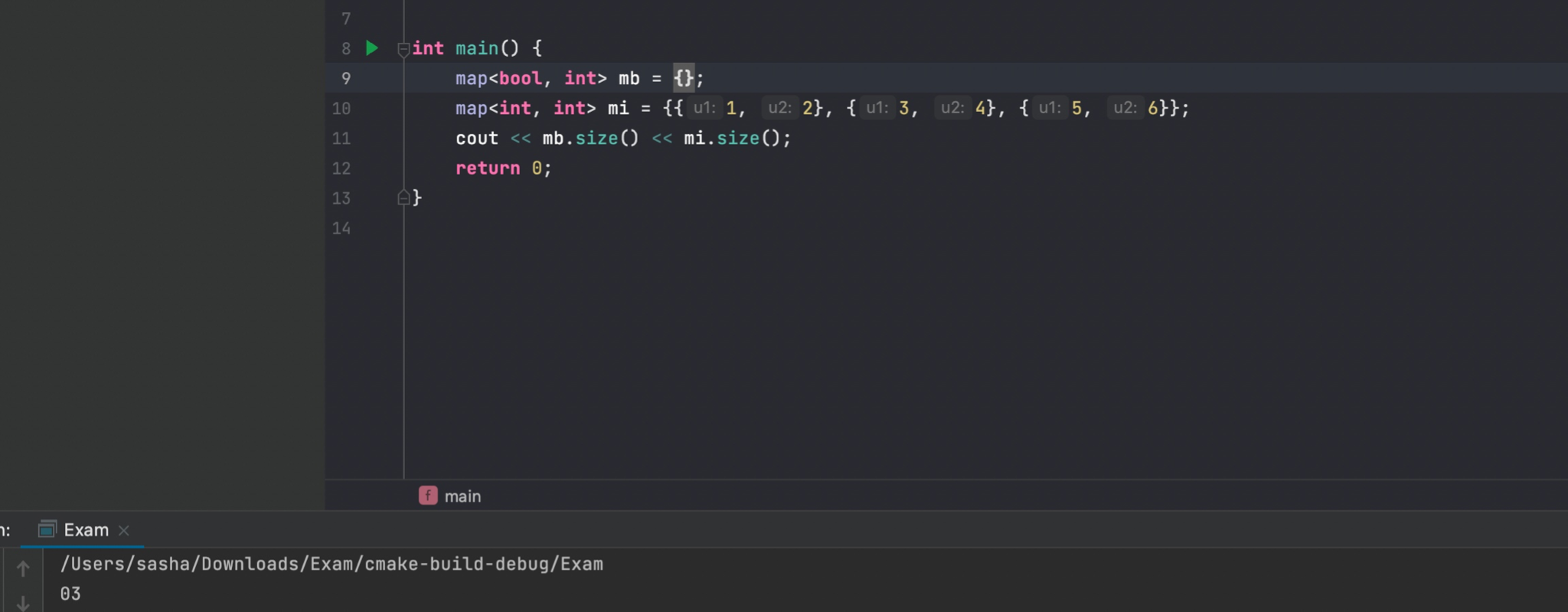
Разница в том, что наши мапы в качестве ключей хранят bool и int. Если у нас в качестве ключа стоит булевое значение, то плюсы увидят только пару {1, 2}, она у нас одна. Пары {3, 4} и {5, 0} не могут быть обработаны, так как числа 3 и 5 не могут представлены как булевые значения (наверно так)

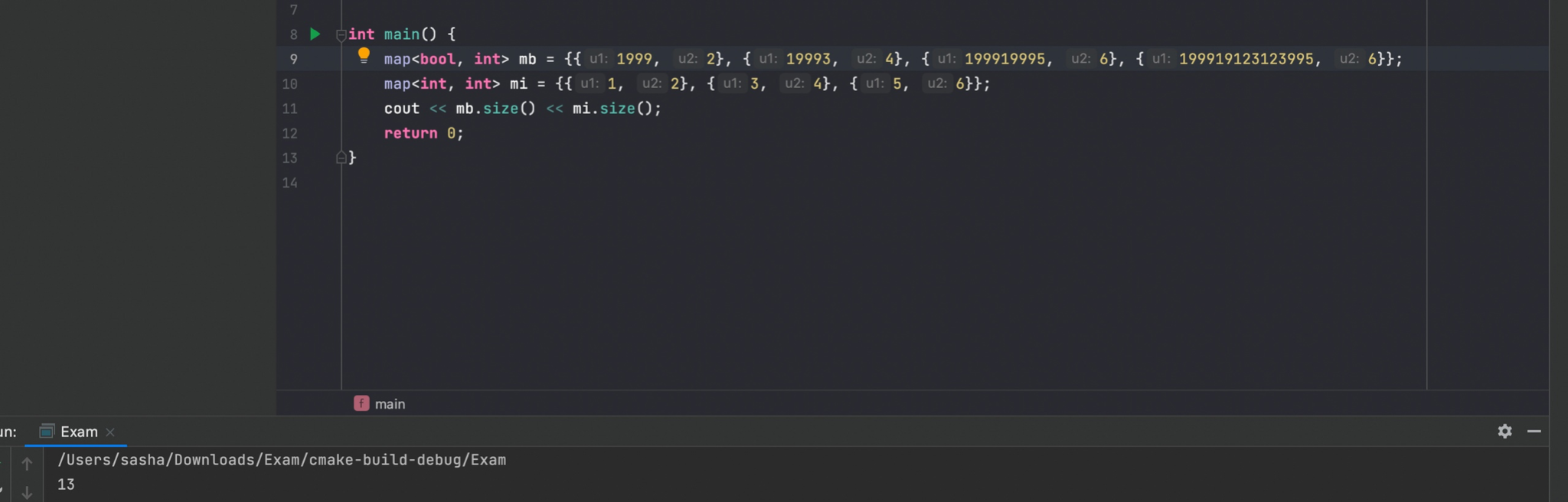
Мнение #3 - bool в качестве ключа проверяет просто наличие элементов в мапе, если там хоть что-то есть - возвращает единицу, иначе - ноль

Тоже выскажусь: вероятно, значения ключей просто неявным образом приводятся к bool(то есть берётся только последний бит). Поскольку они все нечётные, получаем три пары с ключём 1.

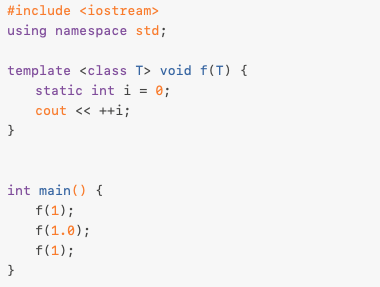
Так как структура хранимых элементов в массиве однородна, а первым параметром (ключом) в мапе идёт булево значение, компилятор объединяет все сущности одной природы в одну и рассматривает это как единое целое - ключ для нашей пары;

**NB** если бы мы дописали к первому мапу {A().a, A().b}, где A - это структура с полями a и b, то размер данного мапа был бы 2





## *Что выведет программа? (С шаблоном / template)*



✓ ~~123~~

✓ ~~111~~

✓ **112**

✓ ~~Ошибка компиляции~~

Почему 112? Все дело в static. Эта приписка позволяет хранить текущее значение переменной и не создавать новую при вызове функции. Так как мы вызываем два раза метод f() для **int** и один раз для **double**, то получаем 112:

1 - int

1 - double (уже новый тип, «новая» переменная)

2 - снова int, плюсуем старое значение

## *Что выведет программа? (С 3 шаблонами / 3 template)*



✓ ~~1~~

✓ ~~2~~

✓ 3

✓ ~~Ошибка компиляции~~

1 не выведется потому что зашаблонированная функция принимает на вход T а не T\* (T, а не массив из T)

2 не выведется поскольку шаблон не имеет заданных параметров, поэтому форма вызова функции 2 будет выглядеть как f<int\*>(p); - **можно и не писать это**

int\* может быть корректно обработан **любым** из этих шаблонов, однако вызывается именно 3й шаблон и вот почему:

1. Второй шаблон - это специализированный шаблон от первого: в нём конкретно прописаны типы данных, при которых он вызывается, при этом без объявления первого шаблона второй не может существовать
2. Более того, для вызова второго шаблона при данном наборе **и данном порядке следования** функций нам нужно явно написать f<int\*>(p);
3. Вызов именно третьего шаблона связан с тем, что он был объявлен позже всех и в памяти он лежит фактически самым первым при поиске нужной функции для вызова, соответственно, когда мы вызываем f(p), программа начинает искать подходящую функцию для вызова, и первой же попавшейся функцией является подходящий нам шаблон, который выводит 3
4. **NB** если бы последней функцией был объявлен специализированный шаблон, выводящий 2, то был бы вызван именно он

## *Что выведет программа? (Cо структурой / struct)*



✓ ~~AACC~~

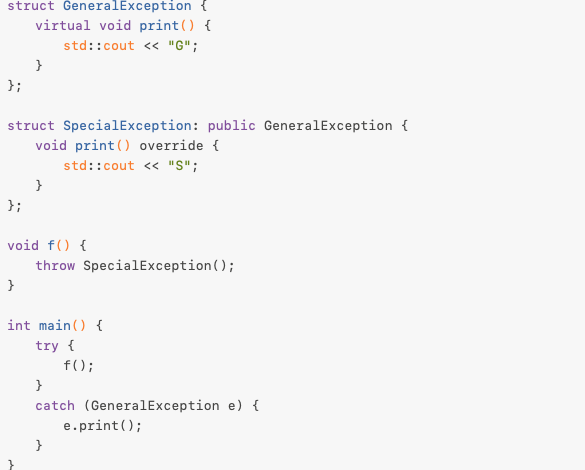
✓ ~~AABBCC~~

✓ **AABCBC**

✓ ~~Ошибка компиляции~~

Сначала вы вызываем 2 конструктора во время инициализации массива. Далее, когда у нас начинается цикл, мы проходимся по каждой структуре из этого массива, создавая экземпляр нашей структуры. В связи с этим вызывается второй конструктор (конструктор копирования, так как мы конструируем элемент структуры A из элемента x), который выводит В, а затем уже выводится наш метод f().

## *Что выведет программа? (C эксепшинами / exception)*



✓ ~~Ничего~~

✓ **G**

✓ ~~S~~

✓ ~~Ошибка компиляции~~

Мы вызываем метод f(), который выкидывает SpecialException, а он, в свою очередь, наследуется от GeneralException. В такой ситуации у нас сработает блок catch, который выведет G

Аналогично задаче, где функция принимает экземпляр класса A, но передаем в нее мы экземпляр класса B, который унаследован от A

## Что выведет программа? (Куча структур X)



✓ ~~abc~~

✓ **abbc**

✓ ~~abcc~~

✓ ~~bbcc~~

Пройдемся по строчкам в main().

1 строка вызывает обычный конструктор.

2 строка вызывает конструктор копирования, выводит b.

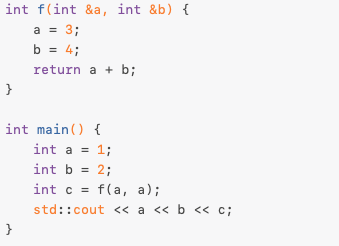
3 строка аналогична второй, также вызывается второй конструктор (эти две записи равнозначны).

По идее это ссылка

4 строка вызывает перегруженный оператор присваивания для уже существующей структуры, так что у нас будет выводится **c**

## 

## *Что выведет программа? (Функция из 2 аргументов f)*



✓ ~~127~~

✓ ~~347~~

✓ ~~427~~

✓ **428**

Перед присвоением некоторого значения переменной **c**, f() вызывается, принимая в качестве аргументов 2 числа с помощью унарного оператора взятия адреса.

Так как мы передали переменную **a** в 2 аргумента по адресу **a** (фактически передали 2 одинаковых адреса), то код, исполнившись построчно, присвоит переменной **a** 3, а после этого переприсвоит ей значение 4.

Мы же понимаем, что название принимаемых аргументов не играют никакой роли, так как они находятся в зоне видимости исключительно данной функции, и никаким образом не перекликаются с переменными, находящимися в мейне, так что нам важно именно то, что мы **передаем** в функцию, а не как там называются переменные

А как получилось 8? Так как переменная у нас одна, следовательно и адрес один, значит мы перекинули 4 дважды в один адрес, получилась сумма

## *Что выведет программа? (Структуры Base, Derived)*



✓ ~~11~~

✓ **22**

✓ ~~12~~

✓ ~~21~~

Сначала у нас идет переопределение метода на 2 строчке, это не проблема. Далее мы создаем референс на object, но сам по себе он хранит в себе уже переопределенный метод f, поэтому выводится 22