Оглавление

[Терминология и основы 2](#_Toc129457487)

[Модификаторы доступа 4](#_Toc129457488)

[Геттеры/Сеттеры 6](#_Toc129457489)

[Инкапсуляция 7](#_Toc129457490)

[Конструктор класса 8](#_Toc129457491)

[Деструктор класса 10](#_Toc129457492)

[Ключевое слово this 11](#_Toc129457493)

[Конструктор копирования 12](#_Toc129457494)

[Перегрузка операторов 13](#_Toc129457495)

[Дружественные функции 16](#_Toc129457496)

[Определение методов все класса 17](#_Toc129457497)

[Дружественный метод класса 18](#_Toc129457498)

[Дружественные классы 19](#_Toc129457499)

[Ключевое поле **static** и статические поля класса 20](#_Toc129457500)

[Статические методы класса 21](#_Toc129457501)

# Терминология и основы

4 принципа ООП

1. Инкапсуляция
2. Наследование
3. Полиморфизм
4. (Абстракция)

**Класс** – пользовательский **тип** данных. Структура, в которую введены методы для обработки полей

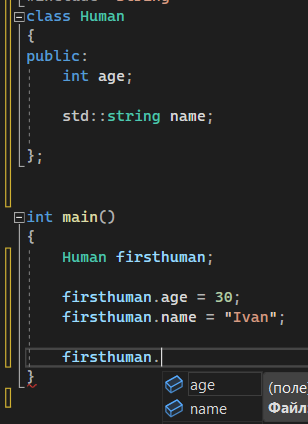
**Поля класса** – переменные, которые содержит в себе класс

**Объект класса (переменная класса)** – переменная с типом нашего класса

Обращение к объекту класса происходит следующим образом

* объект.поле\_класса

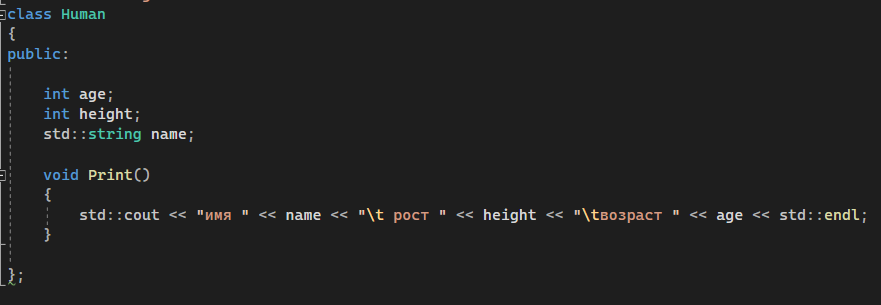
Дальше можем работать с полем в соответствии с проставленым типом данных нашего поля.



**Метод класса (функция класса)** –

Пишется в теле класса по аналогии с обычной функцией.

Метод класса взаимодействует с полями своего объекта. Допустим, у нас есть 3 экземпляра класса, где разные имя, возраст и рост, то мы можем к каждому объекту вызвать метод, и будет взаимодействовать непосредственно с данными, которые хранят поля данного объекта



# Модификаторы доступа

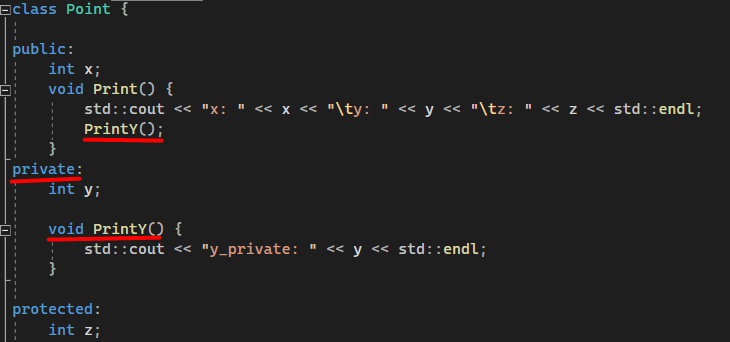
Если явно не указывать какой у поля модификатор доступа, то он будет private

* public: доступен везде
* private: открыт только для элементов внутри класса и дружественным классам и функциям
* protected: как и private, только участвует в наследовании.
  + Когда мы наследует от базового класса дочерний с модификатором доступа private,то в дочернем классе эта переменная будет недоступна. Однако, если указать protected, то уже будет доступна.

Модифицирует не одно поле, а область.

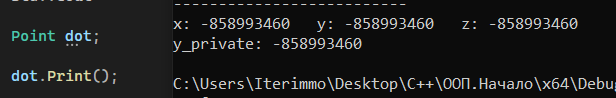
Влияют как на поля, так и на методы класса.

Пример вызова private функии из public



Не смотря на то, что Y у нас находится в области private, мы можем его использовать в public функциях. То же самое относится и к функциям: мы вызвали private-функцию в момент вызова public-функции

Вывод:



(в переменной лежит мусор, так как она неинициализирована)

Сокрытие части функционала от пользователя – инкапсуляция.

Свойства класса - ?

# Геттеры/Сеттеры

В ООП принято данные отделять от методов (переменные от функций).

Доступ к полям принято организовывать через геттеры и сеттеры – это основа инкапсуляции.

Геттеры/сеттеры – это функции класса, которые необходимы для взаимодействия с каким-то конкретным полем.

Они должны быть public-методами.

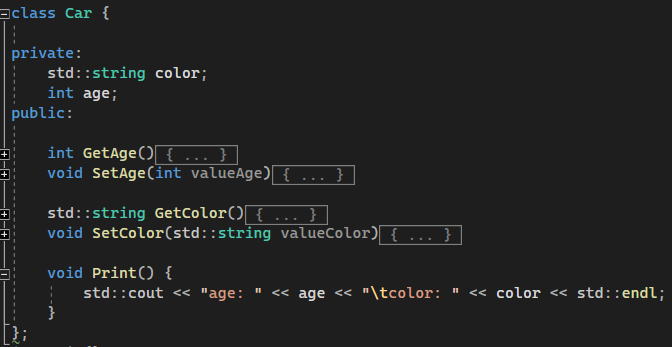
Геттер – получает данные, Сеттер – изменяет.

Тип данных геттера совпадает с типом поля, с которым он взаимодействует;

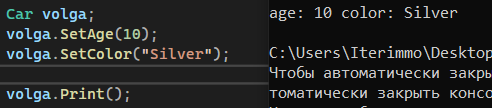
Тип данных сеттера всегда void, однако параметр принимаемой переменной должен совпадать с типом данных поля, которое мы изменяем

Геттеры и сеттеры не обязательно должны идти вместе. Иногда один из них нам не нужен.

**Пример** реализации геттера и сеттера для двух полей класса.



Продолжение. Работа с геттерами и сеттерами и вывод (справа)

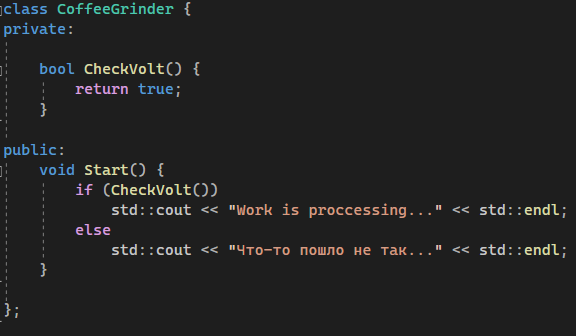


# Инкапсуляция

Инкапсуляция – это один из ключевых принципов ООП, который заключается в том, чтоб скрыть часть функционала класса от пользователя. Предоставить ему только интерфейс и скрыть реализацию.

Пример инкапсуляции (учебный):

Перед тем, как начать работу нашей кофемолки мы проверяем достаточный ли вольтаж поступает на неё. Так как для пользователя было бы неудобно (а иногда он бы и вовсе забывал) проверять вольтаж отдельной кнопкой перед началом работы, мы реализовали метод проверки вольтажа в модификаторе класса private, чтобы он был недоступен для пользователя и включили его в другой метод, который уже, в свою очередь, находится в public. Теперь при нажатии на кнопку старт у нас будет сразу проверяться вольтаж, однако пользователь не может проверить его отдельно. Это и есть хороший пример инкапсуляции данных.



# Конструктор класса

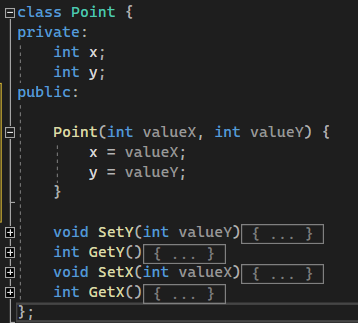
Создается в теле классе (можно создать несколько). Если мы не создали его вручную,то он создается по умолчанию.

Конструктор используется для создания и инициализации полей

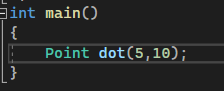
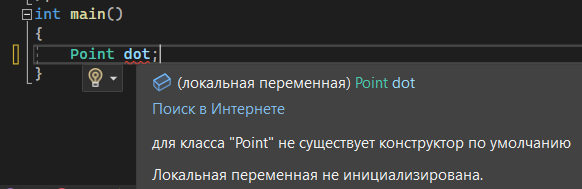
Является функцией, именем которой является имя класса.

Конструктор вызывается каждый раз при создании объекта (переменной) класса.

Выглядит это следующим образом



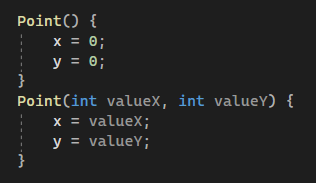
После назначения такого конструктора мы не можем просто создать объект класса Point без инициализации



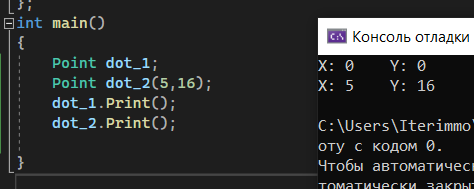
**Перегрузка конструкторов**

Является наглядным примером полиморфизма, когда функция ведет себя по разному в зависимости от ситуации.

Создадим еще один конструктор Point, который будет инициализировать наши переменные нулями, если пользователь ничего не указал.



Таким образом, нам доступны стали следующие варианты создания объекта класса Point:



В этом и заключается полиморфизм: в зависимости от передаваемых параметров конструктор ведет себя по-разному

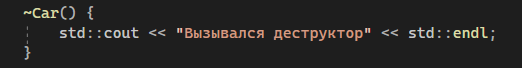
# Деструктор класса

Конструктор используется для создания, **деструктор** – для удалении объекта класса

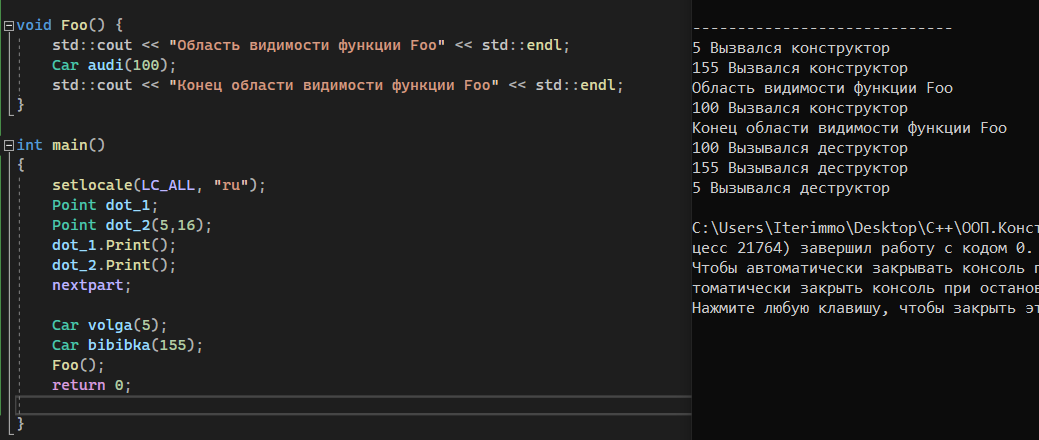
Всегда только **один** деструктор класса

В основном используется для очистки памяти.

Объявляется как конструктор с « ~ » в начале.



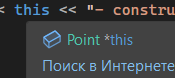
Порядок удаления переменных идет в обратном порядке. Т.е. первым удаляется последний созданный объект. Наглядный пример:



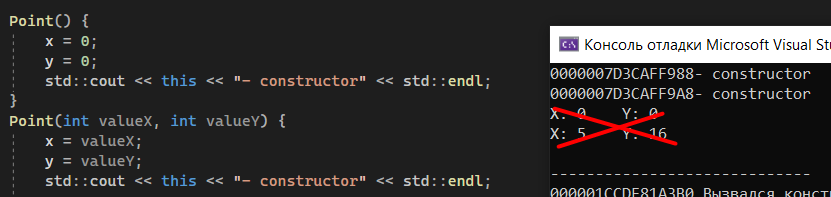
# Ключевое слово this

При обращении через точку object.method(), метод обращается к адресу, в котором расположен наш объект.

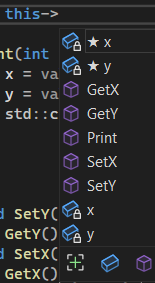
**this** – это указатель объекта на самого себя (адрес, где лежит объект)



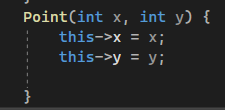
Пример обращения к this в конструкторе



Обращение через this происходят через оператор ->



Пример использования при вызове конструктора



# Конструктор копирования

Существует конструктор копирования по умолчанию.

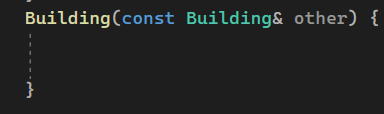
**Конструктор копирования** по умолчанию копирует значения полей объекта, в новый объект.

Если поле представляет указатель, то будет копироваться его адрес, т.е. они будут указывать на одну и ту же область памяти, что может повлечть за собой ошибки.

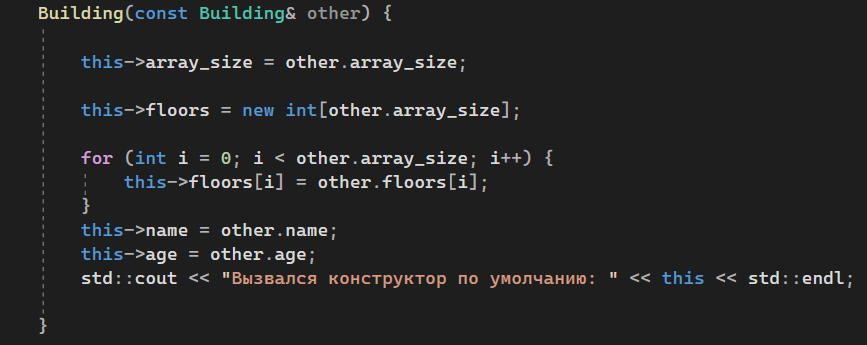
Конструктор по умолчанию будет корректно работать для статической памяти. При работе с динамической памятью нам будет необходимо удалять объекты в деструкторе. В итоге возникнет такая сиутация, что у нас компилятор **дважды** отправляет запрос на очищение области памяти, что вызовет ошибку.

Конструктор копирования задается следующим образом:

<имя класса> (const <имя класса>& var){} // часто вместо var пишут other



Пример его реализации, где указатели используются для динамического массива.



# Перегрузка операторов

Позволяет определеить какие действия будет выполнять оператор

Создать новые операторы нельзя, но можно перегрузить старые

Представляет собой функцию, где после имени стоит ключевое слово **operator** и сам оператор

Может быть как член-функция (метод) класса, так и вне его.

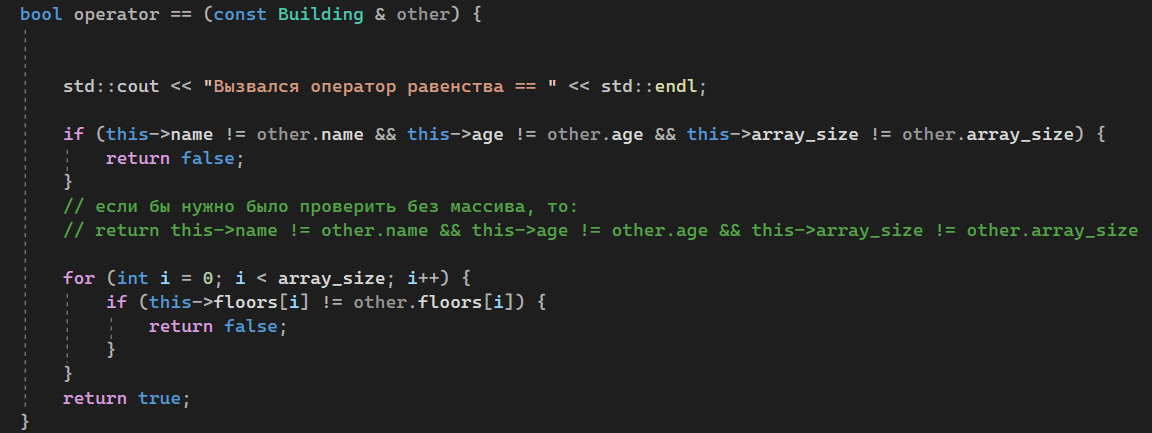
Внутри класса перегружаются

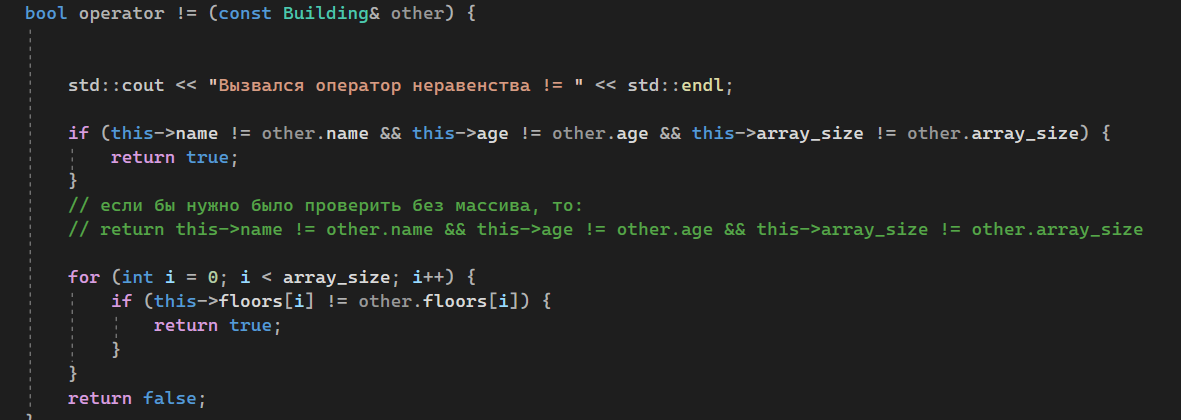
* Инкремент/декремент
* Присовение +=
* Индексирование []
* Вызова ()
* Доступа к члену по указателю ->

Т.е. в классе перегружаются те операторы, которые изменяют состояние объекта или которые с ним непосредственно связаны.

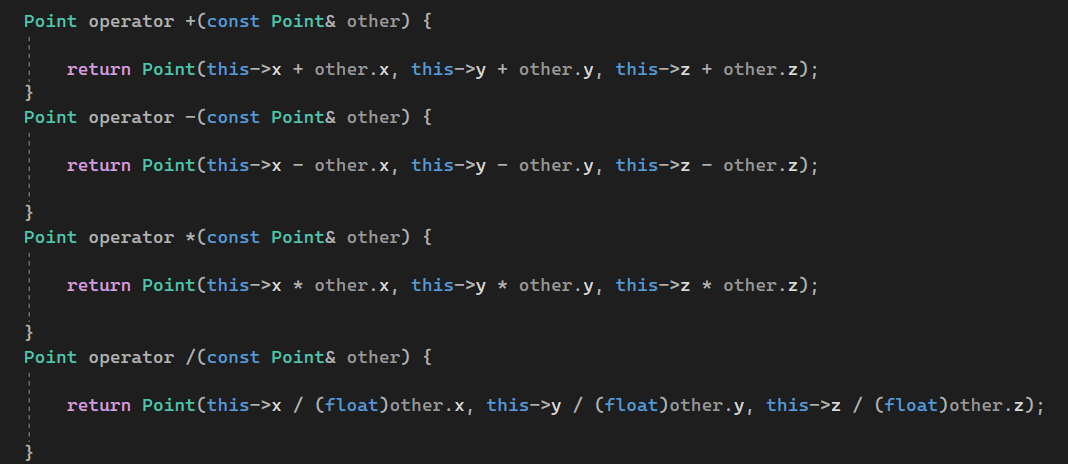
Если нам не нужно создавать новый объект, а надо выполнить действие над текущим, не задействуя ничего иного, то объект нужно передавать по **ссылке**

Пример реализации оператора **равенства и неравенства** внутри класса



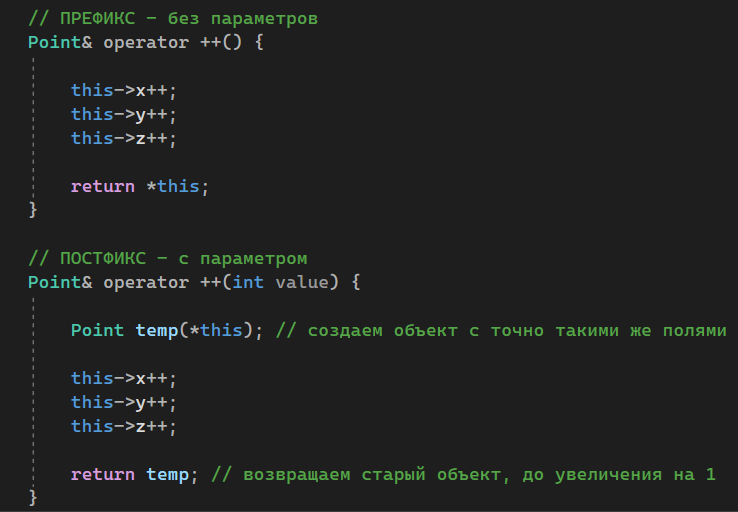


Пример реализации перегрузки **арифметических операторов**



Пример перегрузки **постфиксного и префиксного инкремента**

Для реализации постфикса нам необходимо создать временную переменную, которая хранила бы состояние старого объекта, но имела точно такие же поля. Потом мы прибавляем по каждому из полей, и возвращаем наш старыйобъект. Таким образом, мы вернули старый объект, но увеличили его поля.



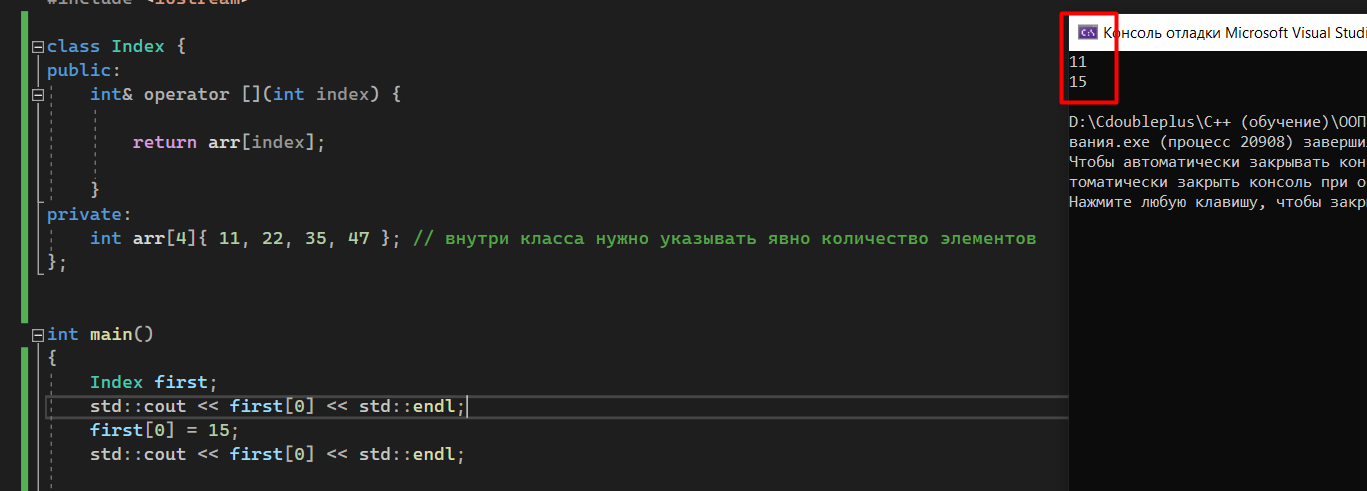
Перегрузка оператора **индексирования**

Получить индекс массива при том, что сам массив находится в Private уровне доступа

Взаимодействие с объектом будет точно также как будто мы взаимодействуем с изначальным массивом.

Если объект необходимо менять, а не только получить, необходимо передавать по ссылке.

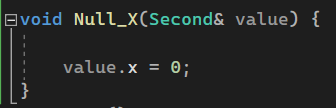
Пример реализации



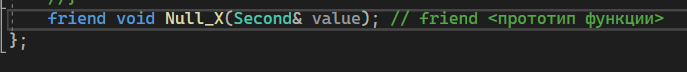
# Дружественные функции

Дружественной функцией по отношению к классу называется та функция, которая хотя и не является членом класса, она имеет доступ к закрытым полям класса (private, protected).

Вот пример её реализации (находится **ВНЕ** класса). Для изменения необходимо передавать по **ссылке**



Для того, чтобы открыть для неё свои private-поля необходимо в самом классе в прописать, что она является дружественной



Практически как прототип функции, только с добавлением слова **friend** в начало.

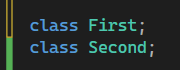
!!! **this** в дружественных функциях не работает !!!

Модификаторы доступа не распостраняются на дружественные функции

Функция может быть дружественна сразу к нескольким классам, однако надо, чтобы во всех классах была объявлена эта функция как дружественная **!**и, объявить все классы, учавствующие в этом, как прототипы классов в самом начале**!**

(чисто технически, надо не все, а только те, которые классу еще неизвестны, но проще так объяснить)

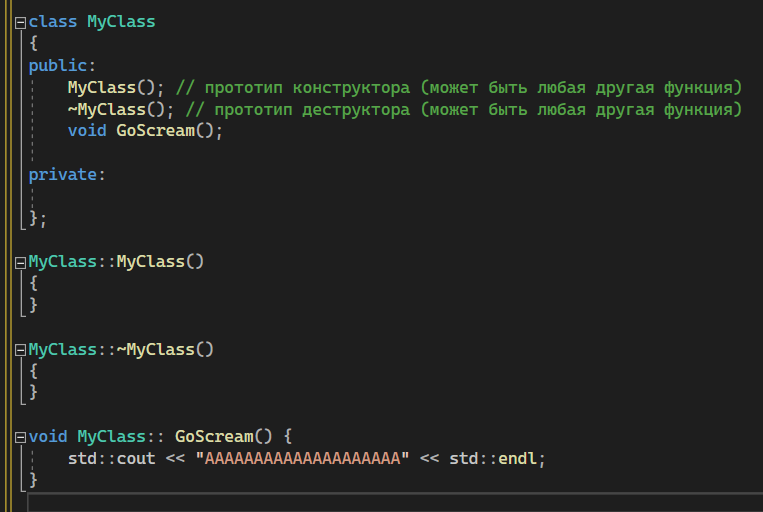
Объявление прототипа класса 1 в 1 как у прототипа функции.



# Определение методов все класса

Выглядит это следующим образом:

* В классе сперва задается прототип функции (метода)
* Вне класса происходит реализация по шаблону
  + <тип данных> <имя класса> :: <имя функции> (параметры)



Напоминает работу std::cout, где мы явно указывали пространство имен, т.е. откуда брать нашу функцию и где она лежит. То же самое здесь делается с классом MyClass.

!!! прототип функции должен находится в public, если мы хотим иметь к нему доступ в любое время !!!

Используется, когда у нас довольно объемные методы и, чтобы улучшить читаемость, мы оставляем в классе прототипы, а саму реализацию выносим вне.

С такого функционала класс **разбивают на файлы**

Видимость одного класса другим можно определить тоже с помощью вынесение методов вне классе. (Об этом речь пойдет далее)

# Дружественный метод класса

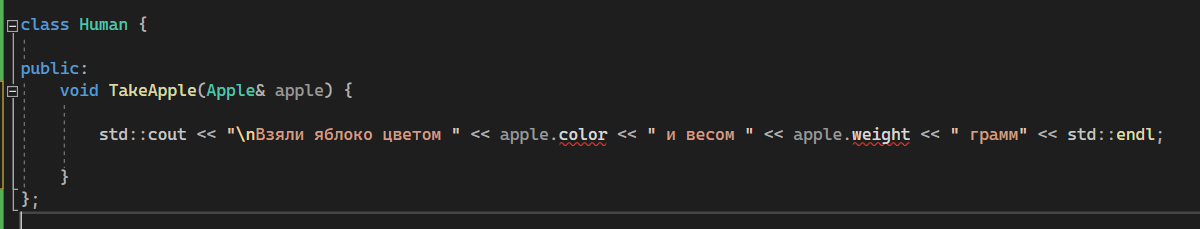
Используется для использования private-полей одного класса для методов другого класса (как и с функциями).

Инициализация:



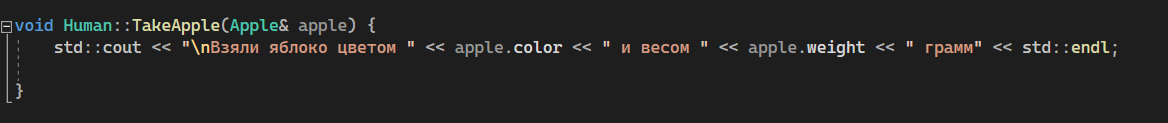
Как с функцией, только теперь мы явно определяем, какому классу она принадлежит.

Есть ньюансы, например:



Хотя мы в классе Apple определили функцию как дружественную, взаимодейстовать из класса с private полями Apple y нас не получится. Для этого необходимо **вынести метод вне класса**

По аналогии с предыдущим разделом



# Дружественные классы

Если нам нужно, чтобы один класс имел доступ к всем полям и методом другого класса, то во втором классе мы прописываем, что этот класс является дружественным первому. То есть, если нам нужно, чтобы класс Human мог работать с полями и методами класса Apple, нам в Apple необходимо указать, что класс Human является дружественным

Инициализация очень простая

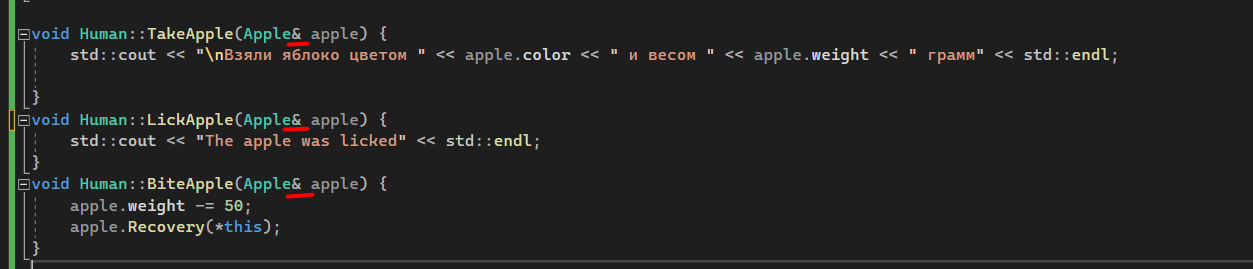
**friend <**имя класса>



В остальном же все будет работать так, как будто это один класс.

Без надобности не надо объявлять дружественность классов (!!!)

!!! Важно: надо передавать по ссылке параметры, чтобы не копировать весь класс целиком и не выделять пол эту копию лишнюю память



# Ключевое поле **static** и статические поля класса

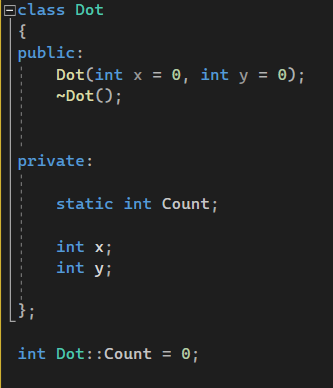
static – одно поле на все объекты класса.

При изменени статического поля в одном экземпляре класса, в других экземплярах оно тоже поменяется.

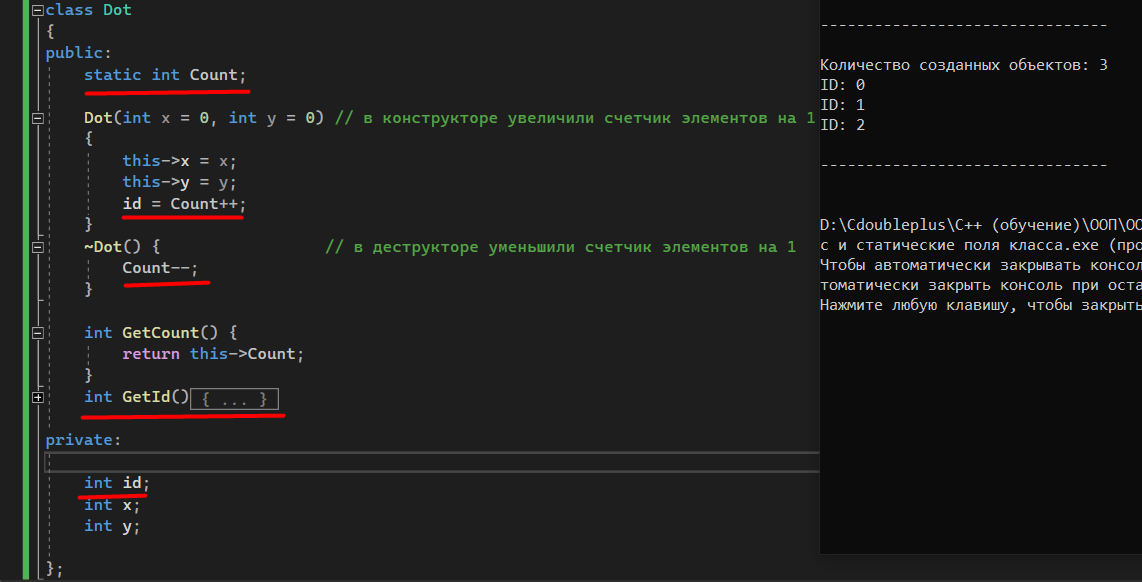
Можно с ней работать через имя класса, а не только через объекты.

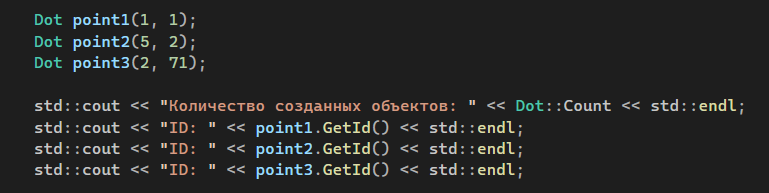
Инициализация происходит ВНЕ класса, а объявление – внутри

**!!!** **должен находится в private** (иначе будут проблемы с инкапсуляцией) **!!!**



Можно использовать, например, для генерации ключей или id



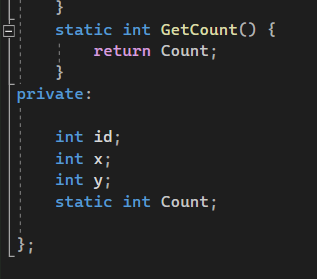


# Статические методы класса. Модификатор **static**

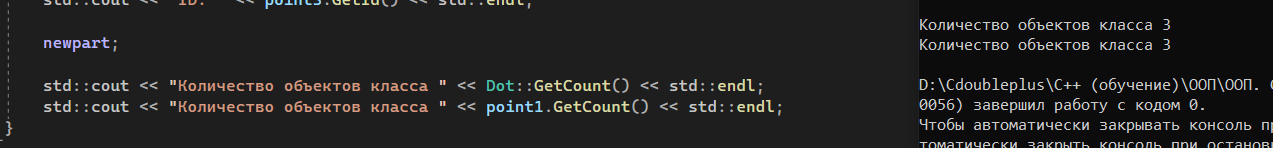
Имеет такое же поведение, как и static поля

Метод будет один единственный для всех экземпляров класса и будет иметь доступ без экземпляров – через имя класса

Объявляется как обычная функция только с использованием модификатора **static**

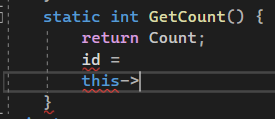


Её вызов может осуществляться не только через объекты, но и **через** **класс**

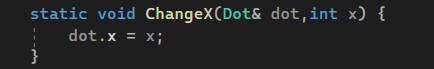


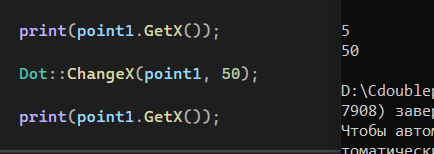
**this нельзя** **использовать** в статическом методе, так как this ссылается на конкретный экземпляр класса, а наш статический метод существует сам по себе, на уровне описания класса.

**Нельзя** обратиться в статических методах **НЕ** к статической переменной



Однако для передачи в статический метод нестатической переменной есть способ: необходимо передать как параметр сам объект класса по ссылке и, например, то, что мы будем изменять.





Было 5, стало 50. Все работает.

# Вложенные классы

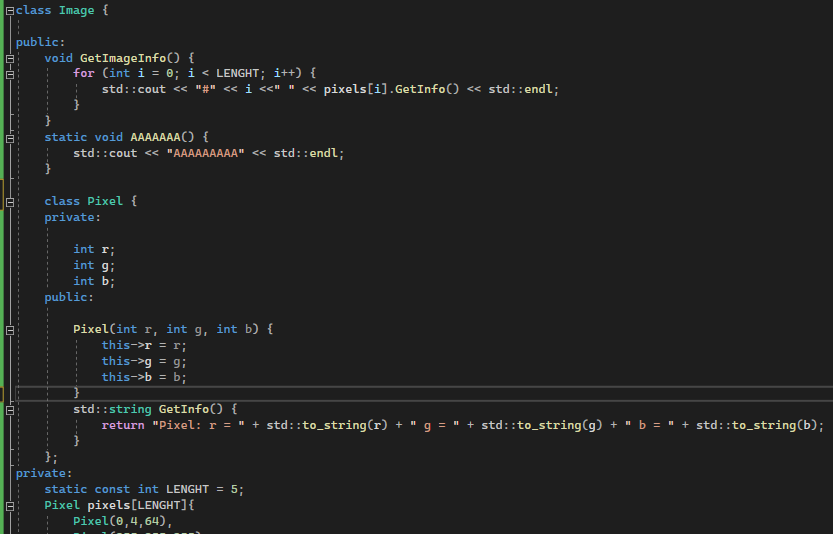
**Вложенный класс** – это класс, реалезация которого описана внутри другого класса.

Также называются внутренними или inner-классами, а тот класс, в который мы вкладываем, называется **объемлющий** класс

Используют для внутрених нужд для того класса, в котором он вложен.

Private поля вложенного класса **недоступны** для объемлющего. В то же время **даже public** поля объемлющего класса недоступны вложенному классу

Например, класс Пиксель может быть внутренним классом класса Картинка

**Пример** реализации вложенного цикла

Класс Pixel расположен внутри класса Image и характеризует цвет пикселя данной картинки (по хорошему надо было задать матрицу с размерностью картинки + ограничить цвеиа до 255…)

создать объект класса Pixel (если тот в Public) можно следующим образом

<имя объемлющего класса>: : < имя внутреннего класса> и как обычно далее

