Оглавление

[Терминология и основы 2](#_Toc129038765)

[Модификаторы доступа 4](#_Toc129038766)

[Геттеры/Сеттеры 6](#_Toc129038767)

[Инкапсуляция 7](#_Toc129038768)

[Конструктор класса 8](#_Toc129038769)

[Деструктор класса 10](#_Toc129038770)

# 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

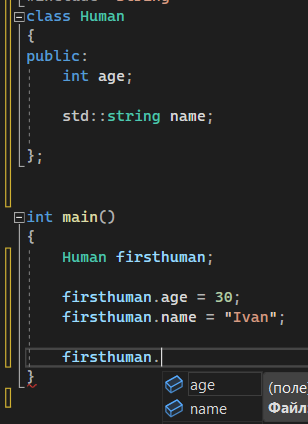
**Поля класса** – переменные, которые содержит в себе класс

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

Обращение к объекту класса происходит следующим образом

* объект.поле\_класса

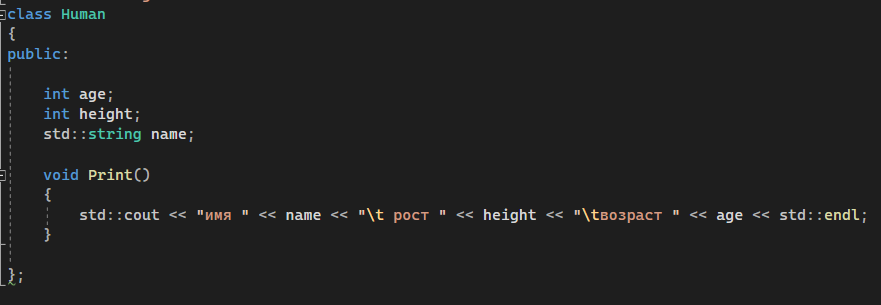
Дальше можем работать с полем в соответствии с проставленым типом данных нашего поля.



**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

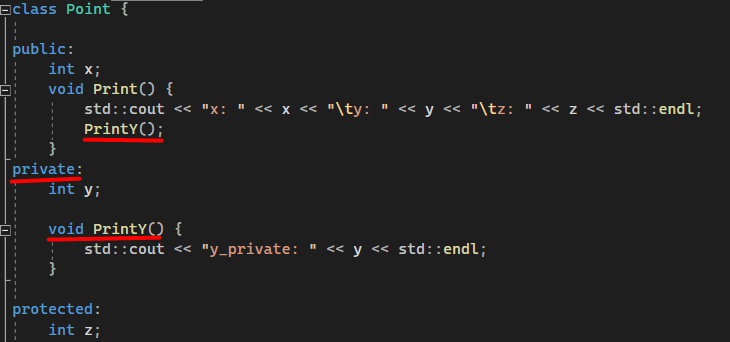
Если явно не указывать какой у поля модификатор доступа, то он будет private

* 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.

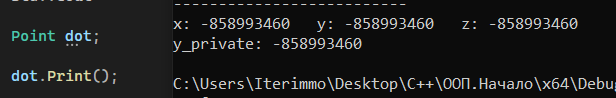
Влияют как на поля, так и на методы класса.

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.

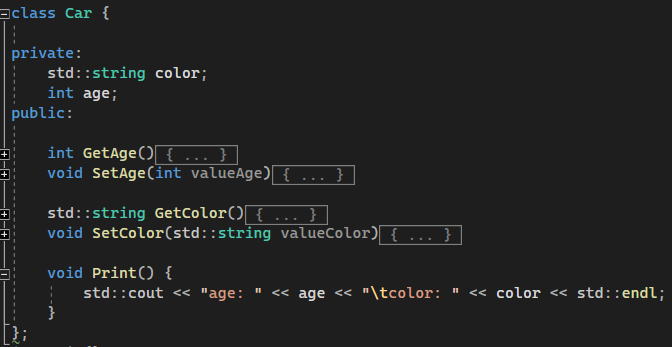
Геттер – получает данные, Сеттер – изменяет.

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

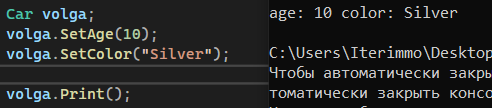
Тип данных сеттера всегда void, однако параметр принимаемой переменной должен совпадать с типом данных поля, которое мы изменяем

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)

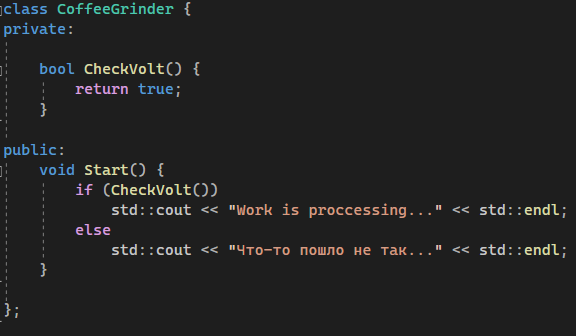


# Инкапсуляция

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

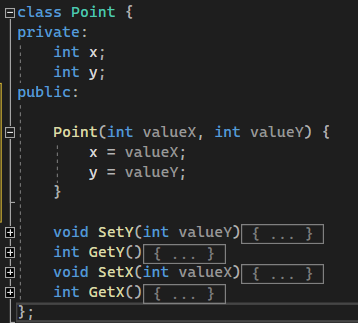
Создается в теле классе (можно создать несколько). Если мы не создали его вручную,то он создается по умолчанию.

Конструктор используется для создания и инициализации полей

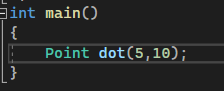
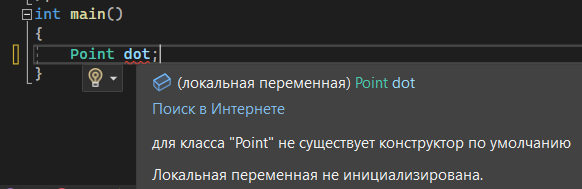
Является функцией, именем которой является имя класса.

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

Выглядит это следующим образом



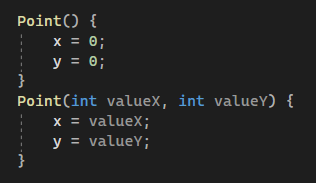
После назначения такого конструктора мы не можем просто создать объект класса Point без инициализации



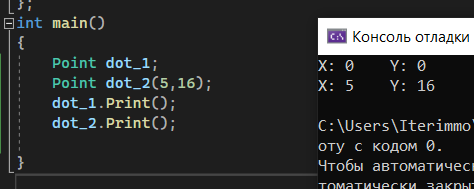
**Перегрузка конструкторов**

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

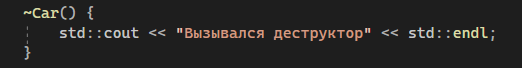
# Деструктор класса

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

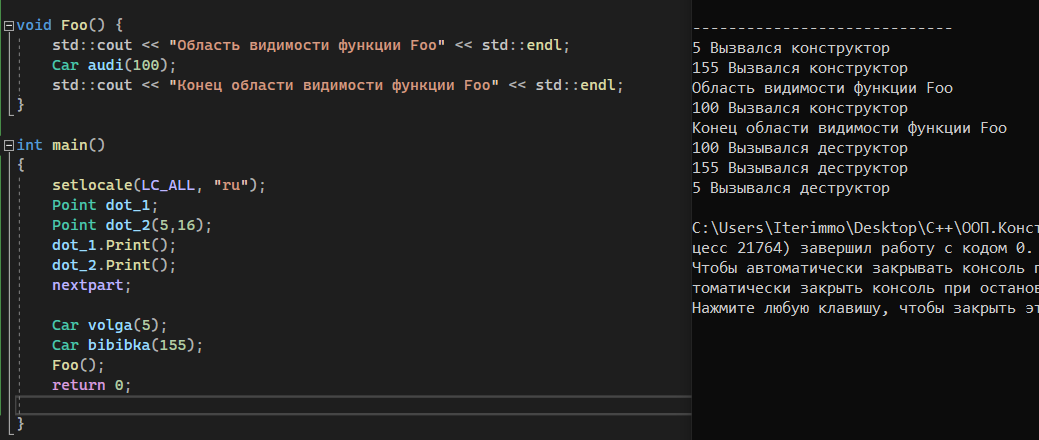
Всегда только **один** деструктор класса

В основном используется для очистки памяти.

Объявляется как конструктор с « ~ » в начале.



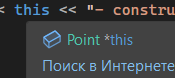
Порядок удаления переменных идет в обратном порядке. Т.е. первым удаляется последний созданный объект. Наглядный пример:



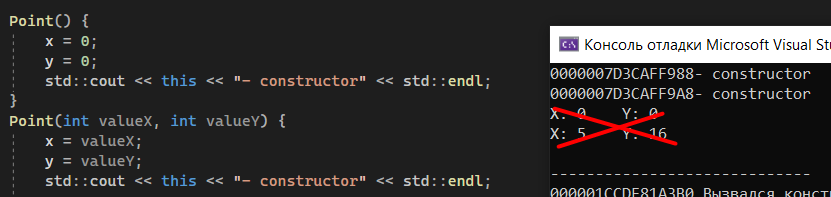
# Ключевое слово this

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

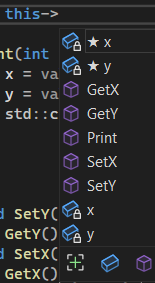
**this** – это указатель объекта на самого себя (адрес, где лежит объект)



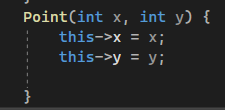
Пример обращения к this в конструкторе



Обращение через this происходят через оператор ->



Пример использования при вызове конструктора



# Конструктор копирования

Существует конструктор копирования по умолчанию.

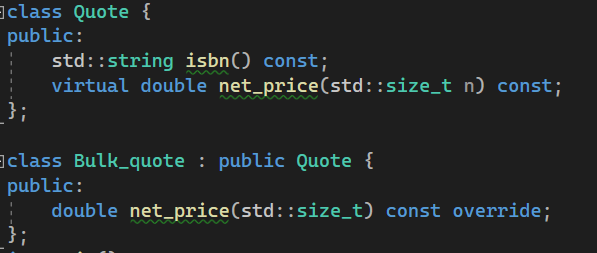
(динамическое связывание,наследование и абстракция данных)

Наследование

Базовый класс (общие во всей иерархии) 🡪 производные классы (специфичные члены)

Производный класс унаследует базовый путем использования списка наследования класса – двоеточие – :

class \*имя производного класса\*:\*имя базового класса\*



Поскольку класс Bulk\_quote использует в списке наследования спецификатор public,его объекты можно использовать так, как будто они являются объектами класса Quote.

**Виртуальные функции –** те функции, которые производные классы должны определять самостоятельно ( базовый класс определяет как virtual)