C++

Классы и объекты в С++ являются основными концепциями объектно-ориентированного программирования — ООП.  Объектно-ориентированное программирование — расширение структурного программирования, в котором основными концепциями являются понятия классов и объектов. Основное отличие языка программирования С++ от С состоит в том, что в С нету классов, а следовательно язык С не поддерживает ООП, в отличие от С++.

Чтобы понять, для чего же в действительности нужны классы, проведём аналогию с каким-нибудь объектом из повседневной жизни, например,  с велосипедом. Велосипед — это объект, который был построен согласно чертежам. Так вот, эти самые чертежи играют роль классов в ООП. Таким образом классы — это некоторые описания, схемы, чертежи по которым создаются объекты. Теперь ясно, что для создания объекта в ООП необходимо сначала составить чертежи, то есть классы. Классы имеют свои функции, которые называются методами класса. Передвижение велосипеда осуществляется за счёт вращения педалей, если рассматривать велосипед с точки зрения ООП, то механизм вращения педалей — это метод класса. Каждый велосипед имеет свой цвет, вес, различные составляющие — всё это свойства. Причём у каждого созданного объекта свойства могут различаться. Имея один класс, можно создать  неограниченно количество объектов (велосипедов), каждый из которых будет обладать одинаковым набором методов, при этом можно не задумываться о внутренней реализации механизма вращения педалей, колёс, срабатывания системы торможения, так как всё это уже будет определено в классе. Разобравшись с назначением класса, дадим ему грамотное определение.

**Классы в С++** - это абстракция описывающая методы, свойства, ещё не существующих объектов. **Объекты** – экз, класса, имеющее свои свойства и методы. Созданные объекты на основе одного класса называются экземплярами этого класса. Эти объекты могут иметь различное поведение, свойства, но все равно будут являться объектами одного класса. В ООП существует три основных принципа построения классов:

1. **Инкапсуляция** - это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.
2. **Наследование** - это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.
3. **Полиморфизм** - свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Каждое свойство построения классов мы рассмотрим подробно по мере необходимости, а пока просто запомните эти три. А теперь вернёмся к классам, для начала рассмотрим структуру объявления классов.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | // объявление классов в С++  class /\*имя класса\*/  {    private:    /\* список свойств и методов для использования внутри класса \*/    public:    /\* список методов доступных другим функциям и объектам программы \*/    protected:    /\*список средств, доступных при наследовании\*/  }; |

Объявление класса начинается с зарезервированного ключевого слова **class**,послекоторого пишется имя класса. В фигурных скобочках, **строки 3 — 10** объявляется тело класса, причём после закрывающейся скобочки обязательно нужно ставить точку с запятой, **строка 10**. В теле класса объявляются три метки спецификации доступа, **строки 4, 6, 8,**после каждой метки нужно обязательно ставить двоеточие. В **строке 4** объявлена метка спецификатора доступа private. Все методы и свойства класса, объявленные после спецификатор доступа private будут доступны только внутри класса. В **строке 6** объявлен спецификатор доступа public, все методы и свойства класса, объявленные после спецификатора доступа  public будут доступны другим функциям и объектам в программе. Пока на этом остановимся, спецификатор доступа protected разбирать сейчас не будем, просто запомните, что он есть. При объявлении класса, не обязательно объявлять три спецификатора доступа, и не обязательно их объявлять в таком порядке. Но лучше сразу определиться с порядком объявления спецификаторов доступа, и стараться его придерживаться.

Начнем с того, что когда мы создаем элементы (переменные) класса, мы не можем присвоить им значения в самом определении класса. Компилятор выдаст ошибку. Поэтому нам необходимо создавать отдельный метод (так называемую set-функцию) класса, с помощью которого и будет происходить инициализация элементов. При этом, если необходимо создать, к примеру, 20 объектов класса, то чтобы инициализировать элементы потребуется 20 раз вызвать set-функции.

Тут нам как раз сможет помочь конструктор класса. Кстати, конструктор (от слова construct — создавать) – это специальный метод класса, который предназначен для инициализации элементов класса некоторыми начальными значениями.

В отличии от конструктора, деструктор (от слова destruct — разрушать) — специальный метод класса, который служит для уничтожения элементов класса. Чаще всего его используют тогда, когда в конструкторе,при создании объекта класса, динамически был выделен участок памяти и необходимо эту память очистить, если эти значения уже не нужны для дальнейшей работы программы.

Важно запомнить:

1. конструктор и деструктор, мы всегда объявляем в разделе public;
2. при объявлении конструктора, тип данных возвращаемого значения не указывается, в том числе — void!!!;
3. у деструктора так же нет типа данных для возвращаемого значения, к тому же деструктору нельзя передавать никаких параметров;
4. имя класса и конструктора должно быть идентично;
5. имя деструктора идентично имени конструктора, но с приставкой ~ ;
6. В классе допустимо создавать несколько конструкторов, если это необходимо. Имена, согласно пункту 2 нашего списка, будут одинаковыми. Компилятор будет их различать по передаваемым параметрам (как при перегрузке функций). Если мы не передаем в конструктор параметры, он считается конструктором по умолчанию;
7. Обратите внимание на то, что в классе может быть объявлен только один деструктор;

Сразу хочу привести пример, который доступно покажет, как работает конструктор:

**ПРИМЕР 1**

# include <iostream>

using namespace std;

class AB //класс

{

    private:

    int a;

    int b;

    public:

    AB()    //это конструктор:  1) у конструктора нет типа возвращаемого значения! в том числе void!!!

    //   2) имя должно быть таким как и у класса (в нашем случае AB)

    {

        a = 0;//присвоим начальные значения переменным

        b = 0;

        cout << "Работа конструктора при создании нового объекта: " << endl;//и здесь же их отобразим на экран

        cout << "a = " << a << endl;

        cout << "b = " << b << endl << endl;

    }

    void setAB() // с помощью этого метода изменим начальные значения заданные конструктором

    {

        cout << "Введите целое число а: ";

        cin >> a;

        cout << "Введите целое число b: ";

        cin >> b;

    }

    void getAB() //выведем на экран измененные значения

    {

        cout << "a = " << a << endl;

        cout << "b = " << b << endl << endl;

    }

};

int main()

{

    setlocale(LC\_ALL, "rus");

    AB obj1;     //конструктор сработает на данном этапе (во время создания объекта класса)

    obj1.setAB();   //присвоим новые значения переменным

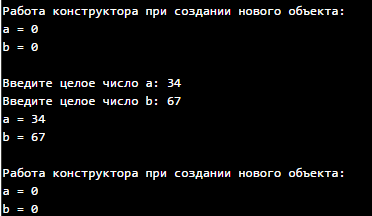
    obj1.getAB();   //и выведем их на экран

    AB obj2;     //конструктор сработает на данном этапе (во время создания 2-го объекта класса)

return 0;

}

Результат работы программы:



Как видно из результата работы программы, конструктор срабатывает сразу, при создании объектов класса, поэтому, явно вызывать конструктор не нужно, он сам «приходит» :)

Хочется еще добавить, что, как и обычным функциям, мы можем передавать конструктору параметры. Через параметры, конструктору можно передавать любые данные, которые будут необходимы при инициализации объектов класса.

Рассмотрим еще один пример, это все та же программа, только в код внесены некоторые изменения. Тут же покажем принцип работы деструктора:

**ПРИМЕР 2**

# include <iostream>

using namespace std;

class AB //класс

{

    private:

    int a;

    int b;

    public:

    AB(int A, int B) //эти параметры мы передадим при создании объекта в main

    {

        a = A;//присвоим нашим элементам класса значения параметров

        b = B;

        cout << "Тут сработал конструктор, который принимает параметры: " << endl;//и здесь же их отобразим на экран

        cout << "a = " << a << endl;

        cout << "b = " << b << endl << endl;

    }

    void setAB()

    {

        cout << "Введите целое число а: ";

        cin >> a;

        cout << "Введите целое число b: ";

        cin >> b;

    }

    void getAB()

    {

        cout << "a = " << a << endl;

        cout << "b = " << b << endl << endl;

    }

    ~AB() // это деструктор. не будем заставлять его чистить память, пусть просто покажет где он сработал

    {

        cout << "Тут сработал деструктор" << endl;

    }

};

int main()

{

setlocale(LC\_ALL, "rus");

AB obj1(100, 100);  //передаем конструктору параметры

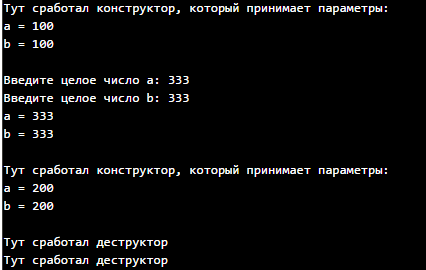
obj1.setAB();   //присвоим новые значения переменным

obj1.getAB();   //и выведем их на экран

AB obj2(200, 200);  //передаем конструктору параметры

}

Смотрим результат работы программы:



Деструктор срабатывает в тот момент, когда завершается работа программы и уничтожаются все данные. Мы его не вызывали – он сработал сам. Как видно, он сработал 2 раза, так как и конструктор. Уже от себя добавлю, что, в первую очередь, он удалил второй созданный объект (где a = 200, b = 200), а затем первый (где a = 100, b = 100). «Последним пришёл — первым вышел».

C#

По аналогии выполним тоже самое, но на языке С#:

ПРИМЕР 1

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication1

{

class AB

{

private int a;

private int b;

public AB() //Конструктор по умолчанию

{

a = 0;

b = 0;

Console.WriteLine("Работа конструктора при создании нового объекта: ");

Console.WriteLine("a = {0}", a);

Console.WriteLine("b = {0}", b);

Console.WriteLine();

}

public void setAB()

{

Console.WriteLine("Введите целое число а: ");

a = int.Parse(Console.ReadLine());

Console.WriteLine("Введите целое число b: ");

b = int.Parse(Console.ReadLine());

Console.WriteLine();

}

public void getAB()

{

Console.WriteLine("a = {0}", a);

Console.WriteLine("b = {0}", b);

Console.WriteLine();

}

}

class Program

{

static void Main(string[] args)

{

AB obj1 = new AB();

obj1.setAB();

obj1.getAB();

AB obj2 = new AB();

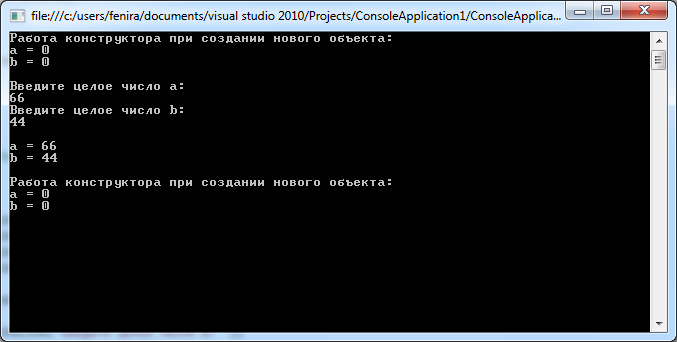
Console.ReadKey();

}

}

}

Результат выполнения:



ПРИМЕР 2:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace ConsoleApplication1

{

class AB

{

private int a;

private int b;

public AB(int A, int B)

{

a = A;

b = B;

Console.WriteLine("Тут сработал конструктор, который принимает параметры: ");

Console.WriteLine("a = {0}", a);

Console.WriteLine("b = {0}", b);

Console.WriteLine();

}

public void setAB()

{

Console.WriteLine("Введите целое число а: ");

a = int.Parse(Console.ReadLine());

Console.WriteLine("Введите целое число b: ");

b = int.Parse(Console.ReadLine());

Console.WriteLine();

}

public void getAB()

{

Console.WriteLine("a = {0}", a);

Console.WriteLine("b = {0}", b);

Console.WriteLine();

}

~AB()

{

Console.WriteLine("Тут сработал деструктор");

}

}

class Program

{

static void Main(string[] args)

{

AB obj1 = new AB(100, 100);

obj1.setAB();

obj1.getAB();

AB obj2 = new AB(200, 200);

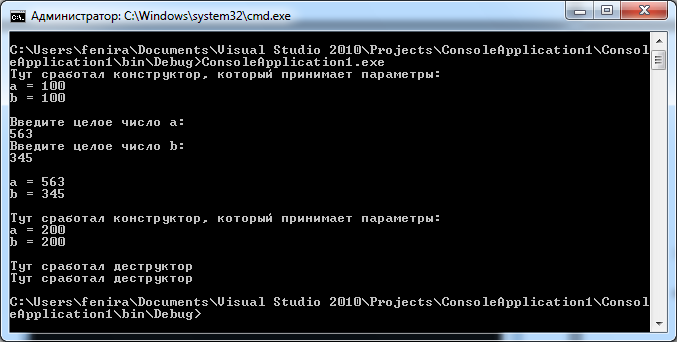
Console.ReadKey();

}

}

}

Результат работы:



Отличия от С++:

* Стоит заметить, что деструктор в примере используется только в обучающих целях, т.к. в языке C# имеется встроенный «Сборщик мусора» и необходимость в деструкторе отпадает автоматически.
* Так же в языке C# принято каждый раз перед описанием метода или параметра ставить модификатор доступа (private, public, protected).

JAVA

По аналогии выполним тоже самое, но на языке Java:

ПРИМЕР 1

package main;

import java.util.Scanner;

public class AB {

private int a;

private int b;

public AB() {

a = 0;

b = 0;

System.out.println("Работа конструктора при создании нового объекта:");

System.out.print("a = "); System.out.println(a);

System.out.print("b = "); System.out.println(b);

}

public void setAB() {

Scanner in = new Scanner(System.in);

System.out.println("Введите целое число а:");

a = in.nextInt();

System.out.println("Введите целое число b:");

b = in.nextInt();

in.close();

}

public void getAB() {

System.out.print("a = "); System.out.println(a);

System.out.print("b = "); System.out.println(b);

}

}

package main;

public class test {

public static void main(String[] args) {

AB obj1 = new AB();

obj1.setAB();

obj1.getAB();

AB obj2 = new AB();

}

}

Результат работы:

Работа конструктора при создании нового объекта:

a = 0

b = 0

Введите целое число а:

23

Введите целое число b:

32

a = 23

b = 32

Работа конструктора при создании нового объекта:

a = 0

b = 0

Пример 2

package main;

import java.util.Scanner;

public class AB {

private int a;

private int b;

public AB(int A, int B) {

System.out.println("Тут сработал конструктор, который принимает параметры:");

a = A;

b = B;

System.out.print("a = "); System.out.println(a);

System.out.print("b = "); System.out.println(b);

}

public void setAB() {

Scanner in = new Scanner(System.in);

System.out.println("Введите целое число а:");

a = in.nextInt();

System.out.println("Введите целое число b:");

b = in.nextInt();

in.close();

}

public void getAB() {

System.out.print("a = "); System.out.println(a);

System.out.print("b = "); System.out.println(b);

}

}

package main;

public class test {

public static void main(String[] args) {

AB obj1 = new AB(100, 100);

obj1.setAB();

obj1.getAB();

AB obj2 = new AB(200, 200);

}

}

Результат работы программы:

Тут сработал конструктор, который принимает параметры:

a = 100

b = 100

Введите целое число а:

123123

Введите целое число b:

3123

a = 123123

b = 3123

Тут сработал конструктор, который принимает параметры:

a = 200

b = 200

Отличия от С++ и С#:

* В Java отсутствуют деструкторы в принципе.
* Реализации двух классов должны находиться в разных файлах.

Ход работы:

1. Создать класс описывающий сущность согласно варианту.
2. Создать конструктор без параметров
3. Создать конструктор с параметрами
4. Создать деструктор
5. Создать методы для чтения данных в объект и вывода из объекта.
6. Реализовать программы на С++, С#, Java
7. \* Создать конструктор копирования для объекта на языке С++.

Варианты заданий (количество полей данных, не менее 6):

1. Автомобиль
2. Самолет
3. Студент
4. Корабль
5. Город
6. Животное
7. Мобильный телефон
8. Ноутбук
9. Музыкальная группа
10. Завод
11. Фирма
12. Танк
13. Компьютерная игра
14. Фильм
15. Картина