

# Конструкторы, Деструкторы

Лекция + практика

# Что такое конструктор?

**Конструктор** — это особый тип метода класса, который автоматически вызывается при создании объекта этого класса. Конструкторы обычно используются для инициализации переменных-членов класса значениями, которые предоставлены по умолчанию/пользователем, или для выполнения любых шагов настройки, необходимых для класса (например, открыть файл или базу данных).

При создании объекта класса, который не содержит ни одного конструктора, будет вызываться **неявно заданный конструктор по умолчанию**, выделяющий память для объекта класса.

В классе можно объявить собственный конструктор по умолчанию. Такой конструктор называется: **явно заданный конструктор по умолчанию**.

## Особенности при работе с конструкторами:

- Имя совпадает с именем класса;
- Не имеет типа возврата (**даже void**);
- Не используется для повторной инициализации;
- Конструкторов может быть несколько (перегрузка функции);
- Всегда имеется хотя бы 1 конструктор (явный или неявный).
- Как правило, расположен в разделе **public**.



# Многообразие конструкторов.

*Конструкторы  
строят объекты из  
пыли.*

```
class A {  
    public:  
        A() {...} // конструктор  
  
        ~A() {...} // деструктор  
  
        void AnotherMeth();  
  
};
```

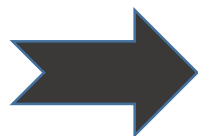


# Виды инициализации

**Прямая**  
`A obj(5);`

Используйте прямую инициализацию или uniform-инициализацию с объектами ваших классов.

**Uniform**  
`A obj{5};`



```
class A
{
    int a;
public:
    A(int a = 1) {}
};
```

**Копирующая**  
`A obj = A(5);`

Не используйте копирующую инициализацию с объектами ваших классов

Инициализация –  
присваивание  
значения при  
определении.

```
static = 0;
dynamic = NaN;
auto = NaN;
```

# Конструктор по умолчанию

Конструктор, который не имеет пар-ов (или содержит пар-ры, которые все имеют знач-я по умолчанию).

Вызывается, если пользователем не указаны знач-я для инициализации.

«Конструктор по умолчанию» автоматически создается и вызывается, если в классе не объявлено ни одного конструктора.

Как только объявлен «конструктор с параметрами», то при объявлении объекта **mp**:

**CMyPoint mp;** ➡ **ошибка!**

// компилятору предложили вызвать констр. по  
//умолчанию (явный или неявный, а его нет!)

Если не задан пользователем **явно** будет создан компилятором – **неявно** !

// класс, точки на координатной плоскости

**class CMyPoint**

{

int x, y;

public:

// явно заданный констр-р по умолчанию

**CMyPoint()** {

**x = y = 0;**

} // явно заданный констр-р с параметрами

**CMyPoint(int a, int b)** {

**x = a;**

**y = b;**

}

// методы класса

};

Каждый класс может иметь только один конструктор по умолчанию (либо явный, либо неявный).

# Конструктор с параметрами (пользователя)

Конструктор может иметь любое количество параметров.

- «*Пользовательский конструктор*» – это конструктор принимающий параметры от пользователя.
- Задача «*пользовательского конструктора*» – инициализация полей предопределенными пользователем значениями.

```
// класс, который определяет дату
class CMyDate
{
    int day;
    int month;
    int year;

public:
    // конструкторы класса
    CMyDate(); // констр-р по умолчанию
    CMyDate(int d, int m, int y=1); // констр-р с 3 парам-и

    // методы класса...
};
```

# Какой конструктор использовать?

Может быть,  
с парам-и по умолчанию?



```
class Fraction
{
private:
    int m_numerator;
    int m_denominator;
public:
    // Явный констр-р по умолчанию
    Fraction(int numerator = 0, int denominator = 1)
    {
        assert(denominator != 0);
        m_numerator = numerator;
        m_denominator = denominator;
    }
};
```

# Список инициализации членов класса

- Списки инициализации членов позволяют инициализировать члены, а не присваивать им значения.
- Это единственный способ инициализации констант и ссылок, которые являются переменными-членами вашего класса.
- Во многих случаях использование списка инициализации может быть более результативным, чем присваивание значений переменным-членам в теле конструктора.
- Списки инициализации работают как с переменными фундаментальных типов данных, так и с членами, которые сами являются классами.

```
class Values
```

```
{
```

```
private:
```

```
    int  m_val1;
```

```
    char m_val2;
```

```
    float m_val3;
```

```
public:
```

```
    Values(char value2='d', int value1, float value3=17.5)  
        : m_val2(value2), m_val1(value1), m_val3(value3)
```

```
    {
```

```
        .....
```

```
    }
```

```
};
```

Порядок  
инициализации  
будет таким !

Переменные в списке инициализации не инициализируются в том порядке, в котором они указаны. Вместо этого они инициализируются в том порядке, в котором **объявлены в классе**



# Список инициализации членов класса.

## Продолжение.

Следует соблюдать следующие рекомендации при работе со списком инициализации:

- Не инициализируйте переменные-члены таким образом, чтобы они зависели от других переменных-членов, которые инициализируются первыми (другими словами, убедитесь, что все ваши переменные-члены правильно инициализируются, даже если порядок в списке инициализации отличается).
- Инициализируйте переменные в списке инициализации в том порядке, в котором они объявлены в классе.

# А как инициализировать массивы?

```
class Values
{
private:
    const int m_array[7];
};
```

```
class Values
{
private:
    const int m_array[7];

public: //используем uniform- инициализацию для
        //инициализации массива
    Values(): m_array { 3, 4, 5, 6, 7, 8, 9 }
    {
    }
};
```

# Классы, содержащие другие классы

Одни классы могут содержать другие классы в качестве переменных-членов.

При создании переменной **b** вызывается конструктор **B()**.

Прежде чем тело конструктора **B()** выполнится, создастся **m\_a**, вызывая констр-р по умолчанию класса **A**.

В результате работы **main()**:

>Constr. A

>Constr. B

```
class A
{
public:
    A() { std::cout << ">Constr. A\n"; }
};

class B
{
private:// B содержит A, как переменную-член
    A m_a;
public:
    B() { std::cout << ">Constr. B\n"; }
};

int main(){
    B b;
    return 0;
}
```

# Конструктор копирования

- **Конструктор копирования** — это особый тип конструктора, который используется для создания нового объекта через копирование существующего объекта. Как в случае с конструктором по умолчанию, если вы не предоставите конструктор копирования для своих классов самостоятельно, то язык C++ создаст **public-конструктор копирования автоматически**.

```
class A
{
private:
    int m_a;
    int m_b;
public: // констр-р с параметрами
    A(int a, int b) : m_a(a), m_b(b)
    {
    }
    // констр-р копирования
    A(const A & obj) : m_a(obj.m_a), m_b(obj.m_b)
    {
    }
};
```

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

Поскольку компилятор мало знает о вашем классе, то по умолчанию созданный конструктор копирования будет использовать **почленную инициализацию**.

# Проблема дублирования кода в конструкторе.

```
class Boo
{
public:
    Boo()
    {
        // Часть кода X
    }
    Boo(int value)
    {
        // Часть кода X
        // Часть кода Y
    }
};
```

Не редкость встретить класс с несколькими конструкторами, которые частично выполняют одно и то же

# Делегирующие конструкторы

- Начиная с C++11, конструкторам разрешено вызывать другие констр-ры. Этот процесс называется **делегированием конструкторов** (или **«цепочкой конструкторов»**).
- Чтобы один конструктор вызывал другой, нужно просто сделать вызов этого конструктора в **списке инициализации членов**.



```
class Boo
{
public:
    Boo()
    {
        // Часть кода X
    }

    Boo(int value): Boo() // используем
                        // конструктор по умолчанию Boo()
                        // для выполнения части кода X
    {
        // Часть кода Y
    }
};
```

# Деструктор класса

**Деструктор** — это специальный тип метода класса, который выполняется при удалении объекта класса. Деструкторы предназначены для очистки памяти после работы класса.

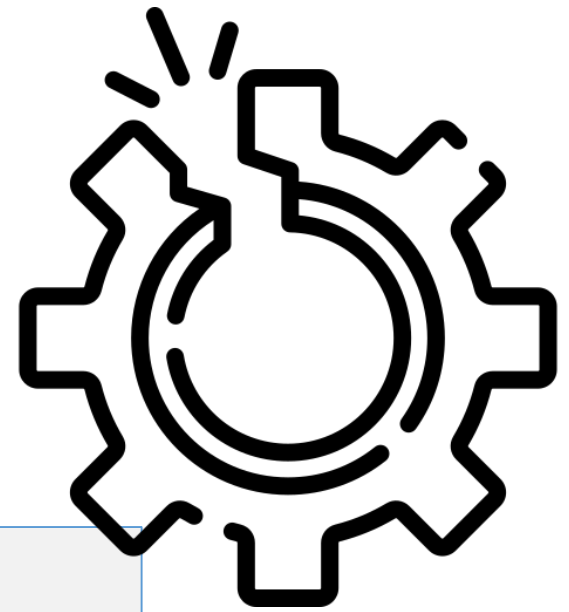
Для простых классов (тех, которые только инициализируют значения обычных переменных-членов) **деструктор не нужен!**

Если объект класса содержит к.л. ресурсы (динамически выделенную память или файл/базу данных), или, если вам необходимо выполнить какие-либо действия до того, как объект будет уничтожен, следует воспользоваться деструктором, поскольку он производит последние действия с объектом перед его окончательным уничтожением.

```
class A {  
    public:  
        A(){...} // конструктор  
        ~A(){...} // деструктор  
};
```

## Особенности при работе с деструктором:

- Имя совпадает с именем класса, но со знаком ~ ;
- Не может принимать аргументы;
- Не имеет типа возврата;
- Не может быть вызван явно, но может быть вызван другим методом или оператором **delete**;
- Может быть только 1 в классе.



# Предупреждение о функции `exit()`

Если вы используете функцию `exit()`, то ваша программа завершится, и никакие **деструкторы не будут вызваны**.

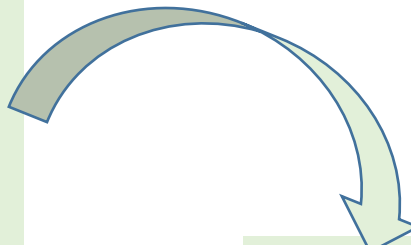
**Будьте осторожны**, если в таком случае вы полагаетесь на свои деструкторы для выполнения необходимой работы по очистке (например, перед тем, как выйти, вы записываете что-нибудь в лог-файл или в базу данных).



# П. Делегирующие конструкторы.

```
#include <iostream>
#include <string>
class Employee
{
private:
    int      m_id;
    std::string m_name;

public:
    Employee(int id=0, const std::string &name=""):
        m_id(id), m_name(name)
    {
        std::cout << "Employee " << m_name << " created.\n";
    }
    // Используем делегирующие конструкторы для
    //сокращения дублированного кода
    Employee(const std::string &name) : Employee(0, name) { }
};
```



```
int main()
{
    Employee a;
    Employee b("Ivan");

    return 0;
}
```

# П. Деструктор

```
#include <iostream>
#include <cassert>
class Massiv
{
private:
    int *m_array;
    int m_length;
public:
    Massiv(int length) // конструктор
    {
        assert(length > 0);
        m_array = new int[length];
        m_length = length;
    }
    ~Massiv() // деструктор
    {
        // Динамически удаляем массив, который выделили ранее
        delete[] m_array ;
    }
    void setValue(int index, int value) { m_array[index] = value; }
    int getValue(int index) { return m_array[index]; }
    int getLength() { return m_length; }
};
```

```
int main()
{
    Massiv arr(15); // выделяем 15 целочисленных значений
    for (int count=0; count < 15; ++count)
        arr.setValue(count, count+1);

    std::cout << "The value of element 7 is " << arr.getValue(7);

    return 0;
} // arr удаляется здесь, и деструктор ~Massiv() вызывается тоже здесь
```

# П. Выполнение конструкторов и деструкторов.

```
#include <iostream>
class Smth
{
private:
    int m_nID;

public:
    Smth(int nID)
    {
        std::cout << "Constr. Smth " << nID << '\n';
        m_nID = nID;
    }

    ~Smth()
    {
        std::cout << "Destruct. Smth " << m_nID << '\n';
    }

    int getID() { return m_nID; }
};
```

```
int main()
{
    // Выделяем объект класса Smth из стека
    Smth object(1);

    std::cout << object.getID() << '\n';

    // Выделяем объект Smth динамически из кучи
    Smth *pObject = new Smth(2);

    std::cout << pObject->getID() << '\n';
    delete pObject;

    return 0;
} // object выходит из области видимости здесь
```

# Домашняя работа # 11

Составить описание класса для представления времени.

Предусмотреть возможности установки времени и изменения его отдельных полей (час, минута, секунда) с проверкой допустимости вводимых значений. В случае недопустимых значений полей выводятся сообщения об ошибке. Создать методы изменения времени на заданное количество часов, минут и секунд.