# Синтаксис ООП в C++, примеры классов

*Программирование*

# Содержание

- Объявление класса, создание объекта
- Создание массива объектов
- Разделение на объявление и реализацию
- Использование ключевого слова this
- Конструктор класса
- Деструктор класса
- Конструктор и создание массивов
- Использование ключевого слова const
- Конструктор копий
- Перегрузка оператора =

# Объявление класса, создание объекта

Объявление класса (можно, например, перед main()):

```
class MyClass {
public:
    void Foo () {
        printf ("Function Foo from MyClass");
    }
};
```

Простое создание и использование объекта:

```
int main (int argc, char* argv[]) {
    MyClass object1;
    object1.Foo();
    return 0;
}
```

в этом месте object1 удаляется автоматически

Динамическое выделение памяти под объект:

```
MyClass* object2 = new MyClass();
object2->Foo();
delete object2;
```

# Создание массива объектов

Статичный массив:

```
MyClass arr[10];
for (int i = 0; i < 10; i++)
    arr[i].Foo();
```

Массив с выделением памяти:

```
MyClass* arr = new MyClass[10];
for (int i = 0; i < 10; i++)
    arr[i].Foo();
delete[] arr;
```

инициализируются все 10 объектов

все объекты удаляются вместе с массивом

Массив указателей на объекты с выделением памяти:

```
MyClass** arr = new MyClass* [10];
for (int i = 0; i < 10; i++) {
    arr[i] = new MyClass();
    arr[i]->Foo();
}
for (int i = 0; i < 10; i++)
    delete arr[i];
delete[] arr;
```

10 указателей на объекты

выделение памяти под конкретный объект

удаление объекта

удаление массива

# Разделение на объявление и реализацию

В языке C++ удобно разделять объявление и реализацию классов, чтобы использовать в проектах как модули:

```
// файлы класса (можно считать модулем)
myclass.h
myclass.cpp

// точка входа и общие алгоритмы
main.cpp
```

Объявление класса – содержимое файла **myclass.h**:

```
#include <stdio.h>

class MyClass {
public:
    void Foo();
    int GetValue();
private:
    int hiddenValue;
};
```

# Разделение на объявление и реализацию

В языке C++ удобно разделять объявление и реализацию классов, чтобы использовать в проектах как модули:

```
// файлы класса (можно считать модулем)
myclass.h
myclass.cpp

// точка входа и общие алгоритмы
main.cpp
```

Реализация методов класса – **myclass.cpp**:

```
#include "myclass.h"

void MyClass::Foo() {
    hiddenValue = 42;
    printf ("hiddenValue initialized");
}

int MyClass::GetValue() {
    return hiddenValue;
}
```

# Разделение на объявление и реализацию

В языке C++ удобно разделять объявление и реализацию классов, чтобы использовать в проектах как модули:

```
// файлы класса (можно считать модулем)
myclass.h
myclass.cpp

// точка входа и общие алгоритмы
main.cpp
```

Использование класса – **main.cpp**:

```
#include <stdio.h>
#include "myclass.h"

int main (int argc, char* argv[]) {
    MyClass obj;
    obj.Foo();
    int k = obj.GetValue();
    printf ("%d", k);
    return 0;
}
```

# Условное деление классов по ролям

Можно выделить два типа классов:

**1. Основные**

Player – игрок
GameField – игровое поле
Account – банковский счёт

**2. Вспомогательные**

MyString – работа со строками
MyConfig – настройки
MyDataBase – хранилище данных
MyWeb – работа с Интернетом

**Основные** описывают предметную область конкретной задачи и создаются прежде всего под её условия.

**Вспомогательные** классы создаются универсальными и используются повторно в других программах.

# Использование ключевого слова this

В методах класса имена переменных вступают в коллизию:

```
class MyClass {
private:
    char buff[100];
    int size;
public:
    void SetBuff (char *buff, int size) {
        this->size = size;
        strncpy ( this->buff, buff, size );
        this->buff[size] = '\0';
    }
};
```

имена параметров функции дублируют имена свойств класса

ключевое слово **this** спешит на помощь

Внутри методов класса **this** является указателем на экземпляр класса, от имени которого вызван метод.

# Конструктор класса

Автоматическую инициализацию объекта можно выполнить при помощи специальных методов – конструкторов:

```
class MyClass {
private:
    int* arrBuff;
    int size;
public:
    MyClass() {
        size = 42;
        arrBuff = new int[size];
        memset( arrBuff, 0, size * sizeof(int) );
    }
    void SetElement (int index, int value)  { arrBuff[index] = value; }
    int GetElementAt (int index)        { return arrBuff[index]; }
};
```

```
MyClass intsArray;
intsArray.SetElement(13, 99345);

printf( "%d", intsArray.GetElementAt(13) );
printf( "%d", intsArray.GetElementAt(14) );
```

# Конструктор класса

Конструктор с параметрами:

```
class MyClass {
private:
    int* arrBuff;
    int size;
public:
    MyClass (int initSize, int defaultValue = 0) {
        size = initSize;
        arrBuff = new int[size];
        for (int i = 0; i < size; i++)
            arrBuff[i] = defaultValue;
    }
};
```

```
MyClass intsArray(10, 42);      // 10 элементов со значением 42 каждый
MyClass zeroIntsArray(17);      // 17 элементов со значением 0

MyClass* ptrIntsArray = new MyClass(20, 100);      // 20 элементов со значением 100
delete ptrIntsArray;
```

```
MyClass errArray;          //! ОШИБКА: нет конструктора по-умолчанию
```

# Конструктор класса

Несколько конструкторов:

```
class MyClass {
private:
    int* arrBuff;
    int size;
public:
    MyClass() {
        arrBuff = new int[0];
        size = 0;
    }
    MyClass (int initSize, int defaultValue = 0) {
        size = initSize;
        arrBuff = new int[size];
        for (int i = 0; i < size; i++)
            arrBuff[i] = defaultValue;
    }
};
```

конструктор по-умолчанию гарантирует, что каждый новый объект правильно заполнен на начальном этапе

```
MyClass intsArray(10, 42);      // 10 элементов со значением 42

MyClass emptyArray;      // вызывается конструктор по-умолчанию
```

# Деструктор класса

Специальный метод, автоматически вызываемый при удалении объекта – деструктор:

```
class MyClass {
private:
    int* arrBuff;
    int size;
public:
    MyClass (int initSize) {
        size = initSize;
        arrBuff = new int[size];
    }
    ~MyClass() {
        delete[] arrBuff;
    }
};
```

```
int main (int argc, char* argv[]) {
    MyClass intsArray(10);
    return 0;
}
```

выделяется память под буфер

удаляется объект, освобождается память

# Конструктор и создание массивов

Если в классе нет конструктора по-умолчанию:

```
class MyClass {
private:
    int* arrBuff;
public:
    MyClass (int initSize) { arrBuff = new int[initSize]; }
    ~MyClass() { delete[] arrBuff; }
};
```

default конструктора нет, но зато есть конструктор с параметрами

```
MyClass arr[10];                  //! ОШИБКА: нет конструктора по-умолчанию
MyClass* arr2 = new MyClass[10];  //! ОШИБКА: снова нет конструктора
```

```
MyClass** arr = new MyClass*[10];
for (int i = 0; i < 10; i++)
    arr[i] = new MyClass(15);

// ... полезная работа

for (int i = 0; i < 10; i++)
    delete arr[i];
delete[] arr;
```

# Синтаксический сахар

При использовании современных языков программирования часто апеллируют к понятию «синтаксический сахар». «Сахар» упрощает работу программиста, делает её «сладкой», удобной и быстрой.

В C++ «синтаксический сахар» извращённо, мазахистски «сладкий». С привкусом слёз радости от того, что это… *наконец скомпилировалось*.

С другой стороны, второй столь же гибкий в использовании язык ещё поискать. В правильных руках C++ превращается в недетский такой аттракцион…

# Использование ключевого слова const

Функция, которая ничего не меняет в объекте:

```
class MyClass {
public:
    void DummyFunc() const {
        printf ("I will not change this object never ever!");
    }
};
```

Функция, результат которой нельзя менять:

```
class MyClass {
public:
    const char* GetString() {
        return "One has not to change this string";
    }
};
```

```
MyClass foo;
char *errStr = foo.GetString();          //! ОШИБКА: нельзя без константы
```

```
MyClass foo;
const char *okStr = foo.GetString();
```

# Использование ключевого слова const

Функция, параметры которой внутри не поменяются:

```
class MyClass {
public:
    void UsefullFunc (int &a) {
        a++;
    }
    void UselessFunc (const int &a) {
        a++;                //! ОШИБКА: нельзя менять значение
    }
};
```

аналог передачи параметра по указателю, только синтаксически *более лучше* в использовании

```
int a = 42;

MyClass val;
val.UsefullFunc(a);
printf ("%d", a);       // уже 43

val.UselessFunc(a);       // переменная точно не изменится внутри
```

# Использование ключевого слова const

Полный const:

```
class MyClass {
public:

    const char* TripleComboHit (const SomeFoo &param) const {
        ...
    }

};
```

Сложно даже представить, зачем может понадобиться такой вариант написания, не так ли? Загляните в код библиотеки `stl`, будете удивлены.

# Конструктор копий

Особенный конструктор копий для клонирования объектов:

```
class MyClass {
private:
    int arr[100];
public:
    MyClass () {
        /* пустой конструктор */
    }
    MyClass (const MyClass &copyFrom) {
        for (int i = 0; i < 100; i++)
            this->arr[i] = copyFrom.arr[i];
    }
};
```

конструктор по-умолчанию нужен для удобства, пусть даже пустой

великий и ужасный конструктор копий

И когда потребуется создать объект на базе другого, то:

```
MyClass src;
...

MyClass dst (src);
```

тут он как раз и действует

# Перегрузка оператора =

Вместо клонирования объекта бывает удобнее применить копирование:

```
class MyClass {
private:
    int arr[100];
public:
    MyClass& operator= (const MyClass &copyFrom) {
        if (this == &copyFrom)
            return *this;
        for (int i = 0; i < 100; i++)
            this->arr[i] = copyFrom.arr[i];
        return *this;
    }
};
```

по адресу источника определяем, не пытаемся ли мы себя в себя скопировать

разыменование адреса

```
MyClass src;
...

MyClass dst;
dst = src;
```

# Конструктор копий и оператор =

Полный вариант выглядит, как правило, так:

```
class MyClass {
private:
    int arr[100];
    void mAssign (const MyClass &copyFrom) {
        for (int i = 0; i < 100; i++)
            this->arr[i] = copyFrom.arr[i];
    }
public:
    MyClass() {
        /* пустой конструктор */
    }
    MyClass (const MyClass &copyFrom) {
        mAssign (copyFrom);
    }
    MyClass& operator= (const MyClass &copyFrom) {
        if (this == &copyFrom) return *this;
        mAssign (copyFrom);
        return *this;
    }
};
```