Введение в классы

Где объявление, а где определение?

```
int f1();
void f2()
{

}

int iGlobalVar;
extern float fGlobalVar;

int f3()
{
    double var;
}
```

Чем равно значение переменной x?

```
int y = 5;

int z = ++y;
int x = z++;
```

Зачем нужны следующие директивы препроцессора?

```
// Файл sample.h

#ifndef SAMPLE_GUARD_HEADER
#define SAMPLE_GUARD_HEADER

// Код

#endif
```

Что будет выведено на экран?

```
#include <iostream>

int main()
{
    int var = 30;
    int* pVar = &var;

    std::cout << pVar << std::endl;
    std::cout << &pVar << std::endl;
    std::cout << *pVar << std::endl;
    std::cout << var << std::endl;
    std::cout << &var << std::endl;
    std::cout << &*pVar << std::endl;
}
```

Что будет выведено на экран?

```
#include <iostream>

int main()
{
    int *array = new int[6] { 1, 2, 3, 4, 5, 6 };
    int *iterator = array;

    iterator++;

    std::cout << iterator << std::endl;
    std::cout << std::boolalpha << (iterator > array) << std::endl;

    iterator = &array[5];

    std::cout << iterator - array + 1 << std::endl;

    delete[] array;
}
```

Порядок вычисления аргументов функции

Стандарт C++ не определяет, в каком порядке будут вычислены фактические параметры функции

```
int f();
int g();
void h(int x, int y);

int main()
{
    h(g(), f());
}
```

Какая из функций будет вызвана первой?

Второй допустимый вариант объявления функции main

```
int main(int argc, char** argv)
```

- argc – количество аргументов, переданных в программу через CLI.
- argv – сами аргументы в формате строки, причем первый из них – имя запускаемой программы

Пример

```
#include <iostream>
#include <cstring>

int main(int argc, char **argv)
{
    if (argc != 3 || strcmp(argv[1], "-n"))
        std::cout << "incorrect format";
    else
        std::cout << "Hello, " << argv[2] << std::endl;
}
```

```
> ./a.exe
> incorrect format
> ./a.exe -f Name
> incorrect format
> ./a.exe -n John
> Hello, John
```

Объявление класса

```
class Имя
{
// Описание класса
} [список объектов];
```

Класс можно объявлять в глобальной области видимости, либо внутри другого класса

Терминология

- Поля – переменные класса
- Методы – функции-члены класса
- Интерфейс – набор всех публичных методов
- Состояние – набор всех полей класса
- Экземпляр – объект класса

Модификаторы доступа

```
class Sample
{
private:
// [Описание полей и методов, видимых только внутри класса
//  в т.ч. и в public, protected методах]
public:
// [Описание открытых полей и методов, к которым можно
обращаться
//  вне класса]
protected:
// [Работает также, как и private, разница будет видна, когда
//  мы дойдем до наследования классов]
};
```

Модификатор доступа по умолчанию

```
struct SampleStruct
{
    // По умолчанию - public
};

class SampleClass
{
    // По умолчанию - private
};
```

Создание экземпляров (объектов) класса

```
class Math
{
    // Описание
};

int main()
{
    Math obj; // Создание объекта в статической памяти
    Math* obj2 = new Math; // Создание объекта в динамической памяти

    delete obj2;

    return 0;
}
```

Неявные параметры методов класса

- Во все методы класса при вызове неявно передаются поля класса

```
class Sample
{
    int x;
    int y;
public:
    void doSomething(double a)
    {

    }
};
```

a – явный параметр, x и y - неявные

Доступ к полям и членам

- Если объект был создан статически, то доступ к его полям и методам осуществляется с помощью оператора «.»
- Если объект был создан в динамической памяти, то доступ осуществляется либо с помощью разыменования и операции «.», либо с помощью оператора «->».

Пример

```
struct Sample
{
    int field;
    void doSomething() { std::cout << field; }
};

int main()
{
    Sample object;
    Sample* anotherObject = new Sample;

    object.field = 4;
    anotherObject -> field = 3;

    object.doSomething();
    anotherObject -> doSomething();
}
```

Определение методов вне класса

```
class Sample
{
public:
    void inline_method()
    {
        // Методы, объявленные внутри класса автоматически
        // становятся inline - методами
    }

    void method();
};

void Sample::method()
{
    // Определяем здесь
}
```

Пример класса – Stack | stack.h

```
#ifndef STACK_GUARD_HEADER
#define STACK_GUARD_HEADER

#include <cstdint>

class Stack
{
    int stack[100];
    uint8_t current;
    bool errorSign;
public:
    void init()
    {
        current = 0;
        errorSign = false;
    }
```

Пример класса – Stack | stack.h [продолжение]

```
    bool getErrorSign() { return errorSign; }
    int size() { return current; }
    int maxSize() { return 100; }
    bool isEmpty() { return size() == 0; }
    void push(int element);
    int pop();
};

#endif
```

Пример класса – Stack | stack.cpp

```
#include "stack.h"

void Stack::push(int element)
{
    if (current == 99)
    {
        errorSign = true;
        return;
    }

    stack[current++] = element;
}
```

Пример класса – Stack | stack.cpp [продолжение]

```
int Stack::pop()
{
    if (current == 0)
    {
        errorSign = true;
        return 0;
    }

    return stack[--current];
}
```

Пример класса – Stack | main.cpp

```cpp
#include <iostream>
#include <cstdlib>
#include "stack.h"

int main()
{
    Stack stack;

    srand(4);
    stack.init();

    while (stack.size() < maxSize())
        stack.push(rand() % 100);

    std::cout << "values: ";
    while(!stack.isEmpty())
        std::cout << stack.pop() << ' ';
}
```

Найдите ошибки в коде

```
clas Sample
{
    int b;
publick:
    int a;
};

struct SampleStruct { int x; };

int main()
{
    Sample* object;
    SampleStruct obj;

    object.b = 3;
    object.a = 4;
    obj -> x -= 41;
}
```

Предварительное объявление

```
struct Point;

struct Line
{
    Point* pt1; // Point* - неполный тип
    Point* pt2;
};

struct Point
{
    int x;
    int y;
};
```

Конструкторы

Конструктор – функция без типа, имя которой совпадает с именем класса, вызывается неявно при создании объекта класса. Класс может иметь несколько перегруженных конструкторов.

```
class Sample
{
public:
    Sample();
    Sample(const char* message);
    Sample(int sign);
    Sample(float entity);
}
```

Деструкторы

Деструктор – функция без типа, предваренная символом «~», которая совпадает с именем класса, вызывается неявно при разрушении объекта. Как правило разрушение происходит либо при вызове delete, либо при выходе объекта из области видимости.

```
class Sample
{
public:
    ~Sample();
}
```

Синтаксис вызова конструктора

```
ИмяКласса имяОбъекта;
ИмяКласса имяОбъекта();
ИмяКласса имяОбъекта(параметры);
ИмяКласса имяОбъекта = ИмяКласса();
ИмяКласса имяОбъекта = ИмяКласса(параметры);
ИмяКласса имяОбъекта {};
ИмяКласса имяОбъекта {параметры};
ИмяКласса имяОбъекта = {};
ИмяКласса имяОбъекта = {параметры};
```

Альтернативный синтаксис инициализации

- Конструкторы с одним параметром позволяют применять альтернативный синтаксис инициализации

```
class ИмяКласса
{
public:
    ИмяКласса(int имяПараметра) {}
};
// ...
ИмяКласса имяОбъекта = 13;
```

Компилятор сам развернет данный вызов в

```
ИмяКласса имяОбъекта = ИмяКласса
(13);
```

Конструктор по умолчанию

Если для класса / структуры вы не определили ни одного конструктора, то компилятор сам добавит его, такой конструктор будет эквивалентен конструктору без параметров с пустым телом.

```
class Class
{
public:
    int f1;
    int f2;
    int f3;
};

int main()
{
    Class object; // объект создается благодаря конструктору по умолчанию
}
```

Пример

```
#include <iostream>

class Sample
{
    char objectName;
public:
    Sample(char name)
    {
        objectName = name;
        std::cout << "Object with name " << objectName << " was created" << std::endl;
    }

    ~Sample()
    {
    std::cout << "Object with name " << objectName << " was destroyed" << std::endl;
    }
};
```

Пример | продолжение

```
int main()
{
    Sample A('A');
    Sample B('B');
    Sample C('C');
}
```

Результа

```
Object with name A was created
Object with name B was created
Object with name C was created
Object with name C was destroyed
Object with name B was destroyed
Object with name A was destroyed
```

Список инициализации

Список инициализации используется для задания начальных значений полям без предварительного вызова конструкторов по умолчанию для полей, являющихся объектами классов.

Синтаксис:

```
Конструктор(тип1 имя1, тип2 имя2, ...) :
поле1(имя1), поле2(имя2), ...
{
    // Тело конструктора
}
```

Пример без использования списка инициализации

```
#include <iostream>

class Point2D
{
    int x;
    int y;
public:
    Point2D()
    {
        std::cout << "Default Point2D was created" << std::endl;
        x = y = 0;
    }
```

Пример без использования списка инициализации | продолжение

```
    Point2D(int xCoord, int yCoord)
    {
        x = xCoord;
        y = yCoord;

        std::cout << "Point2D object was created" << std::endl;
    }

    void setX(int xCoord) { x = xCoord; }
    void setY(int yCoord) { y = yCoord; }
};
```

Пример без использования списка инициализации | продолжение

```
class Point3D
{
    Point2D point;
    int z;
public:
    Point3D(int x, int y, int zCoord)
    {
        point.setX(x);
        point.setY(y);
        z = zCoord;
        std::cout << "Point3D object was created" << std::endl;
    }
};
```

Пример без использования списка инициализации | продолжение

```
int main()
{
    Point3D point(-3, 4, 2);
}
```

```
Default Point2D was created
Point3D object was created
```

Пример с использованием списка инициализации

```
#include <iostream>

class Point2D
{
    int x;
    int y;
public:
    Point2D(int xCoord, int yCoord) : x(xCoord), y(yCoord)
    {
        std::cout << "Point2D object was created" << std::endl;
    }
};
```

Пример с использованием списка инициализации | продолжение

```
class Point3D {
    Point2D point;
    int z;
public:
    Point3D(int x, int y, int zCoord) : point(x, y), z(zCoord) {
        std::cout << "Point3D object was created" << std::endl;
    }
};

int main() {
    Point3D point(-3, 4, 2);
}
```

```
Point2D object was created
Point3D object was created
```

Выбор нужного имени в списке инициализации компилятором

```
class Sample
{
    int field;
public:
    Sample(int field) : field(field)
    {

    }
};
```

Компилятор из контекста сможет понять, что внешнее field – это поле класса, а внутреннее – формальный параметр конструктора