# Объектно ориентированное программирование. Классы.

М. А. Ложников

12 сентября 2019 г.
*Ревизия: 1*

**Это предварительная невыверенная версия. Читайте на свой страх и риск.**

## 1 Объектно ориентированное программирование (ООП)

Язык C++ включает в себя парадигмы объектно ориентированного программирования (ООП). Основная идея ООП заключается в выделении самостоятельных частей программы в отдельные сущности — объекты. Сам по себе объект может содержать произвольный набор данных, а также функции для работы над этими данными.

Рассмотрим пример структуры двумерного вектора `Vector2d`, доступный ещё со времён языка C.

```
1 struct Vector2d {
2   double x;
3   double y;
4 };
```

Теперь рассмотрим небольшой пример его использования:

```
1 /* Объявили переменную типа Vector2d */
2 Vector2d vec;
3
4 /* Структуры в языке C могут только хранить данные, они не могут производить
5    действий над ними. Таким образом, структуры в языке C предназначены в основном
6    для группировки данных по смыслу. */
7 vec.x = 1;
8 vec.y = 2;
```

Переменную типа `Vector2d` можно трактовать не только как набор данных, но и как самостоятельный объект, который хранит внутри себя свои координаты и может производить над этими координатами некоторые действия. Например, вектор может посчитать свою собственную длину или прибавить к самому себе другой вектор.

Работает это следующим образом. Структуры в C++ могут содержать не только поля данных, но и функции-члены, которые могут работать с этими данными.

```
1 #include <cmath>
2
```

```
struct Vector2d {
  double x;
  double y;

  /* Функция-член Length() структуры Vector2d возвращает длину вектора. */
  double Length() {
    /* Важно понимать, что поля x и y у каждой переменной типа Vector2d свои.
       Функция-член Length() работает с полями x и y именно той переменной,
       у которой эта функция вызвана. */
    return std::sqrt(x * x + y * y);
  }

  /* Функция-член Add() прибавляет вектор other к тому самому вектору, у которого
     эта функция была вызвана. */
  void Add(Vector2d other) {
    x += other.x;
    y += other.y;
  }
};
```

Структуры, которые содержат функции-члены, обычно называют классами. Кроме того, при их объявлении обычно используют ключевое слово `class` вместо ключевого слова `struct`. Функции-члены классов (структур) называют методами. Сами переменные, чей тип данных является классом, называют объектами класса. Рассмотрим пример использования структуры `Vector2d`.

```
/* Объявляем два объекта типа Vector2d и сразу инициализируем их поля посредством
   списка инициализации. Полю x присваивается первое значение, а полю y --- второе
   (в соответствии с порядком объявления полей x и y в структуре Vector2d). */
Vector2d vec1 = { 1, 2 };
Vector2d vec2 = { 3.14, 2.7 };

/* В C++ обращение к методам структуры происходит точно так же, как и к полям
   структуры. В случае если объект не является указателем, то ставится точка,
   если объект является указателем, то ставится стрелка (->). В примере ниже
   метод Length() вызывается у объекта vec1 и считает его собственную длину.
   То есть метод Length() работает с полями x и y того объекта, у которого он
   был вызван. */
std::cout << vec1.Length() << std::endl;

/* У объекта vec2 вызывается метод Add(). В соответствии с реализацией метода Add()
   это означает, что к вектору vec2 будет прибавлен вектор vec1. */
vec2.Add(vec1);
```

Теперь рассмотрим пример с указателями на класс.

```
void ExampleWithPointers(Vector2d* vec1, Vector2d* vec2) {
  /* В случае если переменная является указателям на класс (структуру), то
     к её полям обращаются при помощи стрелки. */
  std::cout << vec1->Length() << std::endl;
```

```
5
6   /* Метод Add() принимает объект типа Vector2d, а не указатель, поэтому указатель
7      vec1 нужно разыменовать. */
8   vec2->Add(*vec1);
9 }
```

# 2 Классы

Теперь при создании типов объектов будем использовать ключевое слово `class`, а не `struct`. Рассмотрим некоторые дополнительные особенности классов и структур. В дальнейшем под словом класс мы будем подразумевать либо класс (`class`) либо структуру (`struct`) при условии, что не оговорено точно.

## 2.1 Модификаторы доступа

В C++ можно задавать правила доступа к полям и методам классов. Всего есть три модификатора доступа: `public`, `protected` и `private`. Модификатор доступа `public` говорит о том, что поле или метод могут быть использованы кем угодно. Модификатор доступа `private` говорит о том, что поле или метод могут быть использованы только методами того же самого класса (с некоторыми оговорками). Здесь важно не путать класс с объектом класса. Элементы одного объекта с доступом `private` вполне могут быть изменены методами другого объекта того же самого типа, например, если в этот метод передаётся ссылка на первый объект.

В классах (`class`) по умолчанию доступ к элементам `private`, а в структурах (`struct`) — `public`. Это единственное их отличие.

```
1 #include <iostream>
2
3 class Vector2d {
4  private:
5   double x;
6   double y;
7
8  public:
9   void SetX(double xArg) {
10    x = xArg;
11  }
12
13  double GetX() {
14    return x;
15  }
16
17  void SetY(double yArg) {
18    y = yArg;
19  }
20
21  double GetY() {
22    return y;
23  }
24 };
```

```
25
26 int main() {
27   Vector2d vec;
28
29   /* Полю x объекта vec присваиваем значение 5. */
30   vec.SetX(5);
31
32   /* Выводим значение vec.x. */
33   std::cout << "vec.x = " << vec.GetX() << std::endl;
34
35   /* Ошибка компиляции. К приватным полям и методам класса могут обращаться только
36      методы этого класса (почти). */
37   // vec.x = 10;
38
39   return 0;
40 }
```

Здесь используется следующий синтаксис:

```
class ИМЯ_КЛАССА {
 МОДИФИКАТОР_1:
  поля и/или методы с типом доступа МОДИФИКАТОР_1
 МОДИФИКАТОР_2:
  поля и/или методы с типом доступа МОДИФИКАТОР_2
 ...
 МОДИФИКАТОР_N:
  поля и/или методы с типом доступа МОДИФИКАТОР_N
};
```

Каждый новый модификатор отменяет все предыдущие. Теперь рассмотрим пример изменения приватных полей чужого объекта.

```
1 #include <iostream>
2
3 class Vector2d {
4  private:
5   double x;
6   double y;
7  public:
8   /* Метод AddMeToAnotherVector() прибавляет к своему аргументу объект, у которого
9      он вызван. Аргумент передаётся по ссылке, поэтому он снаружи поменяется. */
10  void AddMeToAnotherVector(Vector2d& other) {
11    other.x += x;
12    other.y += y;
13  }
14
15  void SetX(double xArg) { x = xArg; }
16  double GetX() { return x; }
17  void SetY(double yArg) { y = yArg; }
18  double GetY() { return y; }
```

```
};

int main() {
  Vector2d vec1;
  Vector2d vec2;

  /* Задаём первый вектор. */
  vec1.SetX(1);
  vec1.SetY(2);

  /* Задаём второй вектор. */
  vec2.SetX(3);
  vec2.SetY(4);

  /* Прибавляем vec1 к вектору vec2. */
  vec1.AddMeToAnotherVector(vec2);

  /* Выведет 4 и 6. */
  std::cout << "vec2.x = " << vec2.GetX()
            << " vec2.y = " << vec2.GetY() << std::endl;
  return 0;
}
```

## 2.2 Модификатор const

Методы класса бывают константными. Такие методы не могут менять поля того объекта, у которого этот метод вызван (запрещено компилятором). Если объект является константным, то у него можно вызывать только константные методы. Сами константные методы могут вызывать только константные методы того же самого объекта.

Те методы, которые не меняют полей класса следует делать константными. Во-первых, это предохраняет программиста от ошибок, связанных со случайным изменением полей класса. Во-вторых, если объект является константным, то у него можно вызвать только константные методы.

Например, в классе `Vector2d` методы `GetX()` и `GetY()` не меняют поля класса, поэтому их следует сделать константными. Для этого необходимо в конце прототипа метода указать слово `const`.

```
class Vector2d {
 private:
  double x;
  double y;
 public:
  /* Метод не меняет полей класса, значит его нужно сделать константным. */
  double GetX() const {
    return x;
  }

  /* Метод меняет поле класса, поэтому он не является константным. */
  void SetX(double xArg) {
    x = xArg;
  }

```

```
16   double GetY() const {
17     return y;
18   }
19
20   void SetY(double yArg) {
21     y = yArg;
22   }
23 };
```

## 2.3 Использование ссылок в качестве возвращаемого значения

В случае если функция возвращает ссылку на переменную, то присвоив возвращаемому значению этой функции какое-либо значение, сама переменная также поменяется. Это можно использовать для написания более кратких и лаконичных функций чтения/записи полей класса.

```
1  #include <iostream>
2
3  class Vector2d {
4   private:
5    double x;
6    double y;
7   public:
8   /* Функция для чтения. Она может возвращать либо значение либо константную ссылку.
9      Например, следующий код предпочтителен в случаях, когда возвращаемое значение
10     долго копируется.
11     const double& X() const {
12       return x;
13     }
14  */
15   double X() const {
16     return x;
17   }
18
19   /* Фукция для записи. Возвращает ссылку и не является константной.*/
20   double& X() {
21     return x;
22   }
23
24   double Y() const {
25     return y;
26   }
27
28   double& Y() {
29     return y;
30   }
31 };
32
33 int main() {
34   Vector2d vec;
35
```

```
36   /* Полю x объекта vec присваиваем значение 5. */
37   vec.X() = 5;
38
39   /* Можно сохранить ссылку на поле y объекта vec. */
40   double& yRef = vec.Y();
41
42   /* Полю y объекта vec при помощи ссылки yRef присваиваем значение 10. */
43   yRef = 10;
44
45   /* Выводим поля x и y объекта vec. */
46   std::cout << vec.X() << " " << vec.Y() << std::endl;
47   return 0;
48 }
```

## 2.4 Конструкторы и деструкторы

Классы в C++ могут содержать специальные методы, которые вызываются при создании объекта (конструкторы) и при разрушении объекта (деструкторы).

### 2.4.1 Конструкторы

В простейшем случае объект создаётся при объявлении, а разрушается при выходе из зоны видимости (в конце блока). То есть конструкторы вызываются в порядке объявления объектов. Деструкторы же вызываются в порядке, обратном порядку объявления объектов.

Конструкторы бывают разных видов: стандартный конструктор (конструктор по-умолчанию), конструктор копирования (copy-конструктор) и специальный конструктор. Специальных конструкторов может быть несколько. Стандартный конструктор вызывается при обычном объявлении объекта (переменной). Конструктор копирования вызывается при копировании объекта (только не операцией присваивания, это нечто другое). Специальный конструктор принимает произвольные параметры, которые могут влиять на создание объекта.

Для того, чтобы определить в классе конструктор, нужно объявить метод, название которого совпадает с названием класса. Стандартный конструктор не принимает аргументов. Конструктор копирования всегда принимает константную ссылку на объект того же типа. Специальный конструктор может принимать любые аргументы. Возвращаемое значение у конструкторов не указывается, они всегда возвращают новый объект класса.

Теперь перейдём к примерам. Логично, чтобы стандартный конструктор класса `Vector2d` занулял поля класса. Специальный конструктор, например, может принимать значения, которыми нужно инициализировать поля класса.

```
1  #include <iostream>
2
3  class Vector2d {
4   private:
5    double x;
6    double y;
7   public:
8    // Стандартный конструктор. Вызывается при объявлении переменной -- объекта класса.
9    Vector2d() {
10     x = 0;
11     y = 0;
```

```
12    }
13
14    /* Конструктор копирования. */
15    Vector2d(const Vector2d& other) {
16      x = other.x;
17      y = other.y;
18    }
19
20    /* Специальный конструктор. */
21    Vector2d(double xArg, double yArg) {
22      x = xArg;
23      y = yArg;
24    }
25
26    double X() const { return x; }
27    double Y() const { return y; }
28  };
29
30  int main() {
31    /* Здесь вызывается стандартный конструктор (при объявлении переменной). */
32    Vector2d vec;
33    /* Создаём vecCopy из объекта vec при помощи конструктора копирования. */
34    Vector2d vecCopy(vec);
35    /* Наконец, специальный конструктор позволяет сразу при объявлении инициализировать
36       поля класса (только по той простой причине, что мы его таким написали). */
37    Vector2d anotherVec(10, 5);
38
39    /* Выведет 10 и 5. */
40    std::cout << "anotherVec.x = " << anotherVec.X()
41              << "anotherVec.y = " << anotherVec.Y() << std::endl;
42
43    /* Можно вызвать конструктор класса явно (совпадает с именем класса),
44       он создаст новый объект. Выведет 3 и 4.*/
45    vec = Vector2d(3, 4);
46
47    std::cout << "vec.x = " << vec.X()
48              << "vec.y = " << vec.Y() << std::endl;
49    return 0;
50  }
```

Следует помнить следующие правила:

- Компилятор по умолчанию создаёт copy-конструктор, который копирует объект поэлементно. Таким образом в случае простых классов нет необходимости переопределять конструктор копирования.
- Если в классе не задан стандартный конструктор, то считается, что он определён, но ничего не делает.
- Если в классе определён какой-либо конструктор, то стандартный конструктор не создаётся компилятором автоматически. При этом если программист явно не переопределит стандартный конструктор, то компилятор будет выдавать ошибку при попытке обычного объявления объекта

класса (которое вызывает стандартный конструктор). Таким образом, для объявления объекта класса необходимо будет использовать один из явно объявленных конструкторов класса.

### 2.4.2 Деструктор

В простых классах нет необходимости писать деструктор. Однако, это может понадобиться, например, если класс выделяет память. В таком случае, при удалении объекта эту память следует освободить. Для того, чтобы написать деструктор, необходимо написать метод, который называется `~ИМЯ_КЛАССА()` (“тильда” + имя класса, аргументов у деструктора нет).

```
class VectorNd {
 private:
  double* data;
  int size;
 public:
  /* Конструктор выделяет память. */
  VectorNd(int count) {
    data = new double[count];
    size = count;
  }

  /* Деструктор освобождает память. */
  ~VectorNd() {
    delete[] data;
  }
};
```

**Замечание 2.1** *В примере выше необходим copy-конструктор. Без него класс может некорректно работать. Например, его использование может привести к двойному освобождению указателя.*

## 2.5 Список инициализации

В конструкторах класса есть специальный механизм для инициализации полей класса, называемый списком инициализации. Списки инициализации позволяют инициализировать константные члены класса и поля, являющиеся ссылками. Список инициализации является единственным способом инициализировать такие элементы.

Кроме того, список инициализации позволяет инициализировать поля класса, которые в свою очередь являются объектами каких-либо других классов. В противном случае (если не использовать список инициализации) для таких полей будет вызван стандартный конструктор (его может и не быть, тогда будет ошибка компиляции).

Список инициализации является предпочтительным способом инициализации полей класса.

Для того, чтобы добавить список инициализации в конструктор, нужно поставить двоеточие после прототипа, затем перечислить поля класса и соответствующие им значения через запятую. Значение соответствующего поля указывается в круглых скобках сразу после него. Если поле является объектом некоторого класса, то в скобках можно указать аргументы конструктора класса этого поля.

```
class Base {
 private:
```

```
  FieldType1 field1;
  FieldType2 field2;
  ...
  FieldTypeN fieldN;
 public:
  /* Какой-либо конструктор, необязательно стандартный. Полю field1 присвоится
     значение value1, полю field2 --- value2 и т.д. Если FieldTypeK является
     классом, то valueK является списком аргументов некоторого конструктора класса
     FieldTypeK. */
  Base() :
    field1(value1),
    field2(value2),
    ...
    fieldN(valueN),
  {
    // Тело конструктора
  }
};
```

Необходимо помнить следующие детали:

- Поля класса в списке инициализации необходимо инициализировать строго в том порядке, в котором они объявлены в классе. Если перепутать порядок, то элементы всё равно будут инициализироваться в том порядке, в котором они были объявлены.
- Необязательно перечислять все поля в списке инициализации, можно перечислить только необходимые. Для остальных полей будет вызван стандартный конструктор, если он есть. Однако, в том случае если стандартный конструктор поля удалён, будет ошибка компиляции.

Рассмотрим класс строк.

```
class String {
 private:
  char* data;
  int size;
 public:
  /* В C++11 следует использовать nullptr вместо NULL. */
  String() :
    data(nullptr),
    size(0)
  { }

  String(const String& other) :
    data(nullptr),
    size(other.size)
  {
    if (size > 0) {
      data = new char[size + 1];
    }

    /* Копируем строки посимвольно. */
```

```
21      for (int i = 0; i < size; i++)
22        data[i] = other.data[i];
23
24      /* Прописываем символ конца строки. */
25      data[size] = 0;
26    }
27
28    ~String() {
29      if (data)
30        delete[] data;
31    }
32  };
```

## 2.6 Порядок инициализации и удаления полей класса

Поля класса инициализируются в порядке объявления, а удаляются в порядке, обратном порядку объявления. Проиллюстрируем это на примере.

```
1  #include <iostream>
2  #include <string>
3
4  class Logger {
5   private:
6    std::string name;
7   public:
8    /* Конструктор принимает имя объекта и выводит информацию о том, что он создан. */
9    Logger(const std::string& logerName) {
10     name = logerName;
11     std::cout << "Object '" << name << "' created!" << std::endl;
12   }
13
14   /* Деструктор выводит информацию о том, что объект удалён. */
15   ~Logger() {
16     std::cout << "Object '" << name << "' destroyed!" << std::endl;
17   }
18 };
19
20 class Base {
21  private:
22   Logger field1;
23   Logger field2;
24   std::string name;
25
26  public:
27   /* Имя аргумента метода может совпадать с именем поля класса. В таком случае
28      будет виден только аргумент функции (обратиться к соответствующему полю класа
29      тоже можно, но об этом на следующих занятиях). В данном примере компилятор
30      понимает, что поле name класса инициализируется аргументом name конструктора.
31      Операция сложения для типа std::string создаёт новую строку, являющуюся
32      объединением строк-аргументов операции. */
```

```
33   Base(const std::string& name) :
34     field1(name + ".field1"),
35     field2(name + ".field2"),
36     name(name) // Здесь поле name инициализируется аргументом name
37   {
38     /* А вот здесь name ассоциируется с аргументом конструктора, а не с полем класса.
39       В конструкторах field1 и field2 тоже используется аргумент name, а не поле
40       класса name. */
41
42     std::cout << "Base class '" << name << "' created!" << std::endl;
43   }
44
45   ~Base() {
46     std::cout << "Base class '" << name << "' destroyed!" << std::endl;
47   }
48 };
49
50 int main() {
51   Base base1("base1");
52   Base base2("base2");
53
54   return 0;
55 }
```

После запуска данная программа выведет

```
Object 'base1.field1' created!
Object 'base1.field2' created!
Base class 'base1' created!
Object 'base2.field1' created!
Object 'base2.field2' created!
Base class 'base2' created!
Base class 'base2' destroyed!
Object 'base2.field2' destroyed!
Object 'base2.field1' destroyed!
Base class 'base1' destroyed!
Object 'base1.field2' destroyed!
Object 'base1.field1' destroyed!
```

Из вывода программы можно извлечь следующие правила:

- Объекты создаются в порядке объявления и уничтожаются в порядке, обратном порядку объявления.
- Поля класса инициализируются в порядке своего объявления и уничтожаются в порядке, обратном порядку объявления.
- Поля класса инициализируются перед выполнением тела конструктора класса.
- Поля класса уничтожаются после выполнения тела деструктора класса.

## 2.7 Разделение объявления методов класса и реализации

Реализацию методов класса необязательно писать сразу при объявлении метода. Рассмотрим пример.

Listing 1: Файл vector2d.hpp

```
#ifndef VECTOR2D
#define VECTOR2D

#include <cmath>

class Vector2d {
 private:
  double x;
  double y;
 public:
  /* Реализацию методов ниже напишем в файле исходного кода. */
  Vector2d();
  Vector2d(double xArg, double yArg);

  double Length() const;
  void Add(const Vector2d& other);

  /* А вот эти методы напишем сразу вместе с реализацией. */
  double X() const { return x; }
  double& X() { return x; }
  double Y() const { return y; }
  double& Y() { return y; }
};

#endif // VECTOR2D
```

Listing 2: Файл vector2d.cpp

```
#include <cmath>
// Для использования класса необходимо подключить заголовочный файл с его объявлением.
#include "vector2d.hpp"

/* Для того, чтобы сказать компилятору о том, что мы пишем реализацию метода
   класса Vector2d, а не отдельную функцию, необходимо написать Vector2d:: перед
   названием каждого метода. */
Vector2d::Vector2d() :
  x(0),
  y(0)
{ }

Vector2d::Vector2d(double xArg, double yArg) :
  x(xArg),
  y(yArg)
{ }
```

```

double Vector2d::Length() const {
  return std::sqrt(x * x + y * y);
}

void Vector2d::Add(const Vector2d& other) {
  x += other.x;
  y += other.y;
}
```

Listing 3: Файл main.cpp

```
#include <iostream>
#include "vector2d.hpp"

int main() {
  Vector2d vec(3, 4);
  Vector2d anotherVec(10, 5);

  vec.Add(anotherVec);

  /* Выведет 13 и 9. */
  std::cout << "vec.x = " << vec.X()
            << "vec.y = " << vec.Y() << std::endl;
  return 0;
}
```

## 2.8 Динамическое создание и удаление объектов классов

Объекты классов необязательно создавать при помощи объявления переменных. Можно использовать специальные операторы `new`/`delete`, которые создают и удаляют объект соответственно. Оператор `new` создаёт новый объект и возвращает указатель на него. Оператор `delete` удаляет объект (принимает указатель на объект).

**Не путать с операторами** `new[]`/`delete[]`, которые используются для выделения памяти под массив и освобождению массива. Эти операторы **не совместимы**. Дело в том, что оператор `new[]` выделяет память под некоторую дополнительную информацию, используемую для внутренних нужд, например, под размер массива, а оператор `new` этого не делает, поскольку выделяет память только под один объект. Таким образом, память, выделенная оператором `new[]` должна быть освобождена оператором `delete[]`, а память, выделенная оператором `new` — освобождена оператором `delete`.

Синтаксис:

```
// Объявляем указатель на класс. В данный момент объектов не создаётся.
ClassType* obj;

/* Создаём объект при помощи конструктора класса ClassType, передав в конструктор
   аргументы arg1, arg2,..., argN. */
obj = new ClassType(arg1, arg2,..., argN);
```

```

// Уничтожаем объект.
delete obj;
```

Рассмотрим пример динамического создания объектов типа `Vector2d`.

Listing 4: Файл main.cpp

```
#include <iostream>
#include "vector2d.hpp"

int main() {
  /* В этом случае объект не создаётся, конструкторов не вызывается.
     Дело в том, что мы объявили указатель на объект, а не сам объект. */
  Vector2d* vec;

  /* Создаём объект при помощи специального конструктора. */
  vec = new Vector2d(3, 4);

  /* Для обращения к элементам объекта используется стрелка, поскольку работаем
     с указателями. Выведет 5. */
  std::cout << "Length = " << vec->Length() << std::endl;

  std::cout << "vec->x = " << vec->X()
            << " vec->y = " << vec->Y() << std::endl;

  /* Удаляем объект. */
  delete vec;

  return 0;
}
```

# Список литературы

[1] *Бьерн Страуструп* Язык программирования C++. М:Бином. 2011.

[2] `https://en.cppreference.com`