
Введение в ООП на C++: классы, объекты, поля, методы, операторы

А. Г. Fenster, `fenster@fenster.name`

22 сентября 2009 г.

Конспект семинара №1 по объектно-ориентированному программированию. Пожалуйста, не относитесь к этому как к руководству по C++ или ООП: здесь изложен только самый минимум информации по теме; тем не менее, этого минимума должно хватить для выполнения первого задания.

О найденных ошибках и неточностях, пожалуйста, пишите мне по адресу `fenster@fenster.name`. Также буду рад получить любые комментарии по поводу этого текста.

1 Основные понятия

Вспомним понятие *структуры* (`struct`) в языке C. Структурой мы называли составной тип данных: каждая переменная такого типа содержала в себе несколько переменных других типов. Пример:

```
struct Rational // рациональное число
{
    int numerator;           // числитель
    unsigned int denominator; // знаменатель
};
```

К *полям* структуры `numerator` и `denominator` можно было обращаться при помощи *селектора* `.` (точка):

```
struct Rational a;
a.numerator = 1;
a.denominator = 2;
```

Если же у нас есть указатель на структуру

```
struct Rational *p = &a;
```

то вместо громоздкой записи

```
(*p).numerator = 1;
```

можно использовать удобное сокращение при помощи селектора ->:

```
p->numerator = 1;
```

Важно, что в языке C структура хранила данные и ничего «не знала» о том, как эти данные используются. Для работы с ними можно было, например, создавать функции:

```
void reduce(struct Rational *a)
{ // сокращение дроби
  // функция изменяет аргумент, поэтому передаём адрес
  int d = gcd(abs(a->numerator), a->denominator); // НОД
  a->numerator /= d;
  a->denominator /= d;
}

struct Rational mul(struct Rational a, struct Rational b)
{ // умножение
  struct Rational result;
  result.numerator = a.numerator * b.numerator;
  result.denominator = a.denominator * b.denominator;
  reduce(&result); // сокращаем дробь
  return result;
}
```

Перечислим лишь некоторые из недостатков такого подхода:

1. Не получится записывать арифметические выражения в привычном виде $c = a * b$; вместо этого придётся писать `c = mul(a, b)`;
2. Необходимо всегда помнить, изменяет ли данная функция свой аргумент, и передавать параметром либо структуру, либо её адрес;

3. Отсутствует защита от некорректных действий: скажем, любой может присвоить `a.denominator = 0`; и нет возможности отследить это;
4. Создание рационального числа требует нескольких действий: объявить переменную-структуру, присвоить числитель и знаменатель;
5. Ну и наконец, хорошо, когда структура простая, как в этом примере: функция может смело вернуть структуру (`return result`) и не волноваться о выделении и освобождении памяти. Сложнее будет в том случае, если структура хранит указатель на некоторые данные в памяти.

В языке C++ эти проблемы предлагается решать следующим образом. В самом начале отметим два новых момента:

1. В C++ имя структуры само по себе является именем типа, поэтому для краткости при объявлении переменной вместо

```
struct Rational a;
```

можно писать просто

```
Rational a;
```

2. Для вывода на экран мы будем использовать не `printf`, а более удобный способ:

```
#include <iostream>
...
std::cout << "Значение переменной a равно " << a << "\n";
```

Аналогично делается ввод:

```
int a;
std::cin >> a;
```

Смысл этих «заклинаний» станет понятен позже.

1.1 Методы

Структура может содержать как поля (данные), так и *методы*: функции, связанные с этими данными. Например, вот как можно реализовать метод `reduce`:

```
struct Rational
{
    int numerator;
    unsigned int denominator;
    void reduce();
};

void Rational::reduce()
{
    int d = gcd(abs(numerator), denominator);
    numerator /= d;
    denominator /= d;
}
```

В этом примере метод `reduce` является частью структуры `Rational`, а значит, имеет доступ ко всем её полям. Вызвать метод весьма просто:

```
Rational a;
a.numerator = 2;
a.denominator = 4;
a.reduce();
```

либо аналогично для указателя на структуру:

```
Rational *p = &a;
p->reduce();
```

Иногда удобно вставлять реализацию метода прямо в описание структуры:

```
struct Rational
{
    int numerator;
    unsigned int denominator;
    void reduce()
    {
        int d = gcd(abs(numerator), denominator);
        numerator /= d;
        denominator /= d;
    }
};
```

Для краткости в дальнейшем будет в основном использован именно такой способ записи. Часто рекомендуют обращаться к полям «собственной» структуры при помощи указателя `this`, который всегда указывает на ту структуру, метод которой вызван. Это помогает избавиться от возможных ошибок, явно указывая, что в данной ситуации обращение идёт к полю, а не к какой-либо переменной:

```
struct Rational
{
    int numerator;
    unsigned int denominator;
    void reduce()
    {
        int d = gcd(abs(this->numerator), this->denominator);
        this->numerator /= d;
        this->denominator /= d;
    }
};
```

1.2 Приватность

Принято реализовывать структуры так, чтобы любой доступ к полям происходил только через методы. Это позволяет контролировать значения, которые могут быть присвоены полям (например, запретить присваивать 0 знаменателю).

У такой техники есть поддержка в языке C++: любое поле (и метод) можно занести в особую секцию `private`, после чего он будет доступен только для других методов этой же структуры.¹ Попытка обратиться к приватному полю или методу структуры из другого фрагмента кода вызовет ошибку компиляции. Антоним ключевого слова `private` — слово `public`, разрешающее доступ отовсюду.

```
struct Rational
{
private:
    int numerator;
    unsigned int denominator;
public:
    int getNumerator() const
    {
        return numerator;
    }
    int getDenominator() const
```

¹И *другей* этой структуры — об этом позже.

```

    {
        return denominator;
    }
    void setValue(int n, int d)
    {
        if (d == 0)
        {
            сообщить об ошибке!
            return;
        }
        if (d < 0)
        {
            n = -n;
            d = -d;
        }
        numerator = n;
        denominator = d;
    }
};

```

Слово `const` после заголовка метода показывает компилятору, что метод не изменяет значения полей структуры.

Использовать эти методы можно так:

```

Rational r;
r.setValue(1, 2);
std::cout << r.getNumerator() << "/" <<
            r.getDenominator() << "\n";

```

По умолчанию все поля структуры являются публичными (в основном для сохранения совместимости с C). Но в C++ удобнее делать поля приватными по умолчанию. Для этого нужно вместо структур объявлять *классы*. Два следующих фрагмента кода по сути аналогичны:

<code>struct Rational</code>		<code>class Rational</code>
{		{
private:		

В дальнейшем мы в основном будем пользоваться именно классами, а не структурами. Экземпляр класса (переменную, имеющую тип `class ...`) мы будем называть *объектом*.

1.3 Ссылки

Понятие указателя в языке C++ дополнено новым понятием *ссылки* (reference):

```
int a;  
int &b = a;
```

В этом примере `b` и `a` — синонимы, `b` — другое имя той же самой переменной. Ссылки позволяют передавать параметры в функции «на изменение» без использования указателей:

```
void does_not_work(int a)  
{  
    a++; // изменяется локальная копия  
}  
void increment(int &a)  
{  
    a++; // изменяется переданная переменная  
}
```

Также ссылки можно использовать для того, чтобы ставить вызов функции в левую часть оператора присваивания. Следующий (несколько надуманный) пример демонстрирует это:

```
int a; // переменная  
int& f()  
{  
    return a;  
}
```

Определив такую функцию, можно написать присваивание `f() = 7`; которое присвоит значение 7 переменной `a`.

2 Конструктор, конструктор копирования и деструктор

2.1 Конструктор

Конструктором называется метод, который вызывается при создании нового объекта. Конструктор — это метод, имя которого совпадает с именем класса; тип возвращаемого значения не указывается.

Один класс может содержать несколько конструкторов с различным (по типам) набором параметров или без параметров; при создании объекта автоматически выбирается подходящий конструктор.

Пример ниже содержит одну новую инструкцию `throw`, которая позволяет прервать выполнение функции (и программы) в случае ошибки, сгенерировав некоторое *исключение*; подробнее эта тема будет рассмотрена позже.

```
class Rational
{
    int numerator, denominator;
public:
    void reduce();
    Rational(int n, int d)
    { // создание числа по числителю и знаменателю
        if (d == 0) throw "Zero denominator";
        if (d < 0)
        {
            n = -n;
            d = -d;
        }
        numerator = n;
        denominator = d;
        reduce();
    }
    Rational(int n)
    { // создание из целого числа
        numerator = n;
        denominator = 1;
    }
    Rational()
    { // конструктор по умолчанию (без параметров)
        numerator = 0;
        denominator = 1;
    }
};
```

Использование этих конструкторов:

```
Rational a(3, 5); // использован Rational(int n, int d)
Rational b(3);   // использован Rational(int n)
Rational c;     // использован Rational()
```

В более сложных классах конструкторы могут, помимо инициализации полей, выделять память для хранения чего-либо, открывать файлы и т. п.

Инициализацию полей можно вынести из тела конструктора. Скажем, конструктор по умолчанию из предыдущего примера может быть записан так:

```
class Rational
{
    ...
public:
    Rational(): numerator(0), denominator(1)
    {
    }
};
```

2.2 Конструктор копирования

В случае, когда необходимо физически скопировать объект, вызывается особый конструктор, принимающий параметром ссылку на объект того же типа. Такой конструктор называется конструктором копирования.

```
class Rational
{
    ...
public:
    Rational(const Rational& orig)
    { // конструктор копирования
        numerator = orig.numerator;
        denominator = orig.denominator;
    }
};
```

Конструктор копирования вызывается при инициализации объекта объектом этого же типа:

```
Rational r = r1; // вызов конструктора копирования
```

а также при вызове функции с передачей параметра по значению:

```
void f(Rational r)
{
}
```

```
Rational r;
f(r); // вызов конструктора копирования
```

Синтаксис языка не запрещает «обмануть» конструктор копирования, инициализировав объект самим собой:

```
Rational r = r;
```

В этом случае объект останется неинициализированным. Такую (весьма редкую) ситуацию можно «поймать» в конструкторе копирования проверкой `if (&orig == this)` (конструкция `&a` возвращает адрес переменной `a`).

2.3 Деструктор

Деструктор — метод, который вызывается при удалении объекта из памяти (например, при достижении конца блока, в котором была введена переменная). Деструктор имеет имя, совпадающее с именем класса, но с префиксом `~`. Деструктор не имеет параметров.

```
class Rational
{
    ...
public:
    ~Rational()
    {
    }
};
```

Для простых классов (как `Rational`) деструктор объявлять не обязательно. В более сложных классах он может, например, освободить ранее выделенную память.

3 Операторы

3.1 Арифметические операторы

Для любого созданного пользователем класса можно определить всевозможные стандартные операторы (`+`, `*` и многие другие). Определим

для нашего класса `Rational` оператор умножения. Он принимает один параметр: дробь, на которую производится умножение данного объекта.

```
class Rational
{
    ...
public:
    Rational operator*(const Rational &second) const
    {
        Rational result(enumerator * second.enumerator,
                        denominator * second.denominator);
        return result;
    }
};
```

Параметр оператора передаётся по ссылке, чтобы не производилось лишнее копирование (и лишний вызов конструктора копирования).

После такого определения будет возможно записать выражение в привычном виде:

```
Rational a(1, 2), b(1, 3), c;
c = a * b; // вызов a.operator*(b)
```

Вполне возможно сделать оператор умножения на переменную другого типа (например, на целое число). То есть, можно определить несколько одинаковых операторов, но с разными типами параметров.

3.2 Оператор присваивания

Оператор присваивания тоже может (а для сложных классов — и должен) быть переопределён. Чтобы можно было писать цепочку присваиваний ($a = b = c$), он должен возвращать присвоенное значение в виде ссылки на класс. Пример:

```
class Rational
{
    ...
public:
    Rational& operator=(const Rational &orig)
    {
```

```
        numerator = orig.numerator;
        denominator = orig.denominator;
        return *this;
    }
};
```

Логика оператора присваивания в целом должна повторять логику конструктора копирования. Важно понимать, в каких случаях используется оператор присваивания, а в каких конструктор копирования:

```
Rational a = b; // конструктор копирования
a = b; // оператор присваивания a.operator=(b)
```

В сложных классах, хранящих указатели на выделенную память, в операторе присваивания необходимо проверять, что `orig` не совпадает с текущим объектом, другими словами, что не выполняется присваивания вида `a = a`. Для этого можно сравнить `&orig` и `this`.

3.3 Оператор приведения типа

При помощи специальных операторов можно разрешить использовать класс в качестве любого другого типа. Например, реализуем приведение нашего рационального числа к типу `double`:

```
class Rational
{
    ...
public:
    operator double() const
    {
        return (double)numerator / denominator;
    }
};
```

Это позволит использовать класс `Rational` в тех выражениях, где допустим тип `double`:

```
Rational a(1, 4);
double d = sqrt(a);
```

4 Выделение и освобождение памяти

Вместо функций `malloc` и `free`, при помощи которых выделялась и освобождалась память в C, в C++ нужно использовать операторы `new` и `delete`. Для базовых типов их использование почти не отличается от старого варианта:

<code>int *p = malloc(sizeof(int));</code>		<code>int *p = new int;</code>
<code>free(p);</code>		<code>delete p;</code>
<code>int *q = malloc(5 * sizeof(int));</code>		<code>int *q = new int[5];</code>
<code>free(q);</code>		<code>delete[] q;</code>

При работе с классами основное отличие `new` и `delete` от `malloc` и `free` в том, что `new` и `delete` вызывают, соответственно, конструктор и деструктор для указанного типа. Например, можно создать указатель на объект `Rational`:

```
Rational *p = new Rational(1, 2); // аргументы конструктора
```

Созданный таким образом объект не будет удалён автоматически, так что необходимо будет вызвать деструктор, когда объект перестанет быть нужным:

```
delete p;
```