Примеры определения операторов

А. Г. Фенстер, http://info.fenster.name

16 ноября 2009 г.

Конспект семинара №2 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.

В первой части было показано, как можно определить для своего класса арифметические бинарные операторы (`+`, `-` и подобные), оператор присваивания и операторы приведения типа. Рассмотрим ещё несколько видов операторов, которые бывает нужно определить для своего класса.

1 Операторы сравнения

Операторы сравнения `==` и `!=` необходимо определять как возвращающие тип `bool` (истину или ложь). Обычно определение этих операторов не вызывает проблем, поэтому мы не будем останавливаться на них подробно.

2 Операторы += и подобные

Эти операторы объединяют функциональность оператора присваивания и арифметических операторов. Как и оператор присваивания, операторы `+=`, `-=`, ... должны возвращать ссылку «на себя». Пример для класса `Rational`:

```
class Rational
{
    ...
public:
    Rational& operator*=(const Rational &other)
    {
        numerator *= other.numerator;
        denominator *= other.denominator;
        reduce();
        return *this;
    }
};
```

Главным отличием операторов `+=`, ... от операторов `+`, ... является то, что первые изменяют содержимое текущего объекта, тогда как вторые являются константными для текущего объекта. Сравните:

```
c = a + b;  // c = a.operator+(b); -- a не изменяется
a += b;     // a.operator+=(b); -- a изменилось
c = a += b; // c = a.operator+=(b); -- a изменилось,
            // в c присвоено новое значение a
```

В третьей строке примера как раз используется возвращаемое значение оператора `+=`.

3 Операторы ++ и --

Как известно, в языке C существуют префиксные и постфиксные версии операторов `++` и `--`: и та, и другая изменяет (увеличивает или уменьшает) значение переменной, для которой оператор вызван, но префиксная версия (`++a`) возвращает уже новое значение переменной, тогда как постфиксная (`a++`) возвращает ещё старое значение.

Такое поведение операторов `++` и `--` возможно реализовать и в C++. Реализуем для класса `Rational` префиксную и постфиксную версии оператора `++`, увеличивающие значение объекта на единицу. Префиксный оператор `++` определяется весьма интуитивно:

```
class Rational
{
```

```
    ...
public:
    Rational& operator++()
    {
        numerator += denominator;
        return *this;
    }
};
```

С постфиксным оператором ситуация чуть сложнее: чтобы отличить его от префиксного, ему передаётся (но никак не используется) один параметр типа `int`.

```
class Rational
{
    ...
public:
    Rational operator++(int)
    {
        Rational saved = *this;
        numerator += denominator;
        return saved;
    }
};
```

Определение префиксного и постфиксного оператора `--` выглядит аналогично.

4 Оператор []

Для придания своему классу свойств массива можно определить для него оператор `[]`, имеющий один аргумент: «индекс массива», который, вообще говоря, может иметь любой тип (не только натуральное число).

Рассмотрим, например, простейшую реализацию вектора в $\mathbb{R}^2$. Класс `Vector` будет хранить два числа — координаты `x` и `y`, но также предоставит возможность обращения к ним по индексам 0 и 1 соответственно. Конструкторы класса и прочие операторы в этом примере пропустим.

```
class Vector
{
private:
    float x, y;
public:
    ...
    float operator[](const int index) const
    {
        switch (index)
        {
            case 0: return x;
            case 1: return y;
            default: throw "incorrect index";
        }
    }
};
```

Такое определение позволит обращаться к объекту класса `Vector` как к массиву:

```
Vector v;
...
a = v[0]; // a = v.operator[](0);
```

К сожалению, определённый таким образом оператор нельзя использовать для изменения элементов нашего «массива». Чтобы это стало возможным, необходимо также определить оператор `[]`, возвращающий *ссылку* на элемент массива. Он уже будет не константным. Обратите внимание: отличие от предыдущего примера — только в типе возвращаемого значения и отсутствии слова `const`:

```
class Vector
{
private:
    float x, y;
public:
    ...
    float& operator[](const int index)
    {
```

```
        switch (index)
        {
            case 0: return x;
            case 1: return y;
            default: throw "incorrect index";
        }
    }
};
```

Такой оператор уже позволит присваивать элементу массива:

```
v[0] = 7; // v.operator[](0) = 7;
```

Зачем определять две версии одного и того же оператора, почему недостаточно определить только версию со ссылкой? Дело в том, что если объект, с которым мы работаем, является константным, необходимо иметь возможность обратиться к нему по индексу при помощи константного метода: вызывать неконстантные методы константного объекта запрещено.

5 Оператор ()

Аналогично оператору [] можно определить и оператор (). Это даёт поистине неограниченные возможности: с вашим объектом можно будет работать как с функцией. Несколько искусственный пример:

```
class Succ
{
public:
    int operator()(int z)
    {
        return (z + 1);
    }
};

Succ succ;
a = succ(6); // a = succ.operator()(6);
```

Такие операторы используются при программировании в функциональном стиле, когда удобно иметь возможность смотреть на объект как на функцию. Например, можно определить класс,

который в операторе `()` будет сравнивать два аргумента по какому-либо правилу, и затем передавать экземпляр этого класса в функции поиска и сортировки аналогично тому, как в функцию `qsort` в языке C передаётся адрес функции сравнения, но более безопасно (т. к. отсутствует работа с указателями).

6 Прочие операторы

Язык C++ позволяет определить для своего класса (обычно используется термин *перегрузить* множество операторов. Перегрузка некоторых из них, например, оператора `new`, позволяет управлять выделением памяти под объекты данного класса. Перегрузка оператора `->` позволяет реализовать концепцию «умных указателей» (smart pointers). Рассмотрение этих вопросов выходит за рамки нашего короткого курса.