Перегрузка операций

М. А. Ложников

26 сентября 2019 г.
Ревизия: 1

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

1 Введение

Язык C++ позволяет переопределить стандартные операции как для встроенных типов данных, так и для пользовательский типов. Список операций, которые можно переопределить, довольно обширен, в том числе он включает арифметические, битовые и логические операции, но не ограничивается ими. Рассмотрим несколько примеров.

1.1 Пример 1. Перегрузка арифметических операций

Предположим дан класс комплексных чисел `Complex`. Хотелось бы, чтобы объекты этого класса можно было складывать точно так же, как и обычные числа.

```
Complex c1, c2, c3;

/* Операция сложения не определена для объектов произвольных типов по-умолчанию.
   Для того, чтобы компилятор знал как складывать числа типа Complex, необходимо
   определить операцию сложения для них. */
c3 = c1 + c2;
```

1.2 Пример 2. Перегрузка операции индексации

Предположим, мы пишем класс динамического массива, назовём его `Vector`, для него будут полезны операции индексации.

```
Vector v = ...;

/* Операция [] тоже не определена для произвольных типов данных. Для того, чтобы она
   появилась в типе вектор, её нужно переопределить. */
v[1] = ...;
```

2 Операции приведения типов

Эти операции используются для конвертации типов.

2.1 Конструктор как неявная операция приведения типа

Объявление конструктора, принимающего один параметр некоторого типа данных может служить в качестве неявной операции приведения типа. Рассмотрим пример.

```cpp
#include <iostream>
#include <string>

class Complex {
 private:
  double re;
  double im;
 public:
  Complex() :
    re(0),
    im(0)
  { }

  /* Конструктор принимает только один параметр. В таком случае он задаёт правило
     конвертации типа аргумента (double) в тип данных класса (Complex). */
  Complex(double re) :
    re(re),
    im(0)
  { }

  /* Неявное приведение типа при помощи конструктора можно запретить при помощи
     ключевого слова explicit. В данном случае запрещается неявная конвертация из типа
     std::string в тип Complex. Функция std::stod() из заголовочного файла <string>
     конвертирует строку std::string в тип double. */
  explicit Complex(const std::string& re) :
    re(std::stod(re)),
    im(0)
  { }


  Complex(double re, double im) :
    re(re),
    im(im)
  { }

  double Re() const { return re; }
  double& Re() { return re; }
  double Im() const { return im; }
  double& Im() { return im; }
};

void PrintComplex(const Complex& number) {
  std::cout << "(" << number.Re() << ", " << number.Im() << ")" << std::endl;
}

int main() {
```

```
47   Complex c;
48
49   /* Здесь тип double неявно конвертируется в тип данных Complex при помощи написанного
50      нами конструктора. */
51   c = 5.3;
52
53   std::cout << "(" << c.Re() << ", " << c.Im() << ")" << std::endl;
54
55   /* При вызове функции аргумент конвертируется в тип Complex. */
56   PrintComplex(3.14);
57
58   /* Неявная операция приведения типа std::string в тип Complex запрещена при помощи
59      ключевого слова explicit. */
60   // c = std::string("125.3"); // Ошибка компиляции из-за explicit.
61
62   /* А вот так можно. Явно вызываем конструктор. */
63   c = Complex(std::string("125.3"));
64
65   // И вот так тоже можно. Тип std::string может быть неявно создан из типа const char*.
66   c = Complex("125.3");
67
68   std::cout << "(" << c.Re() << ", " << c.Im() << ")" << std::endl;
69   return 0;
70 }
```

2.1.1 Аргументы, заданные по-умолчанию

В языке C++ для функций и методов последние аргументы могут быть заданы по-умолчанию. То есть при объявлении (и вызове) функции сначала перечисляются обязательные аргументы, а потом необязательные с заданными по-умолчанию значениями.

```
1  #include <iostream>
2
3  class Complex {
4   private:
5    double re;
6    double im;
7   public:
8    /* Этот конструктор служит и в качестве конструктора по-умолчанию и в качестве
9       операции приведения типа, и в качестве специального конструктора. При вызове такого
10      конструктора последние необязательные аргументы можно не задавать, однако, их
11      нельзя пропускать. То есть нельзя задать только аргумент im. */
12   Complex(double re = 0, double im = 0) :
13     re(re),
14     im(im)
15   { }
16
17   double Re() const { return re; }
18   double& Re() { return re; }
19   double Im() const { return im; }
```

```
20   double& Im() { return im; }
21 };
22
23 void PrintComplex(const Complex& number) {
24   std::cout << "(" << number.Re() << ", " << number.Im() << ")" << std::endl;
25 }
26
27 int main() {
28   /* Все способы создания комплексного числа работают. */
29   Complex c;
30   Complex c2(1, 5);
31   Complex c3(4);
32
33   /* Здесь тип double неявно конвертируется в тип данных Complex при помощи написанного
34      нами конструктора. */
35   c = 5.3;
36
37   /* При вызове функции аргумент конвертируется в тип Complex. */
38   PrintComplex(3.14);
39
40   std::cout << "(" << c.Re() << ", " << c.Im() << ")" << std::endl;
41   std::cout << "(" << c2.Re() << ", " << c2.Im() << ")" << std::endl;
42   std::cout << "(" << c3.Re() << ", " << c3.Im() << ")" << std::endl;
43   return 0;
44 }
```

2.2 Конвертация объекта класса в объект другого типа

Для того, чтобы создать правило приведения объекта класса к объекту некоторого другого типа, необходимо создать в классе специальный метод.

```
1 #include <iostream>
2
3 class Complex {
4  private:
5   double re;
6   double im;
7  public:
8   /* Этот конструктор служит и в качестве конструктора по-умолчанию и в качестве
9      операции приведения типа, и в качестве специального конструктора. */
10  Complex(double re = 0, double im = 0) :
11    re(re),
12    im(im)
13  { }
14
15  /* Операция конвертации в тип данных double. Тип может быть любой, он указывается
16     после слова operator. Тип возвращаемого значения указывать не нужно, он совпадает
17     с указанным после слова operator. */
18  operator double() const {
19    /* Просто возвращаем действительную часть числа. */
```

```
20      return re;
21    }
22  };
23
24  int main() {
25    Complex c(1, 3);
26
27    /* Неявная конвертация типа  Complex в тип double. */
28    double re = c;
29
30    std::cout << "Re: " << re << std::endl;
31    return 0;
32  }
```

3 Реализация перегрузки операций

Для того, чтобы перегрузить операцию для некоторого типа данных существует два различных способа:

1. Если указанный тип данных является классом, то можно определить в нём специальный метод, который начинается с ключевого слова `operator`, а затем идёт тип операции. Тогда объект класса, к которому эта операция применена неявно будет одним из аргументов операции.

2. Вместо метода класса можно написать отдельную функцию, которая начинается со слова `operator` и заканчивается типом операции. В таком случае все параметры операции нужно явно указывать в списке аргументов соответствующей функции.

Эти способы не всегда являются взаимозаменяемыми. Некоторые операции могут быть перегружены только одним из предложенных способов.

При перегрузке операций важно помнить следующее:

1. само действие, которое делает перегруженная операция целиком и полностью определяется программистом;

2. возвращаемое значение также целиком и полностью определяется программистом;

3. возможность операции менять свои аргументы также зависит от программиста.

В следующих разделах приведены примеры операций, написанных таким образом, чтобы они делали то, что они должны делать по своему смыслу, однако их реализация не обязана быть именно такой.

4 Перегрузка арифметических операций

Реализуем перегрузку арифметических операций на примере класса комплексных чисел. Язык C++ позволяет перегружать следующие арифметические операции: `+`, `-`, `-` (унарная), `*`, `/`, `%`, `+=`, `-=`, `*=`, `/=`, `%=`.

4.1 Первый способ. Добавление метода в класс

Рассмотрим перегрузку операций +, -, +=, -= и унарной операции - для класса комплексных чисел при помощи добавления в класс специального метода.

```
#include <iostream>

class Complex {
 private:
  double re;
  double im;
 public:

  Complex() :
    re(0),
    im(0)
  { }

  Complex(double re, double im) :
    re(re),
    im(im)
  { }

  double Re() const { return re; }
  double& Re() { return re; }
  double Im() const { return im; }
  double& Im() { return im; }

  /* Переопределяем операцию сложения. По смыслу операция должна возвращать новый
     объект. Операция бинарная, у неё есть два аргумента. Левым аргументом неявно
     является сам объект, у которого эта операция вызвана, а правый аргумент передаётся
     в круглых скобках. По смыслу операция не должна менять своих аргументов, поэтому
     метод константный, а аргумент передаётся по константной ссылке. */
  Complex operator+(const Complex& other) const {
    /* Вызываем конструктор и создаём новый объект. */
    return Complex(re + other.re, im + other.im);
  }

  /* Переопределяем операцию +=. Она также является бинарной, левым аргументом
     неявно является сам объект, у которого вызывается данный метод, правый аргумент
     передаётся в скобках. По смыслу операция может поменять левый аргумент,
     но не должна менять правый. Поэтому метод неконстантный, а правый аргумент
     передаётся по константной ссылке. По смыслу операция не создаёт новый объект,
     однако, обычно можно написать A = (B += C), поэтому операция возвращает ссылку
     на сам объект. */
  Complex& operator+=(const Complex& other) {
    re += other.re;
    im += other.im;

    /* Возвращаем текущий объект по ссылке. Специальная переменная this в классе
       содержит указатель на текущий объект, у которого в данный момент вызван метод.
```

```
47        Пусть есть объект obj класса Cls, у него вызывается метод Method(). Если данный
48        метод не является константным, то внутри него переменная this будет иметь
49        тип Cls* и равняться &obj. Если данный метод является константным, то внутри
50        него переменная this будет иметь тип const Cls* и равняться &obj. То есть
51        значение зависит от того объекта, у которого метод вызывается. */
52      return *this;
53    }
54
55    /* Всё то же самое. */
56    Complex operator-(const Complex& other) const {
57      return Complex(re - other.re, im - other.im);
58    }
59
60    /* Операция - "минус" бывает не только бинарная, но и унарная. Такая операция
61       принимает только один аргумент. В данном случае он уже неявно задан --- это
62       сам объект, у которого метод вызван. По смыслу операция возвращает новый объект
63       и не должна менять свой аргумент. */
64    Complex operator-() const {
65      return Complex(-re, -im);
66    }
67
68    /* Всё то же самое. */
69    Complex& operator-=(const Complex& other) {
70      re -= other.re;
71      im -= other.im;
72
73      return *this;
74    }
75  };
76
77  int main() {
78    Complex c1(1, 2);
79    Complex c2(3, 4);
80    Complex c3;
81
82    // Здесь у объекта c1 будет вызван метод operator+() и в него передастся аргумент c2.
83    c3 = c1 + c2;
84
85    // Это эквивалентно следующей строчке:
86    c3 = c1.operator+(c2);
87
88    /* Пример использования операции +=. */
89    c3 += c2;
90
91    /* Пример использования унарного минуса. */
92    c3 = -c2;
93
94    std::cout << "(" << c3.Re() << ", " << c3.Im() << ")" << std::endl;
95    return 0;
96  }
```

4.2 Второй способ. Добавление отдельных функций

Для перегрузки операций для некоторого класса необязательно добавлять специальный метод в класс, вместо этого можно написать специальную функцию, никак не связанную с классом. В этом случае все аргументы операции нужно указывать явно.

```
#include <iostream>

class Complex {
 private:
  double re;
  double im;
 public:
  Complex() :
    re(0),
    im(0)
  { }

  Complex(double re, double im) :
    re(re),
    im(im)
  { }

  double Re() const { return re; }
  double& Re() { return re; }
  double Im() const { return im; }
  double& Im() { return im; }
};

/* Перегружаем операцию сложения для двух комплексных чисел. Поскольку функция написана
   отдельно от класса, в неё не передаются аргументы неявным образом. Кроме того,
   функция не может обращаться к приватным полям класса напрямую. В остальном всё то же
   самое. По смыслу аргументы не должны меняться и функция создаёт новый объект. */
Complex operator+(const Complex& left, const Complex& right) {
  return Complex(left.Re() + right.Re(), left.Im() + right.Im());
}

/* По смыслу первый аргумент может измениться, второй меняться не должен, операция
   не создаёт новых объектов. */
Complex& operator+=(Complex& left, const Complex& right) {
  left.Re() += right.Re();
  left.Im() += right.Im();

  return left;
}

Complex operator-(const Complex& left, const Complex& right) {
  return Complex(left.Re() - right.Re(), left.Im() - right.Im());
}

/* Унарная операция "минус", мы должны явно указать аргумент. */
```

```
46 Complex operator-(const Complex& argument) {
47   return Complex(-argument.Re(), -argument.Im());
48 }
49
50 Complex& operator-=(Complex& left, const Complex& right) {
51   left.Re() -= right.Re();
52   left.Im() -= right.Im();
53   return left;
54 }
55
56 int main() {
57   Complex c1(1, 2);
58   Complex c2(3, 4);
59   Complex c3;
60
61   // Здесь вызывается функция operator+(), и в неё передаются аргументы c1 и c2.
62   c3 = c1 + c2;
63
64   // Это эквивалентно следующей строчке:
65   c3 = operator+(c1, c2);
66
67   /* Пример использования операции +=. */
68   c3 += c2;
69
70   /* Пример использования унарного минуса. */
71   c3 = -c2;
72
73   std::cout << "(" << c3.Re() << ", " << c3.Im() << ")" << std::endl;
74   return 0;
75 }
```

4.3 Сложение обычного числа и комплексного

Использование методов класса для перегрузки операций выглядит компактнее, однако, в некоторых случаях приходится писать отдельные функции. Рассмотрим пример.

```
1  #include <iostream>
2
3  class Complex {
4   private:
5    double re;
6    double im;
7   public:
8    Complex() :
9      re(0),
10     im(0)
11   { }
12
13   Complex(double re, double im) :
14     re(re),
```

```
    im(im)
  { }

  double Re() const { return re; }
  double& Re() { return re; }
  double Im() const { return im; }
  double& Im() { return im; }

  /* Этот метод перегружает операцию с левым аргументом типа Complex и правым
     аргументом типа double. */
  Complex operator-(double number) const {
    return Complex(re - number, im);
  }
};

/* А вот эта функция перегружает операцию с левым аргументом типа double и правым
   аргументом типа Complex. По смыслу операция не меняет своих аргументов и
   возвращает комплексное число. */
Complex operator-(double number, const Complex& right) {
  return Complex(number - right.Re(), - right.Im());
}

int main() {
  Complex c1(1, 2);
  Complex c2;

  /* Здесь работает метод. */
  c2 = c1 - 5;

  /* А вот здесь метод сработать не может, здесь работает функция. */
  c2 = 3 - c1;

  std::cout << "(" << c2.Re() << ", " << c2.Im() << ")" << std::endl;
  return 0;
}
```

4.4 Сложение обычного числа и комплексного - 2

Код необходимый для сложения обычного числа и комплексного можно сократить при помощи операции неявного приведения типа.

```
#include <iostream>

class Complex {
 private:
  double re;
  double im;
 public:
  Complex(double re = 0, double im = 0) :
    re(re),
```

```
    im(im)
  { }

  double Re() const { return re; }
  double& Re() { return re; }
  double Im() const { return im; }
  double& Im() { return im; }
};

/* Эта функция перегружает операцию вычитания для двух аргументов типа Complex. */
Complex operator- (const Complex& left, const Complex& right) {
  return Complex(left.Re() - right.Re(), left.Im() - right.Im());
}

int main() {
  Complex c1(1, 2);
  Complex c2;

  /* Здесь целое число неявно преобразуется в double, а затем в тип Complex при помощи
     написанного нами конструктора. */
  c2 = c1 - 5;

  /* Здесь целое число неявно преобразуется в double, а затем в тип Complex при помощи
     написанного нами конструктора. */
  c2 = 3 - c1;

  std::cout << "(" << c2.Re() << ", " << c2.Im() << ")" << std::endl;
  return 0;
}
```

В приведённом примере операцию нужно перегружать именно при помощи функции, метод не сработает в случае вычитания из обычного числа комплексного. То есть компилятор не может неявно конвертировать левый аргумент операции в класс и вызвать у объекта класса соответствующий метод.

5 Операции сравнения и логические операции

Операции сравнения и логические операции также можно перегружать (среди них `<`, `>`, `<=`, `>=`, `==`, `!=`, `!` (унарная), `&&`, `||`, `&&=`, `||=`). Рассмотрим пример операции сравнения для комплексных чисел.

```
#include <iostream>

class Complex {
 private:
  double re;
  double im;
 public:
  Complex(double re = 0, double im = 0) :
    re(re),
```

```
    im(im)
  { }

  double Re() const { return re; }
  double& Re() { return re; }
  double Im() const { return im; }
  double& Im() { return im; }

  /* По смыслу операция не должна менять своих аргументов и должна возвращать true
     или false. */
  bool operator<(const Complex& other) const {
    if (re < other.re)
      return true;
    if (re > other.re)
      return false;

    return im < other.im;
  }
};

int main() {
  Complex c1(1, 3);
  Complex c2(2, 4);

  std::cout << (c1 < c2) << std::endl;

  return 0;
}
```

6 Операции инкремента и декремента

Эти операции (++ и --) интересны тем, что они бывают префиксными и постфиксными. По своему смыслу префиксные операции сначала меняют аргумент, а потом возвращают значение. Постфиксные операции сначала возвращают значение, а потом меняют аргумент. Однако при перегрузке это целиком и полностью определяется программистом. Напишем реализацию на примере класса комплексных чисел таким образом, чтобы она делала то, что задумано.

```
#include <iostream>

class Complex {
 private:
  double re;
  double im;
 public:
  Complex(double re = 0, double im = 0) :
    re(re),
    im(im)
  { }

```

```
13   double Re() const { return re; }
14   double& Re() { return re; }
15   double Im() const { return im; }
16   double& Im() { return im; }
17
18   /* Префиксная операция инкремента. Она меняет объект и возвращает его же.
19      Новых объектов не создаётся. */
20   Complex& operator++() {
21     re += 1;
22     return *this;
23   }
24
25   /* Постфиксная операция. При объявлении необходимо указать формальный неиспользуемый
26      аргумент любого типа, например, int для того, чтобы отличить операцию от префиксной
27      (весьма неочевидный костыль C++). Перед изменением объекта его приходится сначала
28      скопировать, чтобы вернуть неизменённую копию. Таким образом, операция создаёт
29      новый объект. */
30   Complex operator++(int) {
31     Complex other(*this);
32     re += 1;
33     return other;
34   }
35 };
36
37 void PrintComplex(const Complex& c) {
38   std::cout << "(" << c.Re() << ", " << c.Im() << ")" << std::endl;
39 }
40
41 int main() {
42   Complex c(3, 1);
43
44   PrintComplex(c);
45
46   /* Префиксная операция. */
47   PrintComplex(++c);
48
49   PrintComplex(c);
50
51   /* Постфиксная операция. */
52   PrintComplex(c++);
53
54   PrintComplex(c);
55   return 0;
56 }
```

7 Операция копирующего присваивания

Речь идёт об обычной операции присваивания = (copy assignment operator). Вообще эта операция может быть определена для создания объекта из любого типа данных, однако стоит отдельно рассмотреть случай копирования объекта из другого объекта того же самого типа данных.

Компилятор создаёт эту операцию (копирование из другого объекта того же типа) по-умолчанию, в таком случае все элементы класса присваиваются поэлементно. Однако, такое поведение не всегда подходит, например, это не будет работать, если класс содержит указатели.

Напишем операцию копирующего присваивания для класса строки.

```
#include <iostream>
#include <cstring> // Это для std::strncpy()

class String {
 private:
  char* data;
  std::size_t size;
 public:
  String() :
    data(nullptr),
    size(0)
  { }

  String(const char* str) :
    data(nullptr),
    size(std::strlen(str))
  {
    data = new char[size + 1];

    /* Копируем поэлементно. */
    std::strncpy(data, str, size + 1);
  }

  ~String() {
    if (data)
      delete[] data;
  }

  /* Операция для копирования объекта из другого объекта того же типа принимает
     константную ссылку на объект того же типа и возвращает обычную ссылку на сам
     объект (для того, чтобы можно было написать A = B = C). */
  String& operator=(const String& other) {
    /* Сначала нужно освободить память, если таковая была выделена. */
    if (data)
      delete[] data;

    data = nullptr;
    size = other.size;

    if (other.data) {
      /* Выделяем память, если есть, что копировать. */
      data = new char[size + 1];

      /* Копируем поэлементно. */
      std::strncpy(data, other.data, size + 1);
```

```
    }

    /* Возвращаем ссылку на себя же. */
    return *this;
  }

  const char* Data() const { return data; }
};

int main() {
  String str("Hello world!");
  String strCopy;

  /* Вот здесь вызывается операция копирующего присваивания. */
  strCopy = str;

  std::cout << strCopy.Data() << std::endl;
  return 0;
}
```

Замечание 7.1 *Для корректной работы классу* `String` *необходима не только операция копирующего присваивания, но и правильный конструктор копирования.*

8 Индексация и функциональный вызов

В языке C++ можно переопределять операцию индексирования `[]` и функционального вызова `()`. Поясним на примере класса матриц. Каждую операцию перегрузим дважды: для чтения и для записи. Операция для чтения будет константной и будет возвращать число, а операция для записи будет возвращать ссылку.

```
#include <iostream>

class Matrix {
 private:
  double* data;
  std::size_t numRows;
  std::size_t numCols;

 public:
  Matrix(std::size_t numRows, std::size_t numCols) :
    data(new double[numRows * numCols]),
    numRows(numRows),
    numCols(numCols)
  {
    /* Зануляем все элементы. */
    for (std::size_t row = 0; row < numRows; row++) {
      for (std::size_t col = 0; col < numCols; col++)
        data[row * numCols + col] = 0;
    }
```

```
  }

  ~Matrix() {
    if (data)
      delete[] data;
  }

  std::size_t NumRows() const { return numRows; }
  std::size_t NumCols() const { return numCols; }

  /* Операция индексации кроме самого объекта принимает ещё один аргумент: индекс.
     Причем индекс может быть любого типа, он не обязан быть целочисленным. Для
     матрицы логично, чтобы индекс был целочисленным. Первая операция предназначена
     для чтения. */
  double operator[](std::size_t index) const {
    return data[index];
  }

  /* Эта операция предназначена для записи. Она возвращает ссылку. */
  double& operator[](std::size_t index) {
    return data[index];
  }

  /* Операция функционального вызова может принимать любое количество аргументов любого
     типа. Для матрицы логично сделать два целочисленных аргумента: номер строки и
     номер столбца. Операция для чтения. */
  double operator()(std::size_t row, std::size_t col) const {
    return data[row * numCols + col];
  }

  /* Операция для записи. */
  double& operator()(std::size_t row, std::size_t col) {
    return data[row * numCols + col];
  }
};

int main() {
  Matrix m(5, 5);

  /* Запись элементов при помощи операции индексирования. */
  m[3] = 10;
  m[14] = 5;

  for (std::size_t row = 0; row < m.NumRows(); row++) {
    for (std::size_t col = 0; col < m.NumCols(); col++) {
      /* Чтение при помощи операции индексирования. */
      std::cout << m[row * m.NumCols() + col] << " ";
    }

    std::cout << std::endl;
  }
```

```
71
72    /* Запись элементов при помощи операции функционального вызова. */
73    m(1, 2) = 11;
74    m(4, 3) = 4;
75
76    for (std::size_t row = 0; row < m.NumRows(); row++) {
77      for (std::size_t col = 0; col < m.NumCols(); col++) {
78        /* Чтение при помощи операции функционального вызова. */
79        std::cout << m(row, col) << " ";
80      }
81
82      std::cout << std::endl;
83    }
84
85    return 0;
86  }
```

9 Битовые операции

Битовые операции тоже можно перегружать (операции `&`, `|`, `~` (унарная), `^`, `<<`, `>>`, `&=`, `|=`, `^=`, `<<=`, `>>=`). Пожалуй, самыми актуальными операциями являются операции побитового сдвига, поскольку они используются для ввода/вывода в `std::cin`/`std::cout`. Напишем пример перегрузки на основе класса комплексных чисел.

9.1 Вывод данных

```
1  #include <iostream>
2
3  class Complex {
4   private:
5    double re;
6    double im;
7   public:
8    Complex(double re = 0, double im = 0) :
9      re(re),
10     im(im)
11   { }
12
13   double Re() const { return re; }
14   double& Re() { return re; }
15   double Im() const { return im; }
16   double& Im() { return im; }
17 };
18
19 /* Определим операцию << для использования в потоках вывода. Будем считать, что любой
20    поток, предназначенный для вывода, например, std::cout, имеет тип std::ostream
21    (почти), определённый в заголовочном файле <ostream>. Соответственно, нам необходимо
22    определить операцию << для типов std::ostream и Complex. */
23 std::ostream& operator<<(std::ostream& out, const Complex& number) {
```

```
  out << "(" << number.Re() << ", " << number.Im() << ")";

  return out;
}

int main() {
  Complex c(1, 3);

  std::cout << "Number: " << c << "\n";

  /* Эквивалентно:
     Операция выполняется слева направо. */
  ((std::cout << "Number: ") << c) << "\n";
  /* Или эквивалентно:
     Поскольку операция возвращает сам поток (то есть ссылку на него), операцию можно
     применять цепочкой. */
  operator<<(operator<<(operator<<(std::cout, "Number: "), c), "\n");

  return 0;
}
```

Из-за особенностей типа `std::ostream` операция должна принимать ссылку на поток, поскольку он может содержать какую-либо внутреннюю информацию, которая меняется при выводе. Кроме того, для того, чтобы операцию можно было применять цепочкой для вывода различных объектов, она должна возвращать ссылку на сам поток.

9.2 Ввод данных

```
#include <iostream>

class Complex {
 private:
  double re;
  double im;
 public:
  Complex(double re = 0, double im = 0) :
    re(re),
    im(im)
  { }

  double Re() const { return re; }
  double& Re() { return re; }
  double Im() const { return im; }
  double& Im() { return im; }
};

/* Определим операцию >> для использования в потоках ввода. Будем считать, что любой
   поток, предназначенный для ввода, например, std::cin, имеет тип std::istream (почти),
   определённый в заголовочном файле <istream>. Соответственно, нам необходимо
   определить операцию >> для типов std::istream и Complex. */
```

```
23 std::istream& operator>>(std::istream& in, Complex& number) {
24   in >> number.Re() >> number.Im();
25
26   return in;
27 }
28
29 int main() {
30   Complex c;
31
32   /* Использование операции для чтения комплексного числа. */
33   std::cin >> c;
34
35   std::cout << c.Re() << " " << c.Im() << std::endl;
36   return 0;
37 }
```

10 Заключение

Перегрузка операций позволяет значительно сократить код, необходимый для работы с классом и повысить читабельность программы. Возможности языка C++ по перегрузке операторов не исчерпываются приведёнными выше примерами. Более подробно про перегрузку операций написано в [1] и [3].

Список литературы

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

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

[3] `https://en.cppreference.com/w/cpp/language/operators`