В.Л. Тарасов

Лекции по программированию на C++

Лекция 12.

Перегрузка операторов

12.1. Функции-операторы

В языке C++ можно определять функции, имена которых совпадают с именами встроенных операторов: `+`, `-`, `*`, `/`, `->`, `new` и т.д. Такие функции определяются с ключевым словом `operator` и называются *функциями-операторами*. При вызове таких функций достаточно указывать только знак оператора, например, `+`. Объявления таких функций имеет вид:

```
тип_результата operator знак_оператора(список_аргументов);
```

Как правило, в функциях-операторах выполняются действия, которые отражаются знаком используемого оператора, что делает программы более краткими и понятными.

Примеры создания и использования функций-операторов приведены в следующей программе.

Программа 12.1. Обыкновенные дроби

Создадим класс `Fraction` для моделирования обыкновенных дробей:

```
// Файл Fraction.h

#ifndef FractionH
#define FractionH

#include <iostream>

using namespace std;

class Fraction{                          // Обыкновенная дробь
    int num;                             // Числитель
    int denom;                           // Знаменатель
public:
    Fraction(int n = 1, int m = 1)       // Конструктор
    {
        num = n;
        denom = m;
    }
```

```
    // Методы доступа к закрытым членам
    int numerator()                     // Числитель
    {
        return num;
    }
    int denominator()                   // Знаменатель
    {
        return denom;
    }
// Функции-операторы, выполняющие действия над дробями
    Fraction operator+(Fraction b);     // Сложение дробей
    Fraction operator-()                // Унарный минус
    {
        return Fraction(-num, denom);   // Изменение знака дроби
    }
    friend Fraction operator-(Fraction a, Fraction b);// Вычитание дробей
    void print()                        // Метод для вывода дроби
    {
        cout << num << '/' << denom;
    }
};
// Независимые функции для умножения и деления дробей
Fraction operator*(Fraction a, Fraction b);
Fraction operator/(Fraction a, Fraction b);
#endif
```

Оператор унарный минус `Fraction operator-()` не имеет аргументов потому, что ему, как члену класса, доступна дробь, для которой он вызывается. Вызов конструктора в этой функции:

```
return Fraction(-num, denom);
```

создает безымянный объект (дробь) с числителем `-num` и знаменателем `denom`, значение которой возвращается из функции. После того, как возвращенная из данной функции в вызывающую программу безымянная дробь будет использована, она уничтожается с вызовом деструктора. Унарный минус реализован непосредственно в объявлении класса, поэтому функция является встроенной.

Далее приведены реализации функций-операторов.

```
// Файл Fraction.cpp
#include "Fraction.h"
Fraction Fraction::operator+(Fraction b)      // Сложение дробей
{
```

```
    int cd = denom * b.denom;                  // Знаменатель суммы
    int ns = num * b.denom + b.num * denom;    // Числитель суммы
    Fraction sum(ns, cd);                      // sum – сумма дробей
    return sum;
}
```

Функция-оператор сложения дробей имеет один аргумент b, который считается вторым слагаемым. Первым слагаемым является тот объект (та дробь), для которого будет вызван эта функция. Для суммы дробей создается локальная переменная sum, числитель и знаменатель которой находятся по правилам сложения обыкновенных дробей. Значение sum возвращается как результат работы функции.

```
// Продолжение файла Fraction.cpp

Fraction operator-(Fraction a, Fraction b)        // Вычитание дробей
{                                                 // это функция-друг
    int cd = a.denom * b.denom;                   // Знаменатель разности
    int ns = a.num * b.denom - b.num * a.denom;   // Числитель разности
    Fraction subtr(ns, cd);                       // subtr – разность дробей
    return subtr;
}
```

Функция-оператор вычитания дробей не является членом класса, поэтому она имеет два аргумента - уменьшаемое и вычитаемое. Эта функция является другом класса Fraction, поэтому ей разрешен доступ к закрытым членам класса – числителю num, и знаменателю denom.

```
// Продолжение файла Fraction.cpp

Fraction operator*(Fraction a, Fraction b)
{
    int np = a.numerator() * b.numerator();       // Числитель произведения
    int dp = a.denominator() * b.denominator();   // Знаменатель произвед.
    return  Fraction(np, dp);             // Создание и возврат произведения
}
```

Функция-оператор умножения дробей, не являясь членом класса Fraction, должна иметь два аргумента – сомножителя. Для доступа к закрытым членам класса – числителю и знаменателю дроби – она использует открытые методы numerator() и denominator(). В данной функции, как и в функции унарного минуса, не создается именованная локальная переменная для произведения. Вместо этого вызывается конструктор, которому передаются числитель и знаменатель произведения, конструктор создает безымянную дробь, значение которой передается вызывающей функции.

```
// Продолжение файла Fraction.cpp

Fraction operator/(Fraction a, Fraction b)
{
```

```
    if(b.numerator() == 0){
        cerr << "The divisor is equal to zero\n";
        exit(1);
    }
    // Умножаем дробь a на дробь, обратную дроби b
    return  a * Fraction(b.denominator(), b.numerator());
}
```

Испытаем класс Fraction.

```
// Файл TestFraction.cpp

#include "Fraction.h"// Файлы в кавычках ищутся сначала в папке проекта

int main()
{
    Fraction f12(1, 2),              // Дробь 1/2
             f13(1, 3),              // Дробь 1/3
             fs, fd, fp, fm, fdiv;   // 5 дробей, равных по умолчанию 1/1
    cout << "f12  = "; f12.print(); cout << endl;
    cout << "f13  = "; f13.print(); cout << endl;

    fs = f12.operator+(f13);         // Явный вызов функции-оператора +
                                     // Неявный вызов: fs = f12 + f13;
    cout << "fs   = "; fs.print(); cout << endl;

    fd = f12 - f13;                  // Неявный вызов функции-оператора -
                                     // Явный: fd = operator-(f12, f13);
    cout << "fd   = "; fd.print(); cout << endl;

    fp = operator*(f12, f13);        // Явный вызов функции-оператора *
                                     // Неявный вызов: fp = f12 * f13;
    cout << "fp   = ";   fp.print(); cout << endl;

    fm = -f12;                       // Неявный вызов унарного минуса
                                     // Явный вызов: fm = f12.operator-();
    cout << "fm   = ";   fm.print(); cout << endl;

    fdiv = f12 / f13;                // Деление дробей
                                     // Явный: fdiv = operator/(f12, f13);
    cout << "fdiv = ";   fdiv.print(); cout << endl;
    return 0;
}
```

Программа выводит:

```
f12  = 1/2
f13  = 1/3
fs   = 5/6
fd   = 1/6
fp   = 1/6
fm   = -1/2
fdiv = 3/2
```

В функции main() для сложения и умножения дробей использованы явные вызовы соответствующих функций-операторов:

```
fs = f12.operator+(f13); // Явный вызов функции-оператора, члена класса
fp = operator*(f12, f13);// Явный вызов функции-оператора, не члена класса
```

Вместо явных вызовов функций-операторов можно использовать неявный, который выглядит более понятно:

```
fs = f12 + f13;
fp = f12 * f13;
```

12.2. Правила перегрузки операторов

Перегружены могут быть любые операторы языка C++ (они перечислены в табл. 2.9), за исключением операторов: *точка* (.), *точка-звездочка* (.*), *два двоеточия* (::) и (?:).

Нельзя менять синтаксис операторов, например, нельзя определить оператор перемножения сразу трех дробей, так как встроенный оператор умножения – бинарный:

```
Fraction operator*(Fraction a, Fraction b, Fraction c);    // Ошибка,
                                                           // 3 сомножителя
```

Для перегруженных операторов сохраняются приоритеты и порядок выполнения, указанные в табл. 2.9, например, в инструкции:

```
fs = fd + fp * fm;
```

сначала выполнится умножение, а затем сложение.

Функция-оператор должна быть или членом класса, или иметь аргумент типа класса. Нельзя определить функцию-оператор с аргументами только встроенных типов, например, нельзя определить функцию-оператор сложения int и double:

```
int operator+(int, double);          // Ошибка, аргументы только
                                     // встроенных типов
```

Нельзя изобрести новый знак оператора, например:

```
Fraction operator@();                // Ошибка, несуществующий оператор @
```

Программа 12.2. Комплексные числа

В данной программе приводятся дополнительные примеры использования перегрузки операторов на примере создания класса комплексных чисел.

В файл Complex.h поместим объявление класса комплексных чисел:

```
// Файл Complex.h

#ifndef ComplexH
#define ComplexH
```

```
#include <math.h>

class Complex
{
    double re, im;       // Действительная и мнимая части комплексного числа

public:

    Complex()                          // Конструктор по умолчанию
    {
        re = 0; im = 0;
    }

    Complex(double x)                  // Конструктор с одним аргументом,
    {                                  // формирование комплексного числа
        re = x; im = 0;                // по одному вещественному
    }

    Complex(double x, double y)        // Конструктор с двумя аргументами
    {
        re = x; im = y;
    }

    double Arg();                      // Аргумент комплексного числа
    double Abs();                      // Модуль комплексного числа

    Complex& operator+=(Complex z)     // Сложение с присваиванием
    {
        re += z.re; im += z.im;
        return *this;
    }

    Complex& operator-=(Complex z)             // Вычитание с присваиванием
    {
        re -= z.re; im -= z.im;
        return *this;
    }

    Complex operator*(Complex);                // Перемножение комплексных
    Complex operator*(double);                 // Умножение комплексного
                                               // на вещественное
    Complex operator/(Complex);                // Деление комплексных чисел
    Complex operator-();                       // Унарный минус
    void Roots(int m, Complex rts[]);          // Извлечение корней степени m
    void Print();                              // Вывод комплексного числа
};

// Объявление функций, не являющихся членами класса Complex

Complex operator+(Complex z, Complex t);       // Сложение
Complex operator-(Complex z, Complex t);       // Вычитание
Complex operator*(double a, Complex z);        // Умножение
Complex Pow(Complex z, int m);                 // Возведение комплексного z
                                               // в целую степень m
// Получение комплексного числа по его модулю и аргументу
Complex Polar(double mod, double arg);
```

```
#endif
```

В классе Complex предусмотрены три конструктора, чтобы можно было создавать комплексные числа любым из следующих способов:

```
Complex z1(1, 2);                // z1.re = 1, z1.im = 2
Complex z2(1);                   // z2.re = 1, z2.im = 0
Complex z3;                      // z3.re = 0, z3.im = 0
```

Наличие конструктора позволяет инициализировать комплексные числа при их определении. Например:

```
Complex z4 = 13;              // z4.re = 13.0, z4.im = 0.0
```

Здесь сначала создается безымянное комплексное число Complex(13) с использованием конструктора Complex(double x), которое затем используется для инициализации переменной z4. При этом сначала целое число 13 преобразуется в число типа double 13.0.

Функции-операторы operator+=(Complex z) и operator-=(Complex z) изменяют свой первый операнд и возвращают ссылку на него.

Реализацию функций для работы с комплексными числами разместим в файле Complex.cpp.

```
// Файл Complex.cpp

#include "Complex.h"

#include <iostream>
#define _USE_MATH_DEFINES                // Для доступа к M_PI

using namespace std;

double Complex::Arg()                    // Аргумент комплексного числа
{
    return atan2(im, re);
}

double Complex::Abs()                    // Модуль комплексного числа
{
    return sqrt(re * re + im * im);
}

Complex Complex::operator*(Complex z)    // Умножение двух комплексных
{
    Complex t(re * z.re - im * z.im, re * z.im + im * z.re);
    return t;
}

Complex Complex::operator*(double a)  // Умножение комплексного на double
{
    return *this * Complex(a);
}

Complex Complex::operator/(Complex z) // Деление
```

```
{
    double r = (re * z.re + im * z.im) / (z.re * z.re + z.im * z.im);
    double i = (-re * z.im + im * z.re) / (z.re * z.re + z.im * z.im);
    return Complex(r, i);
}

Complex Complex::operator-()          // Унарный минус,
{                                     // получение комплексного числа
    return Complex(-re, -im);         // с противоположным знаком
}
```

При нахождении аргумента комплексного число использована функция математической библиотеки (заголовочный файл cmath):

```
double atan2(double y, double x);
```

которая возвращает значение *arctg*(*y*/*x*). Она работает корректно, даже когда угол равен π/2 или -π/2, то есть когда $x = 0$.

Так как функция умножения operator*(Complex z) является членом класса, ей доступно комплексное число, для которого она вызывается. Это число, представленное значениями действительной и мнимой частей re и im, рассматривается как первый сомножитель. Второй сомножитель z передается в функцию как аргумент.

Умножение комплексного числа на число с плавающей точкой в функции operator*(double a) сводится к умножению двух комплексных чисел путем преобразования множителя a в комплексное число конструктором: Complex(a).

В функции деления operator/(Complex z) реализована формула деления комплексных чисел:

$$\frac{re + i \cdot im}{z.re + i \cdot z.im} = \frac{re \cdot z.re + im \cdot z.im + i \cdot (-re \cdot z.im + im \cdot z.re)}{(z.re)^2 + (z.im)^2}.$$

Следующие функции не являются членами класса Complex.

```
// Продолжение файла Complex.cpp

Complex operator+(Complex z, Complex t)              // Сложение
{
    return z += t;
}

Complex operator-(Complex z, Complex t)              // Вычитание
{
    return z -= t;
}

// Умножение вещественного на комплексное

Complex operator*(double a, Complex z)
{
```

```
    return z * a;                   // Умножение комплексного на double
}
```

Обсудим работу функции `operator+(Complex z, Complex t)`. Выражение `z += t` есть краткая запись вызова функции-оператора: `z.operator+=(t)`, которая, как член класса `Complex`, имеет доступ к закрытым членам `z` и изменяет `z`, прибавляя к нему `t`. Полученная сумма комплексных чисел $z+t$ возвращается как результат функции `operator+(Complex z, Complex t)`.

Аналогично, в функции `operator-(Complex z, Complex t)` выражение `z -= t` есть вызов функции-члена `z.operator-=(Complex t)`, которая вычисляет разность `z - t`.

Работа функции `operator*(double a, Complex z)` сводится к вызову функции `Complex::operator*(double a)`.

Напомним, что комплексное число можно представить в алгебраической, показательной или тригонометрической форме:

$$z = x + iy = \rho e^{i\varphi} = \rho\left(cos\,\varphi + i\,sin\,\varphi\right).$$

Здесь x – действительная часть, y – мнимая часть, ρ – модуль комплексного числа, φ – аргумент. Целую степень комплексного числа можно найти по формуле:

$$z^m = \left(x + iy\right)^m = \rho^m e^{i\varphi m}.$$

Корни степени m находятся по формуле:

$$\sqrt[m]{z} = \sqrt[m]{\rho}\,e^{i\frac{\varphi+2k\pi}{m}} = \sqrt[m]{\rho}\left(cos\frac{\varphi+2k\pi}{m} + i\,sin\frac{\varphi+2k\pi}{m}\right), k = \overline{0,m-1}$$

Приведенные формулы использованы в следующих функциях.

```
// Продолжение файла Complex.cpp

// Polar: формирование комплексного числа по его модулю mod и аргументу arg

Complex Polar(double mod, double arg)
{
    return Complex(mod * cos(arg), mod * sin(arg));
}

// Pow: возведение комплексного числа z в степень m

Complex Pow(Complex z, int m)
{
    double angle = z.Arg();             // Аргумент
    double mod = z.Abs();               // Модуль
    angle *= m;                         // Увеличение аргумента в m раз
    double mp = 1.0;                    // m-я степень модуля
    for(int k = 0; k < m; k++)          // находится с помощью
```

```
        mp *= mod;                          // произведения
    return Polar(mp, angle);
}

// Roots: извлекает корень степени m из комплексного числа
// и записывает m корней в массив rts

void Complex::Roots(int m, Complex rts[])
{
    double angle = Arg();                   // Аргумент
    double mod = Abs();                     // Модуль
    mod = pow(mod, 1.0 / m);                // Корень m-й степени из модуля
    for(int k = 0; k < m; k++)              // Получение m корней
        rts[k] = Polar(mod, (angle + 2 * k * M_PI) / m);
}

void Complex::Print()                       // Вывод комплексного числа
{
    cout << "(" << re << ", " << im << ")";
}
```

В качестве примера использования класса комплексных чисел решим приведенное кубическое уравнение:

$$z^3 + pz + q = 0.$$

Его корни можно выразить формулой Кардано:

$$z = \sqrt[3]{-\frac{q}{2} + \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}} + \sqrt[3]{-\frac{q}{2} - \sqrt{\frac{q^2}{4} + \frac{p^3}{27}}}.$$

Каждый кубический корень приведенной формулы имеет три значения. Решение кубического уравнения дают комбинации кубических корней, произведение которых равно $-p/3$.

Функцию `main()` поместим в файл `CubicEquation.cpp`.

```
// Файл CubicEquation.cpp

#include "Complex.h"
#include <iostream>
using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
    const double eps = 0.1E-4;              // Малое число (точность)
    double x, y;

    cout << "Решаем приведенное кубическое уравнение \n"
        " z^3 + p*z + q = 0\n";

    cout << "Введите действительную и мнимую части p: ";
    cin >> x >> y;
```

```
    Complex p(x, y);                        // Создание коэффициента p

    cout << "Введите действительную и мнимую части q: ";
    cin >> x >> y;
    Complex q(x, y);                        // Создание коэффициента q
    Complex inner[2];                       // Массив для квадратных корней

    // Массивы для первого и второго кубического корня
    Complex root3_1[3], root3_2[3];
    Complex D;                          // Дискриминант кубического уравнения
    D = q * q / 4 + p * p * p / 27;
    D.Roots(2, inner);                  // Квадратный корень из дискриминанта

    // Первое выражение под куб. корнем
    Complex D1 = (-0.5) * q + inner[0];

    // Второе выражение под куб. корнем
    Complex D2 = (-0.5) * q - inner[0];
    D1.Roots(3, root3_1);          // Извлечение первого кубического корня
    D2.Roots(3, root3_2);          // Извлечение второго кубического корня
    Complex p3 = -p / 3;                    // Критерий для отбора корней
    Complex eq;                             // Левая часть уравнения
    Complex z;                              // Переменная для корня
    Complex prod;                           // Произведение кубических корней

    // Перебор возможных комбинаций кубических корней
    for(int i = 0; i < 3; i++)              // Перебор значений первого
                                            // кубического корня
        for(int j = 0; j < 3; j++){         // Перебор значений второго
                                            // кубического корня
            prod = root3_1[i] * root3_2[j];       // Произведение
                                                  // кубических корней
            double err = (prod - p3).Abs();       // Отклонение произведения
                                                  // от -p/3
            if(err < eps){                        //Если отклонение мало,
                z = root3_1[i] + root3_2[j];      //это корень
                cout << "\nКорень z =  "; z.Print();
                eq = Pow(z, 3) + p * z + q;       //Подстановка корня
                                                  // в уравнение
                cout << "\nПроверка: z^3 + p*z + q = ";
                eq.Print();
            }
        }
        cout << endl;
        return 0;
}
```

В качестве примера решим уравнение

$$z^3 - 2z + 1 = 0,$$

имеющее корни:

$$z_1 = 1, \quad z_2 = -\left(1+\sqrt{5}\right)/2 = -1.61803, \quad z_3 = \left(\sqrt{5}-1\right)/2 = 0.61803.$$

Далее приводится диалог с программой.

```
Решаем приведенное кубическое уравнение
 z^3 + p*z + q = 0
Введите действительную и мнимую части p: -2 0
Введите действительную и мнимую части q:  1 0

Корень z =  (1, 0)
Проверка: z^3 + p*z + q = (0, 0)
Корень z =  (-1.61803, 2.08167e-016)
Проверка: z^3 + p*z + q = (0, 1.13997e-015)
Корень z =  (0.618034, 0)
Проверка: z^3 + p*z + q = (0, 0)
```

Хотя все корни рассмотренного уравнения действительны, у второго корня мнимая часть оказалась отличной нуля. Это объясняется тем, что числа представляются в памяти не абсолютно точно из-за ограниченности разрядной сетки, поэтому все вычисления являются приближенными. Эта мнимая часть `2.08167e-016` много меньше абсолютной величины действительной части корня: `1.61803`, поэтому во многих случаях такой малой величиной можно пренебречь.