Теория и технология программирования

Основы программирования на языках C и C++

Лекция 12. Проектирование классов. Перегрузка операций.

Глухих Михаил Игоревич, к.т.н., доц.
mailto: glukhikh@mail.ru

Числовые объекты

- Язык C++ позволяет описать собственные правила сложения, вычитания, умножения, деления для нашего типа
- А также описать правила преобразования из примитивного типа к нашему и обратно

Пример проектирования – рациональное число

- Пусть, например, в рамках проекта «арифметический тренажер» необходимо реализовать основные операции с рациональными числами
- Наша задача – спроектировать соответствующий класс

Свойства и действия

- У рационального числа есть числитель (*numerator*, произвольное целое число), и знаменатель (*denominator*, натуральное число)
- Рациональные числа можно складывать, вычитать, умножать и делить
- В арифметическом тренажере также потребуются сравнение на равенство и неравенство
- Кроме этого, скорее всего, потребуется их вводить, выводить, преобразовывать к целому (вещественному) и обратно

Члены-данные класса

```
class Rational
{
    // Числитель
    int numer;
    // Знаменатель (>=1)
    int denom;
    …
};
```

Конструкторы

- Конструктор – функция-член класса, которая вызывается **автоматически** при создании переменной соответствующего типа. Явно вызывать конструктор не требуется. Обычно конструктор задает начальное состояние объекта.

```
Rational r; // 1 раз
Rational* ptr = new Rational[10]; // 10 раз
```

- Имя конструктора всегда **совпадает с именем класса**. Конструктор **не имеет результата**, но **может иметь аргументы**.
- Конструктор без аргументов называется конструктором по умолчанию. В двух примерах выше вызовется именно он.
- Если в классе не определен ни один конструктор, то считается, что в нем присутствует конструктор по умолчанию, не делающий ничего.

Конструкторы с аргументами

- Часто начальное состояние объекта может быть задано по-разному. Для этого используются конструкторы с аргументами.
- Для того, чтобы при создании объекта был вызван конструктор с аргументами, необходимо аргументы записать в скобках, например:

```
// Конструктор с двумя целыми аргументами
Rational r(2,3);
Rational* ptr = new Rational(2,3);
// А здесь только конструктор по умолчанию
Rational* array = new Rational[10];
```

Преобразующие конструкторы

- Конструктор с одним аргументом может быть использован для преобразования типа: тип аргумента→тип объекта
- Такой конструктор называется преобразующим

```
// Обычный вариант
Rational r(3);
// Преобразование типа int→Rational
Rational s=4, t=(Rational)5;
```

- Если мы **не хотим**, чтобы конструктор с одним аргументом был преобразующим, впереди следует указать ключевое слово **explicit**

Создание рационального числа

- Пусть по умолчанию вновь созданное рациональное число равно 0 (при этом числитель равен 0,знаменатель 1). Для этого требуется конструктор по умолчанию.

```
Rational r;
```

- Пусть целое значение n можно присвоить рациональному (при этом числитель равен n, знаменатель 1). Для этого требуется определить конструктор с целым аргументом.

```
Rational r=3; // или
Rational s(3);
```

- Наконец, рациональное число можно создать, задав числитель и знаменатель. Для этого требуется определить конструктор с двумя целыми аргументами

```
Rational r(1,2); // 1/2
```

Конструктор по умолчанию

```
// В файле rational.h
class Rational
{
   …
   Rational();
};
// В файле rational.cpp
Rational::Rational()
{
   numer=0;
   denom=1;
}
```

Конструктор из целого (преобразующий)

```
// В файле rational.h
class Rational
{
    …
    Rational(int number);
};
// В файле rational.cpp
Rational::Rational(int number)
{
    numer=number;
    denom=1;
}
```

Конструктор с двумя аргументами

```
// В файле rational.h
class Rational
{
   …
   Rational(int n, int d);
};
// В файле rational.cpp
Rational::Rational(int n, int d)
{
   numer=n;
   denom=d;
}
```

Конструктор копирования

- **Конструктором копирования** называется конструктор, единственным аргументом которого является ссылка на другой объект того же типа
- Конструктор копирования служит для создания копии объекта, например:

```
Rational r(1,3);
// Вызов конструктора копирования
Rational s(r);
```

- Если конструктор копирования не определен, то используется его версия по умолчанию: побайтное копирование всех данных объекта. Для рационального числа нас это устраивает.

Оператор присваивания

- Объекты одного типа по правилам C/C++ можно присваивать друг другу:

```
Rational r(1,4);
Rational s;
s=r;
Rational t=s; // что будет здесь?
```

- При этом, вызывается функция «оператор присваивания»:

```
Rational& operator =(const Rational& r);
```

- Если данная функция для объекта не определена, происходит побайтное копирование. Для рациональных чисел нас это устраивает.

Сложение

- При реализации сложения (и других арифметических операций тоже) следует помнить, что, помимо обычного сложения (+) существует еще добавление (+=).
- Как правило, при реализации арифметических операций делается функция как для одного, так и для другого, причем вторая использует первую.

Оператор добавления

```
// В файле rational.h
class Rational
{
   …
   Rational& operator +=(const Rational& r);
};
// В файле rational.cpp
Rational& Rational::operator +=(const Rational& r)
{
   numer = (numer*r.denom+denom*r.numer);
   denom *= r.denom;
   // this - указатель на себя
   // *this - ссылка на себя
   return *this;
}
```

Использование добавления

- Например:

```
Rational a(1,2), b(1,6);
a += b; // Получится 8/12
// Эквивалентно
a.operator +=(b);
```

- Лучше было бы получить 2/3…
- Подобная проблема встанет и для других операций
- Поэтому можно предусмотреть функцию упрощения рационального числа, которую мы будем вызывать каждый раз, когда значение числа меняется
- Функция упрощения может быть закрытой – мы ее будем вызывать сами, она не потребуется тем, кто использует данный класс

Оператор добавления

```
// В файле rational.h
class Rational
{
   …
   Rational& operator +=(const Rational& r);
};
// В файле rational.cpp
Rational& Rational::operator +=(const Rational& r)
{
   numer = (numer*r.denom+denom*r.numer);
   denom *= r.denom;
   simplify(); // упрощение
   return *this;
}
```

Функция упрощения

```
// В файле rational.h
class Rational
{
    …
    void simplify();
    …
public:
};
```

Функция упрощения

```
// В файле rational.cpp
void Rational::simplify()
{
    if (denom < 0)
    {
        numer = -numer;
        denom = -denom;
    }
    for (int i=2; i<=abs(denom) && i<=abs(numer); i++)
        if (numer % i == 0 && denom % i == 0)
        {
            numer /= i;
            denom /= i;
            i--;
        }
}
```

Оператор сложения

```
// В файле rational.h
class Rational
{
   …
   Rational operator +(const Rational& r) const;
};
// В файле rational.cpp
Rational Rational::operator +(const Rational &r) const
{
   // this - указатель на себя
   // *this - ссылка на себя
   Rational res(*this);
   // Используем готовую операцию добавления
   return res += r;
}
```

Вычитание

- Вычитание целесообразно выполнять через операцию отрицания, или унарный минус: -r
- Тогда (как в алгебре) вместо a-b мы можем вычислить a+(-b), используя уже определенные операции сложения и отрицания

Оператор отрицания (унарный минус)

```
// В файле rational.h
class Rational
{
   …
   Rational operator -() const;
};
// В файле rational.cpp
Rational Rational::operator -() const
{
   Rational r(-numer, denom);
   return r;
}
```

Оператор уменьшения

```
// В файле rational.h
class Rational
{
   …
   Rational& operator -=(const Rational& r);
};
// В файле rational.cpp
Rational& Rational::operator -=(const Rational& r)
{
   return (*this += (-r));
}
```

Инкремент

- Операция инкремента существует в двух вариантах: префиксный и постфиксный. Оба варианта могут быть переопределены в нашем классе.
- Префиксный оператор должен изменить объект и вернуть измененное значение
- Постфиксный оператор должен запомнить старое значение, затем изменить объект и вернуть его старое значение

Операторы инкремента

```
// В файле rational.h
class Rational
{
    …
    Rational& operator ++(); // префикс
    Rational operator ++(int); // постфикс
};
```

Операторы инкремента

```
// В файле rational.cpp
Rational& Rational::operator ++()
{
   numer += denom;
   return *this;
}
Rational Rational::operator ++(int)
{
   Rational r(*this);
   numer += denom;
   return r;
}
```

Сравнение

- Операторы сравнения имеют два аргумента и логический результат
- В языках C/C++ существуют 6 операций сравнения: ==, !=, >, >=, <=, <. Мы реализуем первые две из них.
- Удобно вторую операцию реализовать на базе первой, инвертировав ее результат.

Операторы сравнения

```
// В файле rational.h
class Rational
{
   …
  bool operator ==(const Rational& r) const;
  bool operator !=(const Rational& r) const;
};
// В файле rational.cpp
bool Rational::operator ==(const Rational& r) const
{
   return (numer==r.numer) && (denom==r.denom);
}
bool Rational::operator !=(const Rational& r) const
{
   return !(*this==r);
}
```

Преобразования типов

- Преобразование встроенных типов **к типу нашего объекта** выполняется с помощью преобразующих конструкторов
- Преобразование нашего типа **к встроенным типам** выполняется с помощью операторов преобразования типа
- Операторы преобразования имеют название **operator type**, где type - соответствующий встроенный тип. Аргументов у них нет, тип результата не указывается (всегда соответствует type).

Операторы преобразования типов

```
// В файле rational.h
class Rational
{
   …
   operator int() const;
   operator double() const;
};
// В файле rational.cpp
Rational::operator int() const
{
   return numer / denom;
}
Rational::operator double() const
{
   return ((double)numer)/denom;
}
```

Операторы ввода-вывода

- Определяются как друзья класса, так как левый аргумент у них имеет другой тип (поток)
- Результат операторов ввода-вывода – ссылка на тот же поток (позволяет каскадные операции вида cout<<a<<b<<c)

Операторы ввода-вывода

```
// В файле rational.h
class Rational
{
   friend istream& operator >>(istream& in, Rational& r);
   friend ostream& operator <<(ostream& out, const Rational& r);
};
// В файле rational.cpp
istream& operator >>(istream& in, Rational& r)
{
   in>>r.numer>>r.denom;
   return in;
}
ostream& operator <<(ostream& out, const Rational& r)
{
   out<<r.numer<<"/"<<r.denom;
   return out;
}
```

Тестирование

- После создания нового объекта целесообразно его оттестировать.
- Для этого можно создать проект, в который войдут вновь созданный модуль *rational* и главная функция
- В главной функции необходимо хотя бы по разу использовать каждую из функций и операций с созданным объектом

Фрагменты тестирования

```
// Тестируем конструктор, +, -, *=, /, <<
Rational a(1,2), b(-1,6);
cout<<"a="<<a<<" b="<<b<<" a+b="<<a+b<<endl;
cout<<"a("<<a<<") *= b("<<b<<")"<<endl;
a *= b;
cout<<"a="<<a<<" b="<<b<<" a-b="<<a-b<<endl;
Rational c=3;
cout<<"b="<<b<<" c="<<c<<" b/c="<<b/c<<endl;
```

Фрагменты тестирования

```
// Тестируем >>, !=
Rational e(7,8), f(5,12);
cout<<"e="<<e<<" f="<<f<<" e+f=?"<<endl;
cout<<"Введите результат g=m/n в формате: m n"<<endl;
Rational g;
cin>>g;
if (e+f!=g)
   cout<<"Неправильно! e+f="<<e+f<<endl;
else
   cout<<"Правильно!"<<endl;
```

Итоги

- Рассмотрено
  - Разработан и оттестирован тип «рациональное число»
  - Поддерживается ряд перегруженных операций
- Далее
  - Лексический и синтаксический анализ