**Московский государственный технический**

**университет им. Н.Э. Баумана**

Факультет «Информатика и системы управления»

Кафедра ИУ5 «Системы обработки информации и управления»

Курс «Программирование на основе классов и шаблонов»

Отчет по лабораторной работе №3

«Разработка класса дробь»

|  |  |  |
| --- | --- | --- |
| Выполнил: |  | Проверил: |
| студент группы ИУ5-22 |  | преподаватель каф. ИУ5 |
| Барышников Михаил |  | Козлов А.Д. |
| Подпись и дата: |  | Подпись и дата: |

Москва, 2020 г.

**Задача**

Создать класс для работы с обыкновенными дробями. Числитель и знаменатель дроби имеют тип int. При выводе и после выполнения арифметических операций дроби сокращаются, то есть числитель и знаменатель не должны иметь общих множителей. Перегрузить операции '+', ‘-‘, ‘\*’, ‘/’ для сложения, вычитания, умножения, деления дробей, дроби и целого числа, дроби и вещественного числа в любых сочетаниях (дробь + целое, целое + дробь, дробь + дробь, дробь + вещественное, вещественное + дробь).

Также перегрузить все операции сравнения: “<, <=, ==, !=, >=, >” для всех сочетаний объектов класса и встроенных типов. Для инициализации объектов разрабатываемого класса обыкновенных дробей предусмотреть соответствующие конструкторы: значения по умолчанию 1/1, копирования, с одним аргументом типа int, с одним аргументом типа double, с двумя аргументами типа int, с тремя аргументами типа int. При перегрузке операций использовать функции - члены класса, а где это невозможно, то функции - друзья класса.

**Схема алгоритма**

****

Приведена в пример схема алгоритма переопределения оператора вывода. Тем самым можно вывести объект класса Fraction в поток вывода. Для того, чтобы переопределить операцию вывода для объектов класса необходимо создать friend-функцию к классу т.к. левый операнд (вызывающий) является объектом другого класса (ostream) потокового вывода. Аналогично и для остальных friend переопределений арифметических операторов. Сам вывод – форматированный, также выводится целая часть (смешанная дробь), если знаменатель меньше числителя. Для форматирования использовалась библиотека <iomanip>.

**Текст программы**

* class\_fraction.h

#pragma once

#include <iostream>

#include <iomanip>

#include <string>

class Fraction

{

int numerator;

int denominator;

bool isNegative;

static uint8\_t accuracy;

public:

Fraction();

Fraction(const Fraction&);

Fraction(int);

Fraction(int, int);

Fraction(int, int, int);

Fraction(double);

~Fraction() {}

// functions

static void set\_accuracy(uint8\_t);

void evqlid();

// (-)

Fraction operator-();

// =

Fraction operator=(const Fraction&);

Fraction operator=(int);

Fraction operator=(double);

// \*

Fraction operator\*(int);

Fraction operator\*(double);

Fraction operator\*(const Fraction&);

friend Fraction operator\*(int, Fraction&);

friend Fraction operator\*(double, Fraction&);

// /

Fraction operator/(int);

Fraction operator/(double);

Fraction operator/(const Fraction&);

friend Fraction operator/(int, Fraction&);

friend Fraction operator/(double, Fraction&);

// +

Fraction operator+(int);

Fraction operator+(double);

Fraction operator+(const Fraction &);

friend Fraction operator+(int, Fraction&);

friend Fraction operator+(double, Fraction&);

// -

Fraction operator-(int);

Fraction operator-(double);

Fraction operator-(const Fraction &);

friend Fraction operator-(int, Fraction&);

friend Fraction operator-(double, Fraction&);

// >

bool operator>(int);

bool operator>(double);

bool operator>(const Fraction&);

friend bool operator>(int, Fraction&);

friend bool operator>(double, Fraction&);

// ==

bool operator==(int);

bool operator==(double);

bool operator==(const Fraction&);

friend bool operator==(int, Fraction&);

friend bool operator==(double, Fraction&);

// !=

bool operator!=(int);

bool operator!=(double);

bool operator!=(const Fraction&);

friend bool operator!=(int, Fraction&);

friend bool operator!=(double, Fraction&);

// <

bool operator<(int);

bool operator<(double);

bool operator<(const Fraction&);

friend bool operator<(int, Fraction&);

friend bool operator<(double, Fraction&);

// >=

bool operator>=(int);

bool operator>=(double);

bool operator>=(const Fraction&);

friend bool operator>=(int, Fraction&);

friend bool operator>=(double, Fraction&);

// <=

bool operator<=(int);

bool operator<=(double);

bool operator<=(const Fraction&);

friend bool operator<=(int, Fraction&);

friend bool operator<=(double, Fraction&);

// stream out

friend std::ostream &operator<<(std::ostream&, const Fraction&);

};

* class\_fraction.cpp

#include "class\_fraction.h"

uint8\_t Fraction::accuracy = 4;

Fraction::Fraction()

{

isNegative = false;

numerator = 1;

denominator = 1;

}

Fraction::Fraction(const Fraction &n)

{

numerator = n.numerator;

isNegative = n.isNegative;

denominator = n.denominator;

evqlid();

}

Fraction::Fraction(int b)

{

numerator = abs(b);

denominator = 1;

isNegative = b < 0 ? true : false;

}

Fraction::Fraction(int b, int c)

{

if (c != 0)

{

numerator = abs(b);

denominator = abs(c);

isNegative = b \* c < 0 ? true : false;

evqlid();

}

else

{

\*this = Fraction();

}

}

Fraction::Fraction(int a, int b, int c)

{

if (c != 0)

{

numerator = abs(b) + abs(a) \* abs(c);

denominator = abs(c);

isNegative = a \* b \* c < 0 ? true : false;

evqlid();

}

else

{

\*this = Fraction();

}

}

Fraction::Fraction(double n)

{

denominator = 1;

for (int i = 0; i != accuracy; ++i)

{

n \*= 10;

denominator \*= 10;

}

numerator = abs(n);

numerator += (int)(n \* 10) % 10 > 4 ? 1 : 0;

isNegative = n < 0 ? true : false;

evqlid();

}

void Fraction::set\_accuracy(uint8\_t n)

{

accuracy = n;

}

void Fraction::evqlid()

{

register int i = 2;

while (numerator >= i && denominator >= i)

{

if (!(numerator % i) && !(denominator % i))

{

numerator /= i;

denominator /= i;

}

else

{

++i;

}

}

if (!numerator)

denominator = 1;

}

Fraction Fraction::operator-()

{

Fraction n = \*this;

n.isNegative = !n.isNegative;

return n;

}

Fraction Fraction::operator=(const Fraction &n)

{

numerator = n.numerator;

isNegative = n.isNegative;

denominator = n.denominator;

return \*this;

}

Fraction Fraction::operator=(int n)

{

return \*this = Fraction(n);

}

Fraction Fraction::operator=(double n)

{

return \*this = Fraction(n);

}

Fraction Fraction::operator\*(int n)

{

return \*this \* Fraction(n);

}

Fraction Fraction::operator\*(double n)

{

return \*this \* Fraction(n);

}

Fraction Fraction::operator\*(const Fraction &n)

{

Fraction new\_f = n;

new\_f.numerator \*= numerator;

new\_f.denominator \*= denominator;

new\_f.isNegative = (new\_f.isNegative || isNegative) && !(new\_f.isNegative && isNegative);

return new\_f;

}

Fraction Fraction::operator/(int n)

{

return \*this / Fraction(n);

}

Fraction Fraction::operator/(double n)

{

return \*this / Fraction(n);

}

Fraction Fraction::operator/(const Fraction &n)

{

Fraction new\_f = n;

int a = new\_f.numerator;

new\_f.numerator = new\_f.denominator;

new\_f.denominator = a;

return \*this \* new\_f;

}

Fraction Fraction::operator+(int n)

{

return \*this + Fraction(n);

}

Fraction Fraction::operator+(double n)

{

return \*this + Fraction(n);

}

Fraction Fraction::operator+(const Fraction &n)

{

Fraction new\_f = n;

numerator \*= isNegative ? -1 : 1;

new\_f.numerator \*= new\_f.isNegative ? -1 : 1;

new\_f.numerator \*= denominator;

new\_f.numerator += numerator \* new\_f.denominator;

new\_f.denominator \*= denominator;

new\_f.isNegative = new\_f.numerator < 0 ? true : false;

new\_f.numerator = abs(new\_f.numerator);

numerator = abs(numerator);

return new\_f;

}

Fraction Fraction::operator-(int n)

{

return \*this - Fraction(n);

}

Fraction Fraction::operator-(double n)

{

return \*this - Fraction(n);

}

Fraction Fraction::operator-(const Fraction &n)

{

return \*this + -const\_cast <Fraction&> (n);

}

bool Fraction::operator>(int n)

{

return \*this > Fraction(n);

}

bool Fraction::operator>(double n)

{

return \*this > Fraction(n);

}

bool Fraction::operator>(const Fraction &n)

{

Fraction new\_f = n;

int n\_1, n\_2;

evqlid();

new\_f.evqlid();

if (!isNegative && new\_f.isNegative)

return true;

else if (isNegative && !new\_f.isNegative)

return false;

n\_1 = numerator \* new\_f.denominator;

n\_2 = new\_f.numerator \* denominator;

n\_1 \*= isNegative ? -1 : 1;

n\_2 \*= new\_f.isNegative ? -1 : 1;

if (n\_1 > n\_2)

return true;

return false;

}

bool Fraction::operator==(int n)

{

return \*this == Fraction(n);

}

bool Fraction::operator==(double n)

{

return \*this == Fraction(n);

}

bool Fraction::operator==(const Fraction &n)

{

Fraction new\_f = n;

evqlid();

new\_f.evqlid();

return numerator == new\_f.numerator && denominator == new\_f.denominator && isNegative == new\_f.isNegative;

}

bool Fraction::operator!=(int n)

{

return \*this != Fraction(n);

}

bool Fraction::operator!=(double n)

{

return \*this != Fraction(n);

}

bool Fraction::operator!=(const Fraction &n)

{

return !(\*this == n);

}

bool Fraction::operator<(int n)

{

return \*this < Fraction(n);

}

bool Fraction::operator<(double n)

{

return \*this < Fraction(n);

}

bool Fraction::operator<(const Fraction &n)

{

return !(\*this > n) && \*this != n;

}

bool Fraction::operator>=(int n)

{

return \*this >= Fraction(n);

}

bool Fraction::operator>=(double n)

{

return \*this >= Fraction(n);

}

bool Fraction::operator>=(const Fraction &n)

{

return !(\*this < n);

}

bool Fraction::operator<=(int n)

{

return \*this <= Fraction(n);

}

bool Fraction::operator<=(double n)

{

return \*this <= Fraction(n);

}

bool Fraction::operator<=(const Fraction &n)

{

return !(\*this > n);

}

Fraction operator\*(int n, Fraction &F)

{

return Fraction(n) \* F;

}

Fraction operator\*(double n, Fraction &F)

{

return Fraction(n) \* F;

}

Fraction operator/(int n, Fraction &F)

{

return Fraction(n) / F;

}

Fraction operator/(double n, Fraction &F)

{

return Fraction(n) / F;

}

Fraction operator+(int n, Fraction &F)

{

return F + n;

}

Fraction operator+(double n, Fraction &F)

{

return F + n;

}

Fraction operator-(int n, Fraction & F)

{

return Fraction(n) - F;

}

Fraction operator-(double n, Fraction & F)

{

return Fraction(n) - F;

}

bool operator>(int n, Fraction &F)

{

return Fraction(n) > F;

}

bool operator>(double n, Fraction &F)

{

return Fraction(n) > F;

}

bool operator==(int n, Fraction &F)

{

return Fraction(n) == F;

}

bool operator==(double n, Fraction &F)

{

return Fraction(n) == F;

}

bool operator!=(int n, Fraction &F)

{

return Fraction(n) != F;

}

bool operator!=(double n, Fraction &F)

{

return Fraction(n) != F;

}

bool operator<(int n, Fraction &F)

{

return Fraction(n) < F;

}

bool operator<(double n, Fraction &F)

{

return Fraction(n) < F;

}

bool operator>=(int n, Fraction &F)

{

return Fraction(n) >= F;

}

bool operator>=(double n, Fraction &F)

{

return Fraction(n) >= F;

}

bool operator<=(int n, Fraction &F)

{

return Fraction(n) <= F;

}

bool operator<=(double n, Fraction &F)

{

return Fraction(n) <= F;

}

std::ostream &operator<<(std::ostream& stream, const Fraction& n)

{

std::string s\_1 = std::to\_string(n.numerator % n.denominator), s\_2 = std::to\_string(n.numerator / n.denominator), s\_3 = std::to\_string(n.denominator);

int length = s\_1.length() > s\_3.length() ? s\_1.length() + 2 : s\_3.length() + 2;

stream << "\n" << std::setw(length + s\_2.length()) << s\_1;

stream << "\n" << (n.isNegative ? '-' : ' ') << s\_2 << ' ';

for (int i = 0; i != length - 2; ++i)

stream << '-';

stream << "\n" << std::setw(length + s\_2.length()) << s\_3;

return stream;

}

* main.cpp

#include "class\_fraction.h"

#include <iomanip>

int main()

{

Fraction a(1.536), b(3, 5), c(1.99994);

Fraction::set\_accuracy(8);

b = - 12 - a + b;

std::cout << (a + Fraction(3, 35)) / c << a <<

b << c << "\nb > a " << (b > a) << "\nc == a " <<

(c == a) << "\nb <= c " << (b <= c);

system("pause");

return 0;

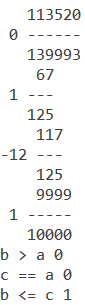
}

**Анализ результатов**

В данной программе в качестве «точки входа» выступает основная функция main(). В ней написан тестируемый код. Ниже представлены результаты выполнения. Для реализации методов данного класса были «с нуля» написаны 3 метода, остальные реализованы с их использованием.

* Сложение дробей
* Умножение дробей
* Операция сравнения “==” для дробей

Тест (в коде программы):



Тест (начальные дроби остались неизменными, а выражение a \* ((b \* c + b) / b)):



Тест (значения – a = 6/7, b = 3/5, c = 99/100, выражение a\*((b\*c - b)/b) + c\*b\*2):



Тест (значения – a = 6/11, b = 3/5, c = 99/101, выражение 3.11 \* a \* ((b \* a - b) / 2) + 4 \* c \* b):



Тесты были проверены вручную. Программа отработала верно. Также были проведены многочисленные тесты логических операций, но так как их много (6 штук), все тесты не были прикреплены. Но основываясь на операциях > и == можно сделать вывод, что остальные работают правильно. Это связано с тем, что остальные операторы были переопределены с использованием этих (базовых).

**Вывод**

В результате написания программы был получен опыт работы с классами (ООП). Было реализованы перегрузки всех основных операторов простых арифметических действий, всех операторов сравнения. В качестве доработки можно было бы сделать шаблонные функции операторов, тогда для каждого оператора исчезли бы по 2 перегрузки функций. Также были переопределены функция вывода и оператор унарного минуса. Еще из минусов – это возможность переполнения int значение (знаменателя или числителя дроби), но и это решаемо, т.к. можно подключить библиотеку для работы с длинной арифметикой на c++. В итоге был написан класс дробей, который удобно использовать для точного вычисления выражений с обыкновенными и смешанными дробями.