#include<iostream>

using namespace std;

class Fraction;

Fraction operator-(Fraction left, Fraction right);

inline Fraction operator\* (Fraction left, Fraction right);

inline Fraction operator/ (Fraction left, Fraction right);

class Fraction

{

int integer; //Целая часть

int numerator; //Числитель

int denominator;//Знаменатель

public:

int get\_integer()const

{

return integer;

}

int get\_numerator()const

{

return numerator;

}

int get\_denomiantor()const

{

return denominator;

}

void set\_integer(int integer)

{

this->integer = integer;

}

void set\_numerator(int numerator)

{

this->numerator = numerator;

}

void set\_denominator(int denominator)

{

this->denominator = denominator ? denominator : 1;

}

// Constructors:

Fraction()

{

integer = numerator = 0;

denominator = 1;

//cout << "DefaultConstructor:" << this << endl;

}

Fraction(int integer)

{

//Конструктор с одним параметром

this->integer = integer;

this->numerator = 0;

this->denominator = 1;

//cout << "1argConstructor:" << this << endl;

}

Fraction(int numerator, int denominator)

{

this->integer = 0;

this->numerator = numerator;

this->denominator = denominator;

//cout << "Constructor:\t" << this << endl;

}

Fraction(int integer, int numerator, int denominator)

{

this->integer = integer;

this->numerator = numerator;

this->set\_denominator(denominator);

//cout << "Constructor:\t" << this << endl;

}

Fraction(const Fraction& other)

{

this->integer = other.integer;

this->numerator = other.numerator;

this->denominator = other.denominator;

//cout << "CopyConstructor:" << this << endl;

}

~Fraction()

{

//cout << "Destructor:\t" << this << endl;

}

// Operators:

Fraction& operator=(const Fraction& other)

{

this->integer = other.integer;

this->numerator = other.numerator;

this->denominator = other.denominator;

cout << "CopyAssignment:" << this << endl;

return \*this;

}

// Methods:

void print()const

{

if (integer)cout << integer;

if (numerator)

{

if (integer)cout << "(";

cout << numerator << "/" << denominator;

if (integer)cout << ")";

}

if (integer == 0 && numerator == 0)cout << 0;

cout << endl;

}

void print\_address()const

{

cout << "Object:\t\t" << this << "\n";

cout << "Integer:\t" << &integer << "\n";

cout << "Numerator:\t" << &numerator << "\n";

cout << "Denominator:\t" << &denominator << "\n";

}

Fraction& proper()

{

integer += numerator / denominator;

numerator%=denominator;

return \*this;

}

Fraction& improper()

{

numerator += integer \* denominator;

integer = 0;

return \*this;

}

Fraction operator\*=(const Fraction & other)

{

return \*this = \*this \* other;

}

Fraction inverted()

{

this->improper();

return Fraction(denominator, numerator);

}

Fraction& reduce() //по алгоритму Евклида

{

int more, less, rest;

if (numerator < denominator)

{

less = numerator;

more = denominator;

}

else

{

less = denominator;

more = numerator;

}

for (rest=1; rest==0; )

{

rest = more % less;

more = less;

less = rest;

}

int GCD = more;

numerator /= GCD;

denominator /= GCD;

return \*this;

}

Fraction& operator/=(const Fraction& other)

{

return \*this = \*this / other;

}

Fraction& operator++() //префикс

{

integer++;

return \*this;

}

Fraction operator++(int) //постфикс

{

Fraction old = \*this;

integer++;

return old;

}

};

bool operator==(const Fraction& left, const Fraction& right)

{

/\*if (left.get\_integer() == right.get\_integer() &&

left.get\_numerator() == right.get\_numerator() &&

left.get\_denomiantor() == right.get\_denomiantor())

return true;

else return false;\*/

return left.get\_integer() == right.get\_integer() &&

left.get\_numerator() == right.get\_numerator() &&

left.get\_denomiantor() == right.get\_denomiantor();

}

ostream& operator<<(ostream& os, const Fraction& obj)

{

if (obj.get\_integer() != 0) os << obj.get\_integer() << ' ';

if (obj.get\_numerator() != 0) os << obj.get\_numerator() << '/' << obj.get\_denomiantor() << ' ';

return os;

}

Fraction operator-(Fraction left, Fraction right)

{

Fraction result(

left.get\_integer() - right.get\_integer(),

left.get\_numerator() \* right.get\_denomiantor() -

right.get\_numerator() \* left.get\_denomiantor(),

left.get\_denomiantor()\* right.get\_denomiantor()

);

return result.proper().reduce();

}

inline Fraction operator\* (Fraction left, Fraction right)

{

left.improper();

right.improper();

Fraction result(left.get\_numerator() \* right.get\_numerator(), left.get\_denomiantor() \* right.get\_denomiantor()) ;

return result.proper().reduce();

}

Fraction operator/ (Fraction left, Fraction right)

{

return left \* right.inverted();

}

void main()

{

setlocale(LC\_ALL, "Russian");

//#define CONSTRUCTORS\_CHECK

#ifdef CONSTRUCTORS\_CHECK

Fraction A; //Default constructor

A.print();

A.print\_address();

Fraction B = 5;//Single argument constructor

B.print();

B.print\_address();

Fraction C(1, 2);

C.print();

C.print\_address();

Fraction D(2, 3, 4);

D.print();

D.print\_address();

#endif // CONSTRUCTORS\_CHECK

//#define arifmetic check

#ifdef arifmetic check

Fraction A(2, 3, 4);

Fraction B(5, 7, 8);

Fraction C;

C = A / B;

C.print();

A /= B;

A.print();

#endif arifmetic check

Fraction A(3, 4);

Fraction B(1, 4);

Fraction C = A - B;

A.print();

B.print();

C.print();

cout << C << endl;

/\*

Fraction i(1, 2);

Fraction j;

j= ++i;

i.print();

j.print();

j = i++;

i.print();

j.print();

\*/

}

/\*

int a, b, c;

a = b = c = 0;

Fraction A, B, C;

cout << "\n---------------------------------------\n";

A = B = C = Fraction(1, 2, 3);

cout << "\n---------------------------------------\n";

//Fraction A(3, 4);

//A.print();

//Fraction B(4, 5);

//B.print();

//Fraction C;

//C = A; //CopyAssignment (operator=)

//C.print();

}\*/

/\*

---------------------------------

Overloading rules:

1. Перегрузить можно только СУЩЕСТВУЮЩИЕ оперторы

+ перегружается

++ перегружается

\* перегружается

\*\* НЕ перегружается

2. НЕ все существующие оперторы можно перегрузить. Не перегружаются

?: - тернарный оператор;

. - оператор прямого доступа;

.\*

:: - оператор разрешения видимости

#

##

3. Перегруженные оперторы сохраняют приоритет;

4. Нельзя изменить поведение операторов со свтроенными типами.

---------------------------------

\*/