**Lab Exercises**

Class: 软件1802 Name: 陈国栋 StuID: 201816040217

Date: 2019.11.24 Score:

**Lab Exercise 1 — String Concatenation**

**I Lab Objectives**

In this lab, you will practice:

1. Overloading the + operator to allow String objects to be concatenated.
2. Writing function prototypes for overloaded operators.
3. Using overloaded operators.

**II Description of the Problem**

String **concatenation（连接）**requires two operands—the two strings that are to be concatenated. In the String case study, we showed how to implement an overloaded concatenation operator that concatenates the second String object to the right of the first String object, thus modifying the first String object. In some applications, it is desirable to produce a concatenated String object without modifying the String arguments. Implement operator+ to allow operations such as

string1 = string2 + string3;

in which neither operand is modified.

**III Sample Output**



**IV Problem-Solving Tips**

1. The overloaded + operator should be a member function of class String and should take one parameter, a const reference to a String.
2. The + operator function should use return type String.
3. The strcat function can be used to concatenate pointer-based strings.

**V Your Solution**

// Fig. 11.9: String.h

// String class definition with operator overloading.

#ifndef STRING\_H

#define STRING\_H

#include <iostream>

using namespace std;

class String

{

friend ostream &operator<<( ostream &, const String & );

friend istream &operator>>( istream &, String & );

public:

String( const char \* = "" ); // conversion/default constructor

String( const String & ); // copy constructor

~String(); // destructor

const String &operator=( const String & ); // assignment operator

const String &operator+=( const String & ); // concatenation operator

const String operator+(const String &); // two String add

bool operator!() const; // is String empty?

bool operator==( const String & ) const; // test s1 == s2

bool operator<( const String & ) const; // test s1 < s2

// test s1 != s2

bool operator!=( const String &right ) const

{

return !( \*this == right );

} // end function operator!=

// test s1 > s2

bool operator>( const String &right ) const

{

return right < \*this;

} // end function operator>

// test s1 <= s2

bool operator<=( const String &right ) const

{

return !( right < \*this );

} // end function operator <=

// test s1 >= s2

bool operator>=( const String &right ) const

{

return !( \*this < right );

} // end function operator>=

char &operator[]( int ); // subscript operator (modifiable lvalue)

char operator[]( int ) const; // subscript operator (rvalue)

String operator()( int, int = 0 ) const; // return a substring

int getLength() const; // return string length

private:

int length; // string length (not counting null terminator)

char \*sPtr; // pointer to start of pointer-based string

void setString( const char \* ); // utility function

}; // end class String

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2010 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Fig. 11.11: fig11\_11.cpp

// String class test program.

#include <iostream>

#include "String.h"

using namespace std;

int main()

{

String s1( "happy" );

String s2( " birthday" );

String s3;

String s4;

// test overloaded equality and relational operators

cout << "s1 is \"" << s1 << "\"; s2 is \"" << s2

<< "\"; s3 is \"" << s3 << '\"'

<< boolalpha << "\n\nThe results of comparing s2 and s1:"

<< "\ns2 == s1 yields " << ( s2 == s1 )

<< "\ns2 != s1 yields " << ( s2 != s1 )

<< "\ns2 > s1 yields " << ( s2 > s1 )

<< "\ns2 < s1 yields " << ( s2 < s1 )

<< "\ns2 >= s1 yields " << ( s2 >= s1 )

<< "\ns2 <= s1 yields " << ( s2 <= s1 )<<endl;

// test for overloaded +

s4 = s1 +s2;

cout<<"s1 = "<<s1<<endl;

cout<<"s2 = "<<s2<<endl;

cout<<"s1 + s2 = "<<s4<<endl;

// test overloaded String empty (!) operator

cout << "\n\nTesting !s3:" << endl;

//cout<<s1<<endl;

//cout<<s2<<endl;

if ( !s3 )

{

cout << "s3 is empty; assigning s1 to s3;" << endl;

s3 = s1; // test overloaded assignment

cout << "s3 is \"" << s3 << "\"";

} // end if

// test overloaded String concatenation operator

cout << "\n\ns1 += s2 yields s1 = ";

s1 += s2; // test overloaded concatenation

cout << s1;

// test conversion constructor

cout << "\n\ns1 += \" to you\" yields" << endl;

s1 += " to you"; // test conversion constructor

cout << "s1 = " << s1 << "\n\n";

// test overloaded function call operator () for substring

cout << "The substring of s1 starting at\n"

<< "location 0 for 14 characters, s1(0, 14), is:\n"

<< s1( 0, 14 ) << "\n\n";

// test substring "to-end-of-String" option

cout << "The substring of s1 starting at\n"

<< "location 15, s1(15), is: "

<< s1( 15 ) << "\n\n";

// test copy constructor

String \*s4Ptr = new String( s1 );

cout << "\n\*s4Ptr = " << \*s4Ptr << "\n\n";

// test assignment (=) operator with self-assignment

cout << "assigning \*s4Ptr to \*s4Ptr" << endl;

\*s4Ptr = \*s4Ptr; // test overloaded assignment

cout << "\*s4Ptr = " << \*s4Ptr << endl;

// test destructor

delete s4Ptr;

// test using subscript operator to create a modifiable lvalue

s1[ 0 ] = 'H';

s1[ 6 ] = 'B';

cout << "\ns1 after s1[0] = 'H' and s1[6] = 'B' is: "

<< s1 << "\n\n";

// test subscript out of range

cout << "Attempt to assign 'd' to s1[30] yields:" << endl;

s1[ 30 ] = 'd'; // ERROR: subscript out of range

} // end main

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2010 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Fig. 11.10: String.cpp

// String class member-function and friend-function definitions.

#include <iostream>

#include <iomanip>

#include <cstring> // strcpy and strcat prototypes

#include <cstdlib> // exit prototype

#include "String.h" // String class definition

#include<bits/stdc++.h>

using namespace std;

// conversion (and default) constructor converts char \* to String

String::String( const char \*s )

: length( ( s != 0 ) ? strlen( s ) : 0 )

{

cout << "Conversion (and default) constructor: " << s << endl;

setString( s ); // call utility function

} // end String conversion constructor

// copy constructor

String::String( const String &copy )

: length( copy.length )

{

cout << "Copy constructor: " << copy.sPtr << endl;

setString( copy.sPtr ); // call utility function

} // end String copy constructor

// Destructor

String::~String()

{

cout << "Destructor: " << sPtr << endl;

delete [] sPtr; // release pointer-based string memory

} // end ~String destructor

//overloaded + operator;

const String String::operator+( const String &right)

{

//string answ;

int len;

len = length + right.length+1;

char \*tempPtr = new char[ len + 1 ];

// String ans;

//answ = \*sPtr + right.sPtr;

strcpy( tempPtr, sPtr );

strcat( tempPtr, right.sPtr);

return String(tempPtr);

}

// overloaded = operator; avoids self assignment

const String &String::operator=( const String &right )

{

cout << "operator= called" << endl;

if ( &right != this ) // avoid self assignment

{

delete [] sPtr; // prevents memory leak

length = right.length; // new String length

setString( right.sPtr ); // call utility function

} // end if

else

cout << "Attempted assignment of a String to itself" << endl;

return \*this; // enables cascaded assignments

} // end function operator=

// concatenate right operand to this object and store in this object

const String &String::operator+=( const String &right )

{

size\_t newLength = length + right.length; // new length

char \*tempPtr = new char[ newLength + 1 ]; // create memory

strcpy( tempPtr, sPtr ); // copy sPtr

strcpy( tempPtr + length, right.sPtr ); // copy right.sPtr

delete [] sPtr; // reclaim old space

sPtr = tempPtr; // assign new array to sPtr

length = newLength; // assign new length to length

return \*this; // enables cascaded calls

} // end function operator+=

// is this String empty?

bool String::operator!() const

{

return length == 0;

} // end function operator!

// Is this String equal to right String?

bool String::operator==( const String &right ) const

{

return strcmp( sPtr, right.sPtr ) == 0;

} // end function operator==

// Is this String less than right String?

bool String::operator<( const String &right ) const

{

return strcmp( sPtr, right.sPtr ) < 0;

} // end function operator<

// return reference to character in String as a modifiable lvalue

char &String::operator[]( int subscript )

{

// test for subscript out of range

if ( subscript < 0 || subscript >= length )

{

cerr << "Error: Subscript " << subscript

<< " out of range" << endl;

exit( 1 ); // terminate program

} // end if

return sPtr[ subscript ]; // non-const return; modifiable lvalue

} // end function operator[]

// return reference to character in String as rvalue

char String::operator[]( int subscript ) const

{

// test for subscript out of range

if ( subscript < 0 || subscript >= length )

{

cerr << "Error: Subscript " << subscript

<< " out of range" << endl;

exit( 1 ); // terminate program

} // end if

return sPtr[ subscript ]; // returns copy of this element

} // end function operator[]

// return a substring beginning at index and of length subLength

String String::operator()( int index, int subLength ) const

{

// if index is out of range or substring length < 0,

// return an empty String object

if ( index < 0 || index >= length || subLength < 0 )

return ""; // converted to a String object automatically

// determine length of substring

int len;

if ( ( subLength == 0 ) || ( index + subLength > length ) )

len = length - index;

else

len = subLength;

// allocate temporary array for substring and

// terminating null character

char \*tempPtr = new char[ len + 1 ];

// copy substring into char array and terminate string

strncpy( tempPtr, &sPtr[ index ], len );

tempPtr[ len ] = '\0';

// create temporary String object containing the substring

String tempString( tempPtr );

delete [] tempPtr; // delete temporary array

return tempString; // return copy of the temporary String

} // end function operator()

// return string length

int String::getLength() const

{

return length;

} // end function getLength

// utility function called by constructors and operator=

void String::setString( const char \*string2 )

{

sPtr = new char[ length + 1 ]; // allocate memory

if ( string2 != 0 ) // if string2 is not null pointer, copy contents

strcpy( sPtr, string2 ); // copy literal to object

else // if string2 is a null pointer, make this an empty string

sPtr[ 0 ] = '\0'; // empty string

} // end function setString

// overloaded output operator

ostream &operator<<( ostream &output, const String &s )

{

output << s.sPtr;

return output; // enables cascading

} // end function operator<<

// overloaded input operator

istream &operator>>( istream &input, String &s )

{

char temp[ 100 ]; // buffer to store input

input >> setw( 100 ) >> temp;

s = temp; // use String class assignment operator

return input; // enables cascading

} // end function operator>>

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2010 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

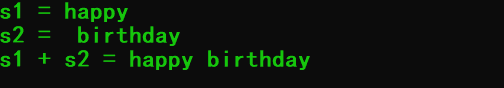
\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/



**Lab Exercise 2 — Huge Integer**

**I Lab Objectives**

In this lab, you will practice:

1. Overloading arithmetic and comparison operators to enhance a huge integer class, HugeInt**.**
2. Writing function prototypes for overloaded operators.
3. Calling overloaded operator functions.

**II Description of the Problem**

A machine with 32-bit integers can represent integers in the range of approximately –2 billion to +2 billion. This fixed-size restriction is rarely troublesome, but there are applications in which we would like to be able to use a much wider range of integers. This is what C++ was built to do, namely, create powerful new data types. Consider class HugeInt of **Figs. 10.17–10.19.** Study the class carefully, then overload the relational and equality operators. [Note: We do not show an assignment operator or copy constructor for class HugeInt, because the assignment operator and copy constructor provided by the compiler are capable of copying the entire array data member properly.]

**III Sample Output**



**IV Problem-Solving Tips**

You can implement the !=, >, >= and <= operators in terms of the overloaded == and < operators.

**V Your Solution**

// Lab 2: Hugeint.h

// HugeInt class definition.

#ifndef HUGEINT\_H

#define HUGEINT\_H

#include <iostream>

using namespace std;

class HugeInt

{

friend ostream &operator<<( ostream &, const HugeInt & );

public:

HugeInt( long = 0 ); // conversion/default constructor

HugeInt( const char \* ); // conversion constructor

// addition operator; HugeInt + HugeInt

HugeInt operator+( const HugeInt & ) const;

// addition operator; HugeInt + int

HugeInt operator+( int ) const;

// addition operator;

// HugeInt + string that represents large integer value

HugeInt operator+( const char \* ) const;

bool operator==(const HugeInt &) const;

bool operator!=(const HugeInt &) const;

bool operator>(const HugeInt &) const;

bool operator<(const HugeInt &) const;

bool operator<=(const HugeInt &) const;

bool operator>=(const HugeInt &) const;

/\* Write prototypes for the six relational and equality operators \*/

int getLength() const;

private:

short integer[ 30 ];

}; // end class HugeInt

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2012 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Lab 2: HugeIntTest.cpp

// HugeInt test program.

#include <iostream>

using namespace std;

#include "Hugeint.h"

int main()

{

HugeInt n1( 7654321 );

HugeInt n2( 7891234 );

HugeInt n3( "99999999999999999999999999999" );

HugeInt n4( "1" );

HugeInt result;

cout << "n1 is " << n1 << "\nn2 is " << n2

<< "\nn3 is " << n3 << "\nn4 is " << n4

<< "\nresult is " << result << "\n\n";

// test relational and equality operators

if ( n1 == n2 )

cout << "n1 equals n2" << endl;

if ( n1 != n2 )

cout << "n1 is not equal to n2" << endl;

if ( n1 < n2 )

cout << "n1 is less than n2" << endl;

if ( n1 <= n2 )

cout << "n1 is less than or equal to n2" << endl;

if ( n1 > n2 )

cout << "n1 is greater than n2" << endl;

if ( n1 >= n2 )

cout << "n1 is greater than or equal to n2" << endl;

result = n1 + n2;

cout << n1 << " + " << n2 << " = " << result << "\n\n";

cout << n3 << " + " << n4 << "\n= " << ( n3 + n4 ) << "\n\n";

result = n1 + 9;

cout << n1 << " + " << 9 << " = " << result << endl;

result = n2 + "10000";

cout << n2 << " + " << "10000" << " = " << result << endl;

} // end main

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2012 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Lab 2: Hugeint.cpp

// HugeInt member-function and friend-function definitions.

#include <iostream>

#include <cctype> // isdigit function prototype

#include <cstring> // strlen function prototype

using namespace std;

#include<bits/stdc++.h>

#include "Hugeint.h" // HugeInt class definition

// default constructor; conversion constructor that converts

// a long integer into a HugeInt object

HugeInt::HugeInt( long value )

{

// initialize array to zero

for ( int i = 0; i <= 29; i++ )

integer[ i ] = 0;

// place digits of argument into array

for ( int j = 29; value != 0 && j >= 0; j-- )

{

integer[ j ] = value % 10;

value /= 10;

} // end for

} // end HugeInt default/conversion constructor

// conversion constructor that converts a character string

// representing a large integer into a HugeInt object

HugeInt::HugeInt( const char \*string )

{

// initialize array to zero

for ( int i = 0; i <= 29; i++ )

integer[ i ] = 0;

// place digits of argument into array

int length = strlen( string );

for ( int j = 30 - length, k = 0; j <= 29; j++, k++ )

if ( isdigit( string[ k ] ) )

integer[ j ] = string[ k ] - '0';

} // end HugeInt conversion constructor

// get function calculates length of integer

int HugeInt::getLength() const

{

int ans;

for ( int i = 0; i <= 29; i++ )

if ( integer[ i ] != 0 ){

ans = 30 -i;

break;

cout<<i;

} // break when first digit is reached

return ans; // length is from first digit (at i) to end of array

} // end function getLength

// addition operator; HugeInt + HugeInt

HugeInt HugeInt::operator+( const HugeInt &op2 ) const

{

HugeInt temp; // temporary result

int carry = 0;

for ( int i = 29; i >= 0; i-- )

{

temp.integer[ i ] =

integer[ i ] + op2.integer[ i ] + carry;

// determine whether to carry a 1

if ( temp.integer[ i ] > 9 )

{

temp.integer[ i ] %= 10; // reduce to 0-9

carry = 1;

} // end if

else // no carry

carry = 0;

} // end for

return temp; // return copy of temporary object

} // end function operator+

// addition operator; HugeInt + int

HugeInt HugeInt::operator+( int op2 ) const

{

// convert op2 to a HugeInt, then invoke

// operator+ for two HugeInt objects

return \*this + HugeInt( op2 );

} // end function operator+

// addition operator;

// HugeInt + string that represents large integer value

HugeInt HugeInt::operator+( const char \*op2 ) const

{

// convert op2 to a HugeInt, then invoke

// operator+ for two HugeInt objects

return \*this + HugeInt( op2 );

} // end function operator+

// equality operator; HugeInt == HugeInt

/\* Write a definition for the == operator \*/

bool HugeInt::operator==( const HugeInt & right) const

{

for(int i=0;i<=29;i++)

if(right.integer[i]!=this->integer[i])

return false;

return true;

}

// inequality operator; HugeInt != HugeInt

/\* Write a definition for the != operator

by calling the == operator \*/

bool HugeInt::operator!=( const HugeInt & right) const

{

for(int i=0;i<=29;i++)

if(right.integer[i]!=this->integer[i])

return true;

return false;

}

// less than operator; HugeInt < HugeInt

/\* Write a definition for the < operator \*/

bool HugeInt::operator<( const HugeInt & right) const

{

int flag = 0;

if(this->getLength() > right.getLength())

return false;

else if(this->getLength() == right.getLength()){

for(int i=0;i<=29;i++){

if(right.integer[i] != this->integer[i])

flag = 1;

if(right.integer[i] < this->integer[i])

return false;

}

if(flag == 0)

return false;

}

else

return true;

}

// less than or equal operator; HugeInt <= HugeInt

/\* Write a definition for the <= operator

by calling the < and == operators \*/

bool HugeInt::operator<=( const HugeInt & right) const

{

int flag = 0;

if(this->getLength() > right.getLength())

return false;

else if(this->getLength() == right.getLength()){

for(int i=0;i<=29;i++){

if(right.integer[i] != this->integer[i])

flag = 1;

if(right.integer[i] < this->integer[i])

return false;

}

if(flag == 0)

return true;

}

else

return true;

}

// greater than operator; HugeInt > HugeInt

/\* Write a definition for the > operator

by calling the <= operator \*/

bool HugeInt::operator>( const HugeInt & right) const

{

int flag = 0;

if(this->getLength() > right.getLength())

return true;

else if(this->getLength() == right.getLength()){

for(int i=0;i<=29;i++){

if(right.integer[i] != this->integer[i])

flag = 1;

if(right.integer[i] > this->integer[i])

return false;

}

if(flag == 0)

return false;

}

else

return false;

}

// greater than or equal operator; HugeInt >= HugeInt

/\* Write a definition for the >= operator

by calling the > and == operators \*/

bool HugeInt::operator>=( const HugeInt & right) const

{

int flag = 0;

if(this->getLength() > right.getLength())

return true;

else if(this->getLength() == right.getLength()){

for(int i=0;i<=29;i++){

if(right.integer[i] != this->integer[i])

flag = 1;

if(right.integer[i] > this->integer[i])

return false;

}

if(flag == 0)

return true;

}

else

return false;

}

// overloaded output operator

ostream& operator<<( ostream &output, const HugeInt &num )

{

int i;

for ( i = 0; ( num.integer[ i ] == 0 ) && ( i <= 29 ); i++ )

; // skip leading zeros

if ( i == 30 )

output << 0;

else

for ( ; i <= 29; i++ )

output << num.integer[ i ];

return output;

} // end function operator<<

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2012 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

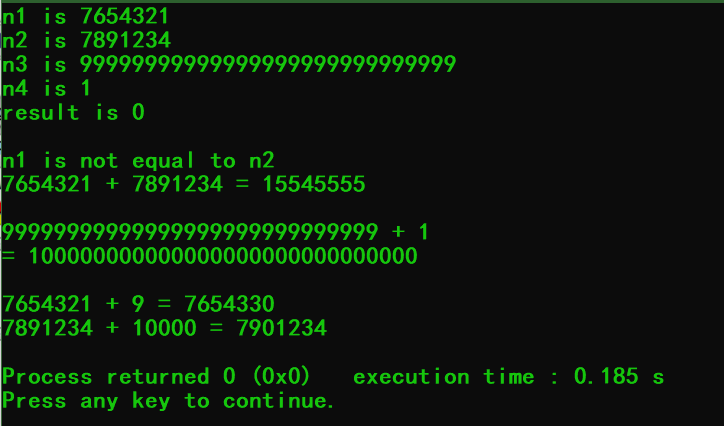
\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

****

**Lab Exercise 3 — Rational Numbers**

**I Lab Objectives**

In this lab, you will practice:

1. Overloading operators to create a class capable of storing rational numbers (fractions) and performing rational number arithmetic.
2. Writing function prototypes for overloaded operators.
3. Implementing overloaded operator functions.

The follow-up questions and activities also will give you practice:

1. Overloading the << operator.
2. Making a class more robust to prevent runtime errors.

**II Description of the Problem**

Create a classRationalNumber(fractions) with the following capabilities:

1. Create a constructor that prevents a 0 denominator in a fraction, reduces or simplifies fractions that are not in reduced form and avoids negative denominators.
2. Overload the addition, subtraction, multiplication and division operators for this class.
3. Overload the relational and equality operators for this class.

**III Sample Output**



**IV Problem-Solving Tips**

* 1. When comparing RationalNumbers, you can cast the numerator to a double and then divide by the denominator to determine the value of that RationalNumber as a double. The <=, >=, > and != operators can be implemented in terms of == and <.
  2. To implement the arithmetic operators, use the following formulas:

Addition: (a/b) + (c/d) = (ad + bc) / (bd).

Subtraction: (a/b)- (c/d) = (ad - bc) / (bd).

Multiplication: (a/b) \* (c/d) = (ac) / (bd).

Division: (a/b) / (c/d) = (ad) / (bc).

Remember to check for division by zero.

**V Your Solution**

// Lab 3: RationalNumber.h

// RationalNumber class definition.

#ifndef RATIONAL\_NUMBER\_H

#define RATIONAL\_NUMBER\_H

class RationalNumber

{

public:

RationalNumber( int = 0, int = 1 ); // default constructor

/\* Write prototype for operator + \*/

RationalNumber operator+(const RationalNumber &) const;

/\* Write prototype for operator - \*/

RationalNumber operator-(const RationalNumber &) const;

/\* Write prototype for operator \* \*/

RationalNumber operator\*(const RationalNumber &) const;

/\* Write prototype for operator / \*/

RationalNumber operator/(const RationalNumber &) const;

// relational operators

/\* Write prototype for operator > \*/

bool operator>(const RationalNumber &) const;

/\* Write prototype for operator < \*/

bool operator<(const RationalNumber &) const;

/\* Write prototype for operator >= \*/

bool operator>=(const RationalNumber &) const;

/\* Write prototype for operator <= \*/

bool operator<=(const RationalNumber &) const;

// equality operators

/\* Write prototype for operator == \*/

bool operator==(const RationalNumber &) const;

/\* Write prototype for operator != \*/

bool operator!=(const RationalNumber &) const;

void printRational() const; // display rational number

private:

int numerator; // private variable numerator

int denominator; // private variable denominator

void reduction(); // function for fraction reduction

}; // end class RationalNumber

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2012 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Lab 3: RationalTest.cpp

// RationalNumber test program.

#include <iostream>

using namespace std;

#include "RationalNumber.h"

int main()

{

RationalNumber c( 7, 3 ), d( 3, 9 ), x;

c.printRational();

cout << " + ";

d.printRational();

cout << " = ";

x = c + d; // test overloaded operators + and =

x.printRational();

cout << '\n';

c.printRational();

cout << " - ";

d.printRational();

cout << " = ";

x = c - d; // test overloaded operators - and =

x.printRational();

cout << '\n';

c.printRational();

cout << " \* ";

d.printRational();

cout << " = ";

x = c \* d; // test overloaded operators \* and =

x.printRational();

cout << '\n';

c.printRational();

cout << " / ";

d.printRational();

cout << " = ";

x = c / d; // test overloaded operators / and =

x.printRational();

cout << '\n';

c.printRational();

cout << " is:\n";

// test overloaded greater than operator

cout << ( ( c > d ) ? " > " : " <= " );

d.printRational();

cout << " according to the overloaded > operator\n";

// test overloaded less than operator

cout << ( ( c < d ) ? " < " : " >= " );

d.printRational();

cout << " according to the overloaded < operator\n";

// test overloaded greater than or equal to operator

cout << ( ( c >= d ) ? " >= " : " < " );

d.printRational();

cout << " according to the overloaded >= operator\n";

// test overloaded less than or equal to operator

cout << ( ( c <= d ) ? " <= " : " > " );

d.printRational();

cout << " according to the overloaded <= operator\n";

// test overloaded equality operator

cout << ( ( c == d ) ? " == " : " != " );

d.printRational();

cout << " according to the overloaded == operator\n";

// test overloaded inequality operator

cout << ( ( c != d ) ? " != " : " == " );

d.printRational();

cout << " according to the overloaded != operator" << endl;

} // end main

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2012 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Lab 3: RationalNumber.cpp

// RationalNumber member-function definitions.

#include <cstdlib>

#include <iostream>

using namespace std;

#include "RationalNumber.h"

// RationalNumber constructor sets n and d and calls reduction

/\* Implement the RationalNumber constructor. Validate d first to ensure that

it is a positive number and set it to 1 if not. Call the reduction utility

function at the end \*/

RationalNumber::RationalNumber(int num,int den)

{

numerator = num;

if(den > 0) denominator = den;

reduction();

}

// overloaded + operator

/\* Write definition for overloaded operator + \*/

RationalNumber RationalNumber::operator+(const RationalNumber & right) const

{

int num, den;

num = this->numerator \* right.denominator + this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

return RationalNumber(num, den);

}

// overloaded - operator

/\* Write definition for overloaded operator - \*/

RationalNumber RationalNumber::operator-(const RationalNumber & right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

return RationalNumber(num, den);

}

// overloaded \* operator

/\* Write definition for overloaded operator \* \*/

RationalNumber RationalNumber::operator\*(const RationalNumber & right) const

{

int num, den;

num = this->numerator \* right.numerator;

den = this->denominator \* right.denominator;

return RationalNumber(num, den);

}

// overloaded / operator

/\* Write definition for overloaded operator /. Check if the client is

attempting to divide by zero and report an error message if so \*/

RationalNumber RationalNumber::operator/(const RationalNumber & right) const

{

if(right.numerator == 0){

cout<<"分母为0";

return RationalNumber(0,0);

}

int num, den;

num = this->numerator \* right.denominator;

den = this->denominator \* right.numerator;

return RationalNumber(num, den);

}

// overloaded > operator

/\* Write definition for operator > \*/

bool RationalNumber::operator>(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den > 0)

return true;

else

return false;

}

// overloaded < operator

/\* Write definition for operator < \*/

bool RationalNumber::operator<(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den < 0)

return true;

else

return false;

}

// overloaded >= operator

/\* Write definition for operator >= \*/

bool RationalNumber::operator>=(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den >= 0)

return true;

else

return false;

}

// overloaded <= operator

/\* Write definition for operator <= \*/

bool RationalNumber::operator<=(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den <= 0)

return true;

else

return false;

}

// overloaded == operator

/\* Write definition for operator == \*/

bool RationalNumber::operator==(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den == 0)

return true;

else

return false;

}

// overloaded != operator

/\* Write definition for operator != \*/

bool RationalNumber::operator!=(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den != 0)

return true;

else

return false;

}

// function printRational definition

void RationalNumber::printRational() const

{

if ( numerator == 0 ) // print fraction as zero

cout << numerator;

else if ( denominator == 1 ) // print fraction as integer

cout << numerator;

else

cout << numerator << '/' << denominator;

} // end function printRational

// function reduction definition

void RationalNumber::reduction()

{

int largest, gcd = 1; // greatest common divisor;

largest = ( numerator > denominator ) ? numerator: denominator;

for ( int loop = 2; loop <= largest; loop++ )

if ( numerator % loop == 0 && denominator % loop == 0 )

gcd = loop;

numerator /= gcd;

denominator /= gcd;

} // end function reduction

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2012 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

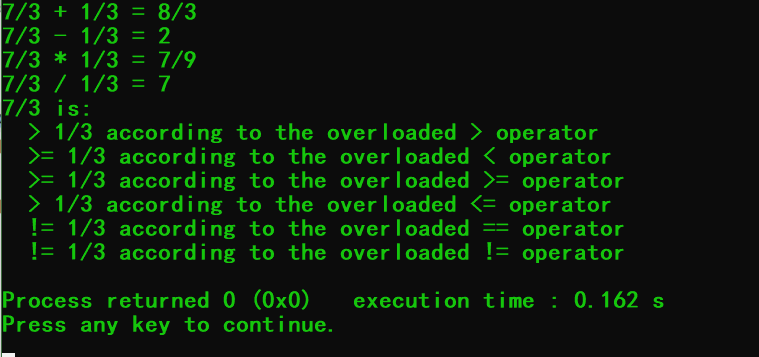
\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/



**VI Follow-Up Questions and Activities**

1. Rewrite the printRational member function as an overloaded << friend function.

// Lab 3: RationalNumber.h

// RationalNumber class definition.

#ifndef RATIONAL\_NUMBER\_H

#define RATIONAL\_NUMBER\_H

#include<bits/stdc++.h>

using namespace std;

class RationalNumber

{

friend void printRational(RationalNumber & right);

public:

RationalNumber( int = 0, int = 1 ); // default constructor

/\* Write prototype for operator + \*/

RationalNumber operator+(const RationalNumber &) const;

/\* Write prototype for operator - \*/

RationalNumber operator-(const RationalNumber &) const;

/\* Write prototype for operator \* \*/

RationalNumber operator\*(const RationalNumber &) const;

/\* Write prototype for operator / \*/

RationalNumber operator/(const RationalNumber &) const;

// relational operators

/\* Write prototype for operator > \*/

bool operator>(const RationalNumber &) const;

/\* Write prototype for operator < \*/

bool operator<(const RationalNumber &) const;

/\* Write prototype for operator >= \*/

bool operator>=(const RationalNumber &) const;

/\* Write prototype for operator <= \*/

bool operator<=(const RationalNumber &) const;

// equality operators

/\* Write prototype for operator == \*/

bool operator==(const RationalNumber &) const;

/\* Write prototype for operator != \*/

bool operator!=(const RationalNumber &) const;

// display rational number

private:

int numerator; // private variable numerator

int denominator; // private variable denominator

void reduction(); // function for fraction reduction

}; // end class RationalNumber

#endif

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2012 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Lab 3: RationalNumber.cpp

// RationalNumber member-function definitions.

#include <cstdlib>

#include <iostream>

using namespace std;

#include "RationalNumber.h"

// RationalNumber constructor sets n and d and calls reduction

/\* Implement the RationalNumber constructor. Validate d first to ensure that

it is a positive number and set it to 1 if not. Call the reduction utility

function at the end \*/

RationalNumber::RationalNumber(int num,int den)

{

numerator = num;

if(den > 0) denominator = den;

reduction();

}

// overloaded + operator

/\* Write definition for overloaded operator + \*/

RationalNumber RationalNumber::operator+(const RationalNumber & right) const

{

int num, den;

num = this->numerator \* right.denominator + this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

return RationalNumber(num, den);

}

// overloaded - operator

/\* Write definition for overloaded operator - \*/

RationalNumber RationalNumber::operator-(const RationalNumber & right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

return RationalNumber(num, den);

}

// overloaded \* operator

/\* Write definition for overloaded operator \* \*/

RationalNumber RationalNumber::operator\*(const RationalNumber & right) const

{

int num, den;

num = this->numerator \* right.numerator;

den = this->denominator \* right.denominator;

return RationalNumber(num, den);

}

// overloaded / operator

/\* Write definition for overloaded operator /. Check if the client is

attempting to divide by zero and report an error message if so \*/

RationalNumber RationalNumber::operator/(const RationalNumber & right) const

{

if(right.numerator == 0){

cout<<"分母为0";

return RationalNumber(0,0);

}

int num, den;

num = this->numerator \* right.denominator;

den = this->denominator \* right.numerator;

return RationalNumber(num, den);

}

// overloaded > operator

/\* Write definition for operator > \*/

bool RationalNumber::operator>(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den > 0)

return true;

else

return false;

}

// overloaded < operator

/\* Write definition for operator < \*/

bool RationalNumber::operator<(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den < 0)

return true;

else

return false;

}

// overloaded >= operator

/\* Write definition for operator >= \*/

bool RationalNumber::operator>=(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den >= 0)

return true;

else

return false;

}

// overloaded <= operator

/\* Write definition for operator <= \*/

bool RationalNumber::operator<=(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den <= 0)

return true;

else

return false;

}

// overloaded == operator

/\* Write definition for operator == \*/

bool RationalNumber::operator==(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den == 0)

return true;

else

return false;

}

// overloaded != operator

/\* Write definition for operator != \*/

bool RationalNumber::operator!=(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den != 0)

return true;

else

return false;

}

// function printRational definition

/\*void RationalNumber::printRational() const

{

if ( numerator == 0 ) // print fraction as zero

cout << numerator;

else if ( denominator == 1 ) // print fraction as integer

cout << numerator;

else

cout << numerator << '/' << denominator;

} // end function printRational

\*/

// function reduction definition

void RationalNumber::reduction()

{

int largest, gcd = 1; // greatest common divisor;

largest = ( numerator > denominator ) ? numerator: denominator;

for ( int loop = 2; loop <= largest; loop++ )

if ( numerator % loop == 0 && denominator % loop == 0 )

gcd = loop;

numerator /= gcd;

denominator /= gcd;

} // end function reduction

void printRational(RationalNumber & right)

{

int numerator = right.numerator;

int denominator = right.denominator;

if ( numerator == 0 ) // print fraction as zero

cout << numerator;

else if ( denominator == 1 ) // print fraction as integer

cout << numerator;

else

cout << numerator << '/' << denominator;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2012 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

// Lab 3: RationalTest.cpp

// RationalNumber test program.

#include <iostream>

using namespace std;

#include "RationalNumber.h"

int main()

{

RationalNumber c( 7, 3 ), d( 3, 9 ), x;

printRational(c);

cout << " + ";

printRational(d);

cout << " = ";

x = c + d; // test overloaded operators + and =

printRational(x);

cout << '\n';

printRational(c);

cout << " - ";

printRational(d);

cout << " = ";

x = c - d; // test overloaded operators - and =

printRational(x);

cout << '\n';

printRational(c);

cout << " \* ";

printRational(d);

cout << " = ";

x = c \* d; // test overloaded operators \* and =

printRational(x);

cout << '\n';

printRational(c);

cout << " / ";

printRational(d);

cout << " = ";

x = c / d; // test overloaded operators / and =

printRational(x);

cout << '\n';

printRational(c);

cout << " is:\n";

// test overloaded greater than operator

cout << ( ( c > d ) ? " > " : " <= " );

printRational(d);

cout << " according to the overloaded > operator\n";

// test overloaded less than operator

cout << ( ( c < d ) ? " < " : " >= " );

printRational(d);

cout << " according to the overloaded < operator\n";

// test overloaded greater than or equal to operator

cout << ( ( c >= d ) ? " >= " : " < " );

printRational(d);

cout << " according to the overloaded >= operator\n";

// test overloaded less than or equal to operator

cout << ( ( c <= d ) ? " <= " : " > " );

printRational(d);

cout << " according to the overloaded <= operator\n";

// test overloaded equality operator

cout << ( ( c == d ) ? " == " : " != " );

printRational(d);

cout << " according to the overloaded == operator\n";

// test overloaded inequality operator

cout << ( ( c != d ) ? " != " : " == " );

printRational(d);

cout << " according to the overloaded != operator" << endl;

} // end main

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2012 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

2. Make the RationalNumber class more robust by providing additional tests for division by zero in each of the relational operators that divides a numerator by a denominator.

// Lab 3: RationalNumber.cpp

// RationalNumber member-function definitions.

#include <cstdlib>

#include <iostream>

using namespace std;

#include "RationalNumber.h"

// RationalNumber constructor sets n and d and calls reduction

/\* Implement the RationalNumber constructor. Validate d first to ensure that

it is a positive number and set it to 1 if not. Call the reduction utility

function at the end \*/

RationalNumber::RationalNumber(int num,int den)

{

numerator = num;

if(den > 0) denominator = den;

reduction();

}

// overloaded + operator

/\* Write definition for overloaded operator + \*/

RationalNumber RationalNumber::operator+(const RationalNumber & right) const

{

int num, den;

num = this->numerator \* right.denominator + this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(den == 0)

{

den = 1;

cout<<"denominator = 0";

}

return RationalNumber(num, den);

}

// overloaded - operator

/\* Write definition for overloaded operator - \*/

RationalNumber RationalNumber::operator-(const RationalNumber & right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(den == 0)

{

den = 1;

cout<<"denominator = 0";

}

return RationalNumber(num, den);

}

// overloaded \* operator

/\* Write definition for overloaded operator \* \*/

RationalNumber RationalNumber::operator\*(const RationalNumber & right) const

{

int num, den;

num = this->numerator \* right.numerator;

den = this->denominator \* right.denominator;

if(den == 0)

{

den = 1;

cout<<"denominator = 0";

}

return RationalNumber(num, den);

}

// overloaded / operator

/\* Write definition for overloaded operator /. Check if the client is

attempting to divide by zero and report an error message if so \*/

RationalNumber RationalNumber::operator/(const RationalNumber & right) const

{

if(right.denominator == 0){

cout<<"分母为0";

return RationalNumber(0,0);

}

int num, den;

num = this->numerator \* right.denominator;

den = this->denominator \* right.numerator;

return RationalNumber(num, den);

}

// overloaded > operator

/\* Write definition for operator > \*/

bool RationalNumber::operator>(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den > 0)

return true;

else

return false;

}

// overloaded < operator

/\* Write definition for operator < \*/

bool RationalNumber::operator<(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den < 0)

return true;

else

return false;

}

// overloaded >= operator

/\* Write definition for operator >= \*/

bool RationalNumber::operator>=(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den >= 0)

return true;

else

return false;

}

// overloaded <= operator

/\* Write definition for operator <= \*/

bool RationalNumber::operator<=(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den <= 0)

return true;

else

return false;

}

// overloaded == operator

/\* Write definition for operator == \*/

bool RationalNumber::operator==(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den == 0)

return true;

else

return false;

}

// overloaded != operator

/\* Write definition for operator != \*/

bool RationalNumber::operator!=(const RationalNumber &right) const

{

int num, den;

num = this->numerator \* right.denominator - this->denominator \* right.numerator;

den = this->denominator \* right.denominator;

if(num \* den != 0)

return true;

else

return false;

}

// function printRational definition

/\*void RationalNumber::printRational() const

{

if ( numerator == 0 ) // print fraction as zero

cout << numerator;

else if ( denominator == 1 ) // print fraction as integer

cout << numerator;

else

cout << numerator << '/' << denominator;

} // end function printRational

\*/

// function reduction definition

void RationalNumber::reduction()

{

int largest, gcd = 1; // greatest common divisor;

largest = ( numerator > denominator ) ? numerator: denominator;

for ( int loop = 2; loop <= largest; loop++ )

if ( numerator % loop == 0 && denominator % loop == 0 )

gcd = loop;

numerator /= gcd;

denominator /= gcd;

} // end function reduction

void printRational(RationalNumber & right)

{

int numerator = right.numerator;

int denominator = right.denominator;

if ( numerator == 0 ) // print fraction as zero

cout << numerator;

else if ( denominator == 1 ) // print fraction as integer

cout << numerator;

else

cout << numerator << '/' << denominator;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* (C) Copyright 1992-2012 by Deitel & Associates, Inc. and \*

\* Pearson Education, Inc. All Rights Reserved. \*

\* \*

\* DISCLAIMER: The authors and publisher of this book have used their \*

\* best efforts in preparing the book. These efforts include the \*

\* development, research, and testing of the theories and programs \*

\* to determine their effectiveness. The authors and publisher make \*

\* no warranty of any kind, expressed or implied, with regard to these \*

\* programs or to the documentation contained in these books. The authors \*

\* and publisher shall not be liable in any event for incidental or \*

\* consequential damages in connection with, or arising out of, the \*

\* furnishing, performance, or use of these programs. \*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

3. Is it possible to add another overloaded operator> function that returns a pointer to the larger of the two rational numbers? Why or why not?

可以，可以返回较大的分数的引用。