**Lab Exercises**

Class: 软件1801 Name: 高福锦 StuID:201816040117

Date: 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.

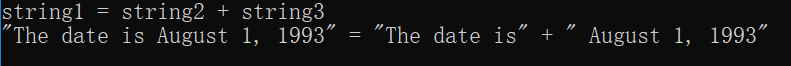
**III Sample Output**



**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**



**// Lab 1: String.h**

**// Header file for class String.**

**#ifndef STRING\_H**

**#define STRING\_H**

**#include <iostream>**

**#include <cstring>**

**#include <cassert>**

**using namespace std;**

**class String**

**{**

**friend ostream &operator<<( ostream &output, const String &s );**

**public:**

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

**String( const String & ); // copy constructor**

**~String(); // destructor**

**const String &operator=( const String & );**

**const String operator+( const String & );**

**private:**

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

**int length; // string length**

**}; // end class String**

**#endif**

**// Lab 1: String.cpp**

**// Member-function definitions for String.cpp**

**#include <iostream>**

**using namespace std;**

**#include <cstring> // strcpy and strcat prototypes**

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

**// conversion constructor: convert a char \* to String**

**String::String( const char \* const zPtr )**

**{**

**length = strlen( zPtr ); // compute length**

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

**assert( sPtr != 0 ); // terminate if memory not allocated**

**strcpy( sPtr, zPtr ); // copy literal to object**

**} // end String conversion constructor**

**// copy constructor**

**String::String( const String &copy )**

**{**

**length = copy.length; // copy length**

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

**assert( sPtr != 0 ); // ensure memory allocated**

**strcpy( sPtr, copy.sPtr ); // copy string**

**} // end String copy constructor**

**// destructor**

**String::~String()**

**{**

**delete [] sPtr; // reclaim string**

**} // end destructor**

**// overloaded = operator; avoids self assignment**

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

**{**

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

**{**

**delete [] sPtr; // prevents memory leak**

**length = right.length; // new String length**

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

**assert( sPtr != 0 ); // ensure memory allocated**

**strcpy( sPtr, right.sPtr ); // copy string**

**}**

**else**

**cout << "Attempted assignment of a String to itself\n";**

**return \*this; // enables concatenated assignments**

**} // end function operator=**

**// concatenate right operand and this object and store in temp object**

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

**{**

**String temp;**

**temp.length=length+right.length;**

**temp.sPtr = new char[ temp.length + 1 ];**

**assert( sPtr != 0 ); // terminate if memory not allocated**

**strcpy(temp.sPtr,sPtr);**

**strcat(temp.sPtr,right.sPtr);**

**return temp;**

**} // end function operator+**

**// overloaded output operator**

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

**{**

**output << s.sPtr;**

**return output; // enables concatenation**

**} // end function operator<<**

**// Lab 1: StringCat.cpp**

**// Demonstrating overloaded + operator that does not modify operands**

**#include <iostream>**

**using namespace std;**

**#include "String.h"**

**int main()**

**{**

**String string1, string2( "The date is" );**

**String string3( " August 1, 1993" );**

**// test overloaded operators**

**cout << "string1 = string2 + string3\n";**

**string1 =string2+string3;**

**cout << "\"" << string1 << "\" = \"" << string2 << "\" + \""**

**<< string3 << "\"" << endl;**

**} // end main**

**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.]

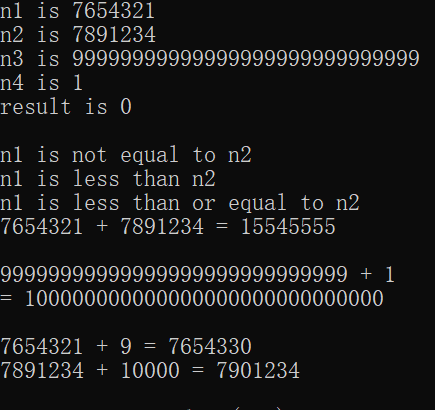
**III Sample Output**



**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 &op2 )const;

bool operator!=( const HugeInt &op2 )const;

bool operator<( const HugeInt &op2 )const;

bool operator<=( const HugeInt &op2 )const;

bool operator>( const HugeInt &op2 )const;

bool operator>=( const HugeInt &op2 )const;

int getLength() const;

private:

short integer[ 30 ];

}; // end class HugeInt

#endif

// 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 "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 i;

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

if ( integer[i] != 0 )

break; // break when first digit is reached

return 30-i; // 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

bool HugeInt::operator==( const HugeInt &op2 )const

{

if(getLength()>op2.getLength()||getLength()<op2.getLength())

return false;

if(getLength()==op2.getLength())

{

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

if(integer[i]!=op2.integer[i])

{

return false;

break;

}

}

return true;

}

// inequality operator; HugeInt != HugeInt

bool HugeInt::operator!=( const HugeInt &op2 )const

{

if(integer==op2.integer)

return false;

return true;

}

// less than operator; HugeInt < HugeInt

bool HugeInt::operator<( const HugeInt &op2 )const

{

if(getLength()>op2.getLength())

return false;

if(getLength()<op2.getLength())

return true;

if(getLength()==op2.getLength())

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

{

if(integer[i]<op2.integer[i])

{

return true;

}

if(integer[i]>op2.integer[i])

{

return false;

}

}

return true;

}

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

bool HugeInt::operator<=( const HugeInt &op2 )const

{

if(HugeInt::operator<(op2)||HugeInt::operator==(op2))

return true;

return false;

}

// greater than operator; HugeInt > HugeInt

bool HugeInt::operator>( const HugeInt &op2 )const

{

if(HugeInt::operator<=(op2))

return false;

return true;

}

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

bool HugeInt::operator>=( const HugeInt &op2 )const

{

if(HugeInt::operator>(op2)||HugeInt::operator==(op2))

return true;

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<<

// 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

**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).

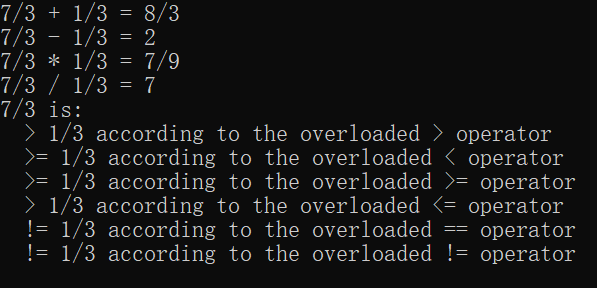
Subtraction: (a/b)- (c/d) = (ad - bc) / (bd).

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

RationalNumber operator+(const RationalNumber&);

RationalNumber operator-(const RationalNumber&);

RationalNumber operator\*(const RationalNumber&);

RationalNumber operator/(const RationalNumber&);

// relational operators

bool operator>(const RationalNumber&);

bool operator<(const RationalNumber&);

bool operator>=(const RationalNumber&);

bool operator<=(const RationalNumber&);

// equality operators

bool operator==(const RationalNumber&);

bool operator!=(const RationalNumber&);

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

// 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

RationalNumber::RationalNumber(int a,int b)

{

numerator =a;

if(b==0)

denominator=1;

else if(b>0)

denominator=b;

else if(b<0)

{

denominator=b\*(-1);

numerator=numerator\*(-1);

}

reduction();

}

// overloaded + operator

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

{

RationalNumber temp;

temp.denominator=denominator\*rigth.denominator;

temp.numerator=(numerator\*rigth.denominator)+(rigth.numerator\*denominator);

temp.reduction();

return temp;

}

// overloaded - operator

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

{

RationalNumber temp;

temp.denominator=denominator\*rigth.denominator;

temp.numerator=(numerator\*rigth.denominator)-(rigth.numerator\*denominator);

temp.reduction();

return temp;

}

// overloaded \* operator

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

{

RationalNumber temp;

temp.denominator=denominator\*rigth.denominator;

temp.numerator=numerator\*rigth.numerator;

temp.reduction();

return temp;

}

// overloaded / operator

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

{

RationalNumber temp;

temp.denominator=denominator\*rigth.numerator;

temp.numerator=numerator\*rigth.denominator;

if(temp.denominator==0)

throw invalid\_argument("error ,denominator can't be zero");

temp.reduction();

return temp;

}

// overloaded > operator

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

{

RationalNumber temp1,temp2;

temp1.denominator=denominator\*rigth.denominator;

temp2.denominator=denominator\*rigth.denominator;

temp1.numerator=numerator\*rigth.denominator;

temp2.numerator=rigth.numerator\*denominator;

if((temp1.numerator-temp2.numerator)>0)

return true;

return false;

}

// overloaded < operator

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

{

RationalNumber temp1,temp2;

temp1.denominator=denominator\*rigth.denominator;

temp2.denominator=denominator\*rigth.denominator;

temp1.numerator=numerator\*rigth.denominator;

temp2.numerator=rigth.numerator\*denominator;

if((temp1.numerator-temp2.numerator)<0)

return true;

return false;

}

// overloaded >= operator

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

{

if(RationalNumber::operator<(rigth))

return false;

return true;

}

// overloaded <= operator

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

{

if(RationalNumber::operator>(rigth))

return false;

return true;

}

// overloaded == operator

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

{

RationalNumber temp1,temp2;

temp1.denominator=denominator\*rigth.denominator;

temp2.denominator=denominator\*rigth.denominator;

temp1.numerator=numerator\*rigth.denominator;

temp2.numerator=rigth.numerator\*denominator;

if((temp1.numerator==temp2.numerator)&&(temp1.denominator==temp2.denominator))

return true;

return false;

}

// overloaded != operator

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

{

if(RationalNumber::operator==(rigth))

return false;

return true;

}

// 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

// 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

**VI Follow-Up Questions and Activities**

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

ostream &operator<<(ostream &output,const RationalNumber &c)

{

output<<c.numerator<<"/"<<c.denominator;

return output;

}//end function operator <<

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.

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

{

RationalNumber temp;

temp.denominator=denominator\*rigth.denominator;

temp.numerator=(numerator\*rigth.denominator)+(rigth.numerator\*denominator);

temp.reduction();

if(temp.denominator==0)

throw invalid\_argument("denominator can't be 0");

return temp;

}

// overloaded - operator

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

{

RationalNumber temp;

temp.denominator=denominator\*rigth.denominator;

temp.numerator=(numerator\*rigth.denominator)-(rigth.numerator\*denominator);

temp.reduction();

if(temp.denominator==0)

throw invalid\_argument("denominator can't be 0");

return temp;

}

// overloaded \* operator

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

{

RationalNumber temp;

temp.denominator=denominator\*rigth.denominator;

temp.numerator=numerator\*rigth.numerator;

temp.reduction();

if(temp.denominator==0)

throw invalid\_argument("denominator can't be 0");

return temp;

}

// overloaded / operator

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

{

RationalNumber temp;

temp.denominator=denominator\*rigth.numerator;

temp.numerator=numerator\*rigth.denominator;

if(temp.denominator==0)

throw invalid\_argument("error ,denominator can't be zero");

temp.reduction();

return temp;

}

// overloaded > operator

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?

It’s possible.

RationalNumber RationalNumber::operator>(const RationalNumber& rigth)

{

RationalNumber temp1,temp2;

temp1.denominator=denominator\*rigth.denominator;

temp2.denominator=denominator\*rigth.denominator;

temp1.numerator=numerator\*rigth.denominator;

temp2.numerator=rigth.numerator\*denominator;

if((temp1.numerator-temp2.numerator)>0)

return temp1;

return temp2;

}