**Lab Exercises**

Class: 软件1803 Name: 苏博文 StuID:\_\_\_\_\_\_\_\_\_201816040327\_\_\_\_\_\_\_

Date: 2019.11.03 Score:

**Lab Exercise 1 — String Concatenation**

**I Lab Objectives**

In this lab, you will practice:

• Overloading the + operator to allow String objects to be concatenated.

• Writing function prototypes for overloaded operators.

• 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**

• The overloaded + operator should be a member function of class String and should take one parameter, a const reference to a String.

• The + operator function should use return type String.

• The strcat function can be used to concatenate pointer-based strings.

**V Your Solution**

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

{

String temp;

temp.length = this->length + A.length;

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

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

strcpy(temp.sPtr,this->sPtr);

strcat(temp.sPtr,A.sPtr);

return temp;

// return temp;

}

cout << "\"" << string1 << "\" = \"" << string2 << "\" + \""<< string3 << "\"" << endl;

private:

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

int length; // string length

**Lab Exercise 2 — Huge Integer**

**I Lab Objectives**

In this lab, you will practice:

• Overloading arithmetic and comparison operators to enhance a huge integer class, HugeInt**.**

• Writing function prototypes for overloaded operators.

• 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**

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

{

return !(\*this==A);

}

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

{

int i;

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

{

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

return false;

else if(this->integer[i]==A.integer[i])

continue;

else

return true;

}

}

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

{

return(\*this==A||\*this<A);

}

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

{

return !(\*this<=A);

}

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

{

return (\*this>A||\*this==A);

}

int getLength() const;

private:

short integer[ 30 ];

**Lab Exercise 3 — Rational Numbers**

**I Lab Objectives**

In this lab, you will practice:

• Overloading operators to create a class capable of storing rational numbers (fractions) and performing rational number arithmetic.

• Writing function prototypes for overloaded operators.

• Implementing overloaded operator functions.

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

• Overloading the << operator.

• Making a class more robust to prevent runtime errors.

**II Description of the Problem**

Create a classRationalNumber(fractions) with the following capabilities:

• 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.

• Overload the addition, subtraction, multiplication and division operators for this class.

• Overload the relational and equality operators for this class.

**III Sample Output**

**IV Problem-Solving Tips**

• 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 <.

• 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**

RationalNumber::RationalNumber(int k,int n)

{

if(n<=0)

denominator=1;

else

denominator=n;

numerator=k;

reduction();

}

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

{

RationalNumber p;

if(denominator==k.denominator)

{

p.denominator=k.denominator;

p.numerator=numerator+k.numerator;}

if(denominator!=k.denominator)

{

p.denominator=denominator\*k.denominator;

p.numerator=numerator\*k.denominator+k.numerator\*denominator;}

return p;

}

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

{

RationalNumber p;

if(denominator=k.denominator)

{

p.denominator=k.denominator;

p.numerator=numerator-k.numerator;

}

if(denominator!=k.denominator)

{

p.denominator=denominator\*k.denominator;

p.numerator=numerator\*k.denominator-k.numerator\*denominator;

}

return p;

}

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

{

RationalNumber p;

p.denominator=denominator\*k.denominator;

p.numerator=numerator\*k.numerator;

}

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

{

RationalNumber p;

p.denominator=denominator\*k.numerator;

p.numerator=numerator\*k.denominator;

}

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

{

if(denominator==k.denominator)

{

if(numerator>k.denominator)

return true;

}

else if(numerator\*k.denominator>k.numerator\*denominator)

{

return true;

}

else

return false;

}

// overloaded < operator

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

{

if(denominator==k.denominator)

{

if(numerator<k.numerator)

return true;

}

else if(numerator\*k.denominator<k.numerator\*denominator)

{

return true;

}

else

return false;

}

// overloaded >= operator

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

{

if(denominator==k.denominator)

{

if(numerator>k.denominator&&numerator==k.numerator)

return true;

}

else if(numerator\*k.denominator>k.numerator\*denominator&&numerator\*k.denominator==k.numerator\*denominator)

{

return true;

}

else

return false;

}

// overloaded <= operator

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

{

if(denominator==k.denominator)

{

if(numerator<k.denominator&&numerator==k.numerator)

return true;

}

else if(numerator\*k.denominator<k.numerator\*denominator&&numerator\*k.denominator==k.numerator\*denominator)

{

return true;

}

else

return false;

}

// overloaded == operator

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

{

if(denominator==k.denominator&&numerator==k.numerator)

return true;

else if(denominator!=k.denominator)

{

if(numerator\*k.denominator==k.numerator\*denominator)

return true;

}

else

return false;

}

// overloaded != operator

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

{

if(denominator==k.denominator&&numerator!=k.numerator)

{

return true;

}

else if(numerator\*k.denominator!=k.numerator\*denominator)

{

return true;

}

else

return false;

}

**VI Follow-Up Questions and Activities**

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

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

{

if ( a.numerator == 0 )

output << a.numerator;

else if ( a.denominator == 1 )

output << a.numerator;

else

output << a.numerator << '/' << a.denominator;

}

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

{

if ( a.numerator == 0 )

output << a.numerator;

else if ( a.denominator == 1 )

output << a.numerator;

else

output << a.numerator << '/' << a.denominator;

}

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.

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?

Yes, the function passes two scores, by calling the operator> that was previously overloaded. If the previous operator> returns true, the function returns a pointer to the first score. Otherwise, it returns a pointer to the second score’s pointer