**Lab Exercises**

Class: Name: NaHCOx

Date: Score:

说明：Lab1 对应中文版十一章中的课后题目11.8;

Lab2 对应中文版十一章中的课后题目11.14;

Lab3 对应中文版十一章中的课后题目11.15;

**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 text, 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 Your Solution**

**String.h**

声明加号的运算符重载

String operator+( const String& ) const;

**String.cpp**

给出函数定义

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

{

   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;

}

**String.cpp**

测试重载的加号

string1 = string2 + string3;

***Output:***

string1 = string2 + string3

"The date is August 1, 1993" = "The date is" + " August 1, 1993"

**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. 11.8–11.10.** 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 Your Solution**

**HugeInt.h**

声明6个关系运算符

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;

**HugeInt.cpp**

第52行， for循环内部声明的变量i在循环结束后被尝试访问，此处是一个bug

for ( int i = 0; i <= 29; i++ )

↓修改为

int i;

for ( i = 0; i <= 29; i++ )

给出小于"<"号和等于号"=="的重载函数，并由此推出其他四个函数

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

{

    for ( int i = 0; i < 30; i++ )

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

            return false;

    return true;

}

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

{

    return !this->operator==(right);

}

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

{

    int i = 0;

    for ( ; integer[i] && right.integer[i] && i<29; i++ );

    return integer[i] < right.integer[i];

}

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

{

    return !right.operator<(\*this);

}

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

{

    return right < \*this;

}

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

{

    return !this->operator<(right);

}

**HugeIntTest.cpp**

（无修改）

***Output:***

n1 is 7654321

n2 is 7891234

n3 is 99999999999999999999999999999

n4 is 1

result is 0

n3

n1 is not equal to n2

n1 is less than or equal to n2

n1 is greater than or equal to n2

7654321 + 7891234 = 15545555

99999999999999999999999999999 + 1

= 100000000000000000000000000000

7654321 + 9 = 7654330

7891234 + 10000 = 7901234

**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
2. rational number arithmetic.
3. Writing function prototypes for overloaded operators.
4. 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 class ***RationalNumber*** (fractions) with the following capabilities:

a) 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.

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

c) Overload the relational and equality operators for this class.

**III Sample Output**



**IV Your Solution**

**RationalNumber.h**

给出4个算术运算符、6个关系运算符的重载函数定义

RationalNumber operator+( const RationalNumber& ) const;

RationalNumber operator-( const RationalNumber& ) const;

RationalNumber operator\*( const RationalNumber& ) const;

RationalNumber operator/( const RationalNumber& ) const;

bool operator<( const RationalNumber& ) const;

bool operator>( const RationalNumber& ) const;

bool operator<=( const RationalNumber& ) const;

bool operator>=( const RationalNumber& ) const;

bool operator==( const RationalNumber& ) const;

bool operator!=( const RationalNumber& ) const;

**RationalNumber.cpp**

给出构造函数定义

RationalNumber::RationalNumber( int n, int d )

{

    if ( d == 0 )   {

        cout << "denominator error" << endl;

        numerator = 0, denominator = 1;

    }

    else if ( n == 0 ) {

        numerator = 0, denominator = 1;

    } else {

        numerator = n, denominator = d;

        reduction();

    }

}

给出加号"+"、乘号"\*"的定义，并由此给出减号"-"、除号"/"的定义

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

{

    RationalNumber temp;

    if ( denominator == right.denominator ) {

        temp.denominator = denominator;

        temp.numerator = numerator + right.numerator;

    } else {

        temp.denominator = denominator \* right.denominator;

        temp.numerator =

            numerator \* right.denominator + denominator \* right.numerator;

    }

    if ( temp.denominator < 0 ) {

        temp.numerator = -temp.numerator;

        temp.denominator = -temp.denominator;

    }

    temp.reduction();

    return temp;

}

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

{

    return \*this + RationalNumber( -right.numerator, right.denominator );

}

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

{

    RationalNumber temp;

    temp.numerator = numerator \* right.numerator;

    temp.denominator = denominator \* right.denominator;

    temp.reduction();

    return temp;

}

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

{

    return \*this \* RationalNumber( right.denominator, right.numerator );

}

给出小于"<"号、等于"=="号的定义，并由此给出剩余4个关系运算符的定义

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

{

    return right < \*this;

}

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

{

    return double(numerator) / denominator <

           double(right.numerator) / right.denominator;

}

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

{

    return !( \*this < right );

}

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

{

    return !( right < \*this );

}

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

{

    return numerator == right.numerator && denominator == right.denominator;

}

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

{

    return !this->operator==(right);

}

**RationalTest.cpp**

（无修改）

***Output:***

7/3 + 1/3 = 8/3

7/3 - 1/3 = 2

7/3 \* 1/3 = 7/9

7/3 / 1/3 = 7

7/3 is:

> 1/3 according to the overloaded > operator

>= 1/3 according to the overloaded < operator

>= 1/3 according to the overloaded >= operator

> 1/3 according to the overloaded <= operator

!= 1/3 according to the overloaded == operator

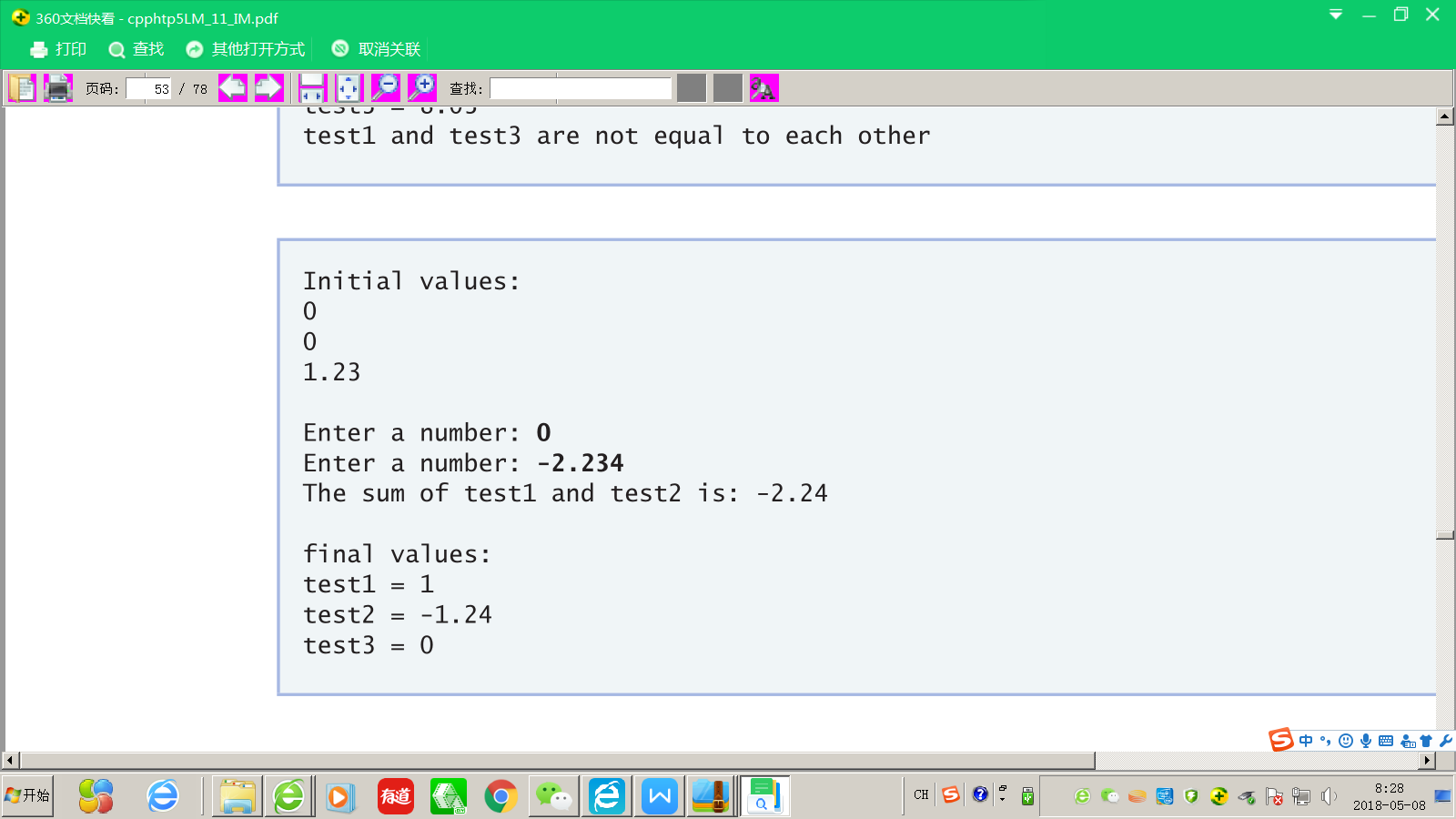
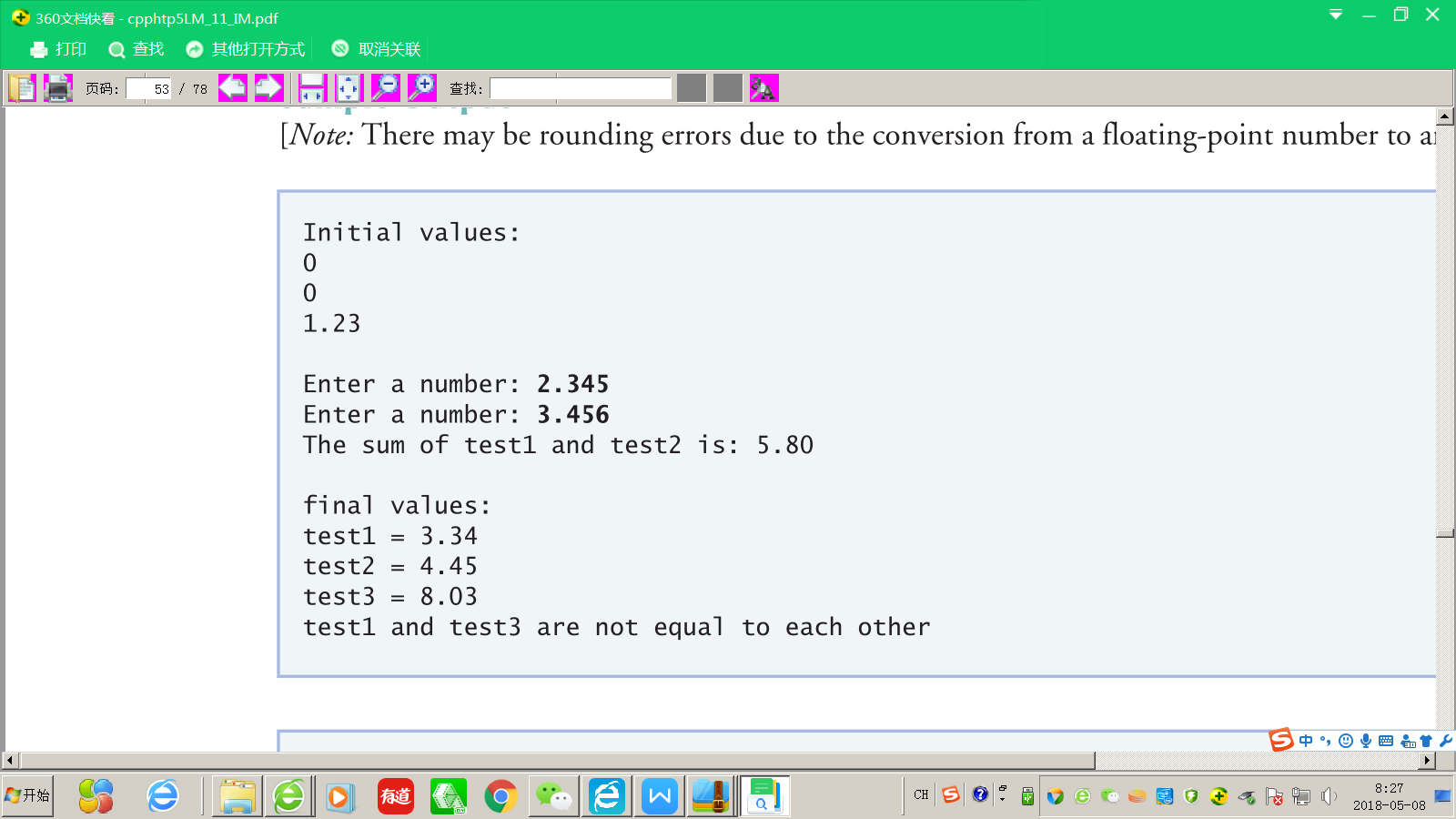
!= 1/3 according to the overloaded != operator

**Debugging**

**问题描述**

本部分程序不能正常运行，修改所有的语法错误使得程序能编译成功。运行程序，输出运行结果，并与下面的输出样例比较，修改所有的逻辑错误。

**输出样例**



**答案**

**Decimal.h**

这次dubug错误较多，先给出正确代码，再说明各处错误，浅灰色底为修改部分

（为节省篇幅，仅给出类的声明部分）

class Decimal

{

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

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

public:

    Decimal( double = 0.0 );

    void setInteger( double );

    void setDecimal( double );

    Decimal& operator=( const Decimal& );

    Decimal operator+( const Decimal& ) const;

    Decimal& operator+=( const Decimal& );

    Decimal& operator++();

    Decimal operator++( int );

    bool operator==( const Decimal& ) const;

    bool operator!=( const Decimal& ) const;

private:

    double integer;

    double decimal;

};

1. 首先，友元函数声明习惯上往往集中在类的开头，且流插入运算符">>"的返回值类型、流提取运算符"<<"的参数表均有错误。
2. "+"和"+="运算符声明时没有operator关键字
3. "+="运算符的返回值应是对象的引用
4. 所有的二元运算符，传参时采取传引用的方式效率更高，传常引用更安全
5. "+="运算符不应该声明为const，
6. 与之对应，根据最小权限原则，"+"、"=="运算符应该声明为const
7. 后置自增运算符"++"的参数表应该为一个int
8. debugging.cpp中调用了"!="运算符，这里补上声明

**Decimal.cpp**

Decimal::Decimal( double n )

{

    decimal = modf( n, &integer );

} // end class Decimal constructor

// function operator<< definition

ostream& operator<<( ostream& output, const Decimal& d )

{

    double n = 0;

    n = floor( d.decimal \* 100 );

    if ( n < 0 )

        n = -n;

    if ( d.decimal != 0 ) {

        output << round( d.integer ) << ".";

        if ( n > 10 )

            output << n;

        else

            output << "0" << n;

    } // end if

    else

        output << d.integer;

    return output;

} // end function operator<<

// function operator>> definition

istream& operator>>( istream &input, Decimal &d )

{

    double n;

    cout << "Enter a number: ";

    input >> n;

    d.decimal = modf( n, &d.integer );

    return input;

} // end function operator>>

// function operator= definition

Decimal &Decimal::operator=( const Decimal& d )

{

    integer = d.integer;

    decimal = d.decimal;

    return \*this;

} // end function operator=

// function setDecimal definition

void Decimal::setDecimal( double d )

{

    decimal = d;

} // end function setDecimal

// function setInteger definition

void Decimal::setInteger( double i )

{

    integer = i;

} // end function setInteger

// function operator+ definition

Decimal Decimal::operator+( const Decimal& d ) const

{

    Decimal result;

    result.setDecimal( decimal + d.decimal );

    result.setInteger( integer + d.integer );

    if ( result.decimal >= 1 )

    {

        result.decimal--;

        result.integer++;

    } // end if

    else if ( result.decimal <= -1 )

    {

        result.decimal++;

        result.integer--;

    } // end if

    return result;

} // end function operator+

// function operator+= definition

Decimal& Decimal::operator+=( const Decimal& d )

{

    \*this = \*this + d;

    return \*this;

} // end function operator+=

// function operator++ definition

Decimal& Decimal::operator++()

{

    integer++;

    return \*this;

} // end function operator++

// function operator++ definition

Decimal Decimal::operator++( int )

{

    Decimal temp = \*this;

    integer++;

    return temp;

} // end function operator++

// function operator== definition

bool Decimal::operator==( const Decimal& d ) const

{

    return ( integer == d.integer && decimal == d.decimal );

} // end function operator==

bool Decimal::operator!=( const Decimal& d ) const

{

    return !this->operator==(d);

} // end function operator==

1. 所有的函数头匹配头文件里做出的更改，包括添加了"!="运算符的重载
2. **流插入运算符"<<"**重载中，第三行的n应该取绝对值；
3. 输出小数的整数部分时应该使用round函数四舍五入而不是向下取整
4. 没有返回ostream&，使得串联调用不能进行
5. **流提取运算符">>"**重载中，应该用istream对象名调用">>"运算符而不是通过类名
6. 访问成员变量时没有指明对象
7. **"+="运算符**的定义中错误地使用了"+="运算符本身
8. **前缀自增运算符"++"**返回值错误，应该返回\*this
9. **后缀自增运算符"++"**返回值错误，应该返回temp

**Debugging.cpp**

（无更改）

***Input1:***

2.345

3.456

***Output1:***

Initial values:

0

0

1.23

Enter a number: 2.345

Enter a number: 3.456

The sum of test1 and test2 is: 5.80

final values:

test1 = 3.34

test2 = 4.45

test3 = 8.03

test1 and test3 are not equal to each other

***Input2:***

0

-2.234

***Output2:***

Initial values:

0

0

1.23

Enter a number: 0

Enter a number: -2.234

The sum of test1 and test2 is: -2.24

final values:

test1 = 1

test2 = -1.24

test3 = 0

test1 and test3 are not equal to each other