

C++作业

罗悦 2016220304022

习题 7.1:

请读者仿照【例 7-2】，为第六章设计的 `cstring` 类重载 `+` 和 `+=` 运算符。这两个运算符的功能是完成两个字符串的链接。请注意运算符函数的参数和返回类型，以及对内存的处理。

所编写的程序如下所示:

```
#ifndef __MYSTRING_H__
#define __MYSTRING_H__
#endif
#include <iostream>
using namespace std;

class cstring                                //类 cstring
{
public:
    cstring();                               //构造函数
    cstring(const char *_str);               //重载构造函数
    cstring(const cstring &_str);           //复制构造函数
    friend class PP;                         //定义 PP 作为友元类
    ~cstring();

public:
    const cstring operator +(const cstring &_str);
    const cstring operator +=(const cstring &_str);
    void operator =(const cstring &_str);

public:
    friend istream & operator >>(istream &_in, cstring &_str);
    friend ostream & operator <<(ostream &_out, cstring &_str);
private:
    char *str;
    int length;
};

cstring::cstring()
```

```

{
    str = new char;
    str[0] = '\0';
    length = 0;
}

cstring::cstring(const char *_str)
{
    length = strlen(_str);
    str = new char[length + 1];
    str[length] = '\0';
    strncpy(str, _str, length+1);
}

cstring::cstring(const cstring &_str)
{
    length = _str.length;
    str = new char[length + 1];
    str[length] = '\0';
    strncpy(str, _str.str, length+1);
}

cstring::~~cstring(){}

const cstring cstring::operator +(const cstring &_str)//运算符“+”的重载
{
    int newLength = length + _str.length;
    char * newStr = new char[newLength + 1];
    newStr[newLength] = '\0';
    strncpy(newStr, str, length);
    strncat(newStr, _str.str, _str.length);

    return cstring(newStr);
}

const cstring cstring::operator +=(const cstring &_str)//运算符“+=”的重载
{
    length+= _str.length;
    int newLength = length + _str.length;
    char * newStr = new char[newLength + 1];
    strncat(str, _str.str, _str.length);
    return cstring(str);
}

```

```

void cstring::operator =(const cstring &_str)           //运算符“=”的重载
{
    str = _str.str;
    length = _str.length;
}

istream & operator >>(istream &_in, cstring &_str) //重载“>>”
{
    _in >> _str.str;
    _str.length = strlen(_str.str);
    return _in;
}

ostream & operator <<(ostream &_out, cstring &_str) //“<<”,打印字符串和长度
{
    _out << _str.str << endl << "length: " << _str.length;
    return _out;
}

int main()
{
    cstring mStr;           //定义对象 mStr
    cstring mStr_1("Luo"); //定义对象 mStr_1
    cstring mStr_2("Yue");  //定义对象 mStr_2

    cout << "mStr: " << mStr << endl;
    cout << "mStr_1: " << mStr_1 << endl;
    cout << "mStr_2: " << mStr_2 << endl;
    cout << "\n" << endl;
    cout << "mStr_1+mStr_2:"<<endl;
    mStr = mStr_1 + mStr_2;
    cout << mStr << endl;

    mStr += mStr_1;
    cout << "mStr += mStr_1:"<<endl;
    cout << mStr << endl;
    return 0;
}

```

如上程序所示，我设计了一个 string 类，这个类对运算符 “+” 和运算符 “+=” 进行了重载，实现了可以对字符串的简单操作。测试结果如下图所示：

```
"C:\Users\n'n\Documents\C-Free\Temp\  
mStr:  
length: 0  
mStr_1: Luo  
length: 3  
mStr_2: Yue  
length: 3  
  
mStr_1+mStr_2:  
LuoYue  
length: 6  
mStr += mStr_1:  
LuoYueLuo  
length: 9  
请按任意键继续. . .
```

其中定义对象 mStr 时，长度为 0，定义 mStr_1 和 mStr_2 时，分别把字符串“Luo”和“Yue”赋给了私有变量*str。最后把 mStr_1 和 mStr_2 两“字符串”相加的值赋给 mStr 并打印出来，然后再进行 mStr += mStr_1 操作验证运算符“+=”，实现“LuoYue”和“Luo”的字符串相加，得到结果并打印出来。

习题 7.4：请读者为复数类重载乘法和除法运算符。注意多编写几个重载版本，以适应不同参与运算的类型。例如，复数乘复数，浮点数乘复数，以及复数乘浮点数等。

所编写的程序如下所示：

复数类：myComplex.h

```
#include <iostream>  
#include <cmath>  
#include "math.h"  
#include "stdlib.h"  
using namespace std;
```

```

class myComplex
{
private:
    double Real;          //定义实部
    double Imaginary;     //定义虚部
    double Modulus;       //定义模
public:
    myComplex(){           //构造函数
        Real=5;
        Imaginary=6;
    }
    myComplex(double a){   //重载构造函数
        Real=a;
        Imaginary=6;
    }
    myComplex(double a,double b){ //重载构造函数
        Real=a;
        Imaginary=b;
    }
    myComplex(const myComplex& v){ //复制构造函数
        Real=v.Real;
        Imaginary=v.Imaginary;
        Modulus=v.Modulus;
    }
    double &getReal(){      //返回实部
        return Real;
    }
    double &getImaginary(){ //返回虚部
        return Imaginary;
    }
    double &getModulus(){   //返回模
        Modulus=sqrt(Real*Real+Imaginary*Imaginary);
        return Modulus;
    }
    myComplex& operator=(const myComplex& r){ //类对象的赋值
        Real=r.Real;
        Imaginary=r.Imaginary;
        return *this;
    }
    friend myComplex operator+(myComplex &m,myComplex &n);
    friend myComplex operator-(myComplex &m,myComplex &n);
    friend myComplex operator*(myComplex &m,myComplex &n);
    friend myComplex operator/(myComplex &m,myComplex &n);
    friend ostream& operator<<(ostream& os,myComplex& c);
}

```

```

        friend istream& operator>>(istream& is, myComplex& c);
};
myComplex operator+(myComplex &m, myComplex &n) { //重载运算符 "+"
    myComplex temp(m.getReal()+n.getReal(), m.getImaginary()+n.getImaginary());
    return temp;
}
myComplex operator-(myComplex &m, myComplex &n) { //重载运算符 "-"
    myComplex temp(m.getReal()-n.getReal(), m.getImaginary()-n.getImaginary());
    return temp;
}
myComplex operator*(myComplex &m, myComplex &n) { //重载运算符 "*"
    myComplex
    temp(m.getReal()*n.getReal()-m.getImaginary()*n.getImaginary(), m.getReal()*n.getImaginary()+m.getImaginary()*n.getReal());
    return temp;
}
myComplex operator*(myComplex &m, float n) { //重载运算符 "*", 修改重载版本
    myComplex temp(m.getReal()*n, m.getImaginary()*n);
    return temp;
}
myComplex operator*(float m, myComplex &n) { //重载运算符 "*", 修改重载版本
    myComplex temp(n.getReal()*m, n.getImaginary()*m);
    return temp;
}
myComplex operator/(myComplex &m, myComplex &n) { //重载运算符 "/"
    myComplex
    temp((m.getReal()*n.getReal()+m.getImaginary()*n.getImaginary())/(n.getReal()*n.getReal()+n.getImaginary()*n.getImaginary()), (n.getReal()*m.getImaginary()-m.getReal()*n.getImaginary())/(n.getReal()*n.getReal()+n.getImaginary()*n.getImaginary()));
    return temp;
}
myComplex operator/(myComplex &m, float n) { //重载运算符 "/" 修改重载版本
    myComplex temp(m.getReal()/n, m.getImaginary()/n);
    return temp;
}
myComplex operator/(float m, myComplex &n) { //重载运算符 "/"
    myComplex
    temp(m*n.getReal()/(n.getReal()*n.getReal()+n.getImaginary()*n.getImaginary()), (-m*n.getImaginary())/(n.getReal()*n.getReal()+n.getImaginary()*n.getImaginary()));
    return temp;
}
ostream& operator<<(ostream& os, myComplex& c) { //重载运算符 "<<"
    os << '(' << c.getReal() << ', ' << c.getImaginary() << ', ' << c.getModulus() << ')';
    return os;
}

```

```

}
istream& operator>>(istream& is, myComplex& c) { //重载运算符">>"
    is >> c.getReal() >> c.getImaginary() >> c.getModulus();
    return is;
}

```

主函数 myComplex.cpp:

```

#include <iostream>
#include <cmath>
#include "mycomplex.h"
using namespace std;

int main() {
    myComplex a1(9,10);           //定义一个对象 a1
    myComplex a2(7,8);           //定义一个对象 a2
    cout<<"定义了两个复数类对象 a1、a2 如下所示，括号中从左到右分别为实
部、虚部、模："<<endl;
    cout<<a1<<endl;
    cout<<a2<<"\n"<<endl;

    myComplex a3;                //定义一个对象 a3
    cout<<"定义了一个复数类对象 a3："<<endl;

    a3=a1+a2;                    //检测运算符"+"
    cout<<"运算 a3=a1+a2 后，a3 的实部、虚部和模分别为："<<endl;
    cout<<a3<<"\n"<<endl;

    a3=a1-a2;                    //检测运算符 "-"
    cout<<"运算 a3=a1-a2 后，a3 的实部、虚部和模分别为："<<endl;
    cout<<a3<<"\n"<<endl;

    a3=a1*a2;                    //检测运算符 "*"
    cout<<"运算 a3=a1*a2 后，a3 的实部、虚部和模分别为："<<endl;
    cout<<a3<<"\n"<<endl;

    a3=a3*3.3;                   //检测运算符 "*" 对于复数乘浮点数的重载
    cout<<"运算 a3=a3*3.3 后，a3 的实部、虚部和模分别为："<<endl;
    cout<<a3<<"\n"<<endl;

    a3=0.3*a3;                   //检测运算符 "*" 对于浮点数乘复数的重载
    cout<<"运算 a3=0.3*a3 后，a3 的实部、虚部和模分别为："<<endl;
    cout<<a3<<"\n"<<endl;

    a3=a1/a2;                    //检测运算符 "/"

```

```

    cout<<"运算 a3=a1/a2 后，a3 的实部、虚部和模分别为："<<endl;
    cout<<a3<<"\n"<<endl;

    a3=a3/0.5;                //检测运算符"/"对于浮点数除以浮点数的重载
    cout<<"运算 a3=a3/0.5 后，a3 的实部、虚部和模分别为："<<endl;
    cout<<a3<<"\n"<<endl;

    a3=15/a3;                //检测运算符"/"对于浮点数除以复数的重载
    cout<<"运算 a3=7/a3 后，a3 的实部、虚部和模分别为："<<endl;
    cout<<a3<<"\n"<<endl;

    return 0;
}

```

如上程序所示，我定义了一个 myComplex.h 头文件，这个头文件主要包括了一个复数类 myComplex，这个类定义了复数的实部、虚部和模的值为私有变量，并且都为 double 类型。在这个类中，定义了一个无参数的构造函数，并且进行了重载，又定义了两个分别有一个参数和两个参数的构造函数。且定义了成员函数来访问私有变量，返回私有变量的值。同时重载了赋值运算符“=”。运用友元的方式，在类外定义了双目运算符的重载函数，分别有“+”“-”“*”“/”，并且对“*”“/”两个运算符进行了多个重载函数的书写，来实现浮点数与复数的乘法与除法运算同时定义了运算符“<<”与“>>”。如上主程序所示，我定义了一个对象 a1，实部为 9，虚部 10，一个对象 a2，实部为 7，虚部 8，并分别打印了出来。之后又定义了一个对象 a3 用来进行双目运算符的重载测试工作，在之后分别进行了“a3=a1+a2”“a3=a1-a2”“a3=a1*a2”“a3=a3*3.3”“a3=0.3*a3”“a3=a1/a2”“a3=a3/0.5”“a3=15/a3”八个操作并且将计算后的 a3 的实部虚部和模的值打印出来。测试结果如下图所示：


```
"C:\Users\n'n\Desktop\My bad\C++\作业三\myComplex.exe"
定义了两个复数类对象a1、a2如下所示，括号中从左到右分别为实部、虚部、模：
(9, 10, 13.4536)
(7, 8, 10.6301)

定义了一个复数类对象a3：
运算a3=a1+a2后，a3的实部、虚部和模分别为：
(16, 18, 24.0832)

运算a3=a1-a2后，a3的实部、虚部和模分别为：
(2, 2, 2.82843)

运算a3=a1*a2后，a3的实部、虚部和模分别为：
(-17, 142, 143.014)

运算a3=a3*3.3后，a3的实部、虚部和模分别为：
(-56.1, 468.6, 471.946)

运算a3=0.3*a3后，a3的实部、虚部和模分别为：
(-16.83, 140.58, 141.584)

运算a3=a1/a2后，a3的实部、虚部和模分别为：
(1.26549, -0.0176991, 1.26561)

运算a3=a3/0.5后，a3的实部、虚部和模分别为：
(2.53097, -0.0353982, 2.53122)

运算a3=7/a3后，a3的实部、虚部和模分别为：
(5.92541, 0.0828729, 5.92599)

请按任意键继续. . .
```

运算结果和实验结果如上图所示

习题 7.7：请读者为 `cstring` 类重载 `<<` 和 `>>` 运算符。注意：在输入 `>>` 操作中，如何处理字符串已有的资源。

所编写程序如下所示：

```
#ifndef __MYSTRING_H__
#define __MYSTRING_H__
#endif
#include <iostream>
using namespace std;

class PP;

class cstring                                //类 cstring
{
```

```

public:
    cstring();                //构造函数
    cstring(const char *_str); //重载构造函数
    cstring(const cstring &_str); //复制构造函数
    friend class PP;          //定义 PP 作为友元类
    ~cstring();

public:
    const cstring operator +(const cstring &_str);
    void operator =(const cstring &_str);

public:
    friend istream & operator >>(istream &_in, cstring &_str);
    friend ostream & operator <<(ostream &_out, cstring &_str);

private:
    char *str;
    int length;
};

cstring::cstring()
{
    str = new char;
    str[0] = '\0';
    length = 0;
}

cstring::cstring(const char *_str)
{
    length = strlen(_str);
    str = new char[length + 1];
    str[length] = '\0';
    strncpy(str, _str, length+1);
}

cstring::cstring(const cstring &_str)
{
    length = _str.length;
    str = new char[length + 1];
    str[length] = '\0';
    strncpy(str, _str.str, length+1);
}

cstring::~~cstring(){}

const cstring cstring::operator +(const cstring &_str)//运算符“+”的重载
{

```

```

    int newLength = length + _str.length;
    char * newStr = new char[newLength + 1];
    newStr[newLength] = '\0';
    strncpy(newStr, str, length);
    strncat(newStr, _str.str, _str.length);

    return cstring(newStr);
}

void cstring::operator =(const cstring &_str)           //运算符“=”的重载
{
    str = _str.str;
    length = _str.length;
}

istream & operator >>(istream &_in, cstring &_str) //重载“>>”
{
    _in >> _str.str;
    _str.length = strlen(_str.str);
    return _in;
}

ostream & operator <<(ostream &_out, cstring &_str) //“<<”,打印字符串和长度
{
    _out << _str.str << endl << "length: " << _str.length;
    return _out;
}

class PP                                     //类 PP
{
public:
    void putstr(cstring &s){                 //putstr 函数
        cout<<s.str<<endl;
    }
};

int main()
{
    cstring mStr;                            //定义对象 mStr
    cstring mStr_0;
    cstring mStr_1("Luo");                  //定义对象 mStr_1
    cstring mStr_2("Yue");                  //定义对象 mStr_2

    cout << "mStr: " << mStr << endl;
    cout << "mStr_1: " << mStr_1 << endl;
}

```

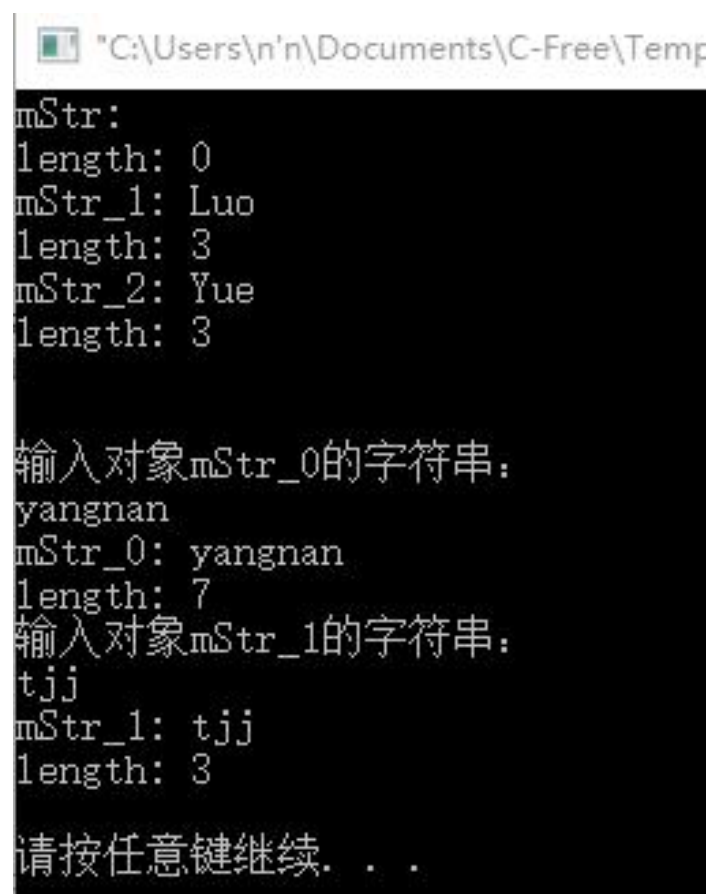
```

    cout << "mStr_2: " << mStr_2 << endl;
    cout << "\n" << endl;
    cout << "输入对象 mStr_0 的字符串: " << endl;
    cin >> mStr_0;
    cout << "mStr_0: " << mStr_0 << endl;
    cout << "输入对象 mStr_1 的字符串: " << endl;
    cin >> mStr_1;
    cout << "mStr_1: " << mStr_1 << endl;
    PP p;                                //定义一个 PP 类的对象 p
    p.putstr(mStr);                       //调用 putstr 函数打印对象 mStr 的字符串

    return 0;
}

```

如上程序所示，我在 string 类中添加了析构函数和重载构造函数。这个类对简单的运算符进行了重载，实现了可以对字符串的简单操作，对“<<”和“>>”运算符进行了重载，并设计了几个对象进行验证。测试结果如下图所示：



```

"C:\Users\n'n\Documents\C-Free\Temp
mStr:
length: 0
mStr_1: Luo
length: 3
mStr_2: Yue
length: 3

输入对象mStr_0的字符串:
yangnan
mStr_0: yangnan
length: 7
输入对象mStr_1的字符串:
tjj
mStr_1: tjj
length: 3

请按任意键继续. . .

```

如上所示，其中定义对象 mStr 时，长度为 0，定义 mStr_1 和 mStr_2 时，分别把字符串“Luo”和“Yue”赋给了私有变量*str。得到结果并打印出来，同时验证了运算符“>>”，进行了简单的字符串输入，其中 mStr 对象无原有字符串，mStr_1 有原有资源，但输入过后进行了覆盖，实现了原有资源的释放。

习题 8.4：请读者修改【例 8-2】，使在 tiger 类中访问 counter 合法，并验证在不同访问控制情况下的结果。

编写程序如下所示：

```
#include<iostream>
#include<string>
using namespace std;

class carnivore{
protected:
    string name;
public:
    static int counter;
    carnivore(string n="carnivore"){
        name=n;
        counter++;
    }
    void prey(){
        cout<<name<<"preys"<<endl;
    }
    string what()const{
        return name;
    }
};
int carnivore::counter=0;

class felid:public carnivore{
protected:
    bool sliPupil;
public:
```

```

        felid(string n="felid",bool s=false){
            name=n;
            sliPupil=s;
        }
};

class tiger:public felid{
public:
    tiger(string n="tiger",bool s=false){
        name=n;
        sliPupil=s;
    }
    void roar(){
        cout<<name<<" roars"<<endl;
        cout<<"counter:"<<counter<<endl;
    }
};

int main(){
    carnivore c;
    felid f;
    tiger t;
    t.roar();
    return 0;
}

```

如上程序所示，我在基类 carnivore 的共有段中定义了一个静态成员变量 counter，并且在构造函数中对 counter 进行 counter++操作，使得派生类中每一次调用构造函数时都会对 counter 进行加一操作，且 felid 使用 public 继承基类 carnivore，类 tiger 使用 public 继承 felid，并在 tiger 类中的 roar 函数中打印 counter 验证访问。实验结果如下图所示：



```

"C:\Users\n'n\Documents\
tiger roars
counter:3
请按任意键继续. . .

```

如上图所示为给出程序的执行情况，但我同时还验证了其他许多继承方式的访问情况，如 felid 私有继承 carnivore，carnivore 的静态成员 counter 声明为私有段成员等继承方式时，都无法在 tiger 类中访问计数器 counter，但只要是共有段成员变量且 public 继承就一定能访问。

习题 8.5：请读者考虑下列两个概念：弦乐器（String）和小提琴（Violin）。二者之间是一种什么关系？如何用程序代码来描述这种关系？

答：Violin 是 String 的一种，即 Violin 是 String 的子类，在程序可以说 Violin 是 String 的子类或派生类。

用程序代码描述如下所示：

```
#include<iostream>
#include<string>
using namespace std;

class String{ //定义弦类乐器基类
protected:
    string name;
public:
    String(string n="String"){
        name=n;
    }
    void show(){
        cout<<"该乐器类别： "<<endl;
        cout<<name<<endl;
        cout<<"-----"<<endl;
    }
};
```

```

class Violin:public String{
private:
    string Vname;
public:
    Violin(string n="Violin"){
        Vname=n;
    }
    void show(){
        cout<<"该乐器类别: "<<endl;
        cout<<name<<endl;
        cout<<Vname<<endl;
        cout<<"-----"<<endl;
    }
};

int main(){
    String s;
    Violin v;
    s.show();
    v.show();
    return 0;
}

```

如上程序所示，我定义了一个基类 String 类，它是弦类乐器的基类，并且派生了一个子类 Violin 继承基类 String，并在两个类中的构造函数分别为各自的名字进行初始化，并书写了一个打印该乐器类别的 show 函数来进行测试，测试结果如下图所示：



```

"C:\Users\n'n\Documents\C-Free\Tem
该乐器类别:
String
-----
该乐器类别:
String
Violin
-----
请按任意键继续. . .

```

如上图所示，我定义了一个基类对象 s，和一个 Violin 类对象 v，并

分别对两个对象的类别进行了打印，其中对象 s 输入弦类乐器，v 属于弦类乐器种的小提琴类。