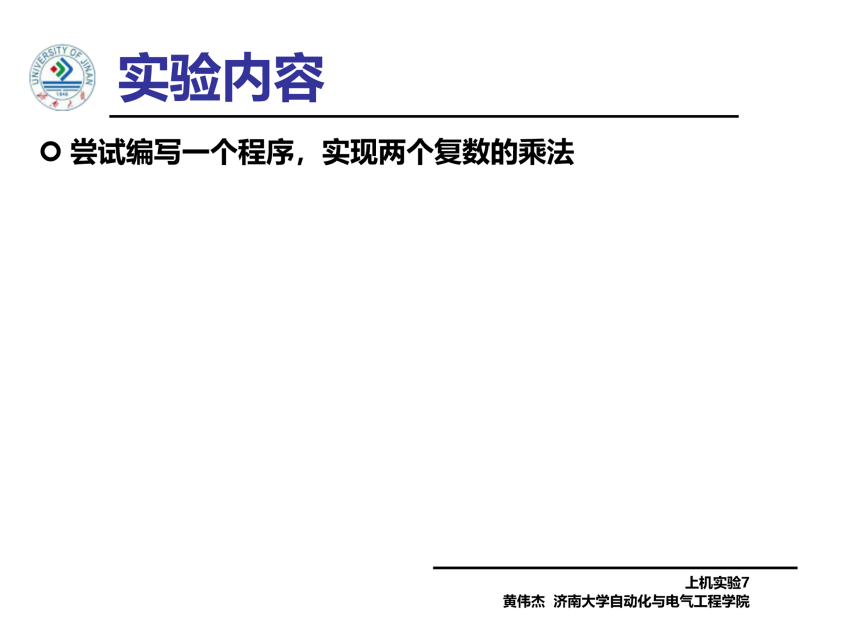
Copyright ©2021-2099 ChenJiacheng. All rights reserved

**实验要求：**



**程序：**

#include<iostream>

using namespace std;

class Complex {

public:

double real;

double imag;

Complex(double r = 0, double i = 0)

{

real = r; imag = i;

}

};

Complex operator\*(Complex co1, Complex co2)

{

Complex temp;

temp.real = co1.real \* co2.real - co1.imag \* co2.imag;

temp.imag = co1.imag \* co2.real + co2.imag \* co1.real;

return temp;

}

int main()

{

cout << "请分别输入两个复数的实部和虚部。" << endl;

double x[2], y[2];

cin >> x[0] >> y[0] >> x[1] >> y[1];

Complex com1(x[0], y[0]), com2(x[1], y[1]), total;

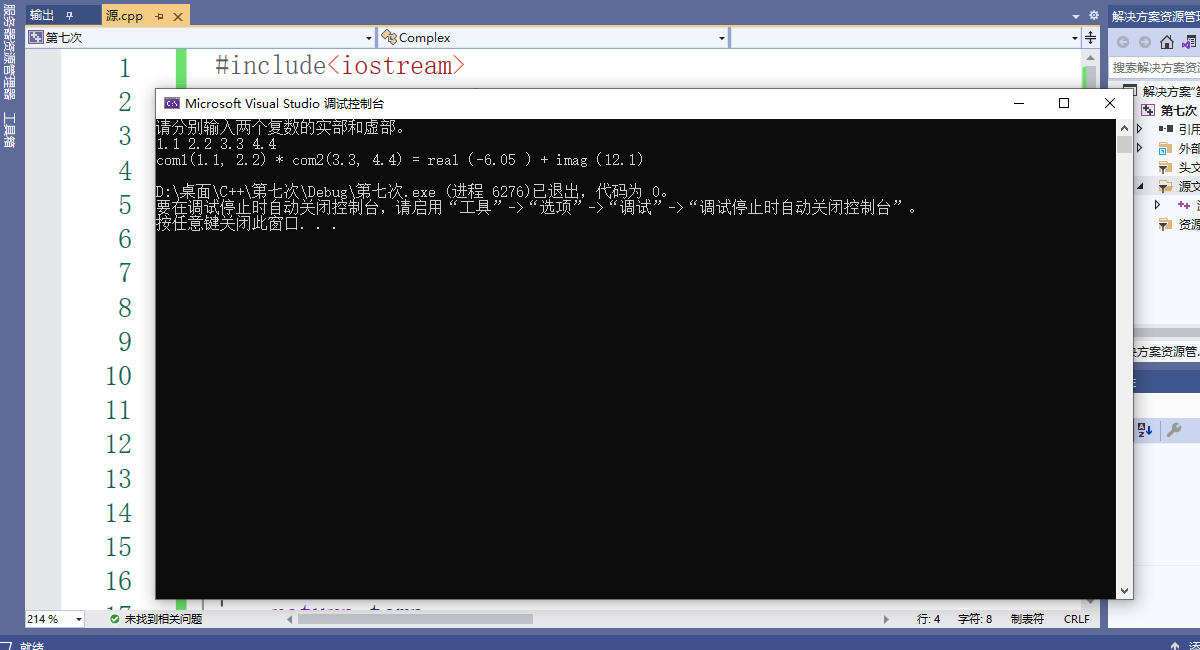
total = com1 \* com2;

cout << "com1(1.1, 2.2) \* com2(3.3, 4.4) = real（" << total.real << " " << "）+ imag（" << total.imag << "）" << endl;

return 0;

}

实验程序输出结果：



**实验总结：**

第一次编译出现错误：输出复数相乘结果不对。

运算符重载是通过创建运算符函数实现的，运算符函数定义了重载的运算符将要进行的操作。运算符函数的定义与其他函数的定义类似，惟一的区别是运算符函数的函数名是由关键字operator和其后要重载的运算符符号构成的。运算符函数定义的一般格式如下：  
  
  <返回类型说明符> operator <运算符符号>(<参数表>)  
  
{  
  
     <函数体>

查阅资料后知道有两处错误，其一复数乘法格式写错了，其二输出结果时把实部与虚部写反了。

认识到了运算符重载的好处：

从开发的角度而言，运算符重载的存在是为了提高开发效率，增加代码的可复用性，很多时候反而是简化了问题。C++强大的原因之一是支持泛型编程。泛型编程使得算法可以独立于数据类型存在。自定义的数据类型通过操作符重载具有了和内建类型同样的使用接口，然后C++的模板加持下，你的算法可以利用这种接口一致性，实现泛化。举个排序的例子，有的时候我们不仅是需要对整数数组来排序，常常也会碰到需要按字母对字符串数组实现排序。通过重载布尔运算符，就可以实现一个排序算法既可用于整数排序，也可以不经修改用于字符串排序，而不是要为两种数据类型写两套排序函数。