（一二二）运算符重载

本章重点是**类设计技术**，而不是通用原理。

C++的重点是**多使用**。

正如我之前写了这么多的学习笔记一样，几乎绝大部分的代码，都是我根据书上的知识，然后理解自行再创造写下的。虽然遇见了不少问题，但解决后，下次再遇见这类情况，解决起来很轻松，绝大多数都可以直接靠记忆书写代码，即使个别忘记了，回头翻一番笔记，基本也能解决，需要靠百度去解决的，已经很少了。

运算符重载是一种形式的C++多态（就像函数重载那样，同名但调用不同的函数）。

要重载运算符，需要使用被称为运算符函数的特殊函数形式。运算符函数格式如下：

**operator 运算符 (argument-list);**

例如：operator +(); 表示重载加号运算符，而operator -()表示重载减号运算符。

其中，运算符必须是有效的C++运算符，而不是是自己捏造的（例如+-\*/之类就是有效的，而@就是无效的）。

但需要注意的是，定义运算符有时需在类中定义（**非必须，具体看下面的运算符重载的限制**），对该类对象有效。运算符前面是的调用该方法的对象，参数中是被调用的对象。

例如：

class Mm

{

int a;

public:

...... ;

Mm operator + (const Mm &bb)

{return this->a + bb.a;}

}

这里实际上就是把+号运算符重载为对对象有用的加号运算符了。

如代码：

#include<iostream>

#include<string>

class Skill //技能类

{

std::string name; //技能名称

double jilv; //几率

double dam; //伤害

public:

Skill(std::string name\_ = "无技能", double jilv\_ = 0.0, double dam\_ = 0.0) { name = name\_;jilv = jilv\_;dam = dam\_; } //默认构造函数，默认几率和伤害都为0

void show();

Skill operator +(const Skill&b); //技能融合函数，将2个技能融合起来变成第三个技能

};

int main()

{

using namespace std;

Skill one("拔刀斩", 20, 500);

Skill two("半月斩", 15, 400);

one.show();

two.show();

Skill three = one + two; //新对象为两个对象之和

cout << "技能已融合！新技能get！" << endl;

three.show();

system("pause");

return 0;

}

void Skill::show()

{

std::cout << "技能：" << name << " 的发动几率为 " << jilv << " %，伤害为： " << dam << " 点。" << std::endl;

}

Skill Skill::operator+(const Skill&b)

{

Skill another;

another.name = name; //名字延续加号前的

another.jilv = (jilv + b.jilv)\*1.25; //几率为两个几率之和，乘以1.2

if (another.jilv > 100)another.jilv = 100; //如果几率大于100，则为100

another.dam = dam + b.dam\*0.5; //伤害为第一个伤害加上第二个伤害的一半

return another; //返回对象（不是引用）

}

输出：

技能：拔刀斩 的发动几率为 20 %，伤害为： 500 点。

技能：半月斩 的发动几率为 15 %，伤害为： 400 点。

技能已融合！新技能get！

技能：拔刀斩 的发动几率为 43.75 %，伤害为： 700 点。

请按任意键继续. . .

总结：

①运算符重载的关键，是运算符左边的调用重载函数，右边的作为参数，所以在码代码的时候，需要注意顺序。

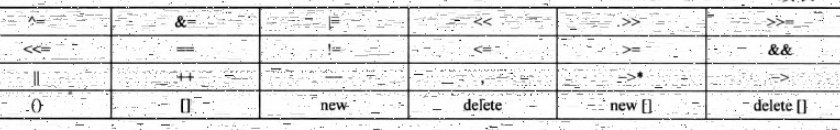
例如，one+two时，按照类方法，新对象three是one的名字，如果是two+one，新对象three是two的名字。

当然，具体是什么，根据函数定义来决定。

**运算符重载的限制：**

多数C++运算符（参见表11.1，如下图，不过看不清……）都可以用这样的方式重载。





但也有一些限制，例如：重载的运算符不必是成员函数，但必须至少有一个操作数是用户定义的类型。

具体限制如下：

①重载后的运算符，必须至少有一个操作数是用户定义的类型（否则不知道你使用的是运算符重载，在运算符重载的函数定义里，也容易出问题），这将防止用户为标准类型重载运算符。

因此，不能把减法运算符重载为2个标准类型（如int或者double）的值的和，而不是他们的差。

②使用运算符时不能违反运算符原来的句法规则。例如，+-\*/以及%这五种运算符，需要两个数来参与。你不能只给他用一个数。例如a+b是可以的，但是+a这种是不行的。而像地址运算符&就是只对一个数使用，如&dizhi。

另外，运算符即使被重载，其运算优先级也与原来的运算符的优先级相同，例如a+b/c，即使对+进行运算符重载，也依然是先计算b/c，再计算a+ (b/c)的结果。

③不能创建新的运算符，只能用已有的

④不能重载下面的运算符。

（1）sizeof sizeof运算符

（2）. ←这是一个句号，成员运算符，比如结构名.结构内变量

（3）.\* ←句号和乘号，成员指针运算符（没见过）

（4）:: ←两个冒号，作用域解析运算符

（5）?: 条件运算符，比如： a>b?c:d

（6）typeid ←一个RTTI运算符（没见过）

（7）const\_cast ←强制类型转换运算符（没见过这种用法）

（8）dynamic\_cast ←强制类型转换运算符（没见过这种用法）

（9）reinterpret\_cast ←强制类型转换运算符（没见过这种用法）

（10）static\_cast ←强制类型转换运算符（没见过这种用法）

而上面那个图片中的运算符，基本都可以用（反正大部分都能用，不能用的是少数）。

⑤大部分运算符：可以通过成员、非成员函数进行重载；

以下几个运算符，**只能通过成员函数**进行重载：

（1）= ←赋值运算符

（2）() ←函数调用运算符

（3）[] ←下标运算符（为什么叫下标？）

（4）-> ←通过指针访问类成员的运算符（不懂，是指this->私有成员这样么？）

⑥运算符重载最好遵循一般逻辑，例如不要把加法运算符（+）重载为两个对象各个值的差这种违反直觉的事情。

⑦重载运算符面对不同对象时，需要定义不同类型的重载运算符函数。

例如，一个对象 + 另一个对象，和一个对象 + 一个int值/double值，是不能使用同一个函数的。

如代码：

#include<iostream>

#include<string>

class Skill //技能类

{

std::string name; //技能名称

double jilv; //几率

double dam; //伤害

public:

Skill(std::string name\_ = "无技能", double jilv\_ = 0.0, double dam\_ = 0.0) { name = name\_;jilv = jilv\_;dam = dam\_; } //默认构造函数，默认几率和伤害都为0

void show()const;

Skill operator +(const Skill&b)const; //技能融合函数，将2个技能融合起来变成第三个技能

void operator\*=(const double &b); //技能增强函数，重载运算符\*=

void operator/=(const double &b); //另一种形式的增强，通过降低概率增加伤害，重载运算符/=

};

int main()

{

using namespace std;

Skill one("拔刀斩", 20, 500);

Skill two("半月斩", 15, 400);

one.show();

two.show();

Skill three = one + two; //新对象为两个对象之和

cout << "技能已融合！新技能get！" << endl;

three.show();

cout << "现将增强你的技能，增强发动几率和伤害，增加比例为1.5。" << endl;

three \*= 1.5; //重载\*=运算符

three.show();

cout << "现在将降低概率，增强伤害，系数为：2" << endl;

three /= 2; //重载/=运算符

three.show();

cout << "bye." << endl;

system("pause");

return 0;

}

void Skill::show()const

{

std::cout << "技能：" << name << " 的发动几率为 " << jilv << " %，伤害为： " << dam << " 点。" << std::endl;

}

Skill Skill::operator+(const Skill&b)const

{

Skill another;

another.name = name; //名字延续加号前的

another.jilv = (jilv + b.jilv)\*1.25; //几率为两个几率之和，乘以1.2

if (another.jilv > 100)another.jilv = 100; //如果几率大于100，则为100

another.dam = dam + b.dam\*0.5; //伤害为第一个伤害加上第二个伤害的一半

return another; //返回对象（不是引用）

}

void Skill::operator\*=(const double &b)

{

jilv \*= b; //几率和伤害都乘以系数

dam \*= b;

}

void Skill::operator/=(const double &b)

{

jilv /= b; //几率除系数

dam \*= b; //伤害乘系数

}

输出：

技能：拔刀斩 的发动几率为 20 %，伤害为： 500 点。

技能：半月斩 的发动几率为 15 %，伤害为： 400 点。

技能已融合！新技能get！

技能：拔刀斩 的发动几率为 43.75 %，伤害为： 700 点。

现将增强你的技能，增强发动几率和伤害，增加比例为1.5。

技能：拔刀斩 的发动几率为 65.625 %，伤害为： 1050 点。

现在将降低概率，增强伤害，系数为：2

技能：拔刀斩 的发动几率为 32.8125 %，伤害为： 2100 点。

bye.

请按任意键继续. . .

（一二二）友元函数

由于C++控制了对类对象的访问（例如不允许访问私有成员）。于是，通常公有类方法（例如：成员函数）提供唯一的访问途径。

这样保护了私有成员，但同时又因为这种限制太严格，以致于不适合特定的编程问题。

在这种情况下，C++提供了另外一种形式的访问权限：**友元**。

友元有三种：

①友元函数；

②友元类；

③友元成员函数。

通过让函数成为类的友元，可以赋予该函数与类的成员函数具有相同的访问权限（例如可以访问、修改私有成员）。

**为什么需要友元函数：**

以类成员函数为例：

Skill Skill::operator+(const Skill&b)const

{

Skill another;

another.name = name; //名字延续加号前的

another.jilv = (jilv + b.jilv)\*1.25; //几率为两个几率之和，乘以1.2

if (another.jilv > 100)another.jilv = 100; //如果几率大于100，则为100

another.dam = dam + b.dam\*0.5; //伤害为第一个伤害加上第二个伤害的一半

return another; //返回对象（不是引用）

}

这个函数是运算符重载，重载对象是加号。本函数如果修改，有以下可能性：

①当我们面对2个类对象时，使用本函数正常；

②当我们遇见一个类对象，一个基本类型的变量时（例如int a），并且类对象处于加号之前，也简单。将参数换位const int a，然后函数内部代码按正常情况修改即可；

③遇见2个基本类型变量，是无法使用运算符重载的，忽视这种可能；

④遇见一个类对象，一个基本类型变量（例如int a），但此时，基本类型处于加号之前。也就是 **对象 + a** 变为 **a + 对象** 这种形式。

按照正常思维，根据加法交换律，这两个相加应该没什么区别。但事实上我们知道，由于运算符重载，在有②号情况函数在的时候，前者可以运行，但后者无法运行（因为没有对应的函数调用）。

这样从逻辑上来讲没什么问题（这里的加号的作用不是面对2个算数值的加号作用），但是这样很不方便。因为这强迫程序员在码代码的时候，必须加号前面是对象，后面是基本类型。换一句话说，不友好。

那么我们在类的成员函数里再加一个运算符重载函数？并不可行。原因在于，类对象位于加号前面时，是作为对象调用运算符重载函数。而一个基本类型位于加号前面时，是无法调用类成员函数的。即对象+a的实质是：

对象.Skill::operator+(const int a)const

我们显然不能书写成：a.Skill::operator+(const Skill&b)const 这种形式，因为a根本不算类对象（他是基本类型）。

但我们的确需要一个运算符重载函数，那么我们写成一般函数？例如：

Skill operator+(const Skill&b)

{  
...

}

即参数是skill对象，返回值也是一个skill类对象。

问题来了（1）int a在哪里？在函数定义里如何书写？

（2）因为他不在类成员函数里，那么b.dam这样的调用自然也是不行的（因为只有在成员函数内才能访问私有成员）；

对第（1）个问题很简单，把int a加到参数里，第一个参数表示加号前面的数字。例如：Skill operator+(int a,const Skill&b);

对于第二个问题，便只能启用友元函数这个概念了。

**友元函数：**

①函数原型位于类的公有成员（public）处进行声明；

②函数原型前加friend，函数定义中不加friend；

③函数定义前，不加类的定义域解析运算符（例如Skill::这样的，不加）；

④重载的运算符使用哪个友元函数，根据对象的类来决定（例如有两个类，Skill和Player，当调用不同的参数时，编译器会根据运算符重载的几个重载函数，来进行重载函数的参数匹配，寻找到参数类型相符的函数（重载函数的调用，是根据参数的匹配程度来决定的）。

如代码：

#include<iostream>

class Skill

{

int a;

public:

Skill(int b = 1) { a = b; }

int operator+(const int b)

{

return a + b;

}

friend int operator+(int a, Skill b); //注意，返回值是a+skill类的私有成员a的和。另外，这里要加friend，但函数定义的时候不加

};

class Player

{

int a;

public:

Player(int b = 5) { a = b; };

int operator+(const int b)

{

return a + b;

}

friend int operator+(int a, Player b); //注意，返回值是a+player类的私有成员a的和

};

int main()

{

using namespace std;

Skill m;

Player n; //这2个的默认构造函数给私有成员赋的值是不一样的，所以最后体现是在使用运算符之后的返回值是不同的

cout << "m + 5 = " << m + 5 << endl;

cout << "5 + m = " << 5 + m << endl;

cout << "n + 5 = " << n + 5 << endl;

cout << "5 + n = " << 5 + n << endl;

system("pause");

return 0;

}

int operator+(int a, Skill b) //注意，这里不加friend

{

return a + b.a;

}

int operator+(int a, Player b)

{

return a + b.a;

}

显示：

m + 5 = 6

5 + m = 6

n + 5 = 10

5 + n = 10

请按任意键继续. . .

总结：

①通过使用运算符重载，使得m+5和5+m这样的效果是一样的；

②另外，由于使用了2个友元函数，分别是不同类的友元，于是使用运算符时，会根据类，调用不同的友元函数（而不是调用相同的友元函数）；

③如果已经有一个 **对象+基本类型** 这种形式的运算符重载了，也可以通过一个友元函数，在函数内部，交换参数位置，把 **基本类型+对象** 变为 **对象+基本类型** 这样，则可以使用已有的运算符重载。

例如：

#include<iostream>

class Skill

{

int a;

public:

Skill(int b = 1) { a = b; }

int operator+(const int b)

{

return a + b;

}

friend int operator+(int a, Skill b); //这里是友元重载函数

};

int main()

{

using namespace std;

Skill m;

cout << "m + 5 = " << m + 5 << endl;

cout << "5 + m = " << 5 + m << endl;

system("pause");

return 0;

}

int operator+(int a, Skill b)

{

return b+a; //把a+b转换为b+a，于是b+a调用类方法中的重载运算符了

}

④能不能使用模板类，作为友元函数，不是很清楚，存疑。

直接使用经试验是不可行的，如：

friend template<class xx, class yy>int operator+(xx , yy )

{

return b + a;

}

无论是像上面这样写（把函数定义放在类的public中）或者是用template<class xx, class yy>int operator+(xx , yy )替换int operator+(int a, Skill b)都是不行的，编译器提示不允许使用template

**常用的友元：重载<<运算符**

首先，我们知道，<<是一个运算符，且他可以被重载。

其次，我们知道，一般我们这么用cout<< abc；于是，运算符前面有cout，后面有变量abc。就像使用加法运算符重载一样，我们可以仿照着去写。

于是有了友元函数（假如是Skill类）：

friend void operator<<(std::ostream& os, const Skill&b); //函数原型

其中，cout是ostream类我们是知道的（在模板时用过，输出到文件或者屏幕），第二个参数是Skill类的引用，被const限定（因此不能被修改）。我们也是知道的。

于是，可以将其放在Skill类的public里作为Skill类的友元函数。

我们编写定义（假设Skill类有两个私有成员，分别是string name和int combat）：

void operator<<(std::ostream &os, Skill & b)

{

os << "name:" << b.name << " , combat is " << b.combat << std::endl;

}

注意，这里要加std:: 因为是ostream类在名称空间std之中。

于是我们可以敲代码：

Skill m;

cout << m;

调用了友元函数，搞定。

但假如因为实际需要，没有在运算符重载的函数定义里的最后输出<<std::endl;

那么在实际使用之中，我们要换行的话，就得这么输入cout<<m<<endl;

似乎这样应该是可以的？

但并不是这样。

原因在于：

<<本身面对cout时，已经被运算符重载了。我们实际经验来看，cout可以输出int，double，long，string类，char类等。

之所以能输出这些，是因为在ostream类中，有所有基本类型的<<运算符重载。

也就是说，就像我们在Skill类进行了<<对skill类的运算符重载一样，ostream类也对<<对所有基本类型进行了运算符重载。

而string类虽然不是基本类型，但是我们也可以像使用基本类型那样，对string类对象进行cout来输出，这说明，string类也进行了<<运算符重载。

运算符重载的实质，是调用函数。

例如对Skill类的对象m使用cout<<m;

实质上是调用了函数operator<<(cout, m);这个函数。

这个函数会输出一段文字，于是cout<<m便输出了cout和m作为参数时输出的文字。

了解了这个实质的前提下，我们又知道，程序是从左往右运行的（在优先级相同的情况下）。那么cout<<m<<endl; 就变成了（operator<<(cout, m)）<<endl;

endl能直接使用么？显然不行，我们需要换行的时候，一般是这么输入：cout<<endl;

而void operator<<(std::ostream &os, Skill & b)这个函数，是无返回值的，因此不能与<<endl;使用。

（注1：<<运算符最初的目的不是输出，而是C和C++的位运算符，将值中的位左移）

（注2：就像我们不能直接用<<endl一样，并没有这样的运算符重载。注意，运算符在使用时，有一个规则是不能违反原来的使用，例如<<必然是左右两个，+的左右也必然是两个，而不是说+a这样使用）

于是，给<<这个运算符重载一个返回值，且这个返回值是ostream类，那么我们就可以继续愉快的使用了。

即std::ostream & operator<<(std::ostream &os, Skill & b)

这样。他返回了一个ostream类引用。

另外，**不要加const** ，推测是因为没有const ostream&的重载函数。

然后新的<<运算符重载定义是：

std::ostream& operator<<(std::ostream &os, Skill & b)

{

os << "name:" << b.name << " , combat is " << b.combat << std::endl;

return os; //输入什么类型的ostream类对象，就输出什么对象

}

如代码：

#include<iostream>

#include<string>

class Skill

{

std::string name;

int combat;

public:

Skill(std::string na= "迪克",int b = 1) { name = na;combat = b; } //默认构造函数

friend std::ostream& operator<<(std::ostream &os, Skill & b);

};

int main()

{

using namespace std;

Skill m;

Skill n("李察", 10);

cout << m << endl << n << endl; //因为返回os，所以在遇见非Skill类时，使用ostream类自己定义的运算符重载

system("pause");

return 0;

}

std::ostream& operator<<(std::ostream &os, Skill & b)

{

os << "name:" << b.name << " , combat is " << b.combat;

return os; //输入什么类型的ostream类对象，就输出什么对象

}

显示：

name:迪克 , combat is 1

name:李察 , combat is 10

请按任意键继续. . .

一切正常。

看到这里，好像友元函数说着说着就又跳到运算符重载了。

但再次强调，友元函数的存在意义，就是让运算符的第一个参数，可以是非类的成员。例如cout、int等，都不是Skill类的成员，如果不使用友元函数，那么是无法进行运算符重载的。

**回过头来重看友元函数的存在意义和不使用友元函数的后果：**

①不使用友元函数则不能调用类的私有成员（友元函数的访问权限同类的成员函数）；

②不使用友元函数不能让非类成员在运算符前面，原因看③和④；

③当类成员在运算符前面时，调用的是运算符重载函数，在后面的作为运算符重载的参数，如：Skill Skill::operator+(const Skill&b)const

④当非类成员在运算符前面时，无法调用类成员函数——因为很多是在ostream类定义的 (例如cout<<a;不过a+b这样的是什么类定义的？) ，于是，他只能调用类外对应的函数重载了。但类是我们自己定义的，毫无疑问，比如ostream类并不知道我们说的是什么，因为他不认识我们自定义的类，所以无法输出，或者进行加减；

⑤因此，我们需要自定义一个运算符重载函数，让类对象作为参数，但是一般情况下，这个运算符重载函数是不能访问类对象的私有成员的，因此必须让他拥有能访问这个类对象私有成员的权限——也就是友元函数的意义。

关键：一个运算符时，在运算符前面的决定运算符重载函数的调用。

**重载运算符：作为成员函数还是非成员函数：**

现在有两种运算符重载的函数格式：（假设类为Player，类对象为m）

一种是面对成员函数的，例如： void operator+(int a);

一种是面对非成员函数的，例如：friend void operator +(int a, Player &b);

前者在调用时：m.operator +(int a); //一个参数，另一个为类对象被隐式的传递了

后者在调用时：operator +(int a, Player &b); //两个参数

**什么时候调用非成员函数（友元函数）呢？三种情况：**

①运算符有两个类对象，且都是同一个类。使用成员函数，且使用一个参数（我的编译器VS2015，提示成员函数的运算符重载函数只能使用一个参数）

③运算符有两个类对象时，且不是同一个类，那么不能使用即是成员函数，又是友元函数的形式（即是一个类的成员函数且是另一个类的友元函数，是不行的，至少我尝试了不行）。

但可以考虑使用 **成员函数的返回值** 的形式，变相得到我们需要的值。例如我们需要B类的私有成员aa，那么我们可以在A类的成员函数中，B类作为参数，函数定义中，使用B类能返回aa值的成员函数，从而得出结果。如代码：

#include<iostream>

class Player

{

int combat;

public:

Player() { combat = 5; }

int getcombat() { return combat; } //成员函数，返回值为私有成员的值

};

class Skill //因为Skill类的成员函数需要调用Player类，所以其必须在Player类的声明之后

{

int combat;

public:

Skill() { combat = 1; };

int operator +(Player& m);

};

int main()

{

using namespace std;

Skill m;

Player n;

cout << m + n << endl;

system("pause");

return 0;

}

int Skill::operator +(Player&m) //类对象作为参数，进行运算符重载

{

int q;

q = combat+m.getcombat(); //调用类方法getcombat()

return q;

}

③一个类对象和一个基本类型。

假如类对象在前，基本类型在后：一般使用成员函数（因为这样更简单），也可以使用友元函数，例如：friend int operator +(const Skill& m,const int b);但这样就相对复杂一些，个人觉得意义不大。

假如基本类型在前，类对象在后：只能使用友元函数。

（一二三）矢量坐标计算

一个类，包含极坐标和直角坐标。

两个这样的类对象，通过运算符重载，可以直接相加/相减，并得出一个新的坐标，而这个新坐标，是原来两个对象的矢量相加的结果。

代码如下：

//Point.h 包含类定义

#pragma once

class Point

{

public:

enum mode { xy = 0, ji = 1 }; //常量

private:

double x; //x,y坐标

double y;

double jiaodu; //极坐标：角度和长度

double changdu;

int mode; //模式，根据模式来决定坐标输出时的形式

void setxy(); //设置x、y坐标（根据极坐标）

void setji(); //设置极坐标（根据x，y坐标）

public:

void cm(int b = 2); //模式转换，默认参数为2（切模式），也可以给参数，改变转换情况

Point(double a = 0, double b = 0, int c = xy); //默认构造函数。默认为x，y坐标赋值

void reset(double a = 0, double b = 0, int c = xy); //重置坐标

void show()const; //输出坐标

Point operator+(const Point& m)const; //运算符+重载，不用引用是因为要创建一个新对象，然后返回这个对象

Point operator+(double m)const; //再次重载，面对对象是一个double值，直接加在x和y坐标上

Point operator-(const Point& m)const; //运算符-重载，

Point operator\*(const double m)const; //运算符\*重载，

Point operator/(const double m)const; //运算符/重载，

friend std::ostream& operator<<(std::ostream &os, const Point&m); //用于cout或其他输出

};

//Point.cpp 包含了Point类的成员函数以及友元函数定义

#include<iostream>

#include<cmath>

#include"Point.h"

void Point::setxy() //设置x、y坐标（根据极坐标），函数内容是拿以前的改改

{

const double jiao\_to\_angle = 57.29577951; //角度/弧度的值大概是这个数

double hudu = jiaodu / jiao\_to\_angle; //角度除以这个值得到弧度

y = sin(hudu)\*changdu; //对边(y) = sin弧度\*斜边。

x = cos(hudu)\*changdu; //临边(x) = sin弧度\*斜边

}

void Point::setji() //设置极坐标（根据x，y坐标）

{

const double jiao\_to\_angle = 57.29577951; //角度/弧度的值大概是这个数

changdu = sqrt(x\*x + y\*y); //利用坐标求距离

jiaodu = atan2(y, x)\*jiao\_to\_angle; //利用坐标求角度

}

void Point::cm(int b)

{

if (b == 2)

{

if (mode == ji)mode = xy;

else mode = ji;

}

else if (b == ji)mode = ji;

else if (b == xy)mode = xy;

else std::cout << "参数输出错误，输出模式转换失败。" << std::endl;

}

Point::Point(double a, double b, int c)

{

if (c == xy)

{

x = a;

y = b;

mode = c;

setji();

}

else if (c == ji)

{

jiaodu = a;

changdu = b;

mode = c;

setxy();

}

else std::cout << "参数输入错误。如果你需要输入x、y坐标，请第三个参数输出0或者不输入第三个参数；如果你需要输入极坐标，请第三个参数输入1。" << std::endl;

}

void Point::reset(double a, double b, int c)

{

if (c == xy)

{

x = a;

y = b;

setji();

}

else if (c == ji)

{

jiaodu = a;

changdu = b;

setxy();

}

else std::cout << "参数输入错误。如果你需要输入x、y坐标，请第三个参数输出0或者不输入第三个参数；如果你需要输入极坐标，请第三个参数输入1。" << std::endl;

}

void Point::show()const //输出坐标

{

if (mode == xy)

{

std::cout << "现在报告x，y坐标：" << std::endl;

std::cout << "x：" << x << "，y：" << y << std::endl;

}

else if (mode == ji)

{

std::cout << "现在报告极坐标：" << std::endl;

std::cout << "长度为：" << changdu << "，角度为：" << jiaodu << "度" << std::endl;

}

}

Point Point::operator+(const Point& m)const //运算符+重载，不用引用是因为要创建一个新对象，然后返回这个对象

{

Point q;

q.x = x + m.x;

q.y = y + m.y;

q.mode = mode; //默认模式为运算符前面的对象的显示模式

q.setji();

return q;

}

Point Point::operator+(double m)const

{

Point q;

q.x = x + m;

q.y = y + m;

q.setji();

return q;

}

Point Point::operator-(const Point& m)const //运算符-重载，

{

Point q;

q.x = x - m.x;

q.y = y - m.y;

q.mode = mode;

q.setji();

return q;

}

Point Point::operator\*(const double m)const //运算符\*重载，

{

Point q;

q.x = x \* m;

q.y = y \* m;

q.mode = mode;

q.setji();

return q;

}

Point Point::operator/(const double m)const //运算符/重载，

{

Point q;

q.x = x / m;

q.y = y / m;

q.mode = mode;

q.setji();

return q;

}

std::ostream& operator<<(std::ostream &os, const Point&m)

{

m.show();

return os;

}

//1.cpp main()函数所在，用于使用和验证类

#include<iostream>

#include"Point.h"

int main()

{

using namespace std;

Point a; //使用默认构造函数

Point b(3, 4);

Point c(4, 5, Point::ji); //最后的Point::ji表示是极坐标，由于enum在类定义里（公有成员），所以需要作用域解析运算符。或者用1替代也可以

a.show(); //分别显示a、b、c

b.show();

c.show();

cout << "将c以x，y坐标显示：" << endl;

c.cm(0); //更改c的输出模式

c.show(); //显示c

cout << "将b以极坐标显示：" << endl;

b.cm(); //更改b的输出模式

b.show(); //显示b

cout << "a+3=" << a + 3 << endl;

cout << "a-2=" << a - 2 << endl;

cout << "b+c=" << b + c << endl; //注意，显示模式根据运算符前面的类对象而定

cout << "b-c=" << b - c << endl;

cout << "b\*2=" << b \* 2 << endl;

cout << "b/2=" << b / 2 << endl;

system("pause");

return 0;

}

显示：

现在报告x，y坐标：

x：0，y：0

现在报告x，y坐标：

x：3，y：4

现在报告极坐标：

长度为：5，角度为：4度

将c以x，y坐标显示：

现在报告x，y坐标：

x：4.98782，y：0.348782

将b以极坐标显示：

现在报告极坐标：

长度为：5，角度为：53.1301度

a+3=现在报告x，y坐标：

x：3，y：3

a-2=现在报告x，y坐标：

x：-2，y：0

b+c=现在报告极坐标：

长度为：9.0949，角度为：28.5651度

b-c=现在报告极坐标：

长度为：4.15726，角度为：118.565度

b\*2=现在报告极坐标：

长度为：10，角度为：53.1301度

b/2=现在报告极坐标：

长度为：2.5，角度为：53.1301度

请按任意键继续. . .

总结：

① - 运算符除了表示减法（例如a-b），也可以表示负号（例如 -a）。因此，重载负号运算符时，可以作为成员函数，且无需使用参数（隐式的调用类对象自己）。

（一二四）给类对象赋值、以及类对象的返回值

**于直接给对象赋值：**

之前学过，如何给对象在初始化时进行赋值。

对于C++11来说，初始化方式有三种：

① man c = man{ "cc",1 };

② man d = { "dd",1 };

③ man f{ "ff",1 };

假如有一类M，他有两个私有成员a和b（int类型）。

于是新建一对象M q; 对象q使用默认构造函数（假如都赋值为0，这个不重要）；

现在，我们想给对象q的第一个私有成员赋值，该怎么办？

这章刚学过运算符重载，难道要写个运算符重载么？隐式调用类对象，参数为赋值符号=后面的值？（我们知道，赋值符号是可以被成员函数重载的）

这显然太麻烦。

如果可以这样q=3，于是q的第一个成员被赋值3就十分方便了。

这是可行的，不是通过运算符重载，而是通过构造函数，构造函数设置一个参数时，使用q=3，则默认3为第一个参数，假设构造函数这么写。

M(int c = 1) { a = c; b = 1.1; };

那么使用q=3时，a将等于3，b将等于1.1（此刻相当于同时给2个私有成员赋值）。

如果我们只想给a赋值，不给b赋值，那么删掉b=1.1也是可以的（但这是默认构造函数，如果删除的话，创建新对象时，调用这个默认构造函数时则无法给b进行初始化）。

那么如果想利用赋值符号，同时给a和b赋值呢，且b使用我们给他的值，而不是上面的默认值1.1。办法是，继续使用构造函数，且这个构造函数有两个参数，赋值时使用列表化进行赋值（大括号）。

如：

M(int c, double d) { a = c;b = d; }

注意：这里不能给默认参数，否则在使用q=3这样时，会有二义性（即能匹配第一个默认构造函数，也能匹配这个构造函数）。

然后使用：q={5,4.4}; 这样的形式，可以同时给a和b赋值。

此时调用了有两个参数的构造函数。另外，需要注意的是，不能省略掉大括号，否则，会导致调用了默认构造函数。

如代码：

#include<iostream>

class M

{

int a;

double b;

public:

M(int c = 1) { a = c; b = 1.1; } //带一个默认参数，默认构造函数

M(int c, double d) { a = c;b = d; } //两个参数，无默认参数，以避免二义性，构造函数

void show() { std::cout << a << ", " << b << std::endl; }; //显示a和b的值

};

int main()

{

using namespace std;

M q; //创建对象q（调用默认构造函数）

q.show(); //输出

q = 5; //使用赋值符号，（此刻调用默认构造函数，但使用给的值作为参数）

q.show();

q = { 5,4.4 }; //调用有两个参数的构造函数

q.show();

q = 9,3.3; //虽然给了两个参数，但是只会调用默认构造函数

q.show();

system("Pause");

return 0;

}

显示：

1, 1.1

5, 1.1

5, 4.4

9, 1.1

请按任意键继续. . .

以上，将某个值直接赋给类对象，其原理是：使用类的构造函数，创造一个临时的对象，然后，将值赋给临时对象，再采用逐成员赋值的方式（这里调用的是复制构造函数），将该临时对象的内容复制到目标对象之中。

这一过程，被称为 **隐式转换** ，因为他是自动进行的，而不需要显式强制类型转换。

这种转换，称为是 **类型转换**。

只有接受一个参数的构造函数，才能作为转换函数。有两个参数的构造函数，是不能用来转换类型的（就像上面说的，需要用大括号把两个参数都括起来）。但如果给第二个参数提供默认值，便又可以用于转换类型了（但前提是不会因此导致 **二义性**）。

将构造函数用于自动类型转换，在某些时候比较方便，但这种特性并非总是合乎需要的（比如说我们某个时候不需要他这么做，暂时举不出例子），因为这可能导致意外的类型转换（比如说只想给一个值赋值，但因为构造函数，给另外一个值赋了默认值）。

因此，C++新增了关键字**explicit**，用于关闭这种隐式转换（但仍允许显式转换），即 **显式强制类型转换**：

例如，double a=int (4.1); 便是一种 显式强制类型转换。double类型的4.1被强制转换为int类型（此时值为4），又被赋值给double类型，于是a=4。

构造函数改为：explicit M(int c = 1) { a = c; b = 1.1; }

此时，M q是可行的，

q=5是不可行的，

但是q = M(5);又是可行的。

另外，需要提一下的是，之前q = 9,3.3;只能导致9被赋值（调用了默认构造函数），但是修改后为 q = M(9,3.3); 则调用的是另一个构造函数了。

假如不使用explicit给构造函数（一个参数的，例如M(int c = 1)），那么，编译器将在什么启动隐式转换（使用构造函数，然后进行赋值）呢？

①声明一个类对象，并进行初始化。例如：M q(1);

②将值赋给对象时，例如：q=5;

③将值传递给接受类对象参数的函数时。例如：void abc(M t = 3); 这样。3相当于是函数的默认参数。

④返回值是类对象时，函数试图返回值时。就比如：

M abc()

{

return 3;

}

这个时候，相当于返回一个临时类对象，然后这个对象被初始化赋值为3，就像M q(3); 然后return q一样。只不过这里不是返回q，是返回一个临时的类对象

⑤在上述任意一种情况下，使用可转换为 可直接赋值给类对象的值的类型的 内置类型。

假如构造函数M(int c = 1) ，这个时候，需要用int类型的值给其赋值，例如q=3; 而我们知道，double类型可以转换为int类型，因此，我们也可以输入q=3.3; 这时，3.3被转换为int类型的3，然后int类型的3被隐式转换，赋值给类对象。这种情况也是允许的。

另外，以上隐式转换发生的前提是，函数不存在二义性，否则会提示错误。

**转换函数：**

隐式转换和显示转换，可以把一个比如说int类型的值，赋给类对象。

那么类对象是否能赋值给一个int类型的变量呢，也是可以的。

要进行这种转换，必须使用特殊的C++运算符函数—— **转换函数**。

转换函数是 用户定义（不定义是不能用的）的强制类型转换，可以像使用强制类型那样使用他们。

**转换函数的创建方法：**

格式：operator 类型名();

关键点：

①转换函数必须是类方法（成员函数）；

②转换函数不能指定返回类型（靠类型名）；

③转换函数不能有参数（靠私有成员的值）。

例如，转换为int类型，其函数原型应该这么写：operator int();

函数定义如下写：

operator int()  
{  
 return a;

}

这样，将返回私有成员a的值。

注意，假如有operator double(); 那么返回int值时调用第一个，返回double值时调用第二个。返回其他类型，会提示冲突。

如代码：

#include<iostream>

class M

{

int a;

double b;

public:

M(int c = 1) { a = c; b = 4.1; } //带一个默认参数，默认构造函数

void show() { std::cout << a << ", " << b << std::endl; }; //显示a和b的值

operator int() { return a; } //转换函数返回int值时调用这个

operator double() { return b; } //转换函数返回double值时调用这个

};

int main()

{

using namespace std;

M q; //创建对象q（调用默认构造函数）

q.show(); //输出

int a = q; //返回int值

double b = q; //返回double值

double c = int(q); //强制类型转换为int，因此返回int值

cout << "a=" << a << ",b=" << b << ",c=" << c << endl;

system("Pause");

return 0;

}

显示：

1, 4.1

a=1,b=4.1,c=1

请按任意键继续. . .

**自动应用类型转换：**

假如，一个类只有一个转换函数，例如强制转换为int。

那么调用这个对象时，显示的便是其转换函数返回的值。

例如假如上面那个类方法，只有operator int()这个转换函数。那么当我们使用cout<<q<<endl; 时，

这个时候，程序便自动应用了类型转换，显示的是“1”。

但假如同时存在两个转换函数，再这样做的话，会提示二义性，因为程序并不知道你想要显示的是int类型的值还是double类型的值，除非你使用了强制类型转换。

**注意：应谨慎的使用隐式转换函数。通常，最好选择仅在被显式的调用时才会被执行的函数。**

例如：用一个返回值是你想要转换函数返回的值的函数，例如，使用operator()返回私有成员a，那么，用int M\_to\_int(){ return a;} 这样的函数，来返回私有成员a，调用时，则使用q.M\_to\_int();

总之：C++为类提供了下面的类型转换：

①利用构造函数，把基本类型转换为类 类型。

②利用转换函数，把类 类型转换为基本类型。

**友元函数（运算符重载）和转换函数：**

有类M，有类对象成员a，有int变量b。

有赋值，M c=a+b;

（1）假如有友元函数（或者运算符重载）的情况下：a+b将调用运算符重载函数，将函数返回值赋给类对象c。

（2）假如有转换函数，且类M的构造函数接受一个int参数。那么a将被转化为int值（根据转换函数），a+b的和（int值）作为参数，赋给类对象c。

假如有转换函数，但类M的构造函数不接受一个int参数（例如有两个参数且无默认参数），那么则提示出错，无法编译。

（3）假如既有运算符重载（“+”运算符），又有转换函数。

那么，根据实际测试来看，优先调用运算符重载函数或友元函数（假如他能完全匹配的话）；

如果不能完全匹配，那么就可能导致二义性错误。

如代码：

#include<iostream>

class M

{

int a;

double b;

public:

M(int c=3) { a = c; b = 4.4; } //带一个默认参数，默认构造函数

void show() { std::cout << a << ", " << b << std::endl; }; //显示a和b的值

operator int() { return a; } //转换函数返回int值时调用这个

friend int operator+(int c, M & q);

int operator+(int c) { return b + c; }

};

int main()

{

using namespace std;

M q; //创建对象q（调用默认构造函数）

M c = 3 + q;

M d = q + 3;

c.show();

d.show();

system("Pause");

return 0;

}

int operator+(int c, M & q)

{

return c + q.b;

}

显示：

7, 4.4

7, 4.4

请按任意键继续. . .

之所以结果为7，是因为优先使用了运算符重载函数（友元函数和成员函数），其用成员函数b和参数c相加，结果为7（也就是第一个数字）。

而若使用转换函数的话，其结果应为6，因为转换函数的返回值是3，3+3，其第一个数字应为6。——但具体为什么，推测是函数列表匹配度问题

①假如把3+q或者q+3，改为3.3+q和q+3.3，那么函数则会提示错误。原因在于，友元函数和成员函数不能完全匹配了，在匹配列表里优先级降低，成为了标准匹配（第三优先级）。而转换函数的返回值也是int类型，需要进行转换，因此也成为了标准转换（优先级也是第三级），因为匹配优先级相同，因此提示错误。

②又假如把q+3改为int(q)+3，那么首先对对象q执行强制类型转换，其为int值3（因为转换函数返回值为3），因此3+3=6，启用构造函数，于是第一个值便变成了6.

由于匹配优先级问题，如果想让q+3结果变为6，那么则在运算符重载函数修改为：int operator+(M&c) { return b + c.b; } 这时，转换函数的优先级则更高。

假如有：

#include<iostream>

class M

{

int a;

double b;

public:

M(int c = 2) { a = c; b = 4.4; } //带一个默认参数，默认构造函数

void show() { std::cout << a << ", " << b << std::endl; }; //显示a和b的值

operator int() { return a; } //转换函数返回int

friend M operator+(M&a,M&c) { return a.b + c.b; }

};

int main()

{

using namespace std;

M q; //创建对象q（调用默认构造函数）

M d = q + 3;

d.show();

system("Pause");

return 0;

}

则d的输出结果为5，而不是8。使用的是转换函数，而不是运算符重载函数。原因我推测为：q+3，通过调用转换函数，被转换为2+3，然后将5赋值给了对象d。

而若使用运算符重载函数，则可能成为了用户定义的类型转换了。因此优先级不如转换函数。即使把q+3改为q+M(3)这种，将3强制转换为类M的临时对象，其优先级也没有转换函数高。

除非改为：M q; M w=3; M d=q+w; 这种形式，才会使用运算符重载函数。

总之，这种容易引起意料之外结果的，我觉得还是转换函数和运算符重载函数，二者取其一吧。或者放弃转换函数，而是改使用显式的转换函数（加关键字explicit）。

（一二五）第十一章复习题

1.使用成员函数为Stonewt类重载乘法运算符，该运算符将数据成员与double类型的值相乘。注意用英石和磅表示时，需要进位。也就是说，将10英石8磅乘以2等于21英石2磅。

答：

Stonewt Stonewt::operator\*(double a)

{

Stonewt q;

q.stone=stone\*a+pounds\*a/14;

q.pounds=(pounds\*a)%14;

}

2.友元函数和成员函数之间的区别是什么？

答：

友元函数：①函数原型需要使用friend；②调用的对象一般是非类对象；③在运算符重载函数里，有两个参数，并且往往一个是类对象、一个是非类对象；④在函数定义里，若不是内联函数，无需使用类作用域解析运算符在函数头中；

成员函数：①函数原型无需使用friend；②调用的对象是类对象；③在运算符重载函数里，类对象被隐式的传递给了函数；④若不是内联函数，那么在函数定义的函数头，需要使用作用域解析运算符。

成员函数是类定义的一部分，通过特定的对象来调用，可以直接调用对象的成员，而无需使用成员运算符（那个句号）。

友元函数不是类的组成部分，不是隐式的访问对象的成员，而是通过参数传递的对象来访问，并且需要使用成员运算符。

3.非成员函数必须是友元才能访问类成员吗？

答：

一般是，因为友元函数才能获得和成员函数一样的权限来访问类成员。

如果不使用友元函数，那么若要访问类成员，则只能访问类的公有成员，而无法访问私有成员。或者，类方法里有返回私有成员的值的公有成员函数。

4.使用友元函数为Stonewt类重载乘法运算符，该运算符将double值与Stone值相乘。

答：

函数原型，位于Stonewt类公有成员区域：friend Stonewt operator \* (double a,const Stone& b);

函数定义：

Stonewt operator \* (double a,const Stone& b)

{

Stonewt q;

q.Stone = b.Stone\*a;

return q;

}

5.哪些运算符不能重载？

答：

（1）sizeof sizeof运算符

（2）. ←这是一个句号，成员运算符，比如结构名.结构内变量

（3）.\* ←句号和乘号，成员指针运算符（没见过）

（4）:: ←两个冒号，作用域解析运算符

（5）?: 条件运算符，比如： a>b?c:d

（6）typeid ←一个RTTI运算符（没见过）

（7）const\_cast ←强制类型转换运算符（没见过这种用法）

（8）dynamic\_cast ←强制类型转换运算符（没见过这种用法）

（9）reinterpret\_cast ←强制类型转换运算符（没见过这种用法）

（10）static\_cast ←强制类型转换运算符（没见过这种用法）

6.在重载运算符=、()、[]和->时，有什么限制？

答：

只能通过类成员函数来进行重载（but，我没试过这个）。

7.为Vector类定义一个转换函数，将Vector类转换为一个double类型的值，后者表示矢量的长度。

答：

以下这个函数直接放在public区域成为内联函数。

operator double()

{

return mag;

}

（一二六）第十一章编程练习

1.修改程序清单11.5，使之将一系列连续的随机漫步者位置写入到文件中。对于每个位置，用步号进行标志。另外，让该程序将初始条件（目标距离和步长）以及结果小结写入到该文件中。该文件的内容与下面类似：

Target Distance: 100, Step Size: 20

0: (x,y) = (0, 0)

1: (x,y) = (-11.4715, 16.383)

2: (x,y) = (-868807, -3.42232)

....

26: (x,y) = (42.2919, -78.2594)

27: (x,y) = (58.6749, -89.7309)

After 27 steps, the subject has the following location:

(x,y) = (58.6749, -89.7309)

or

(m,a) = (107.212, -56.8194)

Average outward distance per step = 3.97081

答：

注：这里我不修改源文件了，在自己的基础上，重新写一个

//Point.h 包含类定义

#pragma once

class Point

{

public:

enum mode { xy = 0, ji = 1 }; //常量

private:

double x; //x,y坐标

double y;

double jiaodu; //极坐标：角度和长度

double changdu;

int mode; //模式，根据模式来决定坐标输出时的形式

void setxy(); //设置x、y坐标（根据极坐标）

void setji(); //设置极坐标（根据x，y坐标）

public:

Point(double a = 0, double b = 0, int c = xy); //默认构造函数。默认为x，y坐标赋值

void reset(double a = 0, double b = 0, int c = xy); //重置坐标

void show(std::ostream &os)const; //输出坐标

double get\_changdu() { return changdu; } //返回极坐标的长度

Point operator+(const Point& m)const; //运算符+重载，不用引用是因为要创建一个新对象，然后返回这个对象

friend std::ostream& operator<<(std::ostream &os, const Point&m); //用于cout或其他输出

};

//Point.cpp 包含了Point类的成员函数以及友元函数定义

#include<iostream>

#include<cmath>

#include<fstream>

#include"Point.h"

void Point::setxy() //设置x、y坐标（根据极坐标），函数内容是拿以前的改改

{

const double jiao\_to\_angle = 57.29577951; //角度/弧度的值大概是这个数

double hudu = jiaodu / jiao\_to\_angle; //角度除以这个值得到弧度

y = sin(hudu)\*changdu; //对边(y) = sin弧度\*斜边。

x = cos(hudu)\*changdu; //临边(x) = sin弧度\*斜边

}

void Point::setji() //设置极坐标（根据x，y坐标）

{

const double jiao\_to\_angle = 57.29577951; //角度/弧度的值大概是这个数

changdu = sqrt(x\*x + y\*y); //利用坐标求距离

jiaodu = atan2(y, x)\*jiao\_to\_angle; //利用坐标求角度

}

Point::Point(double a, double b, int c)

{

if (c == xy)

{

x = a;

y = b;

mode = c;

setji();

}

else if (c == ji)

{

jiaodu = a;

changdu = b;

mode = c;

setxy();

}

else std::cout << "参数输入错误。如果你需要输入x、y坐标，请第三个参数输出0或者不输入第三个参数；如果你需要输入极坐标，请第三个参数输入1。" << std::endl;

}

void Point::reset(double a, double b, int c)

{

if (c == xy)

{

x = a;

y = b;

setji();

}

else if (c == ji)

{

jiaodu = a;

changdu = b;

setxy();

}

else std::cout << "参数输入错误。如果你需要输入x、y坐标，请第三个参数输出0或者不输入第三个参数；如果你需要输入极坐标，请第三个参数输入1。" << std::endl;

}

void Point::show(std::ostream &os)const //输出坐标

{

os << "现在报告x，y坐标：" << std::endl;

os << "x：" << x << "，y：" << y << std::endl;

os << "现在报告极坐标：" << std::endl;

os << "长度为：" << changdu << "，角度为：" << jiaodu << "度" << std::endl;

}

Point Point::operator+(const Point& m)const //运算符+重载，不用引用是因为要创建一个新对象，然后返回这个对象

{

Point q;

q.x = x + m.x;

q.y = y + m.y;

q.mode = mode; //默认模式为运算符前面的对象的显示模式

q.setji();

return q;

}

std::ostream& operator<<(std::ostream &os, const Point&m)

{

os << " (x,y) = (" << m.x << ", " << m.y << ")";

return os;

}

//1.cpp main()函数所在，用于使用和验证类

#include<iostream>

#include<fstream>

#include"Point.h"

#include<ctime>

int main()

{

using namespace std;

srand(time(0));

ofstream q; //创建osftream对象

q.open("111.txt"); //打开111.txt文件

int distance;

cout << "请输入距离：";

cin >> distance;

int step\_size;

cout << "请输入每一步的距离：";

cin >> step\_size;

Point m(0, 0);

q << "0:" << m << endl;

int steps;

for (steps = 0;m.get\_changdu() < distance;steps++)

{

double d = rand() % 360;

Point newone(d, step\_size, 1);

m = m + newone;

q << steps+1 << ":" << m << endl;

}

q << "After " << steps << " steps, the subject has the following location:" << endl;

m.show(q);

q << "Average outward distance per step = " << m.get\_changdu() / steps << endl;

system("pause");

return 0;

}

文本文件结果：

0: (x,y) = (0, 0)

1: (x,y) = (-8.45237, 18.1262)

2: (x,y) = (7.09055, 30.7126)

3: (x,y) = (24.0515, 20.1142)

4: (x,y) = (4.05152, 20.1142)

5: (x,y) = (-0.447505, 0.626777)

6: (x,y) = (-5.62389, 19.9453)

7: (x,y) = (7.75873, 34.8082)

8: (x,y) = (27.0772, 39.9846)

9: (x,y) = (24.6399, 59.8355)

10: (x,y) = (33.4073, 77.8114)

11: (x,y) = (17.4346, 65.7751)

12: (x,y) = (15.6915, 45.8512)

13: (x,y) = (25.6915, 63.1717)

14: (x,y) = (6.20405, 67.6707)

15: (x,y) = (19.0598, 82.9916)

16: (x,y) = (38.0809, 76.8113)

17: (x,y) = (23.6941, 62.9181)

18: (x,y) = (43.1001, 67.7565)

19: (x,y) = (53.9928, 50.9831)

20: (x,y) = (73.3987, 55.8216)

21: (x,y) = (67.5513, 74.9477)

After 21 steps, the subject has the following location:

现在报告x，y坐标：

x：67.5513，y：74.9477

现在报告极坐标：

长度为：100.898，角度为：47.9712度

Average outward distance per step = 4.80465

2.对于Vector类的头文件（程序清单11.13）和实现文件（程序清单11.14）进行修改，使其不再储存矢量的长度和角度，而是在magval()和angval()被调用时计算它们。

应保留公有接口不变（公有方法及其参数不变），但对私有部分（包括一些私有方法）和方法实现进行修改。然后，使用程序清单11.15对修改后的版本进行测试，结果应该与以前相同，因为Vecotr类的公有接口与原来相同。

答：

//vect.cpp

#include<cmath>

#include"vect.h"

using std::sqrt;

using std::sin;

using std::atan;

using std::atan2;

using std::cout;

namespace VECTOR

{

void Vector::set\_x(double mag, double ang)

{

x = mag\*cos(ang);

}

void Vector::set\_y(double mag, double ang)

{

y = mag\*sin(ang);

}

Vector::Vector()

{

x = y = 0.0;

mode = RECT;

}

Vector::Vector(double n1, double n2, Mode form)

{

mode = form;

if (form == RECT)

{

x = n1;

y = n2;

}

else if (form == POL)

{

set\_x(n1, n2);

set\_y(n1, n2);

}

else

{

cout << "Incorrect 3rd argument to Vector() -- ";

cout << "vector set to 0\n";

x = y = 0;

mode = RECT;

}

}

void Vector::reset(double n1, double n2, Mode form)

{

mode = form;

if (form == RECT)

{

x = n1;

y = n2;

}

else if (form == POL)

{

set\_x(n1, n2);

set\_y(n1, n2);

}

else

{

cout << "Incorrect 3rd argument to Vector() -- ";

cout << "vector set to 0\n";

x = y = 0;

mode = RECT;

}

}

Vector::~Vector()

{

}

void Vector::polar\_mode()

{

mode = POL;

}

void Vector::rect\_mode()

{

mode = RECT;

}

Vector Vector::operator+(const Vector&b)const

{

return Vector(x + b.x, y + b.y);

}

Vector Vector::operator-(const Vector &b)const

{

return Vector(x - b.x, y - b.y);

}

Vector Vector::operator-()const

{

return Vector(-x, -y);

}

Vector Vector::operator\*(double n)const

{

return Vector(n\*x, n\*y);

}

Vector operator\*(double n, const Vector &a)

{

return a\*n;

}

std::ostream & operator<<(std::ostream &os, const Vector &v)

{

if (v.mode == Vector::RECT)

os << "(x,y) = (" << v.x << ", "<<v.y << ")";

else if (v.mode == Vector::POL)

{

os << "(m,a) = (" << v.magval() << ", " << v.angval() << ")";

}

else

os << "Vector object mode is invalid";

return os;

}

}

//vect.cpp

#include<cmath>

#include"vect.h"

using std::sqrt;

using std::sin;

using std::atan;

using std::atan2;

using std::cout;

namespace VECTOR

{

void Vector::set\_x(double mag, double ang)

{

x = mag\*cos(ang);

}

void Vector::set\_y(double mag, double ang)

{

y = mag\*sin(ang);

}

Vector::Vector()

{

x = y = 0.0;

mode = RECT;

}

Vector::Vector(double n1, double n2, Mode form)

{

mode = form;

if (form == RECT)

{

x = n1;

y = n2;

}

else if (form == POL)

{

set\_x(n1, n2);

set\_y(n1, n2);

}

else

{

cout << "Incorrect 3rd argument to Vector() -- ";

cout << "vector set to 0\n";

x = y = 0;

mode = RECT;

}

}

void Vector::reset(double n1, double n2, Mode form)

{

mode = form;

if (form == RECT)

{

x = n1;

y = n2;

}

else if (form == POL)

{

set\_x(n1, n2);

set\_y(n1, n2);

}

else

{

cout << "Incorrect 3rd argument to Vector() -- ";

cout << "vector set to 0\n";

x = y = 0;

mode = RECT;

}

}

Vector::~Vector()

{

}

void Vector::polar\_mode()

{

mode = POL;

}

void Vector::rect\_mode()

{

mode = RECT;

}

Vector Vector::operator+(const Vector&b)const

{

return Vector(x + b.x, y + b.y);

}

Vector Vector::operator-(const Vector &b)const

{

return Vector(x - b.x, y - b.y);

}

Vector Vector::operator-()const

{

return Vector(-x, -y);

}

Vector Vector::operator\*(double n)const

{

return Vector(n\*x, n\*y);

}

Vector operator\*(double n, const Vector &a)

{

return a\*n;

}

std::ostream & operator<<(std::ostream &os, const Vector &v)

{

if (v.mode == Vector::RECT)

os << "(x,y) = (" << v.x << ", "<<v.y << ")";

else if (v.mode == Vector::POL)

{

os << "(m,a) = (" << v.magval() << ", " << v.angval() << ")";

}

else

os << "Vector object mode is invalid";

return os;

}

}

//randwalk.cpp

#include<iostream>

#include<cstdlib>

#include<ctime>

#include"vect.h"

int main()

{

using namespace std;

using VECTOR::Vector;

srand(time(0));

double direction;

Vector step;

Vector result(0.0, 0.0);

unsigned long steps = 0;

double target;

double dstep;

cout << "Enter target distance (q to quit):";

while (cin >> target)

{

cout << "Enter step length: ";

if (!(cin >> dstep))

break;

while (result.magval()<target)

{

direction = rand() % 360;

step.reset(dstep, direction, Vector::POL);

result = result + step;

steps++;

}

cout << "After " << steps << " steps, the subjct has the following location:\n";

cout << result << endl;

result.polar\_mode();

cout << " or\n" << result << endl;

cout << "Average outward distance per step = " << result.magval() / steps << endl;

steps = 0;

result.reset(0.0, 0.0);

cout << "Enter target distance (q to quit): ";

}

cout << "Bye!\n";

cin.clear();

while (cin.get() != '\n')

continue;

return 0;

}

3.修改程序清单11.15，使之报告N次测试中的最高、最低和平均步数（其中N是用户输入的整数），而不是报告每次测试的结果。

答：

为了方便，我在编程练习2的基础上修改。

//randwalk.cpp

#include<iostream>

#include<cstdlib>

#include<ctime>

#include"vect.h"

int main()

{

using namespace std;

using VECTOR::Vector;

srand(time(0));

double direction;

Vector step;

Vector result(0.0, 0.0);

unsigned long steps = 0;

double target;

double dstep;

cout << "Enter target distance (q to quit):";

while (cin >> target)

{

cout << "Enter step length: ";

if (!(cin >> dstep))

break;

cout << "输入你想测试的次数：";

int N;

cin >> N;

if (N == 0)break;

while (result.magval()<target)

{

direction = rand() % 360;

step.reset(dstep, direction, Vector::POL);

result = result + step;

steps++;

}

double Max, Min, Average;

Max = Min = Average = steps;

for (int i = 1;i < N;i++)

{

steps = 0;

result.reset(0.0, 0.0);

while (result.magval()<target)

{

direction = rand() % 360;

step.reset(dstep, direction, Vector::POL);

result = result + step;

steps++;

}

if (Max < steps)Max = steps;

if (Min > steps)Min = steps;

Average = Average + steps;

}

Average = Average / N;

cout << "次数：" << N << endl;

cout << "最大步数：" << Max << endl;

cout << "最小步数：" << Min << endl;

cout << "平均步数：" << Average << endl;

cout << "Enter target distance (q to quit):";

}

cout << "Bye!\n";

cin.clear();

cin.sync();

system("pause");

return 0;

}

显示：

Enter target distance (q to quit):50

Enter step length: 2

输入你想测试的次数：1000

次数：1000

最大步数：3083

最小步数：72

平均步数：646.003

Enter target distance (q to quit):50

Enter step length: 2

输入你想测试的次数：1000

次数：1000

最大步数：3034

最小步数：67

平均步数：646.13

Enter target distance (q to quit):q

Bye!

请按任意键继续. . .

4.重新编写最后的Time了示例（程序清单11.10、程序清单11.11和程序清单11.12），使用友元函数来实现所有的重载运算符。

答：

//mytime3.h

#ifndef MYTIME3\_H\_

#define MYTIME3\_H\_

#include<iostream>

class Time

{

private:

int hours;

int minutes;

public:

Time();

Time(int h, int m = 0);

void AddMin(int m);

void AddHr(int h);

void Reset(int h = 0, int m = 0);

friend Time operator+(const Time & t1, const Time & t2);

friend Time operator-(const Time & t1, const Time & t2);

friend Time operator\*(const Time & t1, double n);

friend Time operator\*(double n, const Time & t1);

friend std::ostream & operator<<(std::ostream & os, const Time & t);

};

#endif

//mytime3.cpp

#include"mytime3.h"

Time::Time()

{

hours = minutes = 0;

}

Time::Time(int h, int m)

{

hours = h;

minutes = m;

}

void Time::AddMin(int m)

{

minutes += m;

hours += minutes / 60;

minutes %= 60;

}

void Time::AddHr(int h)

{

hours += h;

}

void Time::Reset(int h, int m)

{

hours = h;

minutes = m;

}

Time operator +(const Time &t1, const Time &t2)

{

int minutes, hours;

minutes = t1.minutes + t2.minutes;

hours = t1.hours + t2.hours + minutes / 60;

minutes %= 60;

Time sum(hours, minutes);

return sum;

}

Time operator-(const Time & t1, const Time & t2)

{

int tot1, tot2;

int minutes, hours;

tot1 = t1.minutes + 60 \* t1.hours;

tot2 = t2.minutes + 60 \* t2.hours;

minutes = (tot1 - tot2) % 60;

hours = (tot1 - tot2) / 60;

Time diff(hours, minutes);

return diff;

}

Time operator\*(const Time & t1, double n)

{

int h, m;

long totalminutes = t1.hours\*n \* 60 + t1.minutes\*n;

h = totalminutes / 60;

m = totalminutes % 60;

Time result(h, m);

return result;

}

Time operator\*(double n, const Time & t1)

{

return t1\*n;

}

std::ostream & operator<<(std::ostream & os, const Time & t)

{

os << t.hours << " hours, " << t.minutes << " minutes";

return os;

}

//usetime3.cpp

#include<iostream>

#include "mytime3.h"

int main()

{

using std::cout;

using std::endl;

Time aida(3, 35);

Time tosca(2, 48);

Time temp;

cout << "Aida and Tosca:\n";

cout << aida << ";" << tosca << endl;

temp = aida + tosca;

cout << "Aida + Tosca: " << temp << endl;

temp = aida\*1.17;

cout << "Aida\*1.17:" << temp << endl;

cout << "10.0\*Tosca: " << 10.0\*tosca << endl;

system("pause");

return 0;

}

5.重新编写Stonewt类（程序清单11.16和程序清单11.17），使它有一个状态成员，由该状态成员控制对象应转换为英石格式、整数磅格式还是浮点磅格式。重载<<运算符，使用它来替换show\_stn()和show\_lbs()方法。重载加法、减法和重发运算符，以便可以对Stonewt值进行加、减、乘运算。编写一个使用所有类方法和友元的小程序，来测试这个类。

答：

// stonewt.h

#ifndef STONEWT\_H\_

#define STONEWT\_H\_

class Stonewt

{

private:

enum { Lbs\_per\_stn = 14 };

enum MODE{STONE,INTPOUNDS,FLOATPOUNDS};

int stone;

double pds\_left;

double pounds;

int Mode;

public:

Stonewt(double lbs);

Stonewt(int stn, double lbs);

Stonewt();

~Stonewt();

void setmode();

friend std::ostream& operator<<(std::ostream& os, Stonewt &m);

Stonewt operator+(Stonewt&m);

Stonewt operator-(Stonewt&m);

Stonewt operator\*(double m);

};

#endif

//stonewt.cpp

#include<iostream>

using std::cout;

using std::endl;

#include "stonewt.h"

enum MODE { STONE, INTPOUNDS, FLOATPOUNDS };

Stonewt::Stonewt(double lbs)

{

stone = int(lbs) / Lbs\_per\_stn;

pds\_left = int(lbs) % Lbs\_per\_stn + lbs - int(lbs);

pounds = lbs;

Mode = INTPOUNDS;

}

Stonewt::Stonewt(int stn, double lbs)

{

stone = stn;

pds\_left = lbs;

pounds = stn\*Lbs\_per\_stn + lbs;

Mode = STONE;

}

Stonewt::Stonewt()

{

stone = pounds = pds\_left = 0;

}

Stonewt::~Stonewt()

{

}

void Stonewt::setmode()

{

cout << "你想要以何种方式输出数据：" << endl;

cout << "a.x英石y磅\tb.xx磅\tc.xx.yy磅" << endl;

char a;

std::cin >> a;

while(!isalpha(a))

{

cout << "输入错误，请重新输入：";

std::cin.clear();

std::cin.sync();

std::cin >> a;

}

switch (a)

{

case'A':

case'a':Mode = STONE;

break;

case'B':

case'b':Mode = INTPOUNDS;

break;

case'C':

case'c':Mode = FLOATPOUNDS;

break;

default:cout << "输入错误，默认以x英石y磅格式输出。" << endl;

Mode = STONE;

break;

}

}

std::ostream& operator<<(std::ostream& os,Stonewt &m)

{

if (m.Mode == STONE)

{

os << m.stone << "英石" << m.pds\_left << "磅";

}

else if (m.Mode == INTPOUNDS)

{

os << int(m.pounds) << "磅";

}

else if (m.Mode == FLOATPOUNDS)

{

os << m.pounds << "磅";

}

else

{

os << "出错，无法输出。";

}

return os;

}

Stonewt Stonewt::operator+(Stonewt&m)

{

Stonewt q;

q.pounds = pounds + m.pounds;

q.stone = int(q.pounds) / Lbs\_per\_stn;

q.pds\_left = int(q.pounds) % Lbs\_per\_stn + q.pounds - int(q.pounds);

q.Mode = STONE;

return q;

}

Stonewt Stonewt::operator-(Stonewt&m)

{

Stonewt q;

q.pounds = pounds - m.pounds;

q.stone = int(q.pounds) / Lbs\_per\_stn;

q.pds\_left = int(q.pounds) % Lbs\_per\_stn + q.pounds - int(q.pounds);

q.Mode = STONE;

return q;

}

Stonewt Stonewt::operator\*(double m)

{

Stonewt q;

q.pounds = pounds\*m;

q.stone = int(q.pounds) / Lbs\_per\_stn;

q.pds\_left = int(q.pounds) % Lbs\_per\_stn + q.pounds - int(q.pounds);

q.Mode = STONE;

return q;

}

#include<iostream>

#include"stonewt.h"

int main()

{

using namespace std;

Stonewt a(100);

Stonewt b(50, 10.3);

cout << "a=" << a << endl;

cout << "b=" << b << endl;

Stonewt c = a + b;

cout << "c=a+b=" << c << endl;

c.setmode();

cout << "c=" << c << endl;

Stonewt d = b - a;

cout << "d=b-a=" << d << endl;

Stonewt e = a\*3.3;

cout << "e=a\*3.3=" << e << endl;

system("pause");

return 0;

}

显示：

a=100磅

b=50英石10.3磅

c=a+b=57英石12.3磅

你想要以何种方式输出数据：

a.x英石y磅 b.xx磅 c.xx.yy磅

c

c=810.3磅

d=b-a=43英石8.3磅

e=a\*3.3=23英石8磅

请按任意键继续. . .

6.重新编写Stonewt类（程序清单11.16和程序清单11.17），重载全部6个关系运算符。运算符对pounds成员进行比较，并返回一个bool值。编写一个程序，它声明一个包含6个Stonewt对象的数组，并在数组声明中初始化前3个对象。然后使用循环来读取用于设置剩余3个数组元素的值。接着报告最小的元素、最大的元素以及大于或等于11英石的元素的数量（最简单的方法是创建一个Stonewt对象，并将其初始化为11英石，然后将其同其他对象进行比较）。

答：

// stonewt.h

#ifndef STONEWT\_H\_

#define STONEWT\_H\_

class Stonewt

{

private:

enum { Lbs\_per\_stn = 14 };

enum MODE{STONE,INTPOUNDS,FLOATPOUNDS};

int stone;

double pds\_left;

double pounds;

int Mode;

public:

Stonewt(double lbs);

Stonewt(int stn, double lbs);

Stonewt();

~Stonewt();

void setmode();

friend std::ostream& operator<<(std::ostream& os, Stonewt &m);

Stonewt operator+(Stonewt&m);

Stonewt operator-(Stonewt&m);

Stonewt operator\*(double m);

bool operator<(Stonewt&m);

bool operator<=(Stonewt&m);

bool operator>(Stonewt&m);

bool operator>=(Stonewt&m);

bool operator==(Stonewt&m);

bool operator!=(Stonewt&m);

};

#endif

//stonewt.cpp

#include<iostream>

using std::cout;

using std::endl;

#include "stonewt.h"

enum MODE { STONE, INTPOUNDS, FLOATPOUNDS };

Stonewt::Stonewt(double lbs)

{

stone = int(lbs) / Lbs\_per\_stn;

pds\_left = int(lbs) % Lbs\_per\_stn + lbs - int(lbs);

pounds = lbs;

Mode = INTPOUNDS;

}

Stonewt::Stonewt(int stn, double lbs)

{

stone = stn;

pds\_left = lbs;

pounds = stn\*Lbs\_per\_stn + lbs;

Mode = STONE;

}

Stonewt::Stonewt()

{

stone = pounds = pds\_left = 0;

}

Stonewt::~Stonewt()

{

}

void Stonewt::setmode()

{

cout << "你想要以何种方式输出数据：" << endl;

cout << "a.x英石y磅\tb.xx磅\tc.xx.yy磅" << endl;

char a;

std::cin >> a;

while(!isalpha(a))

{

cout << "输入错误，请重新输入：";

std::cin.clear();

std::cin.sync();

std::cin >> a;

}

switch (a)

{

case'A':

case'a':Mode = STONE;

break;

case'B':

case'b':Mode = INTPOUNDS;

break;

case'C':

case'c':Mode = FLOATPOUNDS;

break;

default:cout << "输入错误，默认以x英石y磅格式输出。" << endl;

Mode = STONE;

break;

}

}

std::ostream& operator<<(std::ostream& os,Stonewt &m)

{

if (m.Mode == STONE)

{

os << m.stone << "英石" << m.pds\_left << "磅";

}

else if (m.Mode == INTPOUNDS)

{

os << int(m.pounds) << "磅";

}

else if (m.Mode == FLOATPOUNDS)

{

os << m.pounds << "磅";

}

else

{

os << "出错，无法输出。";

}

return os;

}

Stonewt Stonewt::operator+(Stonewt&m)

{

Stonewt q;

q.pounds = pounds + m.pounds;

q.stone = int(q.pounds) / Lbs\_per\_stn;

q.pds\_left = int(q.pounds) % Lbs\_per\_stn + q.pounds - int(q.pounds);

q.Mode = STONE;

return q;

}

Stonewt Stonewt::operator-(Stonewt&m)

{

Stonewt q;

q.pounds = pounds - m.pounds;

q.stone = int(q.pounds) / Lbs\_per\_stn;

q.pds\_left = int(q.pounds) % Lbs\_per\_stn + q.pounds - int(q.pounds);

q.Mode = STONE;

return q;

}

Stonewt Stonewt::operator\*(double m)

{

Stonewt q;

q.pounds = pounds\*m;

q.stone = int(q.pounds) / Lbs\_per\_stn;

q.pds\_left = int(q.pounds) % Lbs\_per\_stn + q.pounds - int(q.pounds);

q.Mode = STONE;

return q;

}

bool Stonewt::operator<(Stonewt&m)

{

if (pounds < m.pounds)return true;

else return false;

}

bool Stonewt::operator<=(Stonewt&m)

{

if (pounds <= m.pounds)return true;

else return false;

}

bool Stonewt::operator>(Stonewt&m)

{

if (pounds > m.pounds)return true;

else return false;

}

bool Stonewt::operator>=(Stonewt&m)

{

if (pounds >= m.pounds)return true;

else return false;

}

bool Stonewt::operator==(Stonewt&m)

{

if (pounds == m.pounds)return true;

else return false;

}

bool Stonewt::operator!=(Stonewt&m)

{

if (pounds != m.pounds)return true;

else return false;

}

#include<iostream>

#include"stonewt.h"

int main()

{

using namespace std;

Stonewt m[6] = { 100,150,170.4 };

for (int i = 3;i < 6;i++)

{

double q;

cout << "请输入第：" << i + 1 << "个成员的重量（单位：磅）：";

cin >> q;

m[i] = q;

}

Stonewt over = 11 \* 14; //标准11英石对象

Stonewt max, min;

max = min = m[0];

int ma, mi, ov=0;

for (int i = 1;i < 6;i++)

{

if (max < m[i])

{

max = m[i];

ma = i + 1;

}

if (min > m[i])

{

min = m[i];

mi = i + 1;

}

if (m[i] >= over)ov++;

}

cout << "最大是的第" << ma << "个，重量为：" << max << endl;

cout << "最小是的第" << mi << "个，重量为：" << min << endl;

cout << "超过" << ov << "个大于等于11英石。" << endl;

system("pause");

return 0;

}

显示：

请输入第：4个成员的重量（单位：磅）：400

请输入第：5个成员的重量（单位：磅）：70

请输入第：6个成员的重量（单位：磅）：190

最大是的第4个，重量为：400磅

最小是的第5个，重量为：70磅

超过3个大于等于11英石。

请按任意键继续. . .

7.复数有两个部分组成：实数部分和虚数部分。复数的一种书写方式是：（3.0,4.0），其中，3.0是实数部分，4.0是虚数部分。假设a=(A, Bi)， c= (C, Di)，则下面是一些复数运算。

①加法：a + c = (A+C, (B+D)i)

②减法：a - c = (A-C, (B-D)i)

③乘法：a \* c = (A\*C-B\*D, (A\*D + B\*C )i)

④乘法：：x \* c = (x \* C, x \* Di)，其中x为实数

⑤共轭： ~a = (A, -Bi)

请定义一个复数类，以便下面的程序可以使用它来获得正确的结果。

#include<iostream>

using namespace std;

#include "complex0.h" //to avoid confusion with complex.h

int main()  
{  
 complex a (3.0, 4.0); // initialize to (3,4i)

complex c;

cout << "Enter a complex number (q to quit):\n";

while (cin >> c)

{

cout << "c is " << c << '\n';

cout << "complex conjugate is " << ~c << '\n';

cout << "a is " << a << '\n';

cout << " a + c is " << a + c << '\n';

cout << " a - c is " << a - c << '\n';

cout << " a \* c is " << a \* c << '\n';

cout << " 2 \* c is " << 2 \* c << '\n';

cout << "Enter a complex number (q to quit):\n";

}

cout << "Done!\n";

return 0;

}

注意，必须重载运算符<<和>>。标准C++使用头文件complex提供了比这个示例更广泛的复数支持，因此应将自定义的头文件命名为complex0.h，以免发生冲突。应尽可能使用const。

下面是该程序的运行情况。

Enter a complex number (q to quit):

real: 10

imaginary: 12

c is (10,12i)

complex conjugate is (10, -12i)

a is (3,4f)

a + c is (13,16i)

a - c is (-7, -8i)

a \* c is (-18, 76i)

2 \* c is (20.24i)

Enter a complex number (q to quit):

real: q

Done!

请注意，经过重载后，cin>>c将提示用户输入实数和虚数部分。

答：

//complex0.h

#ifndef COMPLEX0\_

#define COMPLEX0\_

#include<iostream>

class complex

{

double a;

double b;

public:

complex() {};

complex(double A, double B) { a = A;b = B; }

complex operator+(complex& m)const;

complex operator-(complex& m)const;

complex operator\*(complex& m)const;

complex operator~();

friend complex operator\*(double a, complex& m);

friend std::ostream& operator<<(std::ostream&os, const complex &m);

friend std::istream& operator>>(std::istream&is, complex&m);

};

#endif // !COMPLEX0

//complex0.cpp

#include<iostream>

#include"complex0.h"

complex complex::operator+(complex& m)const

{

complex q;

q.a = a + m.a;

q.b = b + m.b;

return q;

}

complex complex::operator-(complex& m)const

{

complex q;

q.a = a - m.a;

q.b = b - m.b;

return q;

}

complex complex::operator\*(complex& m)const

{

complex q;

q.a = a\*m.a - b\*m.b;

q.b = a\*m.b + b\*m.a;

return q;

}

complex complex::operator~()

{

complex q;

q.a = a;

q.b = -b;

return q;

}

complex operator\*(double a, complex& m)

{

complex q;

q.a = m.a\*a;

q.b = m.b\*a;

return q;

}

std::ostream& operator<<(std::ostream&os, const complex &m)

{

os << "(" << m.a << "," << m.b << "i)";

return os;

}

std::istream& operator>>(std::istream& is, complex&m)

{

double x, y;

std::cout << "real: ";

is >> x;

if (!is)return is;

std::cout << "imagiary: ";

is >> y;

m.a = x;

m.b = y;

return is;

}

//1.cpp main函数，用于测试

#include<iostream>

using namespace std;

#include "complex0.h" //to avoid confusion with complex.h

int main()

{

complex a(3.0, 4.0); // initialize to (3,4i)

complex c;

cout << "Enter a complex number (q to quit):\n";

while (cin >> c)

{

cout << "c is " << c << '\n';

cout << "complex conjugate is " << ~c << '\n';

cout << "a is " << a << '\n';

cout << " a + c is " << a + c << '\n';

cout << " a - c is " << a - c << '\n';

cout << " a \* c is " << a \* c << '\n';

cout << " 2 \* c is " << 2 \* c << '\n';

cout << "Enter a complex number (q to quit):\n";

}

cout << "Done!\n";

system("pause"); //加上这段，否则我的编译器上窗口会一闪而过

return 0;

}

显示结果和要求完全相同。