第10章 运算符重载

10.1 什么是运算符重载

所谓重载，就是重新赋予新的含义。函数重载就是对一个已有的函数赋予新的含义，使之实现新功能。

运算符也可以重载。实际上，我们已经在不知不觉之中使用了运算符重载。

现在要讨论的问题是：用户能否根据自己的需要对C++已提供的运算符进行重载，赋予它们新的含义，使之一名多用。譬如，能否用“+”号进行两个复数的相加。在C++中不能在程序中直接用运算符“+”对复数进行相加运算。用户必须自己设法实现复数相加。例如用户可以通过定义一个专门的函数来实现复数相加。见例10.1。

例10.1 通过函数来实现复数相加。

```
#include <iostream>
using namespace std;
class Complex                          //定义Complex类
{public:
Complex( ){real=0;imag=0;}             //定义构造函数
Complex(double r,double i){real=r;imag=i;}   //构造函数重载
Complex complex_add(Complex &c2);      //声明复数相加函数
void display( );                       //声明输出函数
 private:
double real;                           //实部
double imag;                           //虚部
};

Complex Complex::complex_add(Complex &c2)
{Complex c;
c.real=real+c2.real;
```

```
c.imag=imag+c2.imag;
return c;}

void Complex::display( )                    //定义输出函数
{cout<<"("<<real<<","<<imag<<"i)"<<endl;}

int main( )
{Complex c1(3,4),c2(5,-10),c3;              //定义3个复数对象
c3=c1.complex_add(c2);                      //调用复数相加函数
cout<<"c1="; c1.display( );                 //输出c1的值
cout<<"c2="; c2.display( );                 //输出c2的值
cout<<"c1+c2="; c3.display( );              //输出c3的值
return 0;
}
```

运行结果如下：

```
c1=(3+4i)
c2=(5-10i)
c1+c2=(8,-6i)
```

结果无疑是正确的，但调用方式不直观、太烦琐，使人感到很不方便。能否也和整数的加法运算一样，直接用加号“+”来实现复数运算呢？如

```
c3=c1+c2;
```

编译系统就会自动完成**c1**和**c2**两个复数相加的运算。如果能做到，就为对象的运算提供了很大的方便。这就需要对运算符“+”进行重载。

10.2 运算符重载的方法

运算符重载的方法是定义一个重载运算符的函数，在需要执行被重载的运算符时，系统就自动调用该函数，以实现相应的运算。也就是说，运算符重载是通过定义函数实现的。运算符重载实质上是函数的重载。

重载运算符的函数一般格式如下：

函数类型 **operator** 运算符名称 **(**形参表列**)**

{ 对运算符的重载处理 **}**

例如，想将“+”用于**Complex**类(复数)的加法运算，函数的原型可以是这样的：

```
Complex operator+ (Complex& c1,Complex& c2);
```

在定义了重载运算符的函数后，可以说： 函数**operator+**重载了运算符+。

为了说明在运算符重载后，执行表达式就是调用函数的过程，可以把两个整数相加也想像为调用下面的函数：

```
int operator + (int a,int b)
{return (a+b);}
```

如果有表达式**5+8**，就调用此函数，将**5**和**8**作为调用函数时的实参，函数的返回值为**13**。这就是用函数的方法理解运算符。

可以在例**10.1**程序的基础上重载运算符“**+**”，使之用于复数相加。

例10.2 改写例10.1，重载运算符“+”，使之能用于两个复数相加。

```
#include <iostream>
using namespace std;
class Complex
{public:
Complex( ){real=0;imag=0;}
Complex(double r,double i){real=r;imag=i;}
Complex operator+(Complex &c2);            //声明重载运算符的函数
void display( );
 private:
double real;
double imag;
};
Complex Complex::operator+(Complex &c2)        //定义重载运算符的函数
{ Complex c;
c.real=real+c2.real;
c.imag=imag+c2.imag;
```

```
return c;}

void Complex::display( )
{ cout<<"("<<real<<","<<imag<<"i)"<<endl;}

int main( )
{ Complex c1(3,4),c2(5,-10),c3;
c3=c1+c2;                              //运算符+用于复数运算
cout<<"c1=";c1.display( );
cout<<"c2=";c2.display( );
cout<<"c1+c2=";c3.display( );
return 0;
}
```

运行结果与例**10.1**相同：

```
c1=(3+4i)
c2=(5-10i)
c1+c2=(8,-6i)
```

请比较例**10.1**和例**10.2**，只有两处不同：

(1) 在例**10.2**中以**operator+**函数取代了例**10.1**中的**complex_add**函数，而且只是函数名不同，函数体和函数返回值的类型都是相同的。

(2) 在**main**函数中，以**“c3=c1+c2;”**取代了例**10.1**中的**“c3=c1.complex_add(c2);”**。在将运算符**+**重载为类的成员函数后，**C++**编译系统将程序中的表达式**c1+c2**解释为

```
c1.operator+(c2)                    //其中c1和c2是Complex类的对象
```

即以**c2**为实参调用**c1**的运算符重载函数**operator+(Complex &c2)**，进行求值，得到两个复数之和。

虽然重载运算符所实现的功能完全可以用函数实现，但是使用运算符重载能使用户程序易于编写、阅读和维护。在实际工作中，类的声明和类的使用往往是分离的。假如在声明**Complex**类时，对运算符**+,-,*,/**都进行了重载，那么使用这个类的用户在编程时可以完全不考虑函数是怎么实现的，放心大胆地直接使用**+,-,*,/**进行复数的运算即可，十分方便。

对上面的运算符重载函数**operator+**还可以改写得更简练一些：

```
Complex Complex::operator + (Complex &c2)
{return Complex(real+c2.real, imag+c2.imag);}
```

需要说明的是： 运算符被重载后，其原有的功能仍然保留，没有丧失或改变。

通过运算符重载，扩大了C++已有运算符的作用范围，使之能用于类对象。

运算符重载对C++有重要的意义，把运算符重载和类结合起来，可以在C++程序中定义出很有实用意义而使用方便的新的数据类型。运算符重载使C++具有更强大的功能、更好的可扩充性和适应性，这是C++最吸引人的特点之一。

10.3 重载运算符的规则

(1) C++不允许用户自己定义新的运算符，只能对已有的C++运算符进行重载。

（2） C++允许重载的运算符

C++中绝大部分的运算符允许重载。具体规定见书中表10.1。

不能重载的运算符只有5个：

. (成员访问运算符)

.* (成员指针访问运算符)

:: (域运算符)

sizeof (长度运算符)

?: (条件运算符)

前两个运算符不能重载是为了保证访问成员的功能不能被改变，域运算符和**sizeof**运算符的运算对象是类型而不是变量或一般表达式，不具重载的特征。

(3) 重载不能改变运算符运算对象**(**即操作数**)**的个数。

(4) 重载不能改变运算符的优先级别。

(5) 重载不能改变运算符的结合性。

(6) 重载运算符的函数不能有默认的参数，否则就改变了运算符参数的个数，与前面第**(3)**点矛盾。

(7) 重载的运算符必须和用户定义的自定义类型的对象一起使用，其参数至少应有一个是类对象**(**或类对象的引用**)**。也就是说，参数不能全部是**C++**的标准类型，以防止用户修改用于标准类型数据的运算符的性质。

(8) 用于类对象的运算符一般必须重载，但有两个例外，运算符“=”和“&”不必用户重载。

① 赋值运算符(=)可以用于每一个类对象，可以利用它在同类对象之间相互赋值。

② 地址运算符&也不必重载，它能返回类对象在内存中的起始地址。

(9) 应当使重载运算符的功能类似于该运算符作用于标准类型数据时所实现的功能。

(10) 运算符重载函数可以是类的成员函数(如例**10.2**)，也可以是类的友元函数，还可以是既非类的成员函数也不是友元函数的普通函数。

10.4 运算符重载函数作为类成员函数和友元函数

在本章例**10.2**程序中对运算符“**+**”进行了重载，使之能用于两个复数的相加。在该例中运算符重载函数**operator+**作为**Complex**类中的成员函数。

“**+**”是双目运算符，为什么在例**10.2**程序中的重载函数中只有一个参数呢？实际上，运算符重载函数有两个参数，由于重载函数是**Complex**类中的成员函数，有一个参数是隐含的，运算符函数是用**this**指针隐式地访问类对象的成员。

可以看到，重载函数**operator+**访问了两个对象中的成员，一个是**this**指针指向的对象中的成员，一个是形参对象中的成员。如**this->real+c2.real**，**this->real**就是**c1.real**。

在**10.2**节中已说明，在将运算符函数重载为成员函数后，如果出现含该运算符的表达式，如**c1+c2,**编译系统把它解释为

c1.operator+(c2)

即通过对象**c1**调用运算符重载函数，并以表达式中第二个参数**(**运算符右侧的类对象**c2)**作为函数实参。运算符重载函数的返回值是**Complex**类型，返回值是复数**c1**和**c2**之和**(Complex(c1.real + c2.real,c1.imag+c2.imag))**。

运算符重载函数除了可以作为类的成员函数外，还可以是非成员函数。可以将例**10.2**改写为例**10.3**。

例**10.3** 将运算符“**+**”重载为适用于复数加法，重载函数不作为成员函数，而放在类外，作为**Complex**类的友元函数。

```
#include <iostream>
using namespace std;
class Complex
{public:
Complex( ){real=0;imag=0;}
Complex(double r,double i){real=r;imag=i;}
friend Complex operator + (Complex &c1,Complex &c2);//重载函数作为友元函数
void display( );
 private:
double real;
double imag;
```

```
};

Complex operator + (Complex &c1,Complex &c2)        //定义作为友元函数的重载函数
{return Complex(c1.real+c2.real, c1.imag+c2.imag);}

void Complex∶∶display( )
{cout<<"("<<real<<","<<imag<<"i)"<<endl;}
int main( )
{Complex c1(3,4),c2(5,-10),c3;
c3=c1+c2;
cout<<"c1="; c1.display( );
cout<<"c2="; c2.display( );
cout<<"c1+c2 ="; c3.display( );
}
```

与例**10.2**相比较，只作了一处改动，将运算符函数不作为成员函数，而把它放在类外，在**Complex**类中声明它为友元函数。同时将运算符函数改为有两个参数。在将运算符“**+**”重载为非成员函数后，**C++**编译系统将程序中的表达式**c1+c2**解释为

```
operator+(c1,c2)
```

即执行**c1+c2**相当于调用以下函数：

```
Complex operator + (Complex &c1,Complex &c2)
{return Complex(c1.real+c2.real, c1.imag+c2.imag);}
```

求出两个复数之和。运行结果同例**10.2**。

为什么把运算符函数作为友元函数呢？因为运算符函数要访问**Complex**类对象中的成员。如果运算符函数不是**Complex**类的友元函数，而是一个普通的函数，它是没有权利访问**Complex**类的私有成员的。

在**10.2**节中曾提到过： 运算符重载函数可以是类的成员函数，也可以是类的友元函数，还可以是既非类的成员函数也不是友元函数的普通函数。现在分别讨论这**3**种情况。

首先，只有在极少的情况下才使用既不是类的成员函数也不是友元函数的普通函数，原因是上面提到的，普通函数不能直接访问类的私有成员。

在剩下的两种方式中，什么时候应该用成员函数方式，什么时候应该用友元函数方式？二者有何区别呢？如果将运算符重载函数作为成员函数，它可以通过**this**指针自由地访问本类的数据成员，因此可以少写一个函数的参数。但必须要求运算表达式第一个参数(即运算符左侧的操作数)是一个类对象，

而且与运算符函数的类型相同。因为必须通过类的对象去调用该类的成员函数，而且只有运算符重载函数返回值与该对象同类型，运算结果才有意义。在例**10.2**中，表达式**c1+c2**中第一个参数**c1**是**Complex**类对象，运算符函数返回值的类型也是**Complex**，这是正确的。如果**c1**不是**Complex**类，它就无法通过隐式**this**指针访问**Complex**类的成员了。如果函数返回值不是**Complex**类复数，显然这种运算是没有实际意义的。

如想将一个复数和一个整数相加，如**c1+i**，可以将运算符重载函数作为成员函数，如下面的形式：

```
Complex Complex::operator+(int &i)     //运算符重载函数作为Complex类的成员函数
{return Complex(real+i,imag);}
```

注意在表达式中重载的运算符“+”左侧应为**Complex**类的对象，如

```
c3=c2+i;
```

不能写成

```
c3=i+c2;          //运算符“+”的左侧不是类对象，编译出错
```

如果出于某种考虑，要求在使用重载运算符时运算符左侧的操作数是整型量（如表达式**i+c2**，运算符左侧的操作数**i**是整数），这时是无法利用前面定义的重载运算符的，因为无法调用**i.operator+**函数。可想而知，如果运算符左侧的操作数属于**C++**标准类型**(**如**int)**或是一个其他类的对象，则运算符重载函数不能作为成员函数，只能作为非成员函数。如果函数需要访问类的私有成员，则必须声明为友元函数。可以在**Complex**类中声明：

friend Complex operator+(int &i,Complex &c); //第一个参数可以不是类对象

在类外定义友元函数：

Complex operator+(int &i, Complex &c) //运算符重载函数不是成员函数

{return Complex(i+c.real,c.imag);}

将双目运算符重载为友元函数时，在函数的形参表列中必须有两个参数，不能省略，形参的顺序任意，不要求第一个参数必须为类对象。但在使用运算符的表达式中，要求运算符左侧的操作数与函数第一个参数对应，运算符右侧的操作数与函数的第二个参数对应。如

c3=i+c2; //正确，类型匹配

c3=c2+i; //错误，类型不匹配

请注意，数学上的交换律在此不适用。如果希望适用交换律，则应再重载一次运算符“+”。如

```
Complex operator+(Complex &c, int &i)          //此时第一个参数为类对象
{return Complex(i+c.real,c.imag);}
```

这样，使用表达式**i+c2**和**c2+i**都合法，编译系统会根据表达式的形式选择调用与之匹配的运算符重载函数。可以将以上两个运算符重载函数都作为友元函数，也可以将一个运算符重载函数(运算符左侧为对象名的) 作为成员函数，另一个(运算符左侧不是对象名的)作为友元函数。但不可能将两个都作为成员函数，原因是显然的。

C++规定，有的运算符(如赋值运算符、下标运算符、函数调用运算符)必须定义为类的成员函数，有的运算符则不能定义为类的成员函数(如流插入“<<”和流提取运算符“>>”、类型转换运算符)。

由于友元的使用会破坏类的封装，因此从原则上说，要尽量将运算符函数作为成员函数。但考虑到各方面的因素，一般将单目运算符重载为成员函数，将双目运算符重载为友元函数。在学习了本章第**10.7**节例**10.9**的讨论后，读者对此会有更深入的认识。

说明： 有的**C++**编译系统(如**Visual C++ 6.0**)没有完全实现**C++**标准，它所提供不带后缀**.h**的头文件不支持把成员函数重载为友元函数。上面例**10.3**程序在**GCC**中能正常运行，而在**Visual C++ 6.0**中会编译出错。但是**Visual C++**所提供的老形式的带后缀**.h**的头文件可以支持此项功能，因此可以将程序头两行修改如下，即可顺利运行：

```
#include <iostream.h>
```

以后如遇到类似情况，亦可照此办理。

10.5 重载双目运算符

双目运算符(或称二元运算符)是C++中最常用的运算符。双目运算符有两个操作数，通常在运算符的左右两侧，如3+5,a=b,i<10等。在重载双目运算符时，不言而喻在函数中应该有两个参数。下面再举一个例子说明重载双目运算符的应用。

例**10.4** 定义一个字符串类**String**，用来存放不定长的字符串，重载运算符“==”,“<”和“>”，用于两个字符串的等于、小于和大于的比较运算。

为了使读者便于理解程序，同时也使读者了解建立程序的步骤，下面分几步来介绍编程过程。

(1) 先建立一个**String**类：

```
#include <iostream>
using namespace std;
class String
{public:
String( ){p=NULL;}                    //默认构造函数
String(char *str);                    //构造函数
void display( );
private:
char *p;                              //字符型指针，用于指向字符串
};
```

```
String::String(char *str)                    //定义构造函数
{p=str;}                                     //使p指向实参字符串

void String::display( )                      //输出p所指向的字符串
{cout<<p;}

int main( )
{String string1("Hello"),string2("Book");
string1.display( );
cout<<endl;
string2.display( );
return 0;
}
```

运行结果为

```
Hello
Book
```

(2) 有了这个基础后，再增加其他必要的内容。现在增加对运算符重载的部分。为便于编写和调试，先重载一个运算符“>”。程序如下：

```
#include <iostream>
#include <string>
using namespace std;
class String
{public:
String( ){p=NULL;}
String(char *str);
friend bool operator>(String &string1,String &string2);//声明运算符函数为友元函数
void display( );
private:
char *p;                                    /字符型指针，用于指向字符串
};
String::String(char *str)
{p=str;}
```

```
void String::display( )                    //输出p所指向的字符串
{cout<<p;}
bool operator>(String &string1,String &string2)     //定义运算符重载函数
{if(strcmp(string1.p,string2.p)>0)
return true;
else return false;
}

int main( )
{String string1("Hello"),string2("Book");
cout<<(string1>string2)<<endl;
}
```

程序运行结果为**1**。

这只是一个并不很完善的程序，但是，已经完成了实质性的工作了，运算符重载成功了。其他两个运算符的重载如法炮制即可。

(3) 扩展到对**3**个运算符重载。

在**String**类体中声明**3**个成员函数：

```
friend bool operator> (String &string1, String &string2);
friend bool operator< (String &string1, String &string2);
friend bool operator==(String &string1, String& string2);
```

在类外分别定义**3**个运算符重载函数：

```
bool operator>(String &string1,String &string2)        //对运算符“>”重载
{if(strcmp(string1.p,string2.p)>0)
return true;
else
return false;
}

bool operator<(String &string1,String &string2)        //对运算符“<”重载
{if(strcmp(string1.p,string2.p)<0)
return true;
else
```

```
return false;
}

bool operator==(String &string1,String &string2)      //对运算符“==”重载
{if(strcmp(string1.p,string2.p)==0)
return true;
else
return false;
}
```

再修改主函数：

```
int main( )
{String string1("Hello"),string2("Book"),string3("Computer");
cout<<(string1>string2)<<endl;          //比较结果应该为true
cout<<(string1<string3)<<endl;          //比较结果应该为false
cout<<(string1==string2)<<endl;         //比较结果应该为false
return 0;
}
```

运行结果为

```
1
0
0
```

结果显然是对的。到此为止，主要任务基本完成。

(4) 再进一步修饰完善，使输出结果更直观。下面给出最后的程序。

```
#include <iostream>
using namespace std;
class String
{public:
String( ){p=NULL;}
String(char *str);
friend bool operator>(String &string1,String &string2);
friend bool operator<(String &string1,String &string2);
friend bool operator==(String &string1,String &string2);
```

```
void display( );
private:
char *p;
};
String::String(char *str)
{p=str;}

void String::display( )                    //输出p所指向的字符串
{cout<<p;}

bool operator>(String &string1,String &string2)
{if(strcmp(string1.p,string2.p)>0)
return true;
else
return false;
}

bool operator<(String &string1,String &string2)
```

```
{if(strcmp(string1.p,string2.p)<0)
return true;
else
return false;
}

bool operator==(String &string1,String &string2)
{if(strcmp(string1.p,string2.p)==0)
return true;
else
return false;
}

void compare(String &string1,String &string2)
{if(operator>(string1,string2)==1)
{string1.display( );cout<<">";string2.display( );}
else
if(operator<(string1,string2)==1)
```

```
{string1.display( );cout<<"<";string2.display( );}
else
if(operator==(string1,string2)==1)
{string1.display( );cout<<"=";string2.display( );}
cout<<endl;
}
int main( )
{String string1("Hello"),string2("Book"),string3("Computer"),string4("Hello");
compare(string1,string2);
compare(string2,string3);
compare(string1,string4);
return 0;
}
```

运行结果为

```
Hello>Book
Book<Computer
Hello==Hello
```

增加了一个**compare**函数，用来对两个字符串进行比较，并输出相应的信息。这样可以减轻主函数的负担，使主函数简明易读。

通过这个例子，不仅可以学习到有关双目运算符重载的知识，而且还可以学习怎样去编写**C++**程序。

这种方法的指导思想是： 先搭框架，逐步扩充，由简到繁，最后完善。边编程，边调试，边扩充。千万不要企图在一开始时就解决所有的细节。类是可扩充的，可以一步一步地扩充它的功能。最好直接在计算机上写程序，每一步都要上机调试，调试通过了前面一步再做下一步，步步为营。这样编程和调试的效率是比较高的。读者可以试验一下。

10.6 重载单目运算符

单目运算符只有一个操作数，如**!a**，**-b**，**&c**，***p**，还有最常用的**++i**和**--i**等。重载单目运算符的方法与重载双目运算符的方法是类似的。但由于单目运算符只有一个操作数，因此运算符重载函数只有一个参数，如果运算符重载函数作为成员函数，则还可省略此参数。

下面以自增运算符“**++**”为例，介绍单目运算符的重载。

例10.5 有一个Time类，包含数据成员minute(分)和sec(秒)，模拟秒表，每次走一秒，满60秒进一分钟，此时秒又从0开始算。要求输出分和秒的值。

```
#include <iostream>
using namespace std;
class Time
{public:
Time( ){minute=0;sec=0;}                          //默认构造函数
Time(int m,int s):minute(m),sec(s){ }             //构造函数重载
Time operator++( );                               //声明运算符重载函数
void display( ){cout<<minute<<":"<<sec<<endl;}    //定义输出时间函数
private:
int minute;
int sec;
};
Time Time::operator++( )                          //定义运算符重载函数
{if(++sec>=60)
```

```
{sec-=60;                          //满60秒进1分钟
++minute;}
return *this;                      //返回当前对象值
}
int main( )
{Time time1(34,0);
for (int i=0;i<61;i++)
{++time1;
time1.display( );}
return 0;
}
```

运行情况如下：

```
34:1
34:2
 ┆
34:59
35:0
35:1          (共输出61行)
```

可以看到： 在程序中对运算符“++”进行了重载，使它能用于**Time**类对象。“++”和“--”运算符有两种使用方式，前置自增运算符和后置自增运算符，它们的作用是不一样的，在重载时怎样区别这二者呢？

针对“++”和“--”这一特点，**C++**约定： 在自增(自减)运算符重载函数中，增加一个**int**型形参，就是后置自增(自减)运算符函数。

例**10.6** 在例**10.5**程序的基础上增加对后置自增运算符的重载。修改后的程序如下：

```
#include <iostream>
using namespace std;
class Time
{public:
Time( ){minute=0;sec=0;}
Time(int m,int s):minute(m),sec(s){}
```

```
Time operator++( );                    //声明前置自增运算符“++”重载函数
Time operator++(int);                  //声明后置自增运算符“++”重载函数
void display( ){cout<<minute<<":"<<sec<<endl;}
private:
int minute;
int sec;
};

Time Time::operator++( )               //定义前置自增运算符“++”重载函数
{if(++sec>=60)
{sec-=60;
++minute;}
return *this;                          //返回自加后的当前对象
}

Time Time::operator++(int)             //定义后置自增运算符“++”重载函数
{Time temp(*this);
sec++;
```

```
if(sec>=60)
{sec-=60;
++minute;}
return temp;                    //返回的是自加前的对象
}

int main( )
{Time time1(34,59),time2;
cout<<" time1 : ";
time1.display( );
++time1;
cout<<"++time1: ";
time1.display( );
time2=time1++;                  //将自加前的对象的值赋给time2
cout<<"time1++: ";
time1.display( );
cout<<" time2 :";
time2.display( );               //输出time2对象的值
}
```

请注意前置自增运算符“++”和后置自增运算符“++”二者作用的区别。前者是先自加，返回的是修改后的对象本身。后者返回的是自加前的对象，然后对象自加。请仔细分析后置自增运算符重载函数。

运行结果如下：

time1 : 34:59 **(time1**原值**)**
++time1: 35:0 **(**执行**++time1**后**time1**的值**)**
time1++: 35:1 **(**再执行**time1++**后**time1**的值**)**
time2 : 35:0 **(time2**保存的是执行**time1++**前**time1**的值**)**

可以看到： 重载后置自增运算符时，多了一个**int**型的参数，增加这个参数只是为了与前置自增运算符重载函数有所区别，此外没有任何作用。编译系统在遇到重载后置自增运算符时，会自动调用此函数。

10.7 重载流插入运算符和流提取运算符

C++的流插入运算符“<<”和流提取运算符“>>”是C++在类库中提供的，所有C++编译系统都在类库中提供输入流类istream和输出流类ostream。cin和cout分别是istream类和ostream类的对象。在类库提供的头文件中已经对“<<”和“>>”进行了重载，使之作为流插入运算符和流提取运算符，能用来输出和输入C++标准类型的数据。因此，在本书前面几章中，凡是用“cout<<”和“cin>>”对标准类型数据进行输入输出的，都要用#include <iostream>把头文件包含到本程序文件中。

用户自己定义的类型的数据，是不能直接用“<<”和“>>”来输出和输入的。如果想用它们输出和输入自己声明的类型的数据，必须对它们重载。

对“<<”和“>>”重载的函数形式如下：

istream & operator >> (istream &,自定义类 **&);**

ostream & operator << (ostream &,自定义类 **&);**

即重载运算符“>>”的函数的第一个参数和函数的类型都必须是**istream&**类型，第二个参数是要进行输入操作的类。重载“<<”的函数的第一个参数和函数的类型都必须是**ostream&**类型，第二个参数是要进行输出操作的类。因此，只能将重载“>>”和“<<”的函数作为友元函数或普通的函数，而不能将它们定义为成员函数。

10.7.1 重载流插入运算符“<<”

在程序中，人们希望能用插入运算符“<<”来输出用户自己声明的类的对象的信息，这就需要重载流插入运算符“<<”。

例10.7 在例10.2的基础上，用重载的“<<”输出复数。

```
#include <iostream>
using namespace std;
class Complex
{public:
Complex( ){real=0;imag=0;}
Complex(double r,double i){real=r;imag=i;}
Complex operator + (Complex &c2);                    //运算符“+”重载为成员函数
friend ostream& operator << (ostream&,Complex&);     //运算符“<<”重载为友元函数
private:
```

```
double real;
double imag;
};

Complex Complex：：operator + (Complex &c2)          //定义运算符“+”重载函数
{return Complex(real+c2.real,imag+c2.imag);}
ostream& operator << (ostream& output,Complex& c)    //定义运算符“<<”重载函数
{output<<"("<<c.real<<"+"<<c.imag<<"i)"<<endl;
return output;
}

int main( )
{Complex c1(2,4),c2(6,10),c3;
c3=c1+c2;
cout<<c3;
return 0;
}
```

(在Visual C++ 6.0环境下运行时，需将第一行改为#include <iostream.h>，并删去第2行。)

运行结果为

(8+14i)

可以看到在对运算符“<<”重载后，在程序中用“<<”不仅能输出标准类型数据，而且可以输出用户自己定义的类对象。用“cout<<c3”即能以复数形式输出复数对象c3的值。形式直观，可读性好，易于使用。

下面对怎样实现运算符重载作一些说明。程序中重载了运算符“<<”，运算符重载函数中的形参output是ostream类对象的引用，形参名output是用户任意起的。分析main函数最后第二行：

cout<<c3;

运算符“<<”的左面是**cout**,前面已提到**cout**是**ostream**类对象。“<<”的右面是**c3**，它是**Complex**类对象。由于已将运算符“<<”的重载函数声明为**Complex**类的友元函数，编译系统把“**cout<<c3**”解释为

operator<<(cout,c3)

即以**cout**和**c3**作为实参，调用下面的**operator<<**函数：

```
ostream& operator<<(ostream& output,Complex& c)
{output<<"("<<c.real<<"+"<<c.imag<<"i)"<<endl;
return output;}
```

调用函数时，形参**output**成为**cout**的引用，形参**c**成为**c3**的引用。因此调用函数的过程相当于执行：

```
cout<<"("<<c3.real<<"+"<<c3.imag<<"i)"<<endl; return cout;
```

请注意： 上一行中的“<<”是C++预定义的流插入符，因为它右侧的操作数是字符串常量和double类型数据。执行cout语句输出复数形式的信息。然后执行return语句。请思考： return output的作用是什么？回答是能连续向输出流插入信息。output是ostream类的对象，它是实参cout的引用，也就是cout通过传送地址给output，使它们二者共享同一段存储单元，或者说output是cout的别名。因此，return output就是return cout，将输出流cout的现状返回，即保留输出流的现状。

请问返回到哪里？刚才是在执行

```
cout<<c3;
```

在已知cout<<c3的返回值是cout的当前值。如果有以下输出：

cout<<c3<<c2;

先处理**cout<<c3**，即

(cout<<c3)<<c2;

而执行**(cout<<c3)**得到的结果就是具有新内容的流对象**cout**，因此，**(cout<<c3)<<c2**相当于**cout(**新值**)<<c2**。运算符“**<<**”左侧是**ostream**类对象**cout**，右侧是**Complex**类对象**c2,**则再次调用运算符“**<<**”重载函数，接着向输出流插入**c2**的数据。现在可以理解了为什么**C++**规定运算符“**<<**”重载函数的第一个参数和函数的类型都必须是**ostream**类型的引用，就是为了返回**cout**的当前值以便连续输出。

请读者注意区分什么情况下的“**<<**”是标准类型数据的流插入符，什么情况下的“**<<**”是重载的流插入符。如

cout<<c3<<5<<endl;

有下划线的是调用重载的流插入符，后面两个“<<”不是重载的流插入符，因为它的右侧不是**Complex**类对象而是标准类型的数据，是用预定义的流插入符处理的。

还有一点要说明： 在本程序中，在**Complex**类中定义了运算符“<<”重载函数为友元函数，因此只有在输出**Complex**类对象时才能使用重载的运算符，对其他类型的对象是无效的。如

cout<<time1; **//time1**是**Time**类对象，不能使用用于**Complex**类的重载运算符

10.7.2 重载流提取运算符“>>”

C++预定义的运算符“>>”的作用是从一个输入流中提取数据，如“cin>>i;”表示从输入流中提取一个整数赋给变量i(假设已定义i为int型)。重载流提取运算符的目的是希望将“>>”用于输入自定义类型的对象的信息。

例10.8 在例10.7的基础上，增加重载流提取运算符“>>”，用“cin>>”输入复数，用“cout<<”输出复数。

```
#include <iostream>
using namespace std;
class Complex
{public:
friend ostream& operator << (ostream&,Complex&);    //声明重载运算符“<<”
friend istream& operator >> (istream&,Complex&);    //声明重载运算符“>>”
private:
double real;
double imag;
};
ostream& operator << (ostream& output,Complex& c)       //定义重载运算符“<<”
{output<<"("<<c.real<<"+"<<c.imag<<"i)";
return output;
}
istream& operator >> (istream& input,Complex& c)     //定义重载运算符“>>”
```

```
{cout<<"input real part and imaginary part of complex number:";
input>>c.real>>c.imag;
return input;
}
int main( )
{Complex c1,c2;
cin>>c1>>c2;
cout<<"c1="<<c1<<endl;
cout<<"c2="<<c2<<endl;
return 0;
}
```

运行情况如下：

```
input real part and imaginary part of complex number:3 6↙
input real part and imaginary part of complex number:4 10↙
c1=(3+6i)
c2=(4+10i)
```

以上运行结果无疑是正确的，但并不完善。在输入复数的虚部为正值时，输出的结果是没有问题的，但是虚部如果是负数，就不理想，请观察输出结果。

```
input real part and imaginary part of complex number:3 6↙
input real part and imaginary part of complex number:4 -10↙
c1=(3+6i)
c2=(4+-10i)
```

根据先调试通过，最后完善的原则，可对程序作必要的修改。将重载运算符“<<”函数修改如下：

```
ostream& operator << (ostream& output,Complex& c)
{output<<"("<<c.real;
if(c.imag>=0) output<<"+";              //虚部为正数时，在虚部前加“+”号
output<<c.imag<<"i)"<<endl;             //虚部为负数时，在虚部前不加“+”号
return output;
}
```

这样，运行时输出的最后一行为**c2=(4-10i)** 。

通过本章前面几节的讨论，可以看到： 在**C++**中，运算符重载是很重要的、很有实用意义的。它使类的设计更加丰富多彩，扩大了类的功能和使用范围，使程序易于理解，易于对对象进行操作，它体现了为用户着想、方便用户使用的思想。有了运算符重载，在声明了类之后，人们就可以像使用标准类型一样来使用自己声明的类。类的声明往往是一劳永逸的，有了好的类，用户在程序中就不必定义许多成员函数去完成某些运算和输入输出的功能，使主函数更加简单易读。好的运算符重载能体现面向对象程序设计思想。

在本章的例子中读者应当注意到，在运算符重载中使用引用(reference)的重要性。利用引用作为函数的形参可以在调用函数的过程中不是用传递值的方式进行虚实结合，而是通过传址方式使形参成为实参的别名，因此不生成临时变量(实参的副本)，减少了时间和空间的开销。此外，如果重载函数的返回值是对象的引用时，返回的不是常量，而是引用所代表的对象，它可以出现在赋值号的左侧而成为左值(left value)，可以被赋值或参与其他操作(如保留cout流的当前值以便能连续使用“<<”输出)。但使用引用时要特别小心，因为修改了引用就等于修改了它所代表的对象。

10.8 不同类型数据间的转换

10.8.1 标准类型数据间的转换

在C++中，某些不同类型数据之间可以自动转换，例如

```
int i = 6;
i = 7.5 + i;
```

编译系统对 7.5是作为double型数处理的，在求解表达式时，先将6转换成double型，然后与7.5相加，得到和为13.5，在向整型变量i赋值时，将13.5转换为整数13，然后赋给i。这种转换是由C++编译系统自动完成的，用户不需干预。这种转换称为隐式类型转换。

C++还提供显式类型转换，程序人员在程序中指定将一种指定的数据转换成另一指定的类型，其形式为

类型名**(**数据**)**

如

int(89.5)

其作用是将**89.5**转换为整型数**89**。

对于用户自己声明的类型，编译系统并不知道怎样进行转换。解决这个问题的关键是让编译系统知道怎样去进行这些转换，需要定义专门的函数来处理。

10.8.2 转换构造函数

转换构造函数(**conversion constructor function**) 的作用是将一个其他类型的数据转换成一个类的对象。

先回顾一下以前学习过的几种构造函数：

•默认构造函数。以**Complex**类为例，函数原型的形式为

```
Complex( );                    //没有参数
```

•用于初始化的构造函数。函数原型的形式为

```
Complex(double r,double i);    //形参表列中一般有两个以上参数
```

•用于复制对象的复制构造函数。函数原型的形式为

Complex (Complex &c); //形参是本类对象的引用

•现在又要介绍一种新的构造函数——转换构造函数。

转换构造函数只有一个形参，如

Complex(double r) {real=r;imag=0;}

其作用是将**double**型的参数**r**转换成**Complex**类的对象，将**r**作为复数的实部，虚部为**0**。用户可以根据需要定义转换构造函数，在函数体中告诉编译系统怎样去进行转换。

在类体中，可以有转换构造函数，也可以没有转换构造函数，视需要而定。以上几种构造函数可以同时出现在同一个类中，它们是构造函数的重载。编译系统会根据建立对象时给出的实参的个数与类型选择形参与之匹配的构造函数。

使用转换构造函数将一个指定的数据转换为类对象的方法如下：

（1） 先声明一个类。

（2） 在这个类中定义一个只有一个参数的构造函数，参数的类型是需要转换的类型，在函数体中指定转换的方法。

（3） 在该类的作用域内可以用以下形式进行类型转换：

类名(指定类型的数据)

就可以将指定类型的数据转换为此类的对象。

不仅可以将一个标准类型数据转换成类对象，也可以将另一个类的对象转换成转换构造函数所在的类对象。如可以将一个学生类对象转换为教师类对象，可以在**Teacher**类中写出下面的转换构造函数：

```
Teacher(Student& s){num=s.num;strcpy(name,s.name);sex=s.sex;}
```

但应注意： 对象**s**中的**num,name,sex**必须是公用成员，否则不能被类外引用。

10.8.3 类型转换函数

用转换构造函数可以将一个指定类型的数据转换为类的对象。但是不能反过来将一个类的对象转换为一个其他类型的数据(例如将一个**Complex**类对象转换成**double**类型数据)。

C++提供类型转换函数**(type conversion function)**来解决这个问题。类型转换函数的作用是将一个类的对象转换成另一类型的数据。如果已声明了一个**Complex**类，可以在**Complex**类中这样定义类型转换函数：

```
operator double( )
{return real;}
```

类型转换函数的一般形式为

operator 类型名( )

{实现转换的语句}

在函数名前面不能指定函数类型，函数没有参数。其返回值的类型是由函数名中指定的类型名来确定的。类型转换函数只能作为成员函数，因为转换的主体是本类的对象。不能作为友元函数或普通函数。

从函数形式可以看到，它与运算符重载函数相似，都是用关键字**operator**开头，只是被重载的是类型名。**double**类型经过重载后，除了原有的含义外，还获得新的含义(将一个**Complex**类对象转换为**double**类型数据，并指定了转换方法)。这样，编译系统不仅能识别原有的**double**型数据，而且还会把**Complex**类对象作为**double**型数据处理。

那么程序中的**Complex**类对具有双重身份，既是**Complex**类对象，又可作为**double**类型数据。**Complex**类对象只有在需要时才进行转换，要根据表达式的上下文来决定。

转换构造函数和类型转换运算符有一个共同的功能：当需要的时候，编译系统会自动调用这些函数，建立一个无名的临时对象(或临时变量)。

例10.9 使用类型转换函数的简单例子。

```
#include <iostream>
using namespace std;
class Complex
{public:
Complex( ){real=0;imag=0;}
Complex(double r,double i){real=r;imag=i;}
operator double( ) {return real;}              //类型转换函数
private:
double real;
double imag;
};

int main( )
{Complex c1(3,4),c2(5,-10),c3;
double d;
d=2.5+c1;                    //要求将一个double数据与Complex类数据相加
cout<<d<<endl;
return 0;
}
```

分析：

(1) 如果在**Complex**类中没有定义类型转换函数**operator double**，程序编译将出错。

(2) 如果在**main**函数中加一个语句：

```
c3=c2;
```

由于赋值号两侧都是同一类的数据，是可以合法进行赋值的，没有必要把**c2**转换为**double**型数据。

(3) 如果在**Complex**类中声明了重载运算符“+”函数作为友元函数：

```
Complex operator+ (Complex c1,Complex c2)          //定义运算符“+”重载函数
{return Complex(c1.real+c2.real, c1.imag+c2.imag);}
```

若在**main**函数中有语句

```
c3=c1+c2;
```

由于已对运算符“+”重载，使之能用于两个**Complex**类对象的相加，因此将**c1**和**c2**按**Complex**类对象处理，相加后赋值给同类对象**c3**。

如果改为

```
d=c1+c2;                    //d为double型变量
```

将**c1**与**c2**两个类对象相加，得到一个临时的**Complex**类对象，由于它不能赋值给**double**型变量，而又有对**double**的重载函数，于是调用此函数，把临时类对象转换为**double**数据，然后赋给**d**。

从前面的介绍可知： 对类型的重载和本章开头所介绍的对运算符的重载的概念和方法都是相似的。重载函数都使用关键字**operator**。

因此，通常把类型转换函数也称为类型转换运算符函数，由于它也是重载函数，因此也称为类型转换运算符重载函数(或称强制类型转换运算符重载函数)。

假如程序中需要对一个**Complex**类对象和一个**double**型变量进行**+,-,*,/**等算术运算，以及关系运算和逻辑运算，如果不用类型转换函数，就要对多种运算符进行重载，以便能进行各种运算。这样，是十分麻烦的，工作量较大，程序显得冗长。如果用类型转换函数对**double**进行重载(使**Complex**类对象转换为**double**型数据)，就不必对各种运算符进行重载，因为**Complex**类对象可以被自动地转换为**double**型数据，而标准类型的数据的运算，是可以使用系统提供的各种运算符的。

例**10.10** 包含转换构造函数、运算符重载函数和类型转换函数的程序。

先阅读以下程序，在这个程序中只包含转换构造函数和运算符重载函数。

```
#include <iostream>
using namespace std;
class Complex
{public:
Complex( ){real=0;imag=0;}                          //默认构造函数
Complex(double r){real=r;imag=0;}                   //转换构造函数
Complex(double r,double i){real=r;imag=i;}          //实现初始化的构造函数
friend Complex operator + (Complex c1,Complex c2);  //重载运算符"+"的友元函数
void display( );
private:
double real;
double imag;
};
```

```
Complex operator + (Complex c1,Complex c2)          //定义运算符"+"重载函数
{return Complex(c1.real+c2.real, c1.imag+c2.imag);}

void Complex::display( )
{cout<<"("<<real<<","<<imag<<"i)"<<endl;}

int main( )
{Complex c1(3,4),c2(5,-10),c3;
c3=c1+2.5;                                  //复数与double数据相加
c3.display( );
return 0;
}
```

对程序的分析：

(1) 如果没有定义转换构造函数，则此程序编译出错。

(2) 现在，在类**Complex**中定义了转换构造函数，并具体规定了怎样构成一个复数。由于已重载了算符“+”，在处理表达式**c1+2.5**时，编译系统把它解释为

operator+(c1,2.5)

由于**2.5**不是**Complex**类对象，系统先调用转换构造函数**Complex(2.5)**，建立一个临时的**Complex**类对象，其值为**(2.5+0i)**。上面的函数调用相当于

operator+(c1,Complex(2.5))

将**c1**与**(2.5+0i)** 相加，赋给**c3**。运行结果为

(5.5+4i)

(3) 如果把**“c3=c1+2.5;”**改为**c3=2.5+c1;** 程序可以通过编译和正常运行。过程与前相同。

从中得到一个重要结论： 在已定义了相应的转换构造函数情况下，将运算符**“+”**函数重载为友元函数，在进行两个复数相加时，可以用交换律。

如果运算符函数重载为成员函数，它的第一个参数必须是本类的对象。当第一个操作数不是类对象时，不能将运算符函数重载为成员函数。如果将运算符“+”函数重载为类的成员函数，交换律不适用。

由于这个原因，一般情况下将双目运算符函数重载为友元函数。单目运算符则多重载为成员函数。

(4) 如果一定要将运算符函数重载为成员函数，而第一个操作数又不是类对象时，只有一个办法能够解决，再重载一个运算符“+”函数，其第一个参数为**double**型。当然此函数只能是友元函数，函数原型为

```
friend operator+(double,Complex &);
```

显然这样做不太方便，还是将双目运算符函数重载为友元函数方便些。

(5) 在上面程序的基础上增加类型转换函数：

```
operator double( ){return real;}
```

此时**Complex**类的公用部分为

```
public:
Complex( ){real=0;imag=0;}
Complex(double r){real=r;imag=0;}                    //转换构造函数
Complex(double r,double i){real=r;imag=i;}
operator double( ){return real;}                     //类型转换函数
friend Complex operator+ (Complex c1,Complex c2);    //重载运算符“+”
void display( );
```

其余部分不变。程序在编译时出错，原因是出现二义性。