第3篇

基于对象的程序设计

第8章 类和对象

第9章 关于类和对象的进一步讨论

第10章 运算符重载

第10章 运算符重载

10.1 什么是运算符重载
Operator Overloading

- Operator overloading allows the programmer to define versions of the predefined operators for operands of class type.

- Operator overloading is just "syntactic sugar," which means it is simply another way for you to make a function call.

- The difference is that the arguments for this function don't appear inside parentheses, but instead they surround or are next to characters you've always thought of as immutable operators.

Syntax

- Defining an overloaded operator is like defining a function, but the name of that function is operator@, in which @ represents the operator that's being overloaded.

- Return type Name (parameters list);
- Return type operator@ (parameters list);

Syntax

- The number of arguments depends on two factors:
- 1. Whether it's a unary operator (one argument) or a binary operator (two arguments).

- 2. Whether the operator is defined as a global function (one argument for unary, two for binary) or a member function (zero arguments for unary, one for binary – the object becomes the left-hand argument).

例10.1 通过函数来实现复数相加

```
#include <iostream>
using namespace std;
class Complex
{
public:
    Complex( ) {real=0;imag=0;}
    Complex(double r,double i) {real=r;imag=i;}
    Complex complex_add(Complex &c2);
    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( );
    cout<<"c2="; c2.display( );
    cout<<"c1+c2="; c3.display( );
    return 0;
}
```

10.2运算符重载的方法

Method of Operator Overloading

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 }

将 “+”用于Complex类(复数)的加法运算：

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

例10.2 运算符重载函数作为类成员函数

```
#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);
    void display( );
private:
    double real;
    double imag;
};
```

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

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

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

10.4 运算符重载函数作为类成员函数和友元函数
Overloading Operator as Class Member & Friend Function

“+”是双目运算符，为什么重载函数只有一个参数？

对于c1+c2,编译系统解释为c1.operator+(c2)

c.real=real+c2.real;

c.imag=imag+c2.imag;

this指向c1。real即this->real，就是c1.real。

c.real=c1.real+c2.real;

c.imag=c1.imag+c2.imag;

•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。Why？

例10.3 运算符重载函数作为类的友元函数

```
#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);
    void display( );
private:
    double real;
    double imag;
};
```

```
Complex operator + (Complex &c1,Complex &c2)
{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;          // operator+(c1,c2)
    cout<<"c1="; c1.display( );
    cout<<"c2="; c2.display( );
    cout<<"c1+c2 ="; c3.display( );
    return 0;
}
```

重载为成员函数or友元函数？

如将一个复数和一个整数相加：

```
Complex Complex::operator+(int &i)
{return Complex(real+i,imag);}
c3=c2+i;            //c2.operator+(i)
c3=i+c2;
// i.operator+(c2)
```

，error: no operator defined which takes a left-hand operand of type 'int' (or there is no acceptable conversion)

将运算符重载函数作为成员函数，运算符左侧的操作数必须是一个本类对象。否则运算符重载函数只能作为非成员函数或友元函数。

运算符重载与交换律

```
Complex operator+(int &i, Complex &c)
{return Complex(i+c.real,c.imag);}
```

在运算表达式中操作数与函数参数类型必须匹配。

```
c3=i+c2;                //正确，类型匹配
c3=c2+i;                //错误，类型不匹配
```

如果希望适用交换律，则应再重载一次运算符“+”：

```
Complex operator+(Complex &c, int &i)
{return Complex(i+c.real,c.imag);}
```

故一般将双目运算符重载为友元函数。

C++规定：赋值运算符、下标运算符、函数调用运算符等必须定义为类的成员函数。

流插入“<<”和流提取运算符“>>”、类型转换运算符则不能定义为类的成员函数。

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。但带后缀.h的头文件可以支持此项功能：

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

10.5 重载双目运算符
Binocular Operator Overloading

```
//例10.4 重载String 类双目运算符
// (1) 建立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;
};
```

```
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;
}
```

(2) 重载一个运算符“>”

```
#include <iostream>
#include <string>
using namespace std;
class String
{
public:
    String( ){p=NULL;}
    String(char *str);
    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;
}
```

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

(3)重载3个运算符

```
//在String类体声明中增加2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;
}
```

```
int main( )
{
   String
string1("Hello"),string2("Book"),string3("Computer");
   cout<<(string1>string2)<<endl;
   cout<<(string1<string3)<<endl;
   cout<<(string1==string2)<<endl;
   return 0;
}
```

(4) 进一步修饰完善

```
#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;
}
```

10.6 重载单目运算符

Monocular Operator Overloading

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。

思路：重载自增运算符“++”。

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。

```
#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++( );                         //运算符重载函数
    void display( ){cout<<minute<<":"<<sec<<endl;}
private:
    int minute;
    int sec;
};
```

```
Time Time::operator++( )          //定义运算符重载函数
{
   if(++sec>=60)
      { sec-=60; ++minute;}          //满60秒进1分钟
   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;
}
```

例10.6 后置自增运算符的重载

```
#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 “++”重载函数
    Time operator++(int);  //声明后置 “++”重载函数
    void display( ){cout<<minute<<":"<<sec<<endl;}
private:
    int minute;
    int sec;
};
```

```
Time Time::operator++( )          //定义前置 “++”重载函数
{
   if(++sec>=60) {sec-=60;++minute;}
   return *this;                  //返回自加后的对象
}
Time Time::operator++(int)        //定义后置 “++”重载函数
{
   Time temp(*this);
   sec++;
   if(sec>=60) {sec-=60;++minute;}
   return temp;                   //返回自加前的对象
}
```

```
int main( )
{
    Time time1(34,59),time2;
    cout<<" time1 : ";        time1.display( );
    ++time1;
    cout<<"++time1: ";        time1.display( );
    //将自加前的对象值赋给time2
    time2=time1++;
    cout<<"time1++: ";        time1.display( );
    cout<<" time2 :";         time2.display( );
    return 0;
}
```

10.7 重载流插入运算符和流提取运算符

Stream Insertion & Extraction Operator Overloading

C++的流插入运算符“<<”和流提取运算符“>>”是C++在类库中提供的。

cin和cout分别是输入流类istream和输出流类ostream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，用来输出和输入C++标准类型的数据。

如果想用“<<”和“>>”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 &,classname &);

ostream & operator << (ostream &, classname &);

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“<<”

```
//例10.7 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);
    friend ostream& operator << (ostream&,Complex&);
private:
    double real, imag;
};
```

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

说明

运算符重载函数中的形参output是ostream类对象的引用。

cout<<c3;

“<<”的左面cout是ostream类对象。“<<”的右面c3是Complex类对象。编译系统解释为

operator<<(cout,c3)

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

调用函数的过程相当于执行：

```
    cout<<"("<<c3.real<<"+"<<c3.imag<<"i)"<<endl;
    return cout; //函数中的“<<”是C++预定义的流插入符
```

return output的作用

- 作用是能连续向输出流插入信息。
- output是实参cout的引用即别名，它们二者共享同一段存储单元，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。
- 例如：
- cout<<c3<<c2;
- 先执行cout<<c3，返回值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。
- 运算符“<<”重载函数的第一个参数和函数类型都是ostream类型的引用，就是为了返回cout的当前值以便连续输出。

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 重载流提取运算符“>>”。
#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)
{
    output<<"("<<c.real;
    if(c.imag>=0) output<<“+”; //虚部为正数时在前面加 “+”号
    output<<c.imag<<“i)”<<endl;
    return output;
}
istream& operator >> (istream& input,Complex& c)
{
    cout<<"input real & 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;
}
```

10.8 不同类型数据间的转换
Type Conversion

- In C and C++, if the compiler sees an expression or function call using a type that isn't quite the one it needs, it can often perform an automatic type conversion from the type it has to the type it wants.

- In C++, you can achieve this same effect for user-defined types by defining automatic type conversion functions. These functions come in two flavors: a particular type of constructor and an overloaded operator.

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。

默认构造函数

```
Complex( );
```

带参数的构造函数

```
Complex(double r,double i);
```

复制构造函数

```
Complex (Complex &c);
```

转换构造函数

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

10.8.3 类型转换函数

类型转换函数(type conversion function) 可以将一个类的对象转换成一个指定类型的数据。

类型转换函数只能作为成员函数，不能指定类型，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。其一般形式为

```
operator 类型名( )
{实现转换的语句}
```

如在Complex类中这样定义类型转换函数：

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

例10.10 转换构造函数与类型转换函数

```
#include <iostream>
using namespace std;
class Complex
{
public:
    Complex( ){real=0;imag=0;}
    Complex(double r,double i){real=r;imag=i;}
    Complex(double r) {real=r;imag=0;}  //转换构造函数
    //operator double( ) {return real;}      //类型转换函数
    friend Complex operator+ (Complex c1,Complex c2); //参数不能使用引用，要么使用常引用
    void display( );
private:
    double real;
    double imag;
};
```

```
Complex operator+ (Complex c1,Complex c2)
{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;
    double d1=0,d2=2.5;
    c3=c1+c2;
    //d1=c1+c2;
    c3=c1+d2;
    c3=d2+c1;
    //d1=d2+c1;
    return 0;
}
```

分析：

c3= c1+c2; //OK

d1=c1+c2;

将c1与c2两个类对象相加，得到一个临时的Complex类对象，调用类型转换函数把临时类对象转换为double数据，然后赋给d1。

分析：

c3=c1+d2;

//编译系统把它解释为operator+(c1,2.5)

若定义了转换构造函数，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)。

c3=c1+d2相当于

operator+(c1,Complex(2.5))

分析：

c3=d2+c1; //同上，程序可以通过编译和正常运行。

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。

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。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。

例10.10 转换构造函数与类型转换函数V2

```
#include <iostream>
using namespace std;
class Complex
{
public:
   Complex( ){real=0;imag=0;} //默认构造函数
   Complex(double r,double i){real=r;imag=i;} //带参数的构造函数
   Complex(double r) {cout<<"转换构造函数  called!"<<endl;real=r;imag=0;} //转换构造函数
   operator double( ) {cout<<"类型转换函数   called!"<<endl;return real;}   //类型转换函数
   friend Complex operator+ (Complex c1,Complex c2);
   friend Complex operator+ (Complex c1,double c2);
   friend Complex operator+ (double  c1,Complex c2);
   void display( );
private:
   double real;
   double imag;
};
```

例10.10 转换构造函数与类型转换函数V2

```
Complex operator+ (Complex c1,Complex c2)
{
    cout<<"Complex operator+ (Complex c1,Complex c2)   called!"<<endl;
    return Complex(c1.real+c2.real, c1.imag+c2.imag);
}
Complex operator+ (Complex c1,double c2)
{
    cout<<"Complex operator+ (Complex c1,double c2)   called!"<<endl;
    return Complex(c1.real+c2, c1.imag);
}
Complex operator+ (double c1,Complex c2)
{
    cout<<"Complex operator+ (double c1,Complex c2)   called!"<<endl;
    return Complex(c1+c2.real, c2.imag);
}
```

例10.10 转换构造函数与类型转换函数V2

```
void Complex::display( )
{cout<<"("<<real<<","<<imag<<"i)"<<endl;}
int main( )
{
    Complex c1(3,4),c2(5,-10),c3;
    double d1,d2=2.5;
    c3=c1+c2;
    d1=c1+c2;

    d1=d2+c1;
    c2=c1+d2;
    return 0;
}
```

作业

- **P343：3、6**