第一节 运算符重载的基本概念

C++预定义的运算符，只能用于基本数据类型的运算。基本数据类型包括：整型、实型、字符型、逻辑型等。

在数学上，两个复数可以直接进行+、-运算，但是在 C++中，直接将+、-用在复数对象是不允许的。

有时候也会希望让对象也能通过运算符进行运算，这样代码更简洁、更容易理解，这个时候就需要运算符的重载了。

运算符重载的目的是：扩展 C++中提供的运算符的适用范围，使之能作用于对象。

它的实质是**函数重载**。可以重载为普通函数，也可以成员函数。

把含运算符的表达式转换成对运算符函数的调用，把运算符的操作数转换成运算符函数的参数。

运算符被多次重载时，根据实参的类型决定调用哪个运算符函数。

运算符重载的形式：

```
返回值类型 operator 运算符(形参表)
{
    //函数体
}
```

如下例：

```
class Complex{
    public:
        double real,imag;
        Complex(double r = 0.0, double i = 0.0):real(r),imag(i){ }
        Complex operator-(const Complex & c);
};
Complex operator+(const Complex &a, const Complex &b){
    return Complex(a.real + b.real,a.imag + b.imag);//返回一个临时对象
}
Complex Complex::operator-(const Complex &c)
{
    return Complex(real - c.real, imag - c.imag);//返回一个临时对象
}
int main(){
    Complex a(4,4),b(1,1),c;
    c = a + b;//等价于 c = operator+(a,b);
    cout << c.real <<"," <<c.imag << endl;
    cout<<(a-b).real << "," << (a-b).imag << endl;
    //a-b 等价于 a.operator-(b)
    return 0;
}
```

注意：**重载为成员函数时，参数个数为运算符目数减一（另一个就是我调用这个成员函数的对象）；重载为普通函数时，参数个数为运算符目数。**

第二节 赋值运算符的重载

1.赋值运算符“=”的重载

有时候希望赋值运算符两边的类型可以不匹配，此时就需要重载赋值运算符“=”。赋值运算符“=”**只能重载为成员函数**。

例程如下：

```
class String{
    private:
        char * str;//指向动态分配的数组
    public:
        String():str(new char[1]){str[0] = 0;}
        const char * c_str(){return str;};
        String & operator = (const char*s);
        String::~String(){delete [] str;}
};
String &String::operator = (const char *s)
{
    //重载=以使 obj="hello"能够成立
    delete[] str;
    str = new char[strlen(s)+1];
    strcpy(str,s);
    return * this;
}

int main()
{
    String s;
    s = "Good luck,";//等价于 s.operator=("Good Luck,");
    cout<<s.c_str()<<endl;
    //String s2 = "hello!";//这条语句不注释掉就会出错
    s = "Shenzhou 8!";//等价于 s.operator=("Shenzhou 8!");
    cout <<s.c_str()<<endl;
    return 0;
}
```

对重载函数进行解释。首先把 str 给 delete 掉，然后给 str 重新分配一个空间，大小为 s 字符串的大小+1，然后把 s 的值复制给 str，返回一个本身的引用。要注意，“=”已经被重载，再编写 String s2 = "hello!"后，=已经不是赋值语句，所以必然会出错。

2.相关其它内容

如下代码：

```
class String{
    private:
        char * str;//指向动态分配的数组
    public:
        String():str(new char[1]){str[0] = 0;}
```

```
        const char * c_str(){return str;};
        String & operator = (const char*s){
            delete [] str;
            str = new char[strlen(s)+1];
            strcpy(str,s);
            return * this;
        };
        String::~String(){delete [] str;}
};
```

我们在主函数里面要实现下面功能：

```
String S1,S2;
S1="this";
S2="that";
S1=S2;
```

在没有重载“=”的时候，S1=S2 也可以编译通过，因为它们类型完全相同的。但是，这个“=”会使 S1 每一点都和 S2 一样。那么，这会有什么问题呢？让我们一步一步分解来看。

首先执行 String S1,S2；S1="this";S2="that";那么就会实现这样的效果：

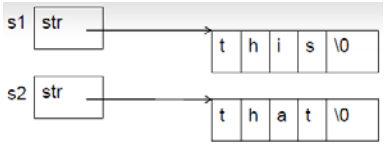

图 2.1 S1="this";S2="that";

再执行 S1=S2，我们发现成了这个样子：

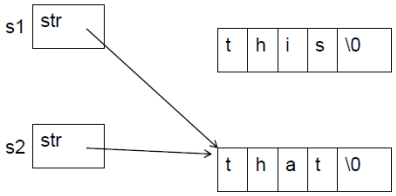

图 2.2 S1=S2 的结果

即 S1 实际上是指向了 S2，两者实际上只指向的一个，原来 S1 的空间失去了指向，与我们想让 S1 中的内容（所指向的空间的内容）和 S2 一样的想法完全不一样。

如果 S1 对象消亡，析构函数将释放 S1.str 指向的空间，则 S2 消亡时还要释放一次，相当于 delete S2 了两次。

如果执行 S1="other"，会导致 S2.str 指向的地方被 delete 掉。所以重载之后，可以避免这样的问题。

考虑下面的语句：

```
String s;
s = "Hello";
s = s;
```

会有什么问题呢？我们重新看重载“=”的成员函数，发现函数第一句和就是把等号左侧对象的空间给 delete 掉了，这在普通的语句下没有什么问题，但是在这里，赋给等号左边对象值的那个对象也是它本身，这样 delete 掉之后，后面的 strcpy 函数就无法复制正确的值给左侧的对象了。为了解决这个问题，需要在这个重载成员函数的函数体开头添加以下语句：

```
if(this == &s)
    return *this;
```

接下来对 operator=的返回值类型进行讨论。**当对运算符进行重载的时候，好的风格是应该尽量保留运算符原来的特性**。

我们考虑 a=b=c，若是 void，那么 b=c 返回值类型就是 void，就没办法再执行 a=操作了，所以可以用 String 类型。

再考虑(a=b)=c。先执行 a=b，在 C++里面，执行=的返回值是**左侧元素的引用**，所以(a=b)的结果是一个 a 的引用，对 a 的引用赋值为 c，那么这个 b 毫无用处。因此不能用 String，而是用 String &这样一个引用格式。this 是当前对象的地址，那么*this 就是当前对象，这解释了为什么要用*this 的原因。

为 String 类编写复制构造函数的时候，会面临和“=”同样的问题（两个对象指向同一个空间），用他同样的方法处理：

```
String(String &s)
{
    str = new char[strlen(s.str)+1];
    strcpy(str,s.str);
}
```

关于浅拷贝和深拷贝待补充

第三节 运算符重载为友元

一般情况下，将运算符重载为类的成员函数是较好的选择。但有时，重载为成员函数并不能满足使用要求，重载为普通函数，又不能访问类的私有成员，所以需要将运算符重载为友元。

如下代码：

```
class Complex
{
    double real,imag;
    public:
    Complex(double r, double i):real(r),imag(i){};
    Complex operator+(double r);
};
Complex Complex::operator+(double r)
{
    return Complex(real+r,imag);
}
```

经过重载以后，c=c+5 有意义，相当于 c=c.operator+(5)

但是，c=5+c 就会出错。所以，为了使得上述表达式成立，需要将+重载为普通函数，这样 c=5+c 就可以通过了。但是普通函数又不能访问私有成员，即不能计算 c=c+5。这样，我们只能用重载为友元函数了。如下：

```
class Complex
{
    double real,imag;
    public:
```

```
    Complex(double r, double i):real(r),imag(i){};
    friend Complex operator + (double r, const Complex & c);
};
Complex Complex::operator+(double r)
{
    return Complex(real+r,imag);
}
```

第四节 运算符重载实例：可变长整型数组

如下代码：

```
int main(){//要编写可变长整型数组类，使之能如下使用
    CArray a;//开始数组是空的
    for(int i = 0; i < 5;++i)
        a.push_back(i);//要用动态分配的内存来存放数组元素需要一个指针成员变量
    CArray a2,a3
    a2 = a;//要重载"="，把 a 中的值复制给 a2
    for(int i = 0; i<a.length;++i)
        cout<<a2[i]<<"";//要重载[]，因为 a2 原来是一个对象
    a2 = a3;//a2 是空的，因为原来的空间被释放了
    for(int i = 0;i<a2.length;++i)//a2.length()返回 0
        cout<<a2[i]<<"";
    cout<<endl;
    a[3]=100;
    CArray a4(a);
    CArray A4(A);//要自己写复制构造函数
    for(int i = 0; i<a4.length;++i)
        cout<<a4[i]<<"";
    return 0;
}
class CArray{
    int size;//数组元素的个数
    int *ptr;//指向动态分配的数组
    public:
    CArray(int s = 0);//s 代表数组元素的个数
    CArray(CArray &a);
    ~CArray();
    void push_back(int v);//用于在数组尾部添加一个元素 v
    CArray & operator=(const CArray &a);
    //用于数组对象间的赋值
    int length(){return size;}//返回数组元素个数
    int & CArray::operator[](int i)
    //返回值不能为 int，不支持 a[i]=4，双目运算符，但是在类内，只有一个运算符
    {//用以支持根据下标访问数组元素，如 n=a[i]和 a[i]=4 这样的语句
```

```
            return ptr[i];
        }
};

CArray::CArray(int s):size(s)
{//构造函数
        if(s ==0)
            ptr = NULL;
        else
            ptr = new int[s];
}
CArray::CArray(CArray &a){//复制构造函数，要实现深复制
        if(!a.ptr){
            ptr = NULL;
            size = 0;
            return;
        }
        ptr = new int [a.size];
        memcpy(ptr,a.ptr,sizeof(int)*a.size);
        size = a.size;
}
CArray：：~CArray()
{
        if(ptr) delete [] ptr;
}
CArray & CArray::operator=(const CArray &a)//深拷贝，而不是浅拷贝
{//赋值号的作用是使“=”左边对象里存放的数组，大小和内容都和右边的对象一致。
        if(ptr == a.ptr)
            return *this;//防止前文所述的出错
        if(a.ptr == NULL){//如果 a 里面的数组是空的
            if(ptr) delete[] ptr;
            ptr = NULL;
            size =0;
            return *this;
        }
        if(size <a.size){//如果原有空间不够，则新建一个足够大的空间
        //如果足够大，就不分配新的空间直接执行 if 后面的语句
            if(ptr)
                delete [] ptr;
            ptr = new int[a.size];
        }
        memcpy(ptr,a.ptr,sizeof(int)*a.size);//空间大小为数目*一个 int 的字节数
        size = a.size;
        return *this;
```

```
}
void CArray::push_back(int v)
{//在数组尾部添加一个元素。先判断原来是否有元素，如果有元素，就新建一个临时空间，
//然后把原来的元素复制过来，然后删除原来的空间，然后把 ptr 指针指向了 tmpPtr 这个临
时空间
//这个元素非常浪费资源
    if(ptr){
        int *tmpPtr = new int[size+1];//重新分配空间
        memcpy(tmpPtr,ptr,sizeof(int)*size);//拷贝原数组内容
        delete[] ptr;
        ptr = tmpPtr;
    }
    else
        ptr = new int[1];//数组原来是空的
    ptr[size++] = v;//加入新的数组元素
}
```

第五节 流插入运算符和流提取运算符的重载

问题 1：cout<<5<<”this”为什么能够成立？

问题 2：cout 是什么？<<为什么能用在 cout 上？

1.流插入运算符的重载

cout 是在 iostream 中定义的 **ostream 类**的对象。之所以<<能用在 cout 上是因为，在 iostream 中对<<进行了重载。

考虑到我们要执行对 5 的操作也要执行对 this 的操作，如果我们定义的重载函数返回值为 void 或者 int 类型，都无法保证后面的两次甚至更多输出能够成立。但是如果我们将其定义为 ostream 类型的话，那么对 5 操作后，还是 ostream 类型，那么就可以继续对 this 操作了，因此要把返回值类型定义为 ostream。即下面的格式：

```
ostream & ostream::operator<<(int n)
{
    //代码
    return *this;
}

ostream & ostream::operator<<(const char *s)
{
    //代码
    return *this;
}
```

cout<<5<<”this”本质上的函数调用形式是：

```
cout.operator<<(5).operator<<("this");
```

例 1：假定下面程序输出为 5hello，该补写些什么？

```
class CStudent{
```

```
    public: int nAge;
};
int main(){
    CStudent s;
    s.nAge = 5;
    cout << s <<"hello";
    return 0;
}
```

需要重载左移运算符，如下：

由于<<已经在ostream中成员函数重载，因此在这里我们只能定义为全局函数进行重载，所以需要两个参数。如下面代码所示，o 其实就是对象 cout。

```
ostream & operator<<(ostream & o, const CStudent & s){
    o<<s.nAge;
    return o;
}
```

例题 2：假定 c 是 Complex 复数类对象，现在希望写"cout << c;"，就能以"a+bi"的形式输出 c 的值；写“cin>>c”就能从键盘接受“a+bi”形式的输入，并且使得 c.real = a，c.imag = b。

```
int main(){
    Complex c;
    int n;
    cin >> c >> n;
    cout << c << "," <<n;
    return 0;
}
```

程序运行结果可以如下：

输入：13.2+133i 87

输出：13.2+133i,87

代码如下：

```
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;
class Complex{
    double real,imag;
public:
    Complex(double r=0,double i =0):real(r),imag(i){};
    friend ostream & operator<<(ostream & os, const Complex &c);
    friend istream & operator >>(istream & is, Compex & c);
};

ostream & operator <<(ostream & os, const Complex & c)
{
    os<<c.real<<"+"<<c.imag<<"i";
    return os;
```

```
}
istream & operator >>(istream & is, Complex & c)
{
	string s;
	is >> s;//将"a+bi"作为字符串读入，中间不能有空格
	int pos = s.find("+",9);
	string sTmp = s.substr(0,pos);//分离出代表实部的字符串
	c.real = atof(sTmp.c_str());//atof 库函数能讲 const char*指针指向的内容转换成 float
	sTmp = s.substr(pos+1,s.length()-pos-2);//分离出代表虚部的字符串
	c.imag = atof(sTmp.c_str());
	return is;
}
int main(){
	Complex c;
	int n;
	cin >> c >> n;
	cout << c << "," <<n;
	return 0;
}
```

选择题：重载“<<”用于将自定义的对象通过 cout 输出时，以下说法正确的是：

C 可以将“<<”重载为全局函数，第一个参数以及返回值，类型都是 ostream &。

第六节 类型转换运算符的重载

代码如下：

```
#include <iostream>

using namespace std;
class Complex{
	double real,imag;
public:
	Complex(double r=0,double i =0):real(r),imag(i){};
	operator double(){return real;}//类型转换运算符重载时不写返回值类型，因为返回值类型就是它本身
};
int main()
{
	Complex c(1.2,3.4);
	cout << (double)c <endl;//输出 1.2
	double n = 2 + c;//c 被自动用类型转换运算符，等价与 double n = 2+c.operator double()
	cout << n;//输出 3.2
}
```

第七节 自增自减运算符的重载

自增运算符++、自减运算符--有前置/后置之分，为了区别所重载的是前置运算符还是后置运算符，C++规定：

（1）前置运算符作为一元运算符重载：

重载为成员函数时：

T & operator++()

T & operator—()

重载为全局函数时：

T1 & operator++(T2)

T1 & operator—(T2)

（2）后置运算符作为二元运算符重载，多写一个没用的参数 int：

重载为成员函数时：

T operator++(int)

T operator--(int)

重载为全局函数时：

T1 operator++(T2, int)

T1 operator--(T2, int)

但是在没有后置运算符重载而有前置运算符重载的情况下，在 vs 中，obj++也调用前置重载，而 dev 则令 obj++编译出错。

例题 1：

```
int main()
{
    CDemo d(5);
    cout<(d++) <<",";//等价于 d.operator++(0);
    cout << d << ",";
    cout << (++d) << ",";//等价于 d.operator++();
    cout << d << endl;
    cout << (d--) << ",";//等价于 d.operator--(0);
    cout << d << ",";
    cout << (--d) << ",";//等价于 d.operator--();
    cout << d << endl;
    return 0;
}
```

输出结果：

5,6,7,7

7,6,5,5

如何编写 CDemo？

```
class CDemo{
    int n;
public:
    CDemo(int i=0):n(i){}
    CDemo & operator++();//前置形式++n 返回值就是 n 的引用，所以这里要用引用
```

```
    CDemo operator++(int);//后置形式，n++返回的是一个临时变量，所以这里不能用引用
    operator int(){return n;}
    friend CDemo & operator--(CDemo &);
    friend CDemo operator--(CDemo &, int);
};

CDemo & CDemo::operator++():
{//前置
    n ++;
    return *this;
}
CDemo CDemo::operator++(int k):
{//后置
    CDemo tmp(*this);//记录修改前的对象
    n++;
    return tmp;//返回修改前的对象
}//s++即为 s.operator++(0);
CDemo & operator--(CDemo & d){//前置
    d.n--;
    return d;
}
CDemo operator--(CDemo &d, int){//后置
    CDemo tmp(d);
    d.n--;
    return tmp;
}//s--即为 operator--(s,0)
```

可以看出，**前置操作因为少一个步骤，所以运算速度快于后置操作**。所以**提倡写++i**。

运算符重载的注意事项：

1.C++不允许定义新的运算符；

2.重载后运算符的含义应该符合日常习惯；

3.运算符重载不改变运算符的优先级；

4.以下运算符不能被重载：“.”、“.*”、“::”、“?:”、sizeof；

5.重载运算符()、[]、->或者赋值运算符=时，运算符重载函数必须生命为成员函数。