CUC 春季学期C++整理7.0

一.对运算符重载的方法

运算符重载的方法是**定义一个重载运算符的函数**，使指定的运算符不仅能够实现原有的功能，而且能实现在函数中指定的新的功能。实际上，**运算符重载实质上是对函数的重载。**

重载函数的一般格式如下：

`函数类型 + operator + 运算符名称（形参表）`

举例说明，如果我们要对"+"进行重载，函数的原型可以是：

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

在上面的一般格式中，`operator`是关键字，是专门用来定义重载运算符的，**运算符名称就是C++已有的运算符**~~这里禁止乱创作~~注意一点：函数名是由`operator`和运算符组成的。上面的`operator +`就是函数名，意思是**"对运算符+的重载"**。其他和一般函数没有什么差别。

举例：复数相加

```
#include<bits/stdc++.h>
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; //实现两个复数的实部相加
    c.imag = imag + c2.imag; //实现两个复数的虚部相加
    return c;
}
void Complex :: display()
{
    cout << "<" << real << "," << imag << ")" << endl;
}
int main()
```

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

运行结果：

```
c1 = (3,4i);
c2 = (5,-10i);
c3 = c1 + c2 = (8,-6i);
```

代码注解：

- C++系统在编译中讲`c1 + c2`解释为：`c1.operator+(c2)`
- 对上面的重载函数还可以这样写：

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

  `return`中的语句是建立一个临时对象，他没有对象名，是一个无名对象。在建立对象的过程中调用构造函数。`return`语句将此临时对象作为函数返回值
- 如果将表达式写为：`c3 = 3 + c2`是错误的，因为形参类型不匹配，应写成对象形式，如`c3 = Complex(3,0) + c2`

二.运算符重载函数作为类成员函数和友元函数

对运算符的重载函数有两种处理方式：**把运算符重载的函数作为类的成员函数；或是在类中声明成友元函数。**

1.作为类的成员

我们还是以上面的例子说明：

```
#include<bits/stdc++.h>
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;
}
void Complex :: display()
{
    cout << "<" << real << "," << imag << ")" << endl;
}
int main()
{
    Complex c1(3,4),c2(5,-10),c3;
    c3 = c1 + c2;
    cout << "c1 = ";
    c1.display();
    cout << "c2 = ";
    c2.display();
    cout << "c3 = c1 + c2 = ";
    c3.display();
    return 0;
}
```

注意到：**R20中的重载函数中只有一个参数**！但是我们知道`+`是一个双目运算符，这里运算符重载应该有两个参数，注意到**重载函数是`Complex`中的成员函数，因此有一个参数是隐含的，是用`this`指针隐式地访问了类对象中的成员**。所以说，重载函数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->imag + imag`

2.作为类的友元函数

```
#include<bits/stdc++.h>
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);//声明友元函数
        void display();
    private:
        double real;
        couble imag;
```

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

程序注解：

- 相较于之前，运算符重载函数有了两个参数。此时c++的解释为：`operator+(c1,c2)`
- 提出问题：友元函数和成员函数之间的区别和适用情形有何不同？

  解答：如果将函数定义为成员函数，它**可以通过`this`指针去访问本类的数据成员，因此可以少写一个函数的参数。但是必须要求运算表达式`(c1 + c2)`中的第一个参数（即运算符左边的操作数）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。而友元函数可以通过改变形参的顺序来实现`c1 + i(int)`和`i(int) + c1`的等同。

三.重载赋值运算符

为什么要重载赋值运算符?

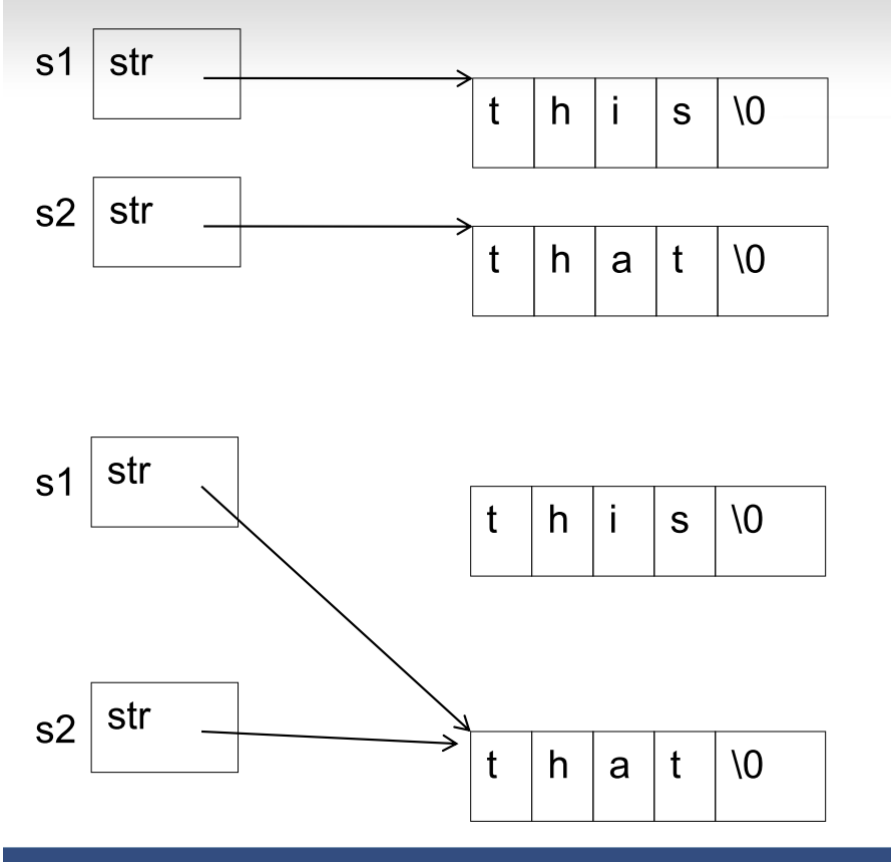
在前面的内容中说明了初始化和赋值的区别：在定义的同时进行赋值叫做初始化，定义完成以后再赋值（不管在定义的时候有没有赋值）就叫做赋值。**初始化只能有一次，赋值可以有多次。**

当以拷贝的方式初始化一个对象时，会调用拷贝构造函数；当给一个对象赋值时，会调用重载过的赋值运算符。即使没有显式的重载赋值运算符，编译器也会以默认地方式重载它。默认重载的赋值运算符功能很简单，就是***将原有对象的所有成员变量一一赋值给新对象***，这和默认拷贝构造函数的类似。看下面的代码：

```
#include<bits/stdc++.h>
using namespace std;
class String
{
    private:
        char * str;
    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( ) { delete [] str; }
}
int main()
{
    String s1,s2;
    s1 = "this";
    s2 = "that";
    s1 = s2;
    return 0;
}
```

如果不定义自己的赋值运算符，那么`S1=S2`实际上导致`S1.str`和`S2.str`指向同一地方。



其次，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，不妥。
另外，如果执行`S1 = "other"`；会导致`S2.str`指向的地方被delete

所以，有些时候我们需要对赋值运算符进行重载。

但是如果发生类似：`s = s`的现象，那么系统就无法处理了，所以为了保险起见，我们将代码修改：

```
#include<bits/stdc++.h>
using namespace std;
class String
{
    private:
        char * str;
    public:
        String ():str(new char[1]) { str[0] = 0;}
        const char * c_str() { return str; };
```

```
        String & operator = (const char * s){
            if(this == &s) return *this; //保证万无一失
            delete [] str;
            str = new char[strlen(s)+1];
            strcpy( str, s);
            return * this;
        };
        ~String( ) { delete [] str; }
}
int main()
{
    String s1,s2;
    s1 = "this";
    s2 = "that";
    s1 = s2;
    return 0;
}
```

对于返回值的几点说明：

- 其实对于重载赋值运算符，返回值是引用或者不是都行，代码都可以运行，**之所以用引用是为了提高代码效率**
- 对于返回值是`Complex`时（非引用）。返回时，会自动生成一个临时对象来保存返回的这个内容，然后通过拷贝构造函数返回给调用这个函数的对象（*this）。但是当返回值是Complex&时，将直接返回(意思就是在函数中实际操作的就是调用这个函数的对象，不存在临时对象)，就不存在这个临时对象了。节省了空间时间。为此效率较高。

结论：返回值是类型的引用，只是为了我们节省效率的一种方式，还有，临时对象具有常性（const）

四.重载单目运算符

单目运算符只有一个操作数，但是其重载方法类似于双目运算符，在这里以重载单目运算符++为例，介绍单目运算符的重载：

注意“++”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，为了区分他们，C++约定，**在自增（自减）运算符重载函数中，增加一个int类型的形参，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;
        }
        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
}
//++s相当于s.operator++();

Time Time::operator++(int)
{
    Time temp(*this);//建立一个临时无名对象
    sec++;
    if (sec >= 60){
        sec-=60;
        ++minute;
    }
    return temp;        // 返回自增前的对象
}
//s++相当于s.operator++(0);

int main()
{
    Time time1(34,59),time2;
    ++time1;
    time1.display();
    time2 = time1 ++;
    time2.display();
    return 0;
}
```

代码解释：

- **首先对于两种运算符的说明：`++a`的返回值是一个a的引用，所以我们在定义函数时，使用了类的引用来定义，相对的`a++`，其返回的不是引用了，而是a这个对象在自增之前的值，所以只需要常规定义就可。**
- **再有，注意区别自增运算符和自减运算符，前者是先自加，返回自加后的结果，后者是先返回当前结果，然后再执行自加操作。**