操作符重载

Andrew Huang<bluedrum@163.com>

课程内容

- 操作符重载的概念
- 操作符重载函数设计
- 几种特殊的操作符
  - >>,<<
  - =
  - [] 操作符

什么是操作符重载

- C/C++ 非常多的运算符(又称为操作符)，只能对基本数据进行运算.
- 在实际编程，用户自定义的的数据结构，如struct ,class 的实例之间也是需要进行运算的.
- 在C 的中，只能通过函数来扩展自定义结构运算

```
//C 计算两个时间之差的
#include <time.h>
double difftime(time_t timer1,time_t timer0);
//实际上相当于 timer1-timer0
```

什么是操作符重载(2)

- C++ 除了通过函数方法来扩展外，有一种更为简洁的扩展方案，即把操作符赋与新的含意，以便能处理新的类型
- 相对于扩展函数，操作符重载的形式更为简洁，易懂. 并能参与连续的链式运算.

```
如 obj3= obj1+obj2;
比 obj3.addobject(obj1,obj2);简洁
```

- 最重要一点是，在不改变原有的体系和代码情况下，操作符重载提供一个直接使用和扩展原有机制
  - 例如：任意一个类重载了<< 操作符，即可以把自己的输出无缝加入到cout 输出

操作符重载的本质

- 重载的操作符本质上也是一个函数，名字比较特殊的函数.
  - 这样设计一个操作符重载的实际上是设计一个特殊函数.
  - 如a+b 可以理解为是一个名字为 operator+(a ，b) 的函数 ，表达式结果就是这个函数的返回值
  - 所有操作符重载函数名，是保留字¡±operator¡± 加上操作符本身构成.
    - << 的重载写成 operator <<
  - 类的操作符重载，隐藏表示当前对象是表达式的第一个操作数

- Class A::operator+(B)　相当于A+B

一个操作符重载实例

```
class temp
{
private :
      int x, y;
public :
      void operator ++(void);
};
//重载++操作符,相当是实现了一个名叫 operator ++的函数
void temp::operator ++(void)
{      x++;      y++;}
//调用
temp objTmp;
objTmp++;
}
```

操作符重载函数的设计

- 操作符有一元和二元操作符.
- 重载操作符虽然类似重载函数，但最重大的区别在
  - c++ 的运算符所能操作的操作数的个数的个数是规定好的。在重载二元运算符时只能指定两个参数，而重载一元运算符时只能指定一个参数。
  - 在重载函数时，可以指定任意的返回类型，甚至可以指定void 类型作为返回类型.但c++ 的运算符是用在表达式中，一个运算符的运算结果要供别的运算符使用，因此任何运算符都指定有非void 类型的返回类型

C++ 允许重载的操作符

+	—	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	— —	<<	>>	= =	!=	&&	||
+=	— =	*=	/=	%=	^=	&=	! =
<<=	>>=	[ ]	( )	— >	— >*	new	delete

C++ 不允许重载的操作符

运算符	运算符的含义	不允许重载的原因
? :	三目运算符	在C++中没有定义一个三目运算符的语法
.	成员操作符	为保证成员操作符对成员访问的安全性，故不允许重载
*	成员指针操作符	同上
: :	作用域运算符	因该操作符左边的操作数是一个类型名，而不是一个表达式
sizeof	求字节数操作符	其操作数是一个类型名，而不是一个表达式

一元操作符重载

一元操作符的重载

- 一元运算符
  - +(正号),-(负号),!(逻辑否),~(位取反),&(取地址)
  - ++(自增),--(自减)
- 一元表达式格式为
  - < 操作符>< 表达式> 如 !a,++a,
  - < 表达式> ++/--, 如 i++,b¡ª
- 一元表达式重载实现方法
  - 1. 用不带参数类方法重载
  - 2. 用带一个参数的全局函数实现，这个函数通常是友元函数，这个方法用的比较少
- 重载的实现(@ 表达一元操作符)
  - 表达式 object @ 重载成 object对应类的方法 operator @ ()
  - 如 object ++ 重载成
- 重载函数的定义格式
  - <类型><类名>::operator <一元运算符>(形参)

自增，自减的重载

- ++,--比较特殊,因为前缀,后缀之分.如果重载不加区别,两者名称一样,无法区别
- 因此C++规定,为了区分前缀和后缀两种形式，后缀形式的自增和自减操作符接受一个额外的int型形参。使用后缀形式的操作符时，编译器提供0作为这个形参的实参，在实现操作符时可以不对这个形参命名。
- 为了与内置操作符一致，后缀形式的操作符应返回旧值，并且，作为值返回，而不是返回引用。通常，后缀形式的实现是调用前缀形式实现的。

```
Demo& Demo::operator++() {
// prefix form
    // do something here
    return *this;
  }
```

```
Demo Demo::operator++(int) { // postfix form
    Demo ret(*this);
    ++*this;
    return ret;
  }
```

自增，自减的重载

```
//原始表达式
class MyClass;
  MyClass MyObject;
++MyObject;
MyObject++;
```

```
//实现代码
class MyClass;
  MyClass MyObject;
MyClass &Myclass::operator ++(){ }
MyClass MyClass::operator ++(int){}
```

```
//显式调用 operator 函数,等价,
class MyClass;
  MyClass MyObject;
MyObject.operator ++();
MyObject.operator ++(0);
```

二元操作符重载

二元操作符重载

- C/C++ 大部分是二元操作符
  - +,-,*,/ ...
  - 表达式 c=a+b ，表示 a 与b 相加，得到中间结果，再赋值给c
- 二元操作符的重载方法
  - 二元操作符的表达式有牵涉到3 个类型，两个参加运算的参数和一个结果．实现方法有两种
  - **第1种,用普通全局函数实现,为了方便,通常是友元函数**，这样两个参与运算的就是，友元函数两个参数，结果对象是函数返回值
  - **第2种,用类的方法实现**，表达式左边为类本身，表达式右边为重载函数唯的一参数，结果对象是函数返回值

用友元函数实现操作符重载

- 实现某一个类之间的加法

```
class CComplex
{
public:
        double m_fReal;
        double m_fImag;
        char m_szStatus[32];
        CComplex(){
                m_fReal=0;
                m_fImag=0;
        }
        CComplex(double fReal,double fImage){
                m_fReal=fReal;
                m_fImag=fImage;
        }
        friend CComplex operator+(const CComplex &cpx1,const
CComplex &cpx2);
};
```

运算符重载为类的成员函数

```
class CComplex
{
public:
        double m_fReal;
        double m_fImag;
        char m_szStatus[32];
        CComplex(){
                m_fReal=0;
                m_fImag=0;
        }
        CComplex(double fReal,double fImage){
                m_fReal=fReal;
                m_fImag=fImage;
        }
CComplex operator+(const CComplex &cpx);
};
```

常用操作符重载

- 流操作符<< 和>>
- 赋值操作符=
- 下标运算符[]

流操作符<< 和>>

- 流操作符<< 和>> 分别对应C++ 标准库中的istream 和ostream ，一般而言，应该为

大多数类重载此操作符；因为输出操作符可用于测试和调试，并方便调用库函数的上层应用者

- 要重载流操作符，通常将此操作符定义为类的友元函数，否则类本身必须使istream 和ostream 的派生类；
- 另外在重载操作符时，还通常返回istream& 和ostream& ，这样返回值可以作为下一个操作符的输入，从而实现操作符的链接，如cout << a1 << a2 << endl ，其中a1 ，a2 分别为实现了流操作符重载的类A 的实例。
- 如果是用友元函数来实现，流操作符的第一个参数时istream 和ostream 的引用，这是因为需要更新流的内部状态；而第二个参数为类的引用，这对<< 而言是为了效率（可以定义为常量引用），对>> 而言是需要接受修改的内容。

流操作符<< 和>>(2)

```
class Complex;
ostream operator<<(ostream& os, Complex& c);
istream operator>>(istream& is, Complex& c);

//输出操作符应做的格式化应尽量少。
ostream operator<<(ostream& os, Complex& c)
{
        os<<"complex value is:("<<c.real<<","<<c.imag<<")"<<endl;
        return os;
}
//输入操作符的重载应加入错误或文件结束的处理
istream operator>>(istream& is, Complex& c)
{
        cout<<"input a complex :\n"<<endl;
        is>>c.real>>c.imag ;
        return is;
}
```

赋值操作符=

- 赋值操作符的最重要问题就是类深拷贝的问题，即对指针成员，需要重新申请空间，在拷贝内容，以免两个实例的成员指向同一块地址；
- 默认赋值操作符是浅拷贝的，这时指针成员会指向同一块地址。
- 浅拷贝是指拷贝对象的普通成员数扰，深拷贝还拷贝对象内部动态分配空间
- 需要在重载时返回*this 的引用，以实现连续赋值，即a1=a2=1
- operator= 必须为成员函数。可以重载。但右边只准出现一个参数。

赋值操作符=

```
//字符串拷贝
CMyString &CMyString::operator=(char *pszData)
{
        delete []m_pszData;
        m_pszData=new char[strlen(pszData)+1];
        strcpy(m_pszData,pszData);
        return *this;
}
```

下标运算符[]

- 标准情况下，[] 运算符用于访问数组的元素。我们可以通过重载下标运算符为类运算符。使得可以象访问数组元素一样的访问对象中的数据成员。
- C++ 只允许把下标运算符重载为非静态的成员函数。
- [] 既可以做左值也可以做右值，即即可以在表达式右边(只读)，也可以在表达式左边(可写). 因此一般需要定义两个版本
  - A[1] = ¡®a¡¯ ； 左值，c = B[2]； 右值
- [] 可以用数字做下标，也能用其它类型，如字符串作下标
  - MyList[5];
  - MyObject[¡®hxy¡¯]=a;

下标运算符[]

```
char CMyString::operator[](int iIndex)
{
        if(iIndex<strlen(m_pszData))
                return m_pszData[iIndex];
        return 0;
}
```

下标运算符[]

```
class Foo
{
public:
 int & operator[] ( const size_t);
 const int & operator[] (const size_t) const;
private:
 int data[10];
}
int & Foo::operator[] (const size_t index)
{
 return data[index];
}
const int& Foo::operator[] ( const size_t index) const
{
 return data[index];
}
```

课堂练习

- 1. 将CDate 操作全部转成重载，至少要实现如下几种
  - >>,<<,=,++,--
- 2. 自行实现一个string 类，里面用new/delete 动态分配空间
  - 要求new []，来分配字符串空间.
  - 可以直接用= 进行字符串赋值

- 支持>>,<< 重载
- 支持用[] 直接访问某一个字符
- 支持用 == 来判断字符串是否相等
- 支持用+ 进行字符串连接
- 有字符串长度的方法
- 可参考 afx.h 和MFC/src/strcore.cpp 的实现代码