

第6章 运算符重载

- 本章主要教学内容

- 运算符重载的原理、基本原则和实现方法
- 一元运算符重载、二元运算符重载
- 运算符重载方式：成员运算符重载、友元运算符重载、普通函数重载运算的区别和编程方法
- 重载特殊运算符：++，--，[]，=，类型转换运算符
- 输入、输出运算符重载
- 仿函数

- 本章教学重点

- 运算符重载的基本原则和实现方法
- 以成员函数和友元方式重载一元运算符和二元运算符重载
- ++、--、[]和类型转换运算符重载
- 输入输出运算符重载

- 教学难点

- ++、--运算符重载（前缀、后缀运算符重载的区别、参数和返回类型设计）
- 输入、输出运算符重载
- 二元运算符重载为成员函数和普通函数（包括友元）的区别（对形参的类型转换）

第6章 运算符重载

- 本章介绍C++运算符重载的相关内容，包括：
 - 以类成员函数、友元和普通函数方式进行运算符重载的方法
 - 输入/输出运算符重载
 - 某些特殊运算符（如++、--、[]、()等）重载。

6.1 运算符重载基础

1、运算符重载的概念

- 运算符重载是C++的一项强大功能。通过重载，可以扩展C++运算符的功能，使它们能够操作用户自定义的数据类型，增加程序代码的直观性和可读性。
- C++的运算符对语言预定义类型是重载的
 - `int i=2+3;`
 - `double j=2+4.8;`
 - `float f=float(3.1)+float(2.0);`
- 对于上面的3个加法表达式，C++系统提供了类似于下面形式的运算符重载函数：
 - `int operator+(int,int);`
 - `double operator+(int,double);`
 - `float operator+(float,float);`

6.1 运算符重载基础

- C++允许程序员通过重载扩展运算符的功能
 - 使重载后的运算符能够对用户自定义的数据类型进行运算。比如，设有复数类**Complex**，其形式如下：

```
class Complex{  
    double real,image;  
public:  
    .....  
};
```

- 假设要实现下面两个复数相加的运算。

```
Complex c1,c2,c3;  
.....  
c1=c2+c3;
```

这条语句是错误的，除非用下面的方法重载“+”运算符，为它增加复数相加的运算能力：

```
Complex operator+(Complex c1,Complex c2)  
{.....}
```

6.1 运算符重载基础

- **why?**
 - 使程序便于编写和阅读
 - 使程序定义类型与语言内建类型更一致
- **how?**
 - 使用特殊的成员函数
 - 使用自由函数，一般为友元

6.1 运算符重载基础

2. 运算符重载限制

(1) 可以重载的运算符

预定义的运算符才能够被重载，这些运算符如下：

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

(2) 不能被重载的运算符

. * :: ?:

(3) 只能被重载为类成员函数的运算符

= [] () ->

6.1 运算符重载基础

(4) 运算符重载过程中的基本原则（限制条件）

- ① 不能改变运算符的优先级。
- ② 不能改变运算符的结合顺序（如+、-、*、/等运算符按照从左到右结合，这个顺序不能改变）。
- ③ 重载运算符不能使用默认参数。
- ④ 不能改变运算符所需要的参数个数。
- ⑤ 不能创造新运算符，只能重载系统已有的运算符。
- ⑥ 不能改变运算符的原有含义。
- ⑦ 若运算符被重载为类的成员函数，则只能是非静态成员函数。

6.1 运算符重载基础

3、运算符重载的语法

- 运算符可以非类成员的形式重载，运算符的计算结果是值，因此运算符函数是要返回值的函数。其重载的语法形式如下：

返回类型 **operator@**(参数表)

- 其中，**operator**是C++的保留关键字，表示运算符函数。@代表要重载的运算符，它可以是前面列举的可重载运算符中的任何一个。例如，

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


6.1 运算符重载基础

3. 与类相关的运算符重载方式

(1) 编译器为类生成的默认运算符重载函数

- 赋值运算 (=)
- 取类对象地址的运算符 (&),
- 成员访问运算符 (如 “.” 和 “->”)。

— 这些运算符不需要重载就可以使用，而其它运算符则只有重载了才能够应用。

(2) 类运算符重载方式

- 重载为类的非静态成员函数
- 重载为类的友元函数
- 重载为普通函数

6.1 运算符重载基础

① 非静态成员函数重载运算符

- 运算符重载为类成员函数时，其第1个参数是由编译器通过**this**指针隐式传递，因此其参数个数要比该运算符实际的参数个数少一个。例如

```
class Complex{  
    double real,image;  
    public:  
        Complex operator+(Complex x){.....}  
    .....  
};
```

- Complex a,b,c;
• a=b+c;

b+c调用了重载运算符+，参数b传递给了隐式参数，而+运算符函数是隐式参数的成员函数，语句“a=b+c;”与“a=b.operator+(c);”等价

6.1 运算符重载基础

② 友元或普通函数重载运算符

- 重载为普通函数或类的友元，参数个数就与运算符实际参数个数相同。形式如下：

```
class Complex{
```

```
.....
```

```
    friend Complex operator+(Complex a,Complex b);
```

```
};
```

```
Complex operator+(Complex a,Complex b){.....}    //友元定义
```

```
Complex operator-(Complex a,Complex b){.....}    //普通函数
```

- 友元和普通函数的区别在于友元可以直接访问类的私有成员，而普通函数只能通过类的公有成员访问其私有成员。

6.1 运算符重载基础

③ 重载为成员与非成员函数的选择

- a) “=, [], (), ->”只能重载为类成员函数。
- b) 一般而言，复合赋值运算符（如+=、-=、*=、/=等）通常应该重载为类成员，但并不是必须这样做（这一点与“=”不同）；
- c) 对于要改变对象状态的运算符，或者与给定类型密切相关的运算符，如++（自增）、--（自减）、解引用运算符，也适宜重载为类成员函数。
- d) 算术运算（+、-、*、/等）、相等与否的比较、关系运算、位运算等运算符具有对称性，通常允许运算符左、右两边的对象进行交换或类型转换，则适宜重载为非成员函数。

6.2 重载二元运算符

二元运算符的调用形式与解析

aa@bb 可解释成 aa.operator@ (bb)

或解释成 operator@(aa,bb)

如果两者都有定义,就按照重载解析

```
class X{  
public:  
    void operator+(int);  
    X(int);  
};  
void operator+(X,X);  
void operator+(X,double);
```

```
void f(X a)  
{  
    a+2; //a.operator+(2)  
    2+a;  
    //::operator+(X(2),a)  
    a+2.0;  
    //::operator+(X,double);  
}
```

6.2.1 类与二元运算符重载

1 作为成员函数重载

- 作为类的非静态成员函数的二元运算符，只能够有一个参数，这个参数是运算符右边的参数，它的第一个参数是通过**this**指针传递的，其重载形式类似于下：

```
class X{  
.....  
    T1 operator@(T2 b){ .....};  
}
```

其中，**T1**是运算符函数的返回类型，**T2**是参数的类型，原则上**T1**、**T2**可以是任何数据类型，但事实上它们常与**X**相同。

6.2.1 类与二元运算符重载

2、作为友元或普通函数重载

- 重载二元运算符为类的友元函数时需要两个参数，其形式如下：

```
class X{
```

```
.....
```

```
    friend T1 operator@(T2 a,T3 b);
```

```
}
```

```
T1 operator@(T2 a,T3 b){ .....} //友元：可直接访问a, b私有成员
```

```
T1 operator#(T2 a,T3 b){ .....} //普通函数：只能访问a, b公有成员
```

- T1、T2、T3代表不同的数据类型，事实上它们常与类X相同。

6.2.1 类与二元运算符重载

3、非静态成员函数、普通函数、友元重载的区别

- ① 以非静态成员函数的方式重载二元运算符时，只能够有一个参数，它实际上是函数的第二个参数（即运算符右边的操作数），其第一个参数（运算符左边的操作数）由C++通过**this**指针隐式传递，而作为普通函数和类的友元函数重载时需要两个参数；
- ② 调用类的重载运算符时，作为类成员函数运算符的左参数必须是一个类对象，而作为友元或普通函数重载的运算符则无此限制。
- ③ 在某些情况下，只有非类成员函数重载才能解决某些特殊情况。

6.2.1 类与二元运算符重载

【例6-1】 设计复数类Complex，利用成员运算符函数重载实现复数的加、减运算，用友元运算符函数重载实现其乘、除等复数运算。

```
#include<iostream>
using namespace std;
class Complex {
private:
    double r, i;
public:
    Complex (double R=0, double I=0):r(R), i(I){ };
    Complex operator+(Complex b);           //L1 复数加法
    Complex operator-(Complex b);           //L2 复数减法
    friend Complex operator*(Complex a,Complex b); //L3 复数乘法
    friend Complex operator/(Complex a,Complex b); //L4 复数除法
    void display();
};
```

```
Complex Complex::operator +(Complex b){return Complex(r+b.r,i+b.i);}
```

```
Complex Complex::operator -(Complex b){return Complex(r-b.r,i-b.i);}
```

```
Complex operator *(Complex a,Complex b){
```

```
    Complex t;
```

```
    t.r=a.r*b.r-a.i*b.i;    t.i=a.r*b.i+a.i*b.r;
```

```
    return t;
```

```
}
```

```
Complex operator /(Complex a,Complex b) {
```

```
    Complex t;
```

```
    double x;
```

```
    x=1/(b.r*b.r+b.i*b.i);    t.r=x*(a.r*b.r+a.i*b.i);    t.i=x*(a.i*b.r-a.r*b.i);
```

```
    return t;
```

```
}
```

```
void Complex::display(){
```

```
    cout<<r;
```

```
    if (i>0) cout<<"+";
```

```
        if (i!=0) cout<<i<<"i"<<endl;
```

```
}
```

6.2.1 类与二元运算符重载

```
void main(void) {  
    Complex c1(1,2),c2(3,4),c3,c4,c5,c6;  
    c3=c1+c2;  
    c4=c1-c2;  
    c5=c1*c2;  
    c6=c1/c2;  
    c1.display();  
    c2.display();  
    c3.display();  
    c4.display();  
    c5.display();  
    c6.display();  
}
```

程序的运行结果如下：

1+2i

3+4i

4+6i

-2-2i

-5+10i

0.44+0.08i

6.2.1 类与二元运算符重载

- 对于程序中的运算符调用

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

```
c4=c1-c2;
```

```
Complex Complex::operator +(Complex b)  
{ return Complex(r+b.r,i+b.i); }
```

```
Complex Complex::operator -(Complex b)  
{ return Complex(r-b.r,i-b.i); }
```

- C++会将它们转换成下面形式的调用语句：

```
c3=c1.operator+(c2);
```

```
c4=c1.operator-(c2);
```

- 从形式上看，这两次函数调用只提供了一个参数c2，但实际上是两个参数，其左参数虽然没有出现在参数表中，但编译器会通过c1对象的this指针传递该参数。

- 总结，在程序中可用下面两种方式调用以类成员函数方式重载的二元运算符：

```
a @ b;
```

```
a.operator@(b)
```

```
//隐式调用二元运算符@
```

```
//显式调用二元运算符@
```

6.2.1 类与二元运算符重载

- 对于c5和c6的计算语句:

c5=c1*c2;

c6=c1/c2;

```
Complex operator *(Complex a,Complex b){  
    Complex t;  
    t.r=a.r*b.r-a.i*b.i;    t.i=b.r*b.i+b.i*b.r;  
    return t;  
}
```

- 因为“*”和“/”通过友元重载实现的，C++编译器会将它们转换成下面的函数调用形式:

c5=operator*(c1,c2);

c6=operator/(c1,c2);

- 总结: 以友元重载运算符函数在程序中可用下面两种形式进行调用:

a@b;

// 隐式调用二元运算符@

operator@(a,b)

// 显式调用二元运算符@

6.2.1 类与二元运算符重载

- 对于程序中的运算符调用：

`c3=c1+c2;`

`c4=c1-c2;`

`c5=c1*c2;`

`c6=c1/c2;`

- C++会将它们转换成下面形式的调用语句：

`c3=c1.operator+(c2);`

`c4=c1.operator -(c2);`

`c5=operator *(c1,c2);`

`c6=operator /(c1,c2);`

- 实际上，在程序中也可以直接写出这样的表达式，显式调用重载的运算符函数

6.2.2 非类成员方式重载二元运算符的特殊用途

1、解决运算符左、右操作数据的次序交换问题

- 对于不要求返回左值且可以交换参数次序的运算符函数（如+、-、*、/等运算符），最好用**非成员形式重载它**（包括友元和普通函数）。

2、解决运算符左操作数据的类型转换问题

- 在调用重载的二元运算符函数时，如果第2个实参与形参的类型不匹配，C++将进行所有的隐式类型转换。
- **对于第一个参数，就要分情况了：**
 - a. 对于非类成员的重载运算符函数，C++编译器在参数不匹配的情况下将**对第一个参数进行隐式类型转换**；
 - b. 对于以类成员重载的运算符函数，**不对第一个参数进行任何隐式类型转换**。

6.2.2 非类成员方式重载二元运算符的特殊用途

【例6-2】 设计复数类**Complex**，使它能够实现下列**L1**，**L2**式的加法运算。

```
void main(){  
    Complex c1,c2(1,2);  
    c1=c2+2;                                //L1  
    c1.display();  
    c1=2+c2;                                //L2  
    c1.display();  
}
```

• 问题分析

- **L1**和**L2**两条语句是数学中的常见运算，“+、-、×、/”等运算的两个操作数可以交换次序。如果用类成员方式重载“+”，则只能完成**L1**语句的运算，**L2**语句则不能实现。
- 解决这样的问题，可以用友元重载“+”运算符。


```
#include <iostream>
using namespace std;
class Complex {
private:
    double r, i;
public:
    Complex(double R = 0, double I = 0) :r(R), i(I) {};
    friend Complex operator+(Complex a, double b) {
        return Complex(a.r + b, a.i);
    }
    friend Complex operator+(double a, Complex b) {
        return Complex(a + b.r, b.i);
    }
    void display();
};
```

解决方案一：

直接用两个友元函数
对不同类型参数进行
运算的加法运算符

6.2.2 非类成员方式重载二元运算符的特殊用途

```
void Complex::display() {  
    cout << r;  
    if (i>0) cout << "+";  
    if (i != 0) cout << i << "i" << endl;  
}  
  
void main(void)  
{  
    Complex c1(1, 2), c2;  
    c2 = c1 + 5;  
    c2.display();           //输出： 6+2i  
    c2 = 5 + c1;  
    c2.display();           //输出： 6+2i  
}
```

```

#include <iostream>
using namespace std;
class Complex {
private: double r, i;
public:
    Complex(double R = 0, double I = 0) :r(R), i(I) {};
    friend Complex operator+(Complex a, Complex b)
    { return Complex(a.r + b.r, a.i+b.i); }
    void display();
};
void Complex::display() {
    cout << r;
    if (i>0) cout << "+";
    if (i != 0) cout << i << "i" << endl;
}
void main(void){
    Complex c1(1, 2), c2;
    c2 = c1 + 5;
    c2.display();           //输出: 6+2i
    c2 = 5 + c1;
    c2.display();           //输出: 6+2i
}

```

解决方案二:

通过1个友元函数重载对两个**Complex**类型相加的加法运算符函数。

这种方案要求: 类应具有能够接受一个参数的构造函数, 此构造函数具有将此参数转换为**Complex**类型的能力!

6.3 重载一元运算符

1、一元运算的概念

一元运算符只需要一个运算参数，如取地址运算符（&）、负数（-）、自增加（++）等。

2、一元运算符常见调用形式

@a 或 a@ //隐式调用形式

a.operator@() //显式调用一元运算符@

– 其中的@代表一元运算符，a代表操作数。

@a代表前缀一元运算，如“++a”；

a@表示后缀运算，如“a++”。

3、@a将被C++解释为下面的形式之一

a.operator@() //成员重载

operator@(a) //友元重载

6.3.1 作为成员函数重载

- 一元运算符作为类成员函数重载时不需要参数，其形式如下：

```
class X{  
.....  
    T operator@(){.....};  
}
```

- T是运算符@的返回类型。从形式上看，作为类成员函数重载的一元运算符没有参数，但实际上它包含了一个隐含参数，即调用对象的this指针。
- 像++、--这样能够实现连续自增、自减的运算符，其重载函数应该返回对象的引用。否则，就不能实现对象的连续运算。

6.3.1 作为成员函数重载

【例6-3】 设计一个时间类Time，能够完成秒钟的自增运算。

//Eg6-3.cpp

```
#include<iostream>
```

```
using namespace std;
```

```
class Time{
```

```
private:
```

```
    int hour,minute,second;
```

```
public:
```

```
    Time(int h,int m,int s);
```

```
    Time& operator++();
```

```
    void display();
```

```
};
```

```
Time::Time(int h,int m,int s) {
```

```
    hour=h;
```

```
    minute=m;
```

```
    second=s;
```

```
    if(hour>=24) hour=0;
```

```
    if(minute>=60) minute=0;
```

```
    if(second>=60) second=0;
```

```
    //若初始小时超过24，重置为0
```

```
    //若初始分钟超过60，重置为0
```

```
    //若初始秒钟超过60，重置为0
```

```
}
```

```

Time &Time::operator ++(){
    ++second;
    if(second>=60) {
        second=0;
        ++minute;
        if(minute>=60){
            minute=0; ++hour;
            if(hour>=24) hour=0;
        }
    }
    return *this;
}

void Time::display(){
    cout<<hour<<":"<<minute<<":"<<second<<endl; }

void main(){
    Time t1(23,59,59);
    t1.display();
    ++ ++t1;                //隐式调用方式
    t1.display();
    t1.operator ++();        //显式调用方式
    t1.display();
}

```

本程序的运行结果如下

```

:
23:59:59
0:0:1
0:0:2

```

6.3.2 作为友元函数重载

- 用友元函数重载一元运算符时需要一个参数。

【例6-4】 用友元重载Time类的自增运算符++。

//Eg6-4.cpp

class Time{

..... **//省略的代码与例6-3相同**

friend Time & operator++(Time &t);

};

6.3.2 作为友元函数重载

```
Time & operator ++(Time &t) {  
    ++t.second;  
    if(t.second>=60){  
        t.second=0;  
        ++t.minute;  
        if(t.minute>=60){  
            t.minute=0;  
            ++t.hour;  
            if(t.hour>=24) t.hour=0;  
        }  
    }  
    return t;  
}  
  
void main(){  
    Time t1(23,59,59);  
    t1.display();  
    ++ ++ t1;  
    t1.display();  
    operator++(t1);  
    t1.display();  
}
```

本程序的运行结果：
23:59:59
0:0:1
0:0:2
此结果与例6-3完全相同

//隐式调用方式

//显式调用方式

6.3.2 作为友元函数重载

- 非类成员重载++、--等的注意事项
 - 在用友元和普通函数重载++、--这类一元运算符函数时，如果用值传递的方式设置函数的参数，就可能会发生错误，不能把运算结果返回给调用对象，也就实现不了自增或自减运算。

重载++运算符的**错误**例子

- 将例6-4中的++运算符函数改为下面的重载形式:

```
class Time{  
    ..... //Time类的其余代码同例6-4
```

```
    friend Time operator++(Time t);
```

```
};
```

```
Time operator++(Time t){
```

```
    ..... //省略的程序代码同例6-4的 operator ++(Time &t)
```

```
    return t;
```

```
}
```

```
.....
```

```
void main(){
```

```
    Time t1(23,59,59);
```

```
    t1.display();
```

```
    ++ ++ t1;
```

```
    t1.display();
```

```
    operator++(t1);
```

```
    t1.display();
```

```
}
```

注意：形参和函数返回值都是值类型

- 本程序的运行结果如下

23:59:59

23:59:59

23:59:59

- 试分析此结果的由来！

6.4 特殊运算符重载

6.4.1 运算符++和--的重载

1、特殊性：区分前缀、后缀

++X;	//前自增
X++;	//后自增
--X;	//前自减
X--;	//后自减

2、将它们重载为类的成员函数时就会都是下面的形式：

```
class X{  
    .....  
    X operator++(){.....};           //前自增  
    X operator++(){.....};           //后自增  
}
```

6.4.1 运算符++和--的重载

3、重载为友元运算符，将都是下面的形式：

```
class X{
```

```
    friend X operator++(X& o); //前自增的友元声明
```

```
    friend X operator++(X& o); //后自增的友元声明
```

```
}
```

4、问题？

无法区分到底是前自增还是后自增运算！

同样的问题发生在自减运算符身上：--

6.4.1 运算符++和--的重载

5、解决方案

- C++编译器通过在运算符函数参数表中是否插入关键字int 来区分这两种方式。
- 自减前缀

operator -- (); //成员函数重载

operator -- (X & x); //

- 自减后缀：加入一个无用的类型参数，表示后缀

operator -- (int);

operator -- (X & x, int);

6.4.1 运算符**++**和**--**的重载

- 自增前缀

operator ++ (); // 成员重载

operator ++ (X & x); // 非成员重载

- 自增后缀，增加一个无用参数

operator ++ (int); // 成员重载

operator ++ (X & x, int); // 非成员重载

【例6-5】 设计一个计数器counter，用数据成员n保存计算器的值，用类成员重载自增运算符实现计数器的自增，用友元重载实现计数器的自减。

```
//Eg6-5.cpp
```

```
#include<iostream>
```

```
using namespace std;
```

```
class Counter {
```

```
private:
```

```
    int n;
```

```
public:
```

```
    Counter(int i = 0) { n = i; }
```

```
    Counter& operator++();
```

```
    Counter operator++(int);
```

```
    friend Counter& operator--(Counter &c);
```

```
    friend Counter operator--(Counter &c, int);
```

```
    void display();
```

```
};
```



```
Counter& Counter::operator++() {  
    ++n;  
    return *this;  
}
```

```
Counter Counter::operator++(int) {  
    Counter t(*this);  
    n++;  
    return t;  
}
```

```
Counter& operator--(Counter &c) {  
    --c.n;  
    return c;  
}
```

```
Counter operator--(Counter &c, int) {  
    Counter temp(c);  
    c.n--;  
    return temp;  
}
```

6.4.1 运算符++和--的重载

```
void Counter::display() {  
    cout << "counter number = " << n << endl;  
}  
void main() {  
    Counter a;  
    ++a;           //调用Counter::operator++()  
    a.display();  
    a++;           //调用Counter::operator++(int)  
    a.display();  
    --a;           //调用operator--(Counter &c)  
    a.display();  
    a-- ;          //调用operator--(Counter &c,int)  
    a.display();  
}
```

6.4.2 下标[]和赋值运算符=

1、重载下标运算符[]

(1) 重载原因

- 在C/C++中，数组不具有检测下标值范围的功能，容易产生数组访问下标越界的错误。通过下标运算符[]重载，可以在访问数组元素时进行下标值检测，禁止越界访问。

(2) []二元运算符的重载形式如下：

```
class X{  
.....  
    X& operator[](int n);  
};
```

(3) 重载[]需要注意的问题

- ① []是一个二元运算符，其第1个参数是通过对象的this指针传递的，第2个参数代表数组的下标
- ② 由于[]既可以出现在赋值符“=”的左边，也可以出现在赋值符“=”的右边，所以重载运算符[]时常返回引用。
- ③ []只能被重载为类的非静态成员函数，不能被重载为友元和普通函数。

6.4.2 下标[]和赋值运算符=

【例6-6】 设计一个工资管理类，它能根据职工的姓名录入和查询职工的工资，每个职工的基本数据有职工姓名和工资。

```
#include <iostream>
#include <string>
using namespace std;
struct Person{                                //职工基本信息的结构
    double salary;
    char *name;
};
class SalaryManage{
    Person *employ;                          //存放职工信息的数组
    int max;                                 //数组下标上界
    int n;                                  //数组中的实际职工人数
public:
    SalaryManage(int Max=0){
        max=Max;
        n=0;
        employ=new Person[max];
    }
```

6.4.2 下标[]和赋值运算符=

```
double &operator[](char *Name) {    //重载[], 返回引用
    Person *p;
    for(p=employ;p<employ+n;p++)
        if(strcmp(p->name,Name)==0)
            return p->salary;
    p=employ + n++;
    p->name=new char[strlen(Name)+1];
    strcpy(p->name,Name);
    p->salary=0;
    return p->salary;
}

void display(){
    for(int i=0;i<n;i++)
        cout<<employ[i].name<<" "<<employ[i].salary<<endl;
}

};
```

6.4.2 下标[]和赋值运算符=

```
void main(){
    SalaryManaege s(3);
    s["杜一为"]=2188.88;
    s["李海山"]=1230.07;
    s["张军民"]=3200.97;
    cout<<"杜一为\t"<<s["杜一为"]<<endl;
    cout<<"李海山\t"<<s["李海山"]<<endl;
    cout<<"张军民\t"<<s["张军民"]<<endl;

    cout<<"-----下为display的输出-----\n\n";
    s.display();
}
```

6.4.2 下标[]和赋值运算符=

2. 重载赋值运算符=

(1) 赋值运算符“=”的重载特殊性

- 赋值运算进行时将调用此运算符
- 只能用成员函数重载
- “=”应用场合较多。在设计类时若没有为它提供赋值运算符成员函数，编译器会自动为它合成一个默认的赋值运算符函数。

(2) 什么时候需要重载“=”

- 如果该类对象没有分配动态存储空间，默认赋值运算符函数能够正确完成对象的赋值拷贝。如果对象构造时分配了动态存储空间，默认赋值运算符函数多数时候都不能正确地进行对象的赋值拷贝，需要为类重载赋值运算符函数。
- 此外，有时还需要通过赋值运算符实现特殊的对象赋值拷贝操作，也需要重载赋值运算符函数。
- 关于重载赋值运算符函数的详细内容，请参考3.8.1节

6.4.3 类型转换运算符

1、关于类型转换运算

- C++是强类型语言，类型转换经常发生
- 两种类型转换
 - 隐式类型转换implicit conversion
 - 显式类型转换explicit conversion
- 隐式类型转换发生的时机
 - 赋值
 - 函数调用
 - 函数返回值

6.4.3 类型转换运算符

— 隐式类型转换示例

```
class X
{ public:
    X (int) {cout<<"X"<<endl;}
};
X f (X) { return 1;}           //将int转换成X类型
void main (void)
{
    int i = 'a';
    X obj = f (i);
    f ('b');
}
```

显示类型转换

```
long i = (long) 1234;
long i = long (1234);
```

6.4.3 类型转换运算符

2、用构造函数实现类的类型转换

若将类Y转换成类X类型，用如下形式的构造函数

```
class X
{
    public: X (Y y) {...};
};
...
Y y;
X x1 = y;
X x2 (y);
x1 = y;
.....
```

【例6-7】 有日期类**Date**，设计其构造函数，能够将整型数据转换成一个**Date**类的对象。

```
#include <iostream>
using namespace std;
class Date{
private:
    int year,month,day;
public:
    Date(int yy=1900,int mm=1,int dd=1){
        year=yy; month=mm; day=dd;
    }
    void Show(){cout<<year<<"-"<<month
        <<"- "<<day<<endl;}
};

void main(){
    Date d(2000,10,11);
    d.Show();
    d=2006;
    d.Show();
}
```

将调用构造函数将
2006转换成Date

6.4.3 类型转换运算符

3. 类型转换运算符

用于将类X转换为类Y类型。语法

```
class X
{
    public: operator Y ( )
    {
        .....
        return  Y类型的数据; };
};
```

- 类型转换函数没有参数，没有返回类型。
- 类型转换函数必须返回将要转换成的type类型数据。
- 一旦定义类型转换运算符，就可以显示或隐式地进行类型转换，如同系统预定义的类型转换一样。

- **【例6-8】** 有一个类**Circle**，设计该类的类型转换函数，当将**Circle**对象转换成**int**型时，返回圆的半径；当将它转换成**double**型时，就返回圆的周长；当将它转换成**float**型时，就返回圆的面积。

//Eg6-8.cpp

#include <iostream>

using namespace std;

class Circle{

private:

double x,y,r;

public:

Circle(double x1,double y1,double r1){ x=x1;y=y1;r=r1; }

operator int(){return int(r);}

operator double(){return 2*3.14*r;}

operator float(){return (float)3.14*r*r;}

};

6.4.3 类型转换运算符

```
void main(){  
    Circle c(2.3,3.4,2.5);  
    int r=c;    //调用operator int(), 将Circle类型转换成int  
    double length=c;    //调用operator double(), 转换成double  
    float area=c;    //调用operator float(), 将Circle类型转换成float  
    double len=(double) c; //将Circle类型对象强制转换成double  
    cout<<r<<endl;  
    cout<<length<<endl;  
    cout<<len<<endl;  
    cout<<area<<endl;  
}
```

6.4.3 类型转换运算符

4. 类型转换的二义性问题

- 无论是定义把其它类型转换成类类型的构造函数，还是定义把类类型转换成其它类型的类型转换运算符函数，都要注意避免转换函数的二义性问题。
- 最常见的情况是定义多个参数都是数值类型的构造函数，或者定义了多个目标类型都是数值类型的类型转换函数。

【例6-9】类B同时设置了int和double类型参数的构造函数，以及将类B转换成int和float的类型转换函数，容易引发二义性问题。

6.4.3 类型转换运算符

```
#include <iostream>
using namespace std;
class B {
    double x;
public:
    B(float a = 0) :x(a) {}
    B(double b=0.0) :x(b) {}
    operator int() { return x; }
    operator float() { return x; }
};
void f(long l) { cout << l << endl; }
void main() {
    B b(4);      //L1, 无法确定调用B::B(float)还是B::B(double)
    f(b);        //L2, 无法确定调用operator int()还是operator float()
}
```


6.5 输入/输出运算符重载

6.5.1 重载输出运算符<<

- 输出运算符<<也称为插入运算符，通过输出运算符<<的重载可以实现用户自定义数据类型的输出。

- <<的重载语法

```
ostream &operator<<(ostream &os,classType object)
{
    .....
    os<< ...           //输出对象的实际成员数据
    return os;         //返回ostream对象
}
```

6.5.2 重载输入运算符>>

- 输入运算符>>也称为提取运算符，用于输入数据。通过输入运算符>>的重载，就能够用它输入用户自定义的数据类型
- >>的重载语法

```
istream &operator>>(istream &is,class_name &object)
{
    .....
    is>> ...    //输入对象object的实际成员数据
    return is;   //返回istream对象
}
```

6.5.3 >>和<<重载的应用

【例6-11】 有一销售人员类**Sales**，其数据成员有姓名**name**，身份证号**id**，年龄**age**。重载输入输出运算符实现对**Sales**类数据成员的输入和输出。

```
#include<iostream>
```

```
#include<string>
```

```
using namespace std;
```

```
class Sales{
```

```
private:
```

```
    string name;
```

```
    string id;
```

```
    int age;
```

```
public:
```

```
    Sales(string Name,string ID,int Age);
```

```
    friend    ostream &operator<<(ostream &os,Sales &s);
```

```
    friend    istream &operator>>(istream &is,Sales &s);
```

```
};
```

6.5.3 >>和<<重载的应用

```
Sales::Sales(string Name,string ID,int Age) {
    name=Name;
    id=ID;
    age=Age;
}

ostream& operator<<(ostream &os,Sales &s) {
    os<<s.name<<"\t";           //输出姓名
    os<<s.id<<"\t";             //输出身份证号
    os<<s.age<<endl;            //输出年龄
    return os;
}

istream &operator>>(istream &is,Sales &s) {
    cout<<"输入雇员的姓名，身份证号，年龄"<<endl;
    is>>s.name>>s.id>>s.age;    //数据成员数据输入
    return is;
}
```

6.5.3 >>和<<重载的应用

```
void main(){  
    Sales s1("杜康","214198012111711",40);  
    cout<<s1;  
    cout<<endl;  
    cin>>s1;  
    cout<<s1;  
}
```

程序运行结果如下：

杜康 214198012111711 40
输入雇员的姓名，身份证号，年龄
Tom 100 23
Tom 100 23

6.6 编程实作：运算符重载编程应用

1. 编程实例一

【例6-12】 设计一个字符串类String，通过运算符重载实现字符串的输入、输出以及+=、==、!=、<、>、>=、[]等运算。

问题分析：

- 标准C++提供了一个String类。String类重载了+、+=、==、>、>=、<、<=等运算符函数，这些重载运算符函数让字符串数据类型的操作变得非常简单，使字符串的赋值、连接与大小比较等操作与C++内置的int和float等类型的数据一样简便。要使用标准C++的string类，需在程序中引用头文件#include <string>。
- 在此可以通过运算符重载实现字符串的+=、==、!=、<、>、>=、[]

```
#include <iostream>
using namespace std;
class String {
private:
    int length;                //字符串长度
    char *sPtr;                //存放字符串的指针
    void setString( const char *s2);
    friend ostream &operator<<(ostream &os, const String &s);
    friend istream &operator>>(istream &is, String &s);
public:
    String( const char * = "" );
    ~String();
    const String &operator=(const String &R);
    const String &operator+=(const String &R);
    bool operator==(const String &R);
    bool operator!=(const String &R);
    bool operator!() ;
    bool operator<(const String &R) const;
    bool operator>(const String &R);
    bool operator>=(const String &R);
    char &operator[](int);
};
```

```
const String &String::operator+=(const String &R) {  
    char *temp = sPtr;  
    length += R.length;  
    sPtr = new char[length+1];  
    strcpy(sPtr,temp );  
    strcat(sPtr,R.sPtr );  
    delete [] temp;  
    return *this;  
}
```

```
bool String::operator==(const String &R){return  
    strcmp(sPtr,R.sPtr)==0;}
```

```
bool String::operator!=(const String & R){return !(*this==R);}
```

```
bool String::operator!(){return length ==0;}
```

```
bool String::operator<(const String &R)const  
{return strcmp(sPtr,R.sPtr)<0;}
```

```
bool String::operator>(const String &R){return R<*this;}
```

```
bool String::operator>=(const String &R){return !(*this<R);}
```

```
char &String::operator[](int subscript){return sPtr[subscript];}
```



```
ostream &operator<<(ostream &os,const  
    String &s){  
    os << s.sPtr;  
    return os;  
}
```

```
istream &operator>>(istream &is,String  
    &s) {  
    char temp[100];  
    s=temp;  
    return is;  
}
```

```
int main(){
```

```
String s1("happy"),s2("new year"),s3;
```

```
cout << "s1 is " << s1 << "\ns2 is " << s2 << "\ns3 is " << s3
```

```
<< "\n比较s2和s1:"
```

```
<< "\ns2 ==s1结果是 " << ( s2 == s1 ? "true" : "false")
```

```
<< "\ns2 != s1结果是 " << ( s2 != s1 ? "true" : "false")
```

```
<< "\ns2 > s1结果是 " << ( s2 > s1 ? "true" : "false")
```

```
<< "\ns2 < s1结果是 " << ( s2 < s1 ? "true" : "false")
```

```
<< "\ns2 >= s1结果是 " << ( s2 >= s1 ? "true" : "false");
```

```
cout << "\n\n测试s3是否为空: ";
```

```
if (!s3){  
    cout << "s3是空串"<<endl;           //L3  
    cout<<"把s1赋给s3的结果是: ";  
    s3 = s1;  
        cout << "s3=" << s3 << "\n";           //L5  
}  
cout << "s1 += s2 的结果是: s1=";           //L6  
s1 += s2;  
cout << s1;           //L7  
  
cout << "\ns1 += to you 的结果是: ";           //L8  
s1 += " to you";  
cout << "s1 = " << s1 <<endl;           //L9  
s1[0] = 'H';  
s1[6] = 'N';  
s1[10] = 'Y';  
cout << "s1 = " << s1 << "\n";           //L10  
return 0;  
}
```

程序运行结果

s1 is happy

s2 is new year

s3 is

比较s2和s1:

s2 == s1结果是false

s2 != s1结果是true

s2 > s1结果是false

s2 < s1结果是true

s2 >= s1结果是false

测试s3是否为空: s3是空串

把s1赋给s3的结果是: s3=happy

s1 += s2 的结果是: s1=happy new year

s1 += to you 的结果是: s1=happy new year to you

s1 = Happy New Year to you

//L1语句输出下面连续的9行

//L2和L3语句输出

//L4和L5语句输出

//L6和L7语句输出

//L8和L9语句输出

//L10语句输出

6.6 编程实作：运算符重载编程应用

• 2. 编程实例二

【例6-13】 改写5.5节的课程结构类，为comFinal、Account、Chemistry类重载输出运算符函数operator<<，使程序能够直接利用cout输出各个类的对象。

（1）重载comFinal类的输出运算符operator<<

- 启动VC++，打开目录C:\course中的com_main.sln工程文件；
- 打开comFinal.h头文件，并在comFinal类的声明中添加operator<<运算符函数的重载声明，如下所示：

6.6 编程实例

```
//comFinal.h  
class comFinal{  
    friend      ostream &operator<<(ostream &os, comFinal &s);  
    .....      //其余代码不作任何修改;  
}
```

- 打开**comFinal.cpp**源文件，并在其中添加**operator<<**的程序代码。代码如下：

```
//comFinal.cpp  
.....  
ostream& operator<<(ostream &out, comFinal &o) {  
    cout << "姓名\t" << "汉语\t" << "数学\t" << "英语\t"  
        << "总分\t" << "平均分" << endl;  
    out<<o.name<<"\t"<<o.chinese<<"\t"<<o.math<<"\t"<<o.english<<"\t"  
        << o.getTotal() << "\t" << o.getAverage() << endl << endl;  
    return out;  
}
```

(2) 重载Account类的输出运算符operator<<

- 在**Account**头文件的类中添加如下函数声明:

```
//Account.h
class Account{
    friend          ostream &operator<<(ostream &os, Account &s);
    .....          //其余代码不作任何修改
}
```

- 添加在**Account.cpp**中的实现代码如下:

```
//Account.cpp
ostream& operator<<(ostream &out, Account &o) {
    cout << "姓名\t" << "汉语\t" << "数学\t" << "英语\t"
           << "会计学\t" << "经济学\t" << "总分\t" << "平均分" << endl;
    out<<o.name<<"\t"<<o.chinese<<"\t"<<o.math<<"\t"<<o.english<<"\t"
       <<o.account<<"\t"<<o.econ<<"\t"<<o.getTotal()+o.account+o.econ
       <<"\t"<<(o.getTotal()+o.account+o.econ)/5<<endl<<endl;
    return out;
}
```

(3) 重载Chemistry类的输出运算符：operator<<

在**Chemistry**类的头文件和源代码中分别添加如下代码：

```
//Chemistry.h  
class Chemistry {  
    friend ostream &operator<<( ostream &os, Chemistry &s);  
    .....  
    其余代码不作任何修改;  
}
```

添加在**Chemistry.cpp**文件中的函数代码如下：

```
//Chemistry.cpp  
ostream& operator<<(ostream &out, Chemistry &o) {  
    cout << "姓名\t" << "汉语\t" << "数学\t" << "英语\t" << "化学\t"  
        << "化学分析\t" << "总分\t" << "平均分" << endl;  
    out << o.name << "\t" << o.chinese << "\t" << o.math << "\t"  
        << o.english << "\t" << o.chemistr << "\t" << o.analy << "\t\t"  
        << o.getTotal()+o.chemistr + o.analy << "\t"  
        << (o.getTotal() + o.chemistr + o.analy )/5<< endl << endl;  
    return out;  
}
```


(4) 验证重载结果

改写应用程序运行结果如下：

```
//com_main.cpp
#include "Chemistry.h"
#include "Account.h"
#include <iostream>

void main(){
    comFinal com("刘科学",89,78);
    Account a1("张三星",87);
    Chemistry c1("光红",89,76,34,56,78);
    cout<<com;           //L1
    cout<<"-----"<<endl;
    cout<<a1;             //L2
    cout<<"-----"<<endl;
    cout<<c1;             //L3
}
```

姓名	汉语	数学	英语	总分	平均分
刘科学	76	89	78	243	81

姓名	汉语	数学	英语	会计学	经济学	总分	平均分
张三星	78	97	98	67	87	427	
	85						

姓名	汉语	数学	英语	化学	化学分析	总分	平均分
光红顺	76	34	89	56	78	333	66