第 6 章　运算符重载

运算符重载是面向对象程序设计的重要特性。运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据时触发不同的响应。C++中经重载后的运算符能直接对用户自定义的数据进行操作运算，这就是 C++语言中的运算符重载所提供的功能。通过本章的学习，读者应该掌握以下内容:

- 运算符重载函数的定义及规则
- 友元运算符重载函数的定义与使用
- 成员运算符重载函数的定义与使用
- 几种常用运算符的重载
- 类型转换运算符的重载

6.1　运算符重载概述

在前一章中曾提到过，C++中运行时的多态性主要通过虚函数来实现，而编译时的多态性由函数重载和运算符重载来实现。本章主要讲解 C++中有关运算符重载方面的内容。在讲解本章之前，有一些基础知识需要我们去理解。而运算符重载的基础就是运算符重载函数，所以本节主要讲的是运算符重载函数。

6.1.1　运算符重载函数的定义

运算符重载是对已有的运算符赋予多重含义，使同一个运算符作用于不同类型的数据时触发不同的响应。比如：

```
int i;
int i1=10,i2=10;
i=i1+i2;
cout<<"i1+i2="<<i<<endl;
double d;
double d1=20,d2=20;
d=d1+d2;
cout<<"d1+d2="<<d<<endl;
```

在这个程序里“+”既完成两个整型数的加法运算，又完成了两个双精度数的加法运算。为什么同一个运算符“+”可以用于完成不同类型的数据的加法运算？这是因为 C++针对预定义的基本数据类型已经对“+”运算符进行了适当的重载。在编译程序编译不同类型数据的加法表达式时，会自动调用相应类型的加法运算符重载函数。

但是 C++中所提供的预定义的基本数据类型毕竟是有限的，在解决一些实际的问题时，往往需要用户自定义数据类型。例如，在解决科学与工程计算问题时，往往要使用复数、矩阵等。

下面定义一个简化的复数类 complex:

```
class complex{
public:
    double real,imag;
    complex(double r=0,double i=0)
    {    real=r; imag=i;}
};
```

若要把类 complex 的两个对象 com1 和 com2 加在一起，下面的语句无法实现此功能：

```
int main()
{
    complex com1(1.1,2.2),com2(3.3,4.4),total;
    total=com1+com2;          //错误
    //…
    return 0;
}
```

会提示没有与这些操作数匹配的“+”运算符的错误。这是因为 complex 类不是预定义类型，系统没用对该类型的数据进行加法运算符函数的重载。C++为运算符重载提供了一种方法，即运算符重载函数。其函数名字为 operator 后紧跟重载运算符。

运算符函数定义的一般格式如下：

```
返回类型 operator 运算符符号(参数表)
{
    函数体
}
```

若要将上述类 complex 的两个对象相加，只要编写一个运算符函数 operator+()，如下所示：

```
complex operator+(complex om1,complex om2)
{
    complex temp;
    temp.real=om1.real+om2.real;
    temp.imag=om1.imag+om2.imag;
    return temp;
}
```

重载后就能方便地使用语句：

```
total=com1+com2;
```

对类 complex 的两个对象 com1 和 com2 相加。当然，在程序中可以使用以下的调用语句，对两个 complex 类对象执行加操作：

```
total=operator+(com1,com2);
```

以上两个调用语句是等价的，但是直接用 com1+com2 的形式更加符合人的书写习惯。

以下就是使用运算符重载函数 operator+()将两个 complex 类对象相加的完整程序。

例 6.1　运算符重载完成两个复数相加。

```
#include<iostream.h>
class complex{
```

```
public:
    double real,imag;
    complex(double r=0,double i=0)
    {    real=r; imag=i;}
};
complex operator+(complex om1,complex om2)
{
    complex temp;
    temp.real=om1.real+om2.real;
    temp.imag=om1.imag+om2.imag;
    return temp;
}
int main()
{
    complex com1(11.1,12.2),com2(13.3,14.4),sum;
    sum=com1+com2;
    cout<<"real="<<sum.real<<" "<<"imag="<<sum.imag<<endl;
    return 0;
}
```

程序运行结果为：

```
real=24.4 imag=26.6
```

6.1.2 运算符重载的规则

C++对运算符重载制定了以下一些规则：

（1）重载运算符与预定义运算符的使用方法完全相同，被重载的运算符不改变原来的操作数个数、优先级和结合性。

（2）重载的运算符只能是运算符集中的运算符，不能另创新的运算符。

（3）在 C++中，大多数系统预定义的运算符可以被重载，但也有些运算符不能被重载，如类属关系运算符“.”、成员指针运算符“.*”及“->*”、作用域标识符“::”、“sizeof”运算符和三目运算符“?:”。

（4）运算符的含义可以改变，但最好不要改变。如实数的加法运算可以用乘法运算符来实现。

（5）不能改变运算符对预定义类型的操作方式，即至少要有一个操作对象是自定义类型，这样做的目的是为了防止用户修改用于基本类型数据的运算符性质。

（6）运算符重载函数不能包括缺省的参数。

（7）除赋值运算符重载函数外，其他运算符重载函数都可以由派生类继承。

（8）运算符的重载实际上是函数的重载。编译程序对运算符重载的选择，遵循函数重载的选择原则。当遇到不很明显的运算符时，编译程序将去寻找参数匹配的运算符函数。

（9）用于类对象的运算符一般必须重载，但有两个例外，运算符“=”和“&”不必用户重载：

① 赋值运算符“=”可以用于每一个类对象，可以利用它在同类对象之间相互赋值。

② 地址运算符“&”也不必重载，它返回类对象在内存中的起始地址。

（10）运算符的重载只能显式重载，不能隐式重载。例如重载了“+”，并不表示可以自动重载“+=”，要想执行“+=”运算，必须再对其进行显式的重载。

6.2　运算符重载函数的两种形式

运算符重载与函数重载相似，其目的是设置某一运算符，让它实现另一种功能，尽管此运算符在原先 C++中代表另一种含义，但它们彼此之间并不冲突。例 6.1 中的运算符重载函数不属于任何类，是全局的函数。因为在 complex 类（复数类）中的数据成员是公有的，所以运算符重载函数可以被访问。但如果定义为私有呢，那该怎么办。实际上，运算符的重载有两种形式，定义为类的友元函数；定义为它将要操作的类的成员函数。前者称为友元运算符函数，后者称为成员运算符函数。

6.2.1　友元运算符重载函数

在 C++中，可以把运算符重载函数定义成某个类的友元函数，称为友元运算符重载函数。

1. 友元运算符重载函数定义的语法形式

（1）在类的内部，定义友元运算符重载函数的格式如下：

```
friend 返回类型 operator 运算符(形参表)
{
    函数体
}
```

（2）友元运算符重载函数也可以只在类中声明友元重载函数的原型，在类外定义。

在类中，声明友元重载函数的原型的格式如下：

```
class X
{
    …
    friend 返回类型 operator 运算符(形参表);
    …
}
```

在类外定义友元运算符重载函数的格式如下：

```
返回类型 operator 运算符(形参表)
{
    函数体
}
```

说明：

- X 是重载此运算符的类名，返回类型指定了友元运算符函数的运算结果的类型；
- operator 是定义运算符重载的关键字；
- 运算符就是需要重载的运算符名称，需满足 6.1.2 节中约定的规则；
- 形参表中给出重载运算符所需要的参数和类型；
- 关键字 friend 表明这是一个友元函数。

当运算符重载函数为友元函数时，将没有隐含的参数 this 指针。这样，对双目运算符友元函数带两个参数，对单目运算符友元函数带一个参数。

2. 友元函数重载双目运算符

双目运算符带两个操作数，通常在运算符的左右两侧，例如：

3+5，24>13

当用友元函数重载双目运算符时，两个操作数都要传递给运算符重载函数。请看下面的例子。

例 6.2 用友元运算符重载函数完成复数的加、减、乘、除。

```
#include<iostream.h>
class complex
{
public:
        complex(double r=0,double i=0)
        {
                real=r;
                imag=i;
        }
        void print();
        friend complex operator+(complex a,complex b);
        friend complex operator-(complex a,complex b);
        friend complex operator*(complex a,complex b);
        friend complex operator/(complex a,complex b);
private:
       double real;
       double imag;
};

complex operator+(complex a,complex b)
{
        complex temp;
        temp.real=a.real+b.real;
        temp.imag=a.imag+b.imag;
        return temp;
}
complex operator-(complex a,complex b)
{
        complex temp;
        temp.real=a.real-b.real;
        temp.imag=a.imag-b.imag;
        return temp;
}
complex operator*(complex a,complex b)
{
        complex temp;
        temp.real=a.real*b.real-a.imag*b.imag;
        temp.imag=a.real*b.imag+a.imag*b.real;
        return temp;
```

```
}
complex operator/(complex a,complex b)
{
    complex temp;
    double t;
    t=1/(b.real*b.real+b.imag*b.imag);
    temp.real=(a.real*b.real+a.imag*b.imag)*t;
    temp.imag=(b.real*a.imag-a.real*b.imag)*t;
    return temp;
}
void complex::print()
{
    cout<<real;
    if(imag>0)cout<<"+";
    if(imag!=0)cout<<imag<<"i\n";
}
void main()
{
    complex A1(2.3,4.6),A2(3.6,2.8),A3,A4,A5,A6;
    A3=A1+A2;
    A4=A1-A2;
    A5=A1*A2;
    A6=A1/A2;
    A1.print();
    A2.print();
    A3.print();
    A4.print();
    A5.print();
    A6.print();
}
```

程序运行结果为：

```
2.3+4.6i
3.6+2.8i
5.9+7.4i
-1.3+1.8i
-4.6+23i
1.01731+0.486538i
```

当用友元函数重载双目运算符时，两个操作数都要传递给运算符函数。一般而言，如果在类 X 中采用友元函数重载双目运算符@，而 aa 和 bb 是类 X 的两个对象，则采用以下两种方法调用是等价的：

```
aa@bb //隐式调用
aa.operator@(bb)//显式调用
```

如上例中，主函数 main()中的语句：

```
A3=A1+A2;
A4=A1-A2;
```

```
A5=A1*A2;
A6=A1/A2;
```

可以改写为：

```
A3=operator+(A1,A2);
A4=operator-(A1,A2);
A5=operator*(A1,A2);
A6=operator/(A1,A2);
```

说明：

（1）在函数返回时，也可直接用类的构造函数来生成一个临时对象，而不对该对象进行命名。

例如：

```
complex operator+(complex a,complex b)
{
    complex temp;
    temp.real=a.real+b.real;
    temp.imag=a.imag+b.imag;
    return temp;
}
```

改为：

```
complex operator+(complex a,complex b)
{
    return complex(a.real+b.real,a.imag+b.imag);
}
```

在返回语句中，通过构造函数建立临时对象作为返回值。这个临时对象没有对象名，是一个无名对象。

（2）Visual C++ 6.0 提供的不带后缀.h 的头文件不支持友元运算符重载函数，在 Visual C++ 6.0 中编译会出错，需要采用带后缀的.h 头文件。

例如：将程序中的

```
#include<iostream>
using namespace std;
```

修改为：

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

编译可以通过。

3. 友元函数重载单目运算符

用友元函数重载单目运算符时，需要一个显式的操作数。

例 6.3 用友元函数重载单目运算符“-”。

```
#include<iostream.h>
class nclass{
    int a,b;
public:
    nclass(int x=0,int y=0)
    { a=x;b=y;}
    friend nclass operator -(nclass obj);
```

```
        void show();
};
nclass operator-(nclass obj)
{
        obj.a=-obj.a;
        obj.b=-obj.b;
        return obj;
}
void nclass::show()
{ cout<<"a="<<a<<" b"<<b<<endl;}
main()
{
        nclass ob1(10,20),ob2;
        ob1.show();
        ob2=-ob1;
        ob2.show();
        return 0;
}
```

程序运行结果为：

```
a=10 b20
a=-10 b-20
```

使用友元函数重载“++”，“--”单目运算符时，可能会出现一些错误。

例 6.4　用友元函数重载单目运算符“++”。

```
#include <iostream.h>
class Point{
private:
        int x;
        int y;
public:
        Point(int x,int y)
        {
                this->x=x;
                this->y=y;
        }
        friend void operator++(Point point);//友元函数重载单目运算符++
        void showPoint();
};
void operator++(Point point)//友元运算符重载函数
{
        ++point.x;
        ++point.y;
}
void Point::showPoint()
{
        cout<<"("<<x<<","<<y<<")"<<endl;
}
```

```
int main()
{
    Point point(10,10);
    point.showPoint();
    ++point;//或 operator++(point)
    point.showPoint();//输出坐标值
    return 0;
}
```

程序运行结果为：

```
(10,10)
(10,10)
```

而我们希望的结果为：

```
(10,10)
(11,11)
```

显然这个结果与我们期望的结果有很大出入，产生这个错误的原因是因为友元函数没有this 指针，所以不能引用 this 指针所指的对象。这个函数采用对象通过值传递的方式传递参数，函数内对 point 的所有修改都无法传到主调函数中去。因此，使用友元函数重载单目运算符“++”或“--”时，采用引用参数来传递操作数，使得函数参数的任何改变都影响产生调用的操作数，从而保持了运算符“++”或“--”的原义。

例 6.5 友元函数采用引用参数重载单目运算符“++”。

```
#include <iostream.h>
class Point{
private:
    int x;
    int y;
public:
    Point(int x,int y)
    {
        this->x=x;
        this->y=y;
    }
    friend void operator++(Point &point);//友元函数重载单目运算符++
    void showPoint();
};
void operator++(Point &point)//友元运算符重载函数
{
    ++point.x;
    ++point.y;
}
void Point::showPoint()
{
    cout<<"("<<x<<","<<y<<")"<<endl;
}
int main()
{
```

```
    Point point(10,10);
    point.showPoint();
    ++point;//或 operator++(point)
    point.showPoint();//输出坐标值
    return 0;
}
```

程序运行结果为：

```
(10,10)
(11,11)
```

从上述程序可以看出，当用友元函数重载单目运算符“++”，参数表中有一个操作数 aa。一般可以采用以下两种方式来调用：

```
Operator++(aa);        //显示调用
++aa;                  //隐式（习惯）调用
```

说明：

（1）运算符重载函数 operator++()可以返回任何类型，甚至可以是 void 类型，但通常返回类型与它所操作的类的类型相同，这样可使重载运算符用在复杂的表达式中。

例如，可以将几个复数连续进行加运算：

```
A4=A3+A2+A1;
```

（2）不能用友元函数重载的运算符是：=、()、[]、->。

（3）C++编译器根据参数的个数和类型来决定调用哪个重载函数。因此，可以为同一个运算符定义几个运算符重载函数来进行不同的操作。

（4）由于单目运算符“-”（取反）可不改变操作数自身的值，所以单目运算符“-”的友元运算符重载函数的原型可写成：

```
friend AB operator-(AB obj);
```

通过传值的方式传送参数。

6.2.2　成员运算符重载函数

在 C++中，可以把运算符重载函数定义成某个类的成员函数，称为成员运算符重载函数。

1. 成员运算符重载函数定义的语法形式

（1）在类的内部，定义成员运算符重载函数的格式如下：

```
返回类型 operator 运算符(形参表)
{
    函数体
}
```

（2）成员运算符重载函数也可只在类中声明成员函数的原型，在类外进行定义。

在类中，声明成员运算符重载函数原型的格式如下：

```
class X
{
    …
    返回类型 operator 运算符(形参表);
    …
}
```

在类外，定义成员运算符重载函数的格式如下：

```
返回类型 X::operator 运算符(形参表)
{
    函数体
}
```

说明：

（1）X 是重载此运算符的类名，返回类型指定了成员运算符重载函数的运算结果类型。

（2）operator 是定义运算符重载的关键字。

（3）运算符就是需要重载的运算符名称，需满足 6.1.2 节中约定的规则。

（4）形参表中给出重载运算符所需要的参数和类型。

根据成员运算符重载函数中操作数的不同，可将运算符分为单目运算符与双目运算符。若为双目运算符，则成员运算符重载函数的参数表中只有一个参数，若为单目运算符，则参数表为空。

2. 用成员函数重载双目运算符

对双目运算符而言，成员运算符重载函数的参数表中仅有一个参数，它作为运算符的右操作数，此时当前对象作为运算符的左操作数。它是通过 this 指针隐含地传递给函数的。例如：

```
class comlpex{
    //…;
    complex operator+(complex a);
    //…;
};
```

在类 complex 中声明了重载“+”的成员运算符函数，返回类型为 complex，它具有两个操作数，一个是当前对象，是左操作数，另一个是对象 a，是右操作数。

例 6.6 采用成员函数重载运算符来完成复数的加、减、乘、除。

```
#include<iostream.h>
class complex
{
public:
     complex(double r=0.0,double i=0.0)
     {
          real=r;
          imag=i;
     }
     void print();
     complex operator+(complex a);
     complex operator-(complex a);
     complex operator*(complex a);
     complex operator/(complex a);
private:
    double   real;
    double   imag;
};
```

```
complex complex::operator+(complex a)        //实现复数加法运算的成员函数
{
        complex    temp;
        temp.real=a.real+real;
        temp.imag=a.imag+imag;
        return    temp;
}
complex complex::operator-(complex a)
{
        complex    temp;
        temp.real=real-a.real;
        temp.imag=imag-a.imag;
        return    temp;
}
complex complex::operator*(complex a)
{
        complex    temp;
        temp.real=real*a.real-imag*a.imag;
        temp.imag=real*a.imag+imag*a.real;
        return    temp;
}
complex complex::operator/(complex a)
{
        complex    temp;
         double t;
         t=1/(a.real*a.real+a.imag*a.imag);
        temp.real=(real*a.real+imag*a.imag)*t;
        temp.imag=(a.real*imag-real*a.imag)*t;
        return    temp;
}
void complex::print()
{
       cout<<real;
       if(imag>0)cout<<"+";
       if(imag!=0)cout<<imag<<"i\n";
}
void main()
{
       complex A1(2.3,4.6),A2(3.6,2.8),A3,A4,A5,A6;
       A3=A1+A2;
       A4=A1-A2;
       A5=A1*A2;
       A6=A1/A2;
       A1.print();
       A2.print();
       A3.print();
```

```
        A4.print();
        A5.print();
        A6.print();
    }
```

程序运行结果为：

```
2.3+4.6i
3.6+2.8i
5.9+7.4i
-1.3+1.8i
-4.6+23i
1.01731+0.486538i
```

在上述例题中，主函数中以下四条语句：

```
A3=A1+A2;
A4=A1-A2;
A5=A1*A2;
A6=A1/A2;
```

实质上是对运算符“+”、“-”、“*”、“/”的重载，程序执行到这四条语句时，C++将其解释为：

```
A3=A1.operator+(A2);
A4=A1.operator-(A2);
A5=A1.operator*(A2);
A6=A1.operator/(A2);
```

由此可知，成员运算符重载函数 operator @实际上是由双目运算符的左边对象 A1 调用的。尽管参数表中只有一个操作数 A2，但另一个操作数由对象 A1 通过 this 指针隐含地传递。

一般而言，采用成员函数重载双目运算符@后，可以采用如下两种方法来调用：

```
aa@bb //隐式调用
aa.operator@(bb)//显式调用
```

成员运算符重载函数 operator@所需要的一个操作数由当前调用成员运算符重载函数的对象 aa 通过 this 指针隐含地传递。因此它的参数表中只有一个操作数 bb。

3. 用成员函数重载单目运算符

对单目运算符而言，成员运算符重载函数的参数表中没有参数，此时当前对象作为运算符的唯一操作数。

例 6.7 成员运算符重载函数重载单目运算符“++”。

```
#include <iostream.h>
class coord{
    int x,y;
public:
    coord(int i=0,int j=0);
    void print();
    coord operator++();
};
coord::coord(int i,int j)
{   x=i;y=j;}
```

```
void coord::print()
{     cout<<"x="<<x<<",y="<<y<<endl;};
coord coord::operator++()
{
      ++x;
      ++y;
      return *this;
}
int main()
{
      coord ob(10,20);
      ob.print();
      ++ob;
      ob.print();
      ob.operator++();
      ob.print();
      return 0;
}
```

程序运行结果为：

```
x=10,y=20
x=11,y=21
x=12,y=22
```

从本例可以看出，当成员函数重载单目运算符时，没有参数被显式地传递给成员运算符重载函数，参数是通过 this 指针隐含地传递给函数。

一般而言，采用成员函数重载单目运算符@后，可采用如下两种方法来调用：

```
@aa //隐式调用
aa.operator@()//显式调用
```

成员运算符重载函数 operator@所需要的操作数由当前调用成员运算符重载函数的对象 aa 通过 this 指针隐含地传递，因此，它的参数表中没有参数。

6.2.3　友元运算符重载函数与成员运算符重载函数的比较

（1）对双目运算符而言，成员运算符重载函数带一个参数，而友元运算符重载函数带两个参数；对单目运算符而言，成员运算符重载函数不带参数，而友元运算符重载函数带一个参数。

（2）双目运算符一般可以重载为友元运算符重载函数或成员运算符重载函数。如果运算符重载函数中两个操作数类型不同，则一般只能采用友元运算符重载函数，而且有些情况必须采用友元函数。

例 6.8　在类 AB 中，用成员运算符函数重载“+”运算符。

```
#include<iostream.h>
class AB
{
public:
      AB(int x=0,int y=0);
```

```
        AB operator+(int x);
        void show();
private:
        int a,b;

};
AB::AB(int x,int y)
{
        a=x;
        b=y;
}
AB AB::operator+(int x)
{
        AB temp;
        temp.a=a+x;
        temp.b=b+x;
        return temp;
}
void AB::show()
{
        cout<<"a="<<a<<" "<<"b="<<b<<endl;
}
void main()
{
        AB ob1(50,60),ob2;
        ob2=ob1+20;
        ob2.show();
        //ob2=30+ob1; 编译报错
        ob2.show();
}
```

在本程序中类 AB 的对象 ob1 与 20 做加法运算，以下语句是正确的：

```
ob2=ob1+20;
```

由于对象 ob1 是运算符“+”的左操作数，所以它调用了“+”运算符重载函数，把一个整数加到了对象 ob1 的某些元素上。然而，下一条语句就不能正常工作了：

```
ob2=30+ob1;
```

程序编译会报错，原因是由于 30 是系统预定义的数据类型，不能对成员函数进行调用，成员函数的调用一般是通过对象来调用的，解决这个问题的办法是用两个友元函数来重载运算符函数“+”，可以将两个参数都显式地传递给运算符函数。这样系统预定义的数据类型就可以出现在运算符的左边，请看下面的例子。

例 6.9 系统预定义数据类型出现在运算符的左边。

```
#include<iostream.h>
class AB
{
public:
        AB(int x=0,int y=0);
```

```
        friend AB operator+(AB ob,int x);
        friend AB operator+(int x,AB ob);
        void show();
    private:
        int a,b;

    };
    AB::AB(int x,int y)
    {
        a=x;
        b=y;
    }
    AB operator+(AB ob,int x)
    {
        AB temp;
        temp.a=ob.a+x;
        temp.b=ob.b+x;
        return temp;
    }
    AB operator+(int x,AB ob)
    {
        AB temp;
        temp.a=x+ob.a;
        temp.b=x+ob.b;
        return temp;
    }
    void AB::show()
    {
        cout<<"a="<<a<<" "<<"b="<<b<<endl;
    }
    void main()
    {
        AB ob1(50,60),ob2;
        ob2=ob1+20;
        ob2.show();
        ob2=40+ob1;
        ob2.show();
    }
```

程序运行结果为：

```
a=70 b=80
a=90 b=100
```

（3）成员运算符重载函数和友元运算符重载函数都有两种调用方式：显式调用与隐式调用。

（4）运算符的优先级决定怎样将一个表达式改写为函数调用形式。如：

```
a+b*c                operator+(a,operator*(b,c))
a+b*!c               operator+(a,operator*(b,c.operator!()))
```

```
a*(b+c)              operator*(a,operator+(b,c))
a+b+c                operator+(operator+(a,b),c)
a=b+=c               operator=(b.opertaor+=( c ))
```

（5）对同一运算符重载时，成员函数重载比友元函数重载少一个参数。

（6）C++的大部分运算符既可以说明为友元运算符重载函数，也可说明为成员运算符重载函数。一般来讲，单目运算符最好重载为成员函数，而双目运算符则最好重载为友元函数。

6.3 几种常用运算符重载

6.3.1 前缀运算符和后缀运算符的重载

针对预定义数据类型，C++提供了自增运算符“++”和自减运算符“--”，这两个运算符都有前缀与后缀两种形式。早期 C++版本虽然能重载这两个运算符，但不能区分它们的两种形式。在后期的 C++版本中，编译器可以通过在运算符函数参数表中是否包含关键字 int 来区分前缀与后缀。以“++”重载运算符为例，其语法格式如下：

（1）成员运算符重载函数

```
<函数类型> operator ++();     //前缀运算
<函数类型> operator ++(int);   //后缀运算
```

在调用后缀方式的函数时，参数 int 一般被传值为 0。

（2）友元运算符重载函数

```
friend <函数类型> operator ++(X &ob);     //前缀运算
friend <函数类型> operator ++(X &ob ,int);   //后缀运算
```

在调用后缀方式的函数时，参数 int 一般被传值为 0。

重载“--”可以采用类似的方法。

例如，已知对象 ob，使用前缀方式++调用形式有以下几种：

```
++ob;//隐式调用
operator++(ob);//显式调用，友元运算符重载函数
ob.operator++();//显式调用，成员运算符重载函数
```

使用后缀方式++调用形式有以下几种：

```
ob++;//隐式调用
operator++(ob,0);//显式调用，友元运算符重载函数
ob.operator++(0);//显式调用，成员运算符重载函数
```

例 6.10 成员运算符函数重载“++”的前缀和后缀方式。

```
#include<iostream.h>
class over{
public:
    void init(int x)
    {     a=x;  }
    void print()
    {     cout<<"a="<<a<<endl;}
    over operator++()
    {
```

```
        ++a;
        return *this;
    }
    over operator++(int)
    {
        a++;
        return *this;
    }
private:
    int a;
};

void main()
{
    over obj1,obj2;
    obj1.init(2);
    obj2.init(4);
    obj1.operator++();//++obj1
    obj2.operator++(0);//obj2++
    obj1.print();
    obj2.print();
}
```

程序运行结果为：

```
a=3
a=5
```

例 6.11　友元运算符函数重载“--”的前缀和后缀方式。

```
#include<iostream.h>
class over{
public:
    void init(int x)
    {    a=x;  }
    void print()
    {    cout<<"a="<<a<<endl;}
    friend over operator--(over &ob)
    {
        --ob.a;
        return ob;
    }
    friend over operator--(over &ob,int)
    {
        ob.a--;
        return ob;
    }
private:
    int a;
};
```

```
void main()
{
    over obj1,obj2;
    obj1.init(2);
    obj2.init(4);
    operator--(obj1);//--obj1
    operator--(obj2,0);//obj2--
    obj1.print();
    obj2.print();
}
```

程序运行结果为：

```
a=1
a=3
```

6.3.2 赋值运算符的重载

对任一类 X，如果没有用户自定义的赋值运算符函数，系统将自动地为其生成一个缺省的赋值运算符函数，完成类 X 中的数据成员间的赋值，例如：

```
X &X::operator=(const X &source）
{
    //…成员间赋值
}
```

若 Obj1 和 Obj2 是类 X 的两个对象，Obj2 已被创建，当编译程序遇到如下语句：

```
Obj1＝Obj2；
```

就调用缺省的赋值运算符函数，将对象 Obj2 的数据成员的值逐个赋给对象 Obj1 的对应数据成员。

如果类中包含指向动态空间的指针，调用默认赋值运算符会导致“浅拷贝”，造成内存泄漏或程序异常。此时需要重载赋值运算符和拷贝构造函数，以实现“深拷贝”。

例 6.12 使用缺省的赋值运算符产生指针悬挂问题。

```
#include<iostream.h>
#include<string.h>
class string{
public:
    string(char *s)
    {
        ptr=new char[strlen(s)+1];
        strcpy(ptr,s);
    }
    ～string()
    {   delete ptr;  }
    void print()
    {  cout<<ptr<<endl; }
private:
    char *ptr;
```

```
};
void main()
{
    string p1("teacher");
    string p2("student");
    p1=p2;
    cout<<"p2: ";
    p2.print();
    cout<<"p1: ";
    p1.print();
}
```

从这个例子可以看出，通过使用缺省的赋值运算符“=”，把对象 p2 的数据成员值逐个赋值给对象 p1 的对应数据成员，从而使得 pl 的数据成员原先的值被覆盖。由于 pl.ptr 和 p2.ptr 具有相同值，都指向 p2 的字符串，p1.ptr 原先指向的内存区不仅没有释放，而且被封锁起来无法再使用，这就是所谓的指针悬挂问题。更为严重的是，对于存储“student”内容的内存在程序结束时被释放了两次，从而导致运行错误。可以通过重载赋值运算符来解决指针悬挂问题。

例 6.13　重载赋值运算符解决指针悬挂问题。

```
#include<iostream.h>
#include<string.h>
class string{
public:
    string(char *s)
    {
        ptr=new char[strlen(s)+1];
        strcpy(ptr,s);
    }
    ~string()
    {    delete ptr;   }
    void print()
    {   cout<<ptr<<endl; }
    string &operator=(const string &);
private:
    char *ptr;
};
string &string::operator=(const string &s)
{
    if(this==&s)return *this;
    delete ptr;
    ptr=new char[strlen(s.ptr)+1];
    strcpy(ptr,s.ptr);
    return *this;
}
void main()
{
```

```
    string p1("teacher");
    string p2("student");
    p1=p2;
    cout<<"p2: ";
    p2.print();
    cout<<"p1: ";
    p1.print();
}
```

运行修改后的程序，就不会产生上面的问题了。因为已释放掉了旧区域，又按新长度分配了新区域，并且将简单的赋值变成了内容拷贝。程序运行结果为：

```
p2: student
p1: student
```

说明：

- 类的赋值运算符“=”只能重载为成员函数，而不能把它重载为友元函数；
- 类的赋值运算符“=”可以被重载，但重载了的运算符函数 operator=()不能被继承。

6.3.3 下标运算符的重载

在 C++中，在重载下标运算符“[]”时认为它是一个双目运算符，第一个运算符是数组名，第二个运算符是数组下标。例如 X[Y]可以看成：

[]——双目运算符

X——左操作数

Y——右操作数

C++不允许把下标运算符函数作为外部函数来定义，它只能是非静态的成员函数。对于下标运算符重载定义形式如下：

```
返回类型 类名::operator[](形参)
{
    //函数体
}
```

其中，类名是重载下标运算符的类，形参在此表示下标，C++规定只能有一个参数，可以是任意类型，如整型、字符型或某个类。返回类型是数组运算的结果，也可以是任意类型，但为了能对数组赋值，一般将其声明为引用形式。假设 X 是某一个类的对象，类中定义了重载“[]”的 operator[]函数，表达式 X[5]可被解释为：

```
X.operator[](5);
```

例 6.14 下标运算符重载的一个实例。

```
#include <iostream.h>
class aInteger{
public:
    aInteger(int size)
    {
        sz=size;
        a=new int[size];
    }
    int& operator[](int i);
```

```
    ～aInteger()
    {    delete []a;}
private:
    int* a;
    int sz;
};
int& aInteger::operator[](int i)
{
    if(i<0||i>sz)
    {
        cout<<"error,leap the pale"<<endl;
    }
    return a[i];
}
int main()
{
    aInteger arr(10);
    for(int i=0;i<10;i++)
    {
        arr[i] = i+1;
        cout<<arr[i]<<"    ";
    }
    cout<<endl;
    int n=arr.operator[](2);
    cout<<"n="<<n<<endl;
    return 0;
}
```

程序运行结果为：

```
1 2 3 4 5 6 7 8 9 10
n=3
```

在整型数组 aInteger 中重载了下标运算符，这种下标运算符能检查越界的错误。现在使用它：

```
aInteger ai(10);
ai[2]=3;
int i=ai[2];
```

ai[2]=3 调用了 ai.operator(2)，返回对 ai::a[2]的引用，接着再调用缺省的赋值运算符，把 3 的值赋给此引用，因而 ai::a[2]的值为 3。注意，假如返回值不采用引用形式，ai.operator(2)的返回值是一临时变量，不能作为左值，因而，上述赋值会出错。对于初始化 i=ai[2]，先调用 ai.operator(2)取出 ai::a[2]的值，然后再利用缺省的拷贝构造函数来初始化 i。

说明：

- 重载下标运算符“[]”的优点之一是可以增加 C++中数组检索的安全性，可以对下标越界做出判断；
- 重载下标运算符“[]”时返回一个 int 的引用，所以可使重载运算符“[]”用在赋值语句的左边。

6.3.4 函数调用运算符的重载

在C++中，在对函数调用运算符“()”重载时认为它是一个双目运算符，第一个运算符是函数名，第二个运算符是函数的形参。例如X(Y)可以看成：

()——双目运算符

X——左操作数

Y——右操作数

其相应的运算符函数名为operator()，必须重载为一个成员函数，则重载运算符“()”的调用由左边的对象产生对operator函数的调用，this指针总是指向发起函数调用的对象。

重载运算符“()”的operator函数可以带任何类型的操作数，返回类型也可以是任何有效的类型。对于函数调用运算符的重载定义形式如下：

```
返回类型 类名::operator()(形参)
{
    //函数体
}
```

例6.15 形参为对象引用的函数调用运算符的重载实例。

```
#include<iostream.h>
class Point{
private:
    int x;
public:
    Point(int x1)
    {    x=x1;}
    const int operator()(const Point& p);
};
const int Point::operator()(const Point& p)
{
    return (x+p.x);
}
int main()
{
    Point a(1);
    Point b(2);
    cout<<"a(b)="<<a(b)<<endl;//a.operator()(b);
    return 0;
}
```

程序运行结果为：

```
a(b)=3
```

在程序中“a(b);”相当于“a.operator()(b);”。对象a是函数调用运算符重载函数“()”的左操作数，而b构成实参，当执行a(b)时，对对象a和对象b的两个私有成员值进行加运算。

例6.16 设数学表达式为$f(x,y)=(x^2+y^2)*z$，通过函数调用运算符的重载实现该表达式。

```
#include <iostream.h>
class fun{
```

```
public:
    double operator()(double x,double y,double z) const;
};
double fun::operator()(double x,double y,double z) const
{
    return (x*x+y*y)*z;
}
void main()
{
    fun f;
    cout<<f(3.0,2.0,5.0)<<endl;
}
```

程序运行结果为：

```
65
```

执行 f(3.0,2.0,5.0)时，对象 f 是函数调用运算符“()”的左操作数，而 3.0、2.0、5.0 构成实参表；其左操作数是 fun 类的对象 f，它将执行对 fun 类的重载函数的调用。

6.4　类型转换

我们在使用重载的运算符时，往往需要在自定义的数据类型和系统预定义的数据类型之间进行转换，或者需要在不同的自定义数据类型之间进行转换。本节介绍 C++中数据类型的转换。

6.4.1　系统预定义类型间的转换

对于系统预定义的数据类型，C++提供了两种类型转换方式，即隐式类型转换（或称标准类型转换）和显式类型转换。

（1）隐式类型转换

隐式类型转换主要遵循以下规则：

- 字符或 short 类型变量与 int 类型变量进行运算时，将字符或 short 类型转换成 int 类型；
- float 型数据在运算时一律转换为双精度（double）型，以提高运算精度（同属于实型）；
- 在赋值表达式 A=B 中，赋值运算符右端 B 的值需转换为 A 类型后再进行赋值；
- 当两个操作对象类型不一致时，在算术运算前，级别低的自动转换为级别高的类型。

（2）显式类型转换

- 强制转换法

(类型名)表达式

例如：

```
int i,j;
cout<<(float)(i+j);
```

- 函数法

类型名(表达式)

例如：

```
int i,j;
cout<<float (i+j)
```

例 6.17 系统预定义类型间转换。

```
#include<iostream.h>
void main()
{
    int a=5,sum;
    double b=5.55;
    sum=a+b;//-------①
    cout<<"隐式转换：a+b="<<sum<<endl;
    sum=(int)(a+b);//-------②
    sum=int(a+b);   //-------③
    cout<<"显式转换：a+b="<<sum<<endl;
}
```

程序运行结果为：

```
隐式转换：a+b=10
显式转换：a+b=10
```

上述代码中的①就是含有隐式类型转换的表达式，在进行“a+b”时，编译系统先将 a 的值 5 转换为双精度 double，然后和 b 相加得到 10.55，在向整型变量 sum 赋值时，将 10.55 转换为整型数 10，赋值给变量 sum。这种转换由 C++编译系统自动完成，不需要用户干预。而上例中的②和③中则涉及到了显式类型转换，它们都是把 a+b 所得结果的值强制转换为整型数。只是②式是 C 语言中用到的形式“(类型名)表达式”，而③式是 C++中采用的形式“类型名(表达式)”。

6.4.2 类类型与系统预定义类型间的转换

对于用户自己定义的类类型而言，编译系统并不知道怎样进行转换。解决这个问题的关键是让编译系统知道怎样去进行这些转换，需要定义专门的函数来处理。C++中提供了如下两种方法：

- 通过转换构造函数进行类型转换；
- 通过类型转换函数进行类型转换。

1. 通过转换构造函数进行类型转换

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。主要有以下几种构造函数：

（1）默认构造函数。以 Complex 类为例，函数原型的形式为：

```
Complex( );  //没有参数
```

（2）用于初始化的构造函数。函数原型的形式为：

```
Complex(double r,double i);        //形参列表中一般有两个以上参数
```

（3）用于复制对象的拷贝构造函数。函数原型的形式为：

```
Complex (Complex &c);              //形参是本类对象的引用
```

（4）再介绍一种新的构造函数——转换构造函数，转换构造函数只有一个形参，如：

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

其作用是将 double 型的参数 r 转换成 Complex 类的对象，将 r 作为复数的实部，虚部为 0。用户可以根据需要定义转换构造函数，在函数体中告诉编译系统怎样去进行转换。

毫无疑问转换构造函数就是构造函数的一种，只不过它具有类型转换的作用。回顾下例 6.1 中两个复数（sum=com1+com2）相加的实例，现在如果想要实现 sum=com1+5.5 该怎么办，也许首先会想到再定义一个关于复数加双精度数的运算符重载函数，这样做的确可以。另外还可以定义一个转换构造函数来解决上述问题。可对 Comlex 类（复数类）进行如下改造。

例 6.18　使用转换构造函数完成双精度数到复数的转换。

```
#include<iostream.h>
class Complex //复数类
{
private://私有
    double real;//实数
    double imag;//虚数
public:
    Complex(double real,double imag)
    {
        this->real=real;
        this->imag=imag;
    }
    Complex(double d=0.0)//转换构造函数
    {
        real=d;//实数取 double 类型的值
        imag=0;//虚数取 0
    }
    Complex operator+(Complex com1);
    void showComplex();
};
Complex Complex::operator+(Complex com1)
{
    return Complex(real+com1.real,imag+com1.imag);
}
void Complex::showComplex()
{
    cout<<real;
    if(imag>0)
        cout<<"+";
    if(imag!=0)
        cout<<imag<<"i"<<endl;
    else
        cout<<endl;
}
int main()
{
    Complex com(10,10),sum;
    sum=com+Complex(5.5);//①
```

```
    sum.showComplex();//输出运算结果
    sum=com+6.6;          //②
    sum.showComplex();//输出运算结果
    sum=5.5;              //③
    sum.showComplex();//输出运算结果
    return 0;
}
```

程序运行结果为：

```
15.5+10i
16.6+10i
5.5
```

main()函数中的语句①显式地调用转换构造函数 Complex(5.5)，将 double 类型的 5.5 转换为无名的 Complex 类的临时对象（5.5+0i），然后执行两个 Complex 类（复数类）对象相加的运算符重载函数。语句②隐式地调用转换构造函数 Complex(6.6)，将 double 类型的 6.6 转换为无名的 Complex 类的临时对象（6.6+0i），然后执行两个 Complex 类（复数类）对象相加的运算符重载函数。语句③隐式调用 Complex(5.5)，将 double 类型的 5.5 转换为无名的 Complex 类的临时对象（5.5+0i），然后调用缺省的赋值运算符函数完成对 sum 的赋值。所以说一般的转换构造函数的定义形式如下：

```
类名(待转换类型)
{
    函数体;
}
```

说明：

（1）转换构造函数只有一个形参。

（2）在类体中，可以有转换构造函数，也可以没有转换构造函数，视需要而定。

（3）当几种构造函数同时出现在同一个类中，它们是构造函数的重载。编译系统会根据建立对象时给出的实参的个数与类型来选择与之匹配的构造函数。

（4）转换构造函数不仅可以将系统预定义类型的数据转换成类对象，也可以将另一个类的对象转换成转换构造函数所在类的对象。

例如，将一个学生类对象转换为教师类对象，可以在 Teacher 类中定义出下面的转换构造函数：

```
Teacher(Student &s)
{
    num=s.num;
    strcpy(name,s.name);
    sex=s.sex;
}
```

但应注意，对象 s 中 num、name 和 sex 必须是公有成员，否则不能在类外被访问。

2. 通过类型转换函数进行类型转换

转换构造函数可以把系统预定义类型转化为自定义类的对象，但是却不能把类的对象转换为基本数据类型。比如，不能将 Complex 类（复数类）的对象转换成 double 类型数据。在 C++中可用类型转换函数（type conversion function）来解决这个问题。类型转换函数的作用是

将一个类的对象转换成另一类型的数据，定义类型转换函数的一般形式为：

```
class 源类类名{
    //…
    operator 目标类型( )
    {
        //…
        return 目标类型的数据;
    }
    //…
};
```

说明：

（1）源类类名为要转换的源类类型，目标类型为要转换成的类型，它既可以是自定义的类型也可以是系统预定义类型。

（2）类型转换函数的函数名前不能指定返回类型，且没有参数。从函数的形式看，它与运算符重载函数相似，都是用 operator 开头，只是被重载的是类型名。

（3）类型转换函数只能定义为一个类的成员函数，因为转换的主体是本类对象。不能为友元函数或普通函数。

（4）类型转换函数中必须有“return 目标类型的数据;”的语句，即必须回传目标类型数据作为函数的返回值。

（5）在类中可以有多个类型转换函数，C++编译器将根据操作数的类型自动地选择一个合适的类型转换函数与之匹配。

例 6.19　类型转换函数显式调用举例。

```
#include<iostream.h>
class Complex{
public:
    Complex(float r=0,float i=0)
    {
        real=r;
        imag=i;
        cout<<"Complex class Constructing...\n";
    }
    operator float()
    {
        cout<<"Complex changed to float...\n";
        return real;
    }
    operator int()
    {
        cout<<"Complex changed to int...\n";
        return int(real);
    }
    void print()
    {   cout<<'('<<real<<','<<imag<<')'<<endl;}
private:
```

```
    float real,imag;
};
int main()
{
    Complex a(1.1,3.3);
    a.print();
    cout<<float(a)*0.5<<endl;
    Complex b(2.5,5.5);
    b.print();
    cout<<int(b)*2<<endl;
    return 0;
}
```

程序运行结果为：

```
Complex class Constructing...
(1.1,3.3)
Complex changed to float...
0.55
Complex class Constructing...
(2.5,5.5)
Complex changed to int...
4
```

上例中两次调用了类型转换函数。第一次采用显式调用，将类对象 a 转换成 float 类型。第二次也是显式调用，将类对象 b 转换为 int 类型。类型转换函数的调用分为显式转换与隐式转换两种形式，下面通过例 6.20 来说明如何进行隐式转换。

例 6.20 类型转换函数隐式调用的举例。

```
#include<iostream.h>
class Complex{
public:
    Complex(int r,int i)
    {
        real=r;
        imag=i;
        cout<<"Complex class Constructing...\n";
    }
    Complex(int i=0) //转换构造函数
    {   real=imag=i/2;}
    operator int()    //类型转换函数
    {
        cout<<"Complex changed to int...\n";
        return real+imag;
    }
    void print()
    {   cout<<"real:"<<real<<"\t"<<"imag:"<<imag<<endl;}
private:
    int real,imag;
```

```
};
int main()
{
    Complex c1(1,2),c2(3,4);
    c1.print();
    c2.print();
    Complex c3;
    c3=c1+c2;
    c3.print();
}
```

程序运行结果为：

```
Complex class Constructing...
Complex class Constructing...
real:1    imag:2
real:3    imag:4
Complex changed to int...
Complex changed to int...
real:5    imag:5
```

上例中没有在类中重载运算符“+”，为何 c3=c1+c2 成立呢？原因在于 C++自动进行了隐式转换，将 c1 与 c2 分别都转换成 int 型，然后调用转换构造函数将整型数据转换为 complex 类。这里类型转换函数和转换构造函数构成了互逆操作，转换构造函数 Complex(int)将一个整型转换成一个 Complex 类型，而类型转换函数 Complex::operator int()将 Complex 类型转换成整型。

一、选择题

1．下列运算符中，_____运算符在 C++中不能被重载。

A．=　　B．()　　C．::　　D．delete

2．下列运算符中, _____运算符在 C++中不能被重载。

A．?:　　B．[]　　C．new　　D．&&

3．下列关于 C++运算符函数的返回类型的描述中，错误的是_____。

A．可以是类类型　　B．可以是 int 类型

C．可以是 void 类型　　D．可以是 float 类型

4．下列运算符不能用友元函数重载的是_____。

A．+　　B．=　　C．*　　D．<<

5．在重载运算符函数时，下面_____运算符必须重载为类成员函数形式。

A．+　　B．-　　C．++　　D．->

6．下列关于运算符重载的描述中，正确的是_____。

A．运算符重载可以改变运算符操作数的个数

B．运算符重载可以改变优先级

C．运算符重载可以改变结合性

D．运算符重载不可以改变语法结构

7．友元运算符 obj>obj2 被 C++编译器解释为_____。

A．operator>(obj1,obj2)　　B．>(obj1,obj2)

C．obj2.operator>(obj1)　　D．obj1.operator>(obj2)

8．在表达式 x+y*z 中，+是作为成员函数重载的运算符，*是作为非成员函数重载的运算符。下列叙述中正确的是_____。

A．operator+有两个参数，operator*有两个参数

B．operator+有两个参数，operator*有一个参数

C．operator+有一个参数，operator*有两个参数

D．operator+有一个参数，operator*有一个参数

9．重载赋值操作符时，应声明为_____函数。

A．友元　　B．虚　　C．成员　　D．多态

10．在一个类中可以对一个操作符进行_____重载。

A．1 种　　B．2 种以下　　C．3 种以下　　D．多种

11．在重载一个运算符时，其参数表中没有任何参数，这表明该运算符是_____。

A．作为友元函数重载的单目运算符　　B．作为成员函数重载的单目运算符

C．作为友元函数重载的双目运算符　　D．作为成员函数重载的双目运算符

12．在成员函数中进行双目运算符重载时，其参数表中应带有_____个参数。

A．0　　B．1　　C．2　　D．3

13．双目运算符重载为普通函数时，其参数表中应带有_____个参数。

A．0　　B．1　　C．2　　D．3

14．如果表达式 a+b 中的“+”是作为成员函数重载的运算符，若采用运算符函数调用格式，则可表示为_____。

A．a.operator+(b)　　B．b.operator+(a)

C．operator+(a,b)　　D．operator(a+b)

15．如果表达式 a==b 中的“==”是作为普通函数重载的运算符，若采用运算符函数调用格式，则可表示为_____。

A．a.operator==(b)　　B．b.operator==(a)

C．operator==(a,b)　　D．operator==(b,a)

16．如果表达式 a++中的“++”是作为普通函数重载的运算符，若采用运算符函数调用格式，则可表示为_____。

A．a.operator++()　　B．operator++(a)

C．operator++(a,1)　　D．operator++(1,a)

17．如果表达式++a 中的“++”是作为成员函数重载的运算符，若采用运算符函数调用格式，则可表示为_____。

A．a.operator++(1)　　B．operator++(a)

C．operator++(a,1)　　D．a.operator++()

18．关于运算符重载，下列说法正确的是_____。

A．重载时，运算符的优先级可以改变　　B．重载时，运算符的结合性可以改变

C．重载时，运算符的功能可以改变　　D．重载时，运算符的操作数个数可以改变

19．关于运算符重载，下列说法正确的是_____。

A．所有的运算符都可以重载

B．通过重载，可以使运算符应用于自定义的数据类型

C．通过重载，可以创造原来没有的运算符

D．通过重载，可以改变运算符的优先级

20．一个程序中数组 a 和变量 k 定义为“int a[5][10],k;”，且程序中包含有语句“a(2,5)=++k*3;”，则此语句中肯定属于重载操作符的是_____。

A．( )　　B．=　　C．++　　D．*

21. 假定 K 是一个类名，并有定义“K k; int j;”，已知在 K 中重载了操作符( )，且语句“j=k(3);”和“k(5)=99;”都能顺利执行，则该操作符函数的原型只可能是_____。

A．K operator( )(int);　　B．int operator( )(int);

C．int & operator( )(int);　　D．K & operator( )(int);

22．假定 M 是一个类名，且 M 中重载了操作符“=”，可以实现 M 对象间的连续赋值，如“m1=m2=m3;”。重载操作符“=”的函数原型最好是_____。

A．int operaotor=(M);　　B．int operator=(M);

C．M operator=(M);　　D．M & operator=(M);

23．下面是重载双目运算符“+”的普通函数原型，其中最符合“+”原来含义的是_____。

A．Value operator+(Value, Value);　　B．Value operator+(Value,int);

C．Value operator+(Value);　　D．Value operator+(int , Value);

24．下面是重载双目运算符“-”的成员函数原型，其中最符合“-”原来含义的是_____。

A．Value Value::operator-(Value);　　B．Value Value::operator-(int);

C．Value Value::operator-(Value,int);　　D．Value Value::operator-(int,Value);

25．在重载运算符时，若运算符函数的形参表中没有参数，则不可能的情况是_____。

A．该运算符是一个单目运算符　　B．该运算符函数有一个隐含的参数 this

C．该运算符函数是类的成员函数　　D．该运算符函数是类的友元函数

二、填空题

1．不能重载“.”、“::”、“.*”、“->*”和_____5 个运算符。

2．运算符重载不能改变运算符的_____。

3．在 C++中，运算符的重载有两种实现方法，一种是通过成员函数来实现，另一种则通过_____来实现。

4．运算符“--”、“delete”、“-=”、“=”、“+”和“->*”中，只能作为成员运算符重载的有_____。

5．运算符“--”、“delete”、“-=”、“=”、“+”和“->*”中，只能重载为静态函数的有_____。

6．运算符“--”、“delete”、“-=”、“=”、“+”和“->*”中，肯定不能重载的是_____。

7．当运算符重载为成员函数时，对象本身就是_____，不在参数表中显式地出现。

8．若以成员函数形式，为类 CSAI 重载“double”运算符，则该运算符重载函数的原型是_____。

9．在表达式“x+=y”中，“+=”是作为非成员函数重载的运算符，若是使用显式的函数调用代替直接使

用运算符“+=”，这个表达式还可以表示为_____。

10．将运算符“>>”重载为类 CSAI 的友元函数的格式是：friend stream& operator >>_____。

三、分析题

1．分析以下程序的执行结果

```
#include<iostream.h>
class   Sample{
      int n;
public:
      Sample(){}
      Sample(int m){n=m;}
      int &operator--(int)
      {
            n--;
            return n;
      }
      void disp()
      {       cout<<"n="<<n<<endl;}
};
void main()
{
      Sample s(10);
      (s--)++;
      s.disp();
}
```

2．分析以下程序的执行结果

```
#include<iostream.h>
class   Sample{
private:
      int x;
public:
      Sample(){x=0;}
      void disp(){cout<<"x="<<x<<endl;}
      void operator++(){x+=10;}
};
void main()
{
      Sample obj;
      obj.disp();
      obj++;
      cout<<"执行 obj++之后"<<endl;
      obj.disp();
}
```

3．分析以下程序的执行结果

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

```
static int dys[]={31,28,31,30,31,30,31,31,30,31,30,31};
class date{
    int mo,da,yr;
public:
    date(int m,int d,int y){mo=m;da=d;yr=y;}
    date(){}
    void disp(){cout<<mo<< " / "<<da<<" / "<<yr<<endl;}
    friend date operator+(date &d,int day) //友元运算符重载函数
    {
        date dt;
        dt.mo=d.mo;
        dt.yr=d.yr;
        day+=d.da;
        while(day>dys[dt.mo-1])
        {
            day-=dys[dt.mo-1];
            if(++dt.mo==13)
            {
                dt.mo=1;
                dt.yr++;
            }
        }
        dt.da=day;
        return dt;
    }
};
void  main()
{
    date dl(2,10,2003),d2;
    d2=dl+365;
    d2.disp();
}
```

4．分析以下程序的执行结果

```
#include<iostream.h>
#include<string.h>
class  Words{
    char *str;
public:
    Words(char *s)
    {
        str=new char[strlen(s)+1];
        strcpy(str,s);
    }
    void disp(){cout<<str<<endl;}
    char operator[](int i)
    {
```

```
            if(str[i]>='A'&& str[i]<='Z')   //大写字母
                return char(str[i]+32);
            else if(str[i]>='a'&&str[i]<='z') //小写字母
                return char(str[i]-32);
            else                //其他字符
                return str[i];
        }
};
void main()
{
        int i;
        char *s="Hello";
        Words word(s);
        word.disp();
        for(i=0;i<strlen(s);i++)
            cout<<word[i];
        cout<<endl;
}
```

5．分析以下程序的执行结果

```
#include<iostream.h>
class   Point{
        int x,y;
public:
        Point(){x=y=0;}
        Point(int i,int j){x=i;y=j;}
        Point operator+(Point);
        void disp(){ cout<<"("<<x<<","<<y<<")"<<endl;}
};
Point Point::operator+(Point p)
{
        this->x+=p.x;
        this->y+=p.y;
        return *this;
}
void main()
{
        Point p1(2,3),p2(3,4),p3;
        cout<<"pl:";
        p1.disp();
        cout<<"p2:";
        p2.disp();
        p3=p1+p2;
        cout<<"执行 p3=pl+p2 后"<<endl;
        cout<<"pl:",p1.disp();
        cout<<"p2:";
        p2.disp();
```

```
    cout<<"p3:";
    p3.disp();
}
```

四、编程题

1．定义一个计数器类 Counter，对其重载运算符“+”。

2．C++在运行期间不会自动检查数组是否越界，设计一个类来检查数组是否越界。

3．有两个矩阵 a 和 b，均为 2 行 3 列，求两个矩阵之和。重载运算符“+”使之能用于矩阵相加，如：c = a + b。

4．对于下面的类 MyString，要求重载一些运算符后可以计算表达式 a=b+c，其中 a、b、c 都是类 MyString 的对象。请重载相应的运算符并编写测试程序。

```
class MyString{
public:
    MyString(char *s){
    str=new char[strlen(s)+1];
    strcpy(str,s);
    }
    ～MyString(){delete[]str;}
private:
    char *str;
};
```

5．设计人民币类 RMB，其数据成员为分、角、元，定义构造函数和数据成员 yuan、jiao、fen，重载这个类的加法、减法运算符。

6．设计一个日期类 Date，包括年、月、日等私有数据成员。要求有实现日期基本运算的成员函数，如某日期加上天数、某日期减去天数、两日期相差的天数等。

7．定义如下集合类的成员函数，并用数据进行测试。

```
class   Set{
    int * elem;                    //存放集合元素的指针
    int count;                     //存放集合中的元素个数
public:
    Set();
    Set(int S[],int n);
    int find(int x)const;          //判断 x 是否在集合中
    Set operator+(const Set&);     //集合的并集
    Set operator-(const Set&);     //集合的差集
    Set operator*(const Set&);     //集合的交集
    void disp();                   //输出集合元素
};
```