高级语言程序设计II

四川大学计算机学院
四川大学计算机学院
陈良银
cly_6666@sina.com
http://cs.scu.edu.cn/~chenliangyin

星期一、第4大节、N1-A509。

面向对象程序设计语言C++

- 第1章　引论
- 第2章　C++语言与C语言的不同
- 第3章　类类型
- 第4章　运算符重载
- 第5章　派生类
- 第6章　流库
- 第7章　模板
- 第8章　面向对象设计技术（自学）
- 第9章　命名空间和例外处理（自学）

回忆

- 实现了——数据封装
- 类的定义,如何定义.类内,类外?
- 对象创建
- 三种类(一般/无名/空类)
- 成员变量和成员函数;类类型常量和const成员.
- 构造和析构函数
- 命名的自动对象/动态对象/子对象/对象数组/局部静态对象/全局对象/表达式局部对象/

回忆

- 拷贝构造函数
- 类对象的初始化(三种方式)
- 指向对象的指针
- 类类型做参数(三种形式)
- this指针
- 静态成员
- 友元

第四章

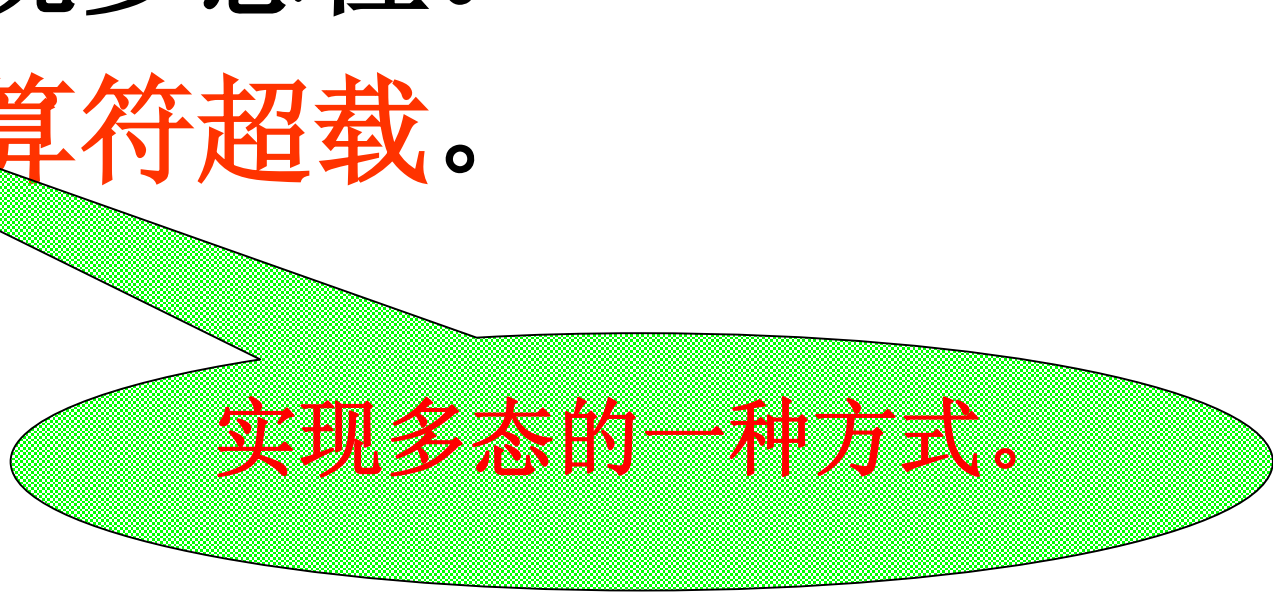
运算符超载

多态-----面向对象的核心概念。

所谓多态，是指一个名字(符号)具有多种含义。

重载可以实现多态性。

有函数和运算符超载。



- 一个类就是用户定义的一个类型。
- 类类型可以作为参数类型，可以作为返回类型；
- 可以定义一个类的单个对象,也可以说明对象数组，甚至还可能有类类型常量。

作用于类的运算符与一般的运算符???

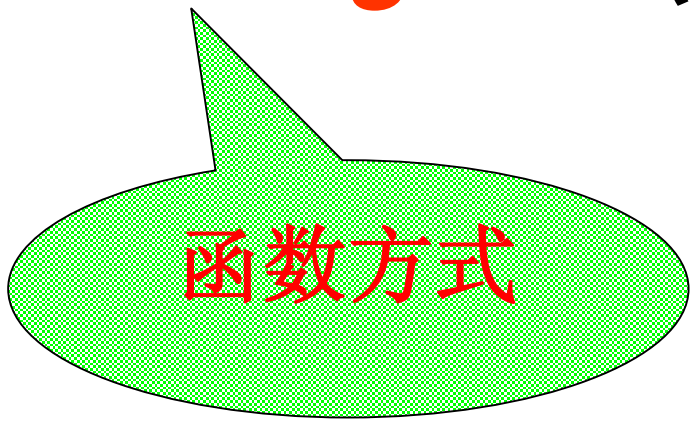
- 预定义运算符表达简单的操作，很直观，但只能对于C++语言定义的类型（的数据）进行操作。
- 在基本数据类型上，系统提供了许多预定义的运算符，它们以一种简洁的方式工作。

重载

- 对于用户定义的类型，某些操作也想使用运算符来表示，则需要对运算符进行超载。

但是，两个串类的对象的合并：

```
class String
{
    public:...
    String string_cat(String);
    ...
};
```

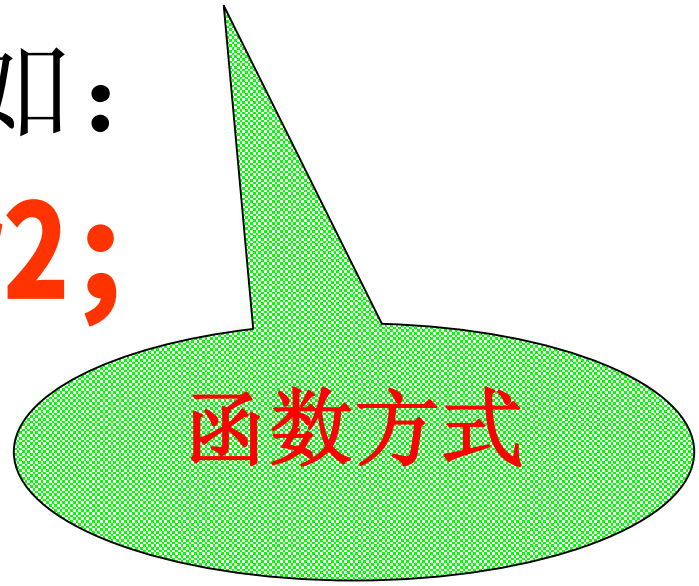


```
String str1,str2;
Str2=str1.string_cat(str2);
```

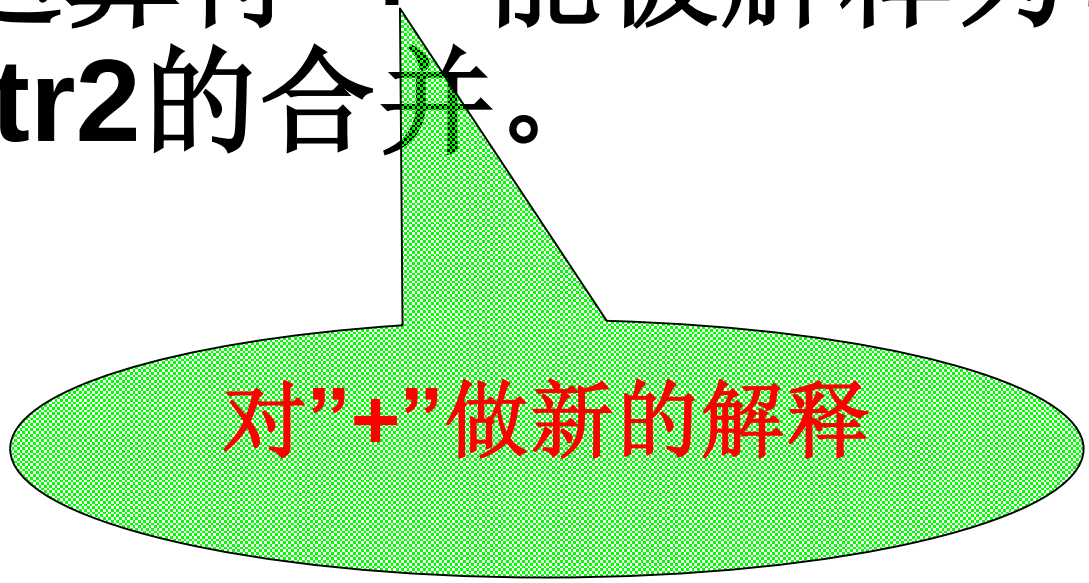
- 表达起来就不如：

```
str2=str1+str2;
```

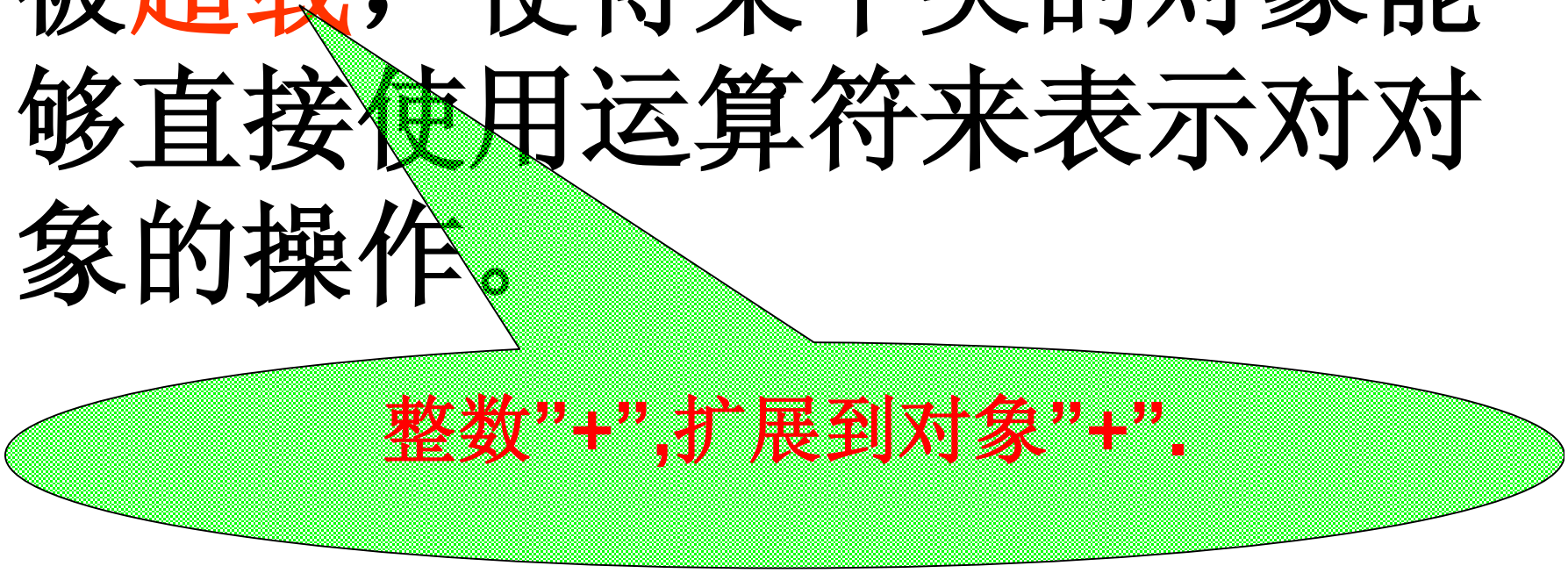
- 那样简洁。



- 为了表达上的方便，希望已预定义的运算符，也可以在特定类的对象上以新的含义进行解释。
- 如在**string**类对象**str1,str2**的环境下，运算符“**+**”能被解释为串**str1**和**str2**的合并。



- 换言之，希望预定义运算符能被超载，使得某个类的对象能够直接使用运算符来表示对对象的操作。

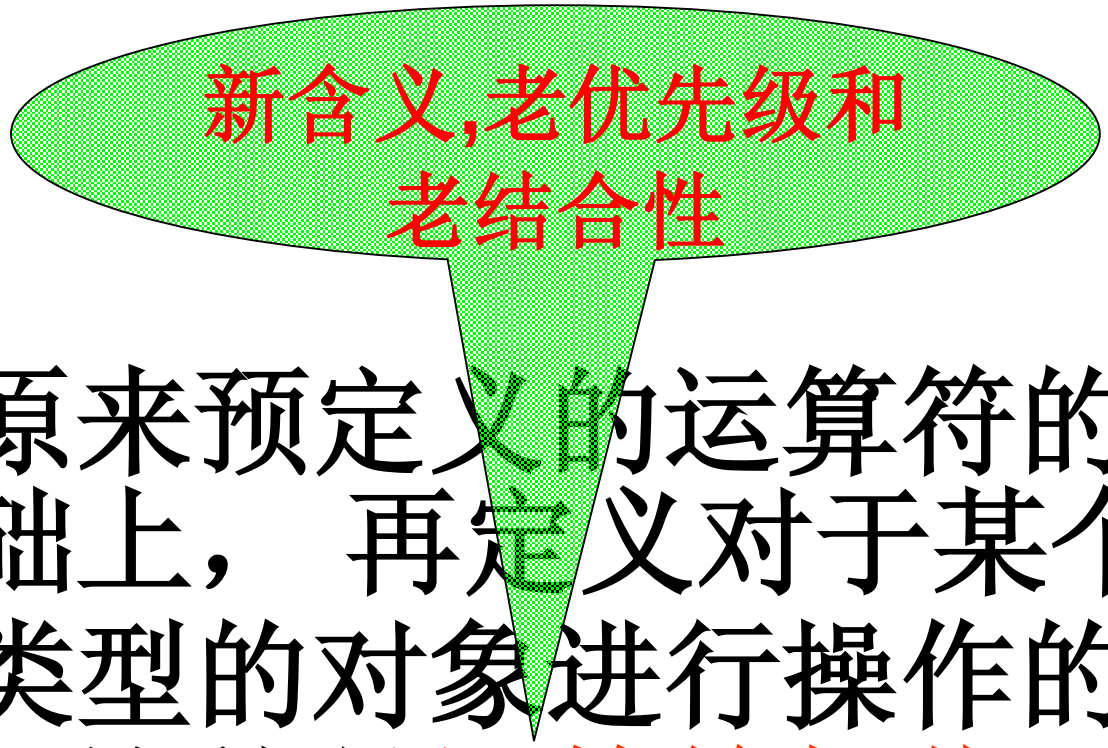


- C语言中，有许多预定义的运算符例如“+”，它可以用于int数据，也可用于float类型数据，虽然使用相同的运算符，但生成的目标代码不相同，这时，“+”运算符具有两种不同的解释(实现代码)。

- 也就是说，像“+”这样的运算符在C语言中已经被超载。不幸的是，C语言仅支持少量有限的运算符超载。

环境变化了.

- C++语言扩充了这个功能，允许预定义运算符由用户在不同的上下文中做出不同的解释。
- 即：如果是类类型的对象使用运算符，则使用的是运算符新的含义；
- 而其他类型的数据使用运算符，使用的是运算符原来的含义。



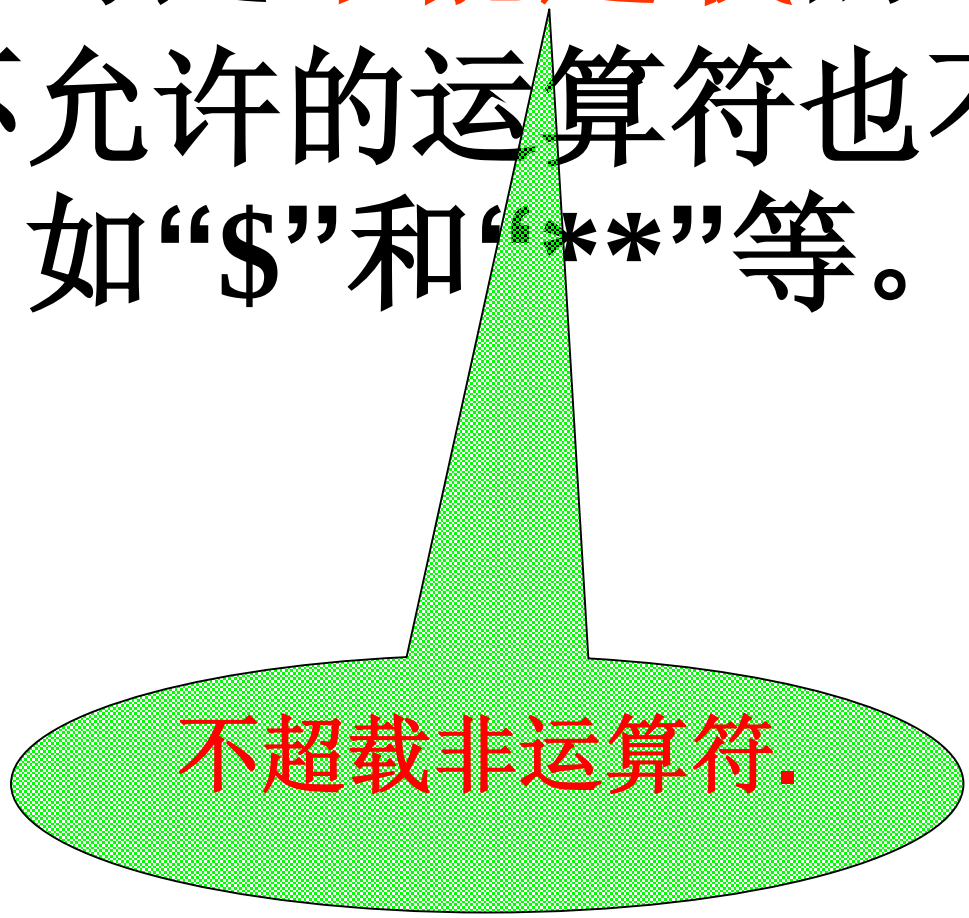
- 在原来预定义的运算符的含义的基础上，再定义对于某个用户定义类型的对象进行操作的新的含义。这就是运算符超载。
- 运算符超载后，优先级和结合性不变。

- 本章介绍：
- 运算符超载的原理和使用；
- new和delete的超载；
- 类类型转换。

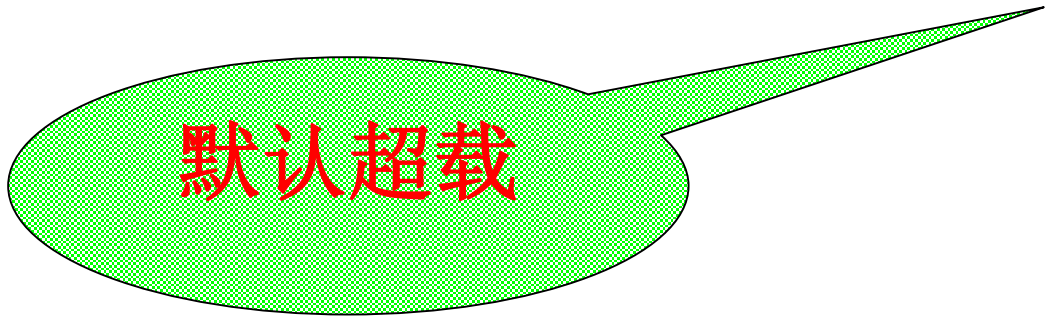
4.1 超载运算符

- 大多数系统预定义的可以通过运算符超载函数定义它们对用户定义类型进行操作的新的含义。只有少数的C++的运算符不能超载:
- :: # ?: . .* *

- 当然，不是运算符的符号，如“；”等是不能超载的，C++语言不允许的运算符也不能超载，如“$”和“**”等。

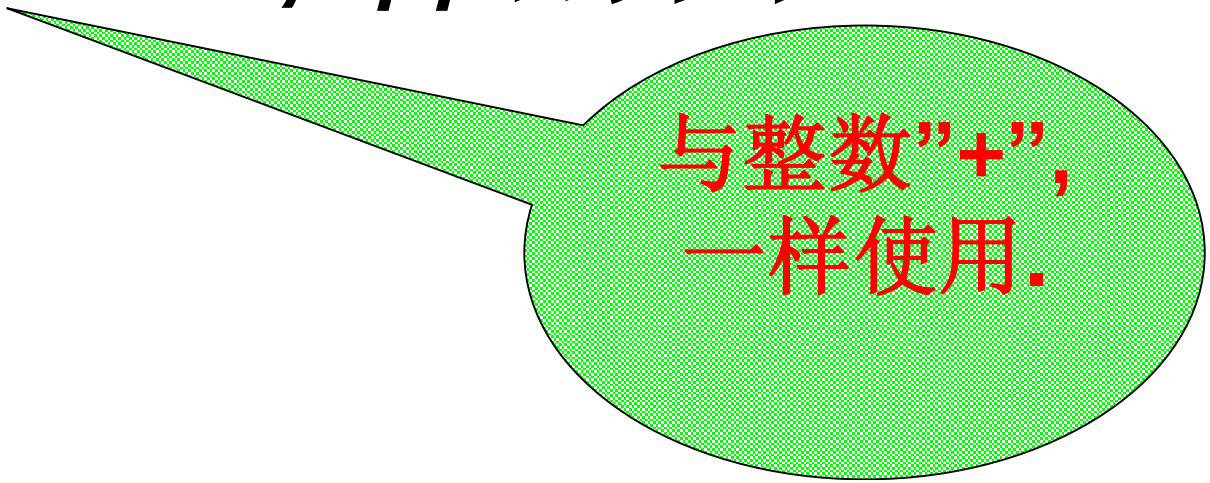


- 超载运算符时，不能改变它们的优先级，不能改变它们的结合性，也不能改变这些运算符所需操作数的数目。
- 对象可以直接使用运算符‘=’和‘&’。



- 超载以后，可以按这些运算符的表达方式使用。
- 例如，在一个**string**类中超载了运算符“**+**”为两个串的合并，则

```
string str1,str2,str3;
str3=str1+str2; //合并串
```



所以重载:
不能改变优先级
不能改变操作数的个数
不能重载没有的符号（自己发明）

单目可以重载
双目可以重载
三目不能重载

运算符重载的能力增强了C++语言的可扩充性

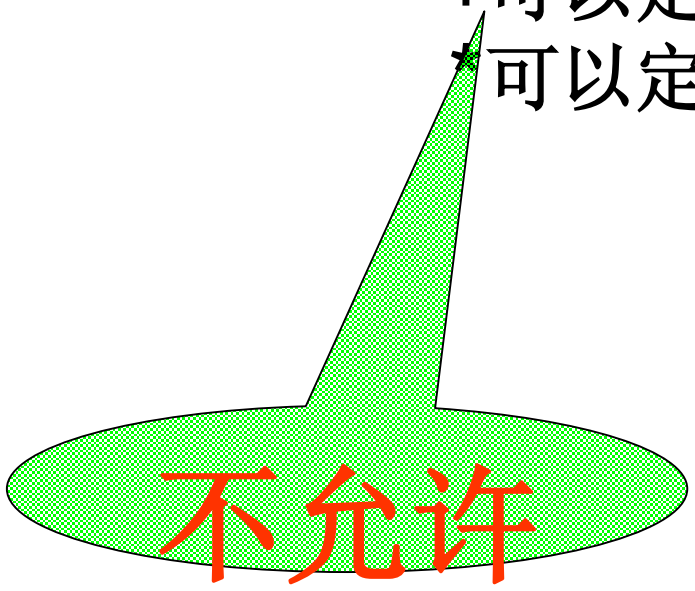
new可以重载
delete可以重载

在C++中，运算符的重载，实际上是一种函数调用的形式

用成员函数重载运算符
用友元函数重载运算符

+可以定义为减运算
*可以定义为除运算

颠倒黑白



对复数的“加法”操作

ComPlus.CPP 不超载不方便.

- add_Com（Com ）实现该操作，使用obj1.add_Com(obj2);
- 表示该函数的调用，不太直观。能不能将2个复数相加表示为：obj1+obj2 呢？

- C++约定，如果一个成员函数的函数名字是特殊的，由关键字operator加上一个运算符构成，
- 如operator+，那么，obj1+obj2就表示该函数的调用，即obj1.operator+(obj2)。

operator关键字

- obj1+obj2称为函数operator+（...）的隐式调用形式；
- obj1.operator+(obj2) 称为函数operator+（...）的显示调用形式；
- 函数operator+（...）称为运算符超载函数。

- 例子:ComPlus1.CPP

4.1.2 运算符超载的语法形式

通过运算符超载函数进行运算符超载。

(1)运算符超载函数是成员函数。

```
type ClassName::operator@(参数表)
{
    //相对于该类而定义的操作
}
```

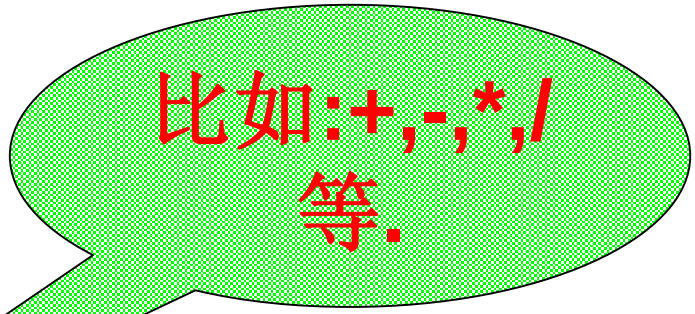
要指明所属类

其中，**type**是返回类型，

“@”是要超载的运算符符号，

ClassName是超载该运算符的类的类名，

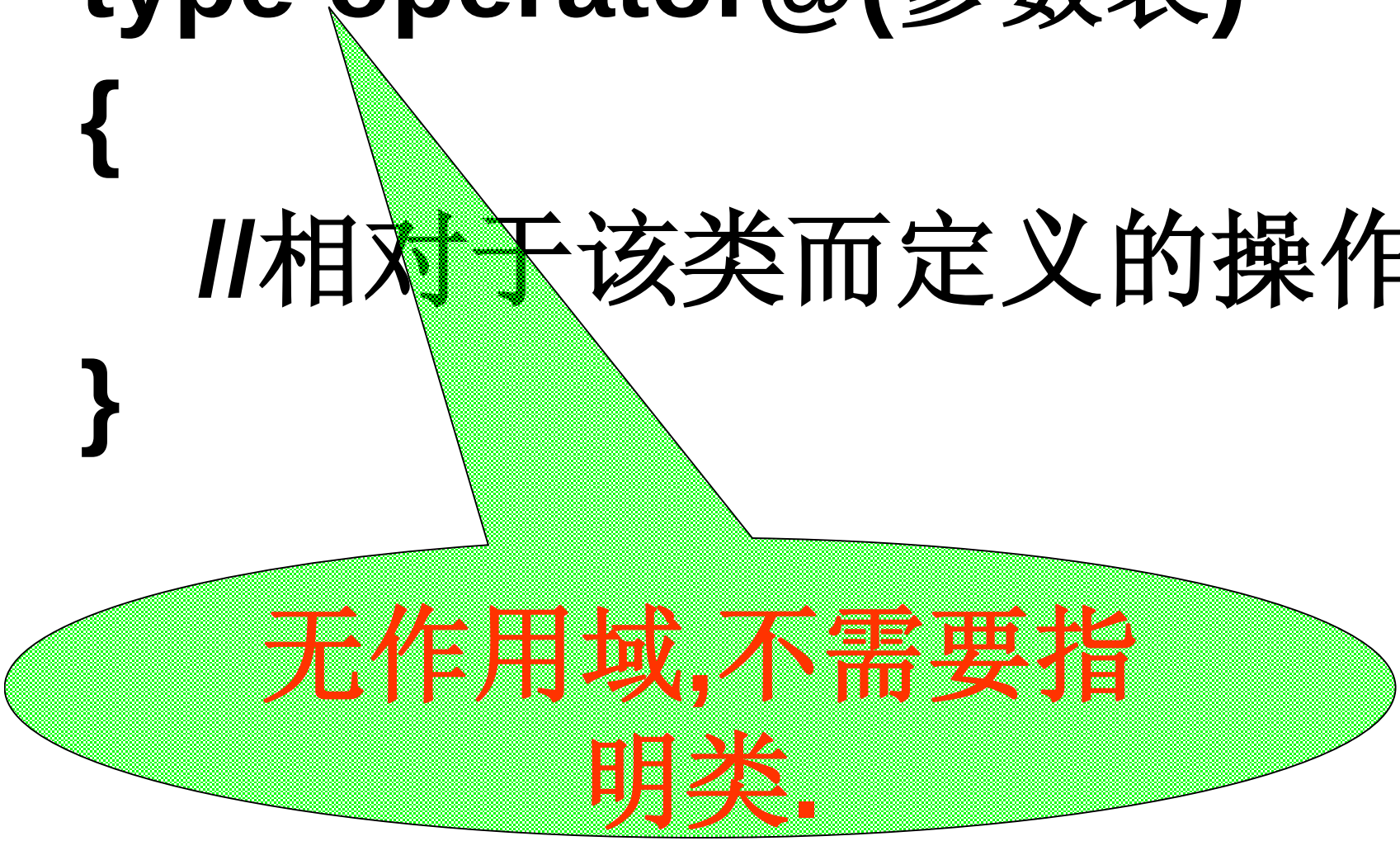
函数名**operator**@是关键字**operator**后跟要超载的运算符符号“@”。



这里用“@”泛指能被超载的运算符。

(2)运算符超载函数是友元函数。

```
type operator@(参数表)
{
    //相对于该类而定义的操作
}
```



运算符分为一元和二元运算符，

一元运算符需要一个操作数，

二元运算符需要两个操作数。

参数表中罗列的就是该运算符所需要的操作数。

运算符函数体对超载的运算符的含义作 出 新 的 解 释.

这里所 解 释 的含义只与超载该运算符的类(即类ClassName)有关。

为何要超载?

当在ClassName类对象的上下文中，该运算符的含义由这个函数体进行 解　释(或称执行)；否则，该运算符具有系统预定义的含义.

因此，当一个运算符被超载时，它所有原先的意义并未失去，只是定义了相对一特定类(这里是ClassName)的一个新运算符。

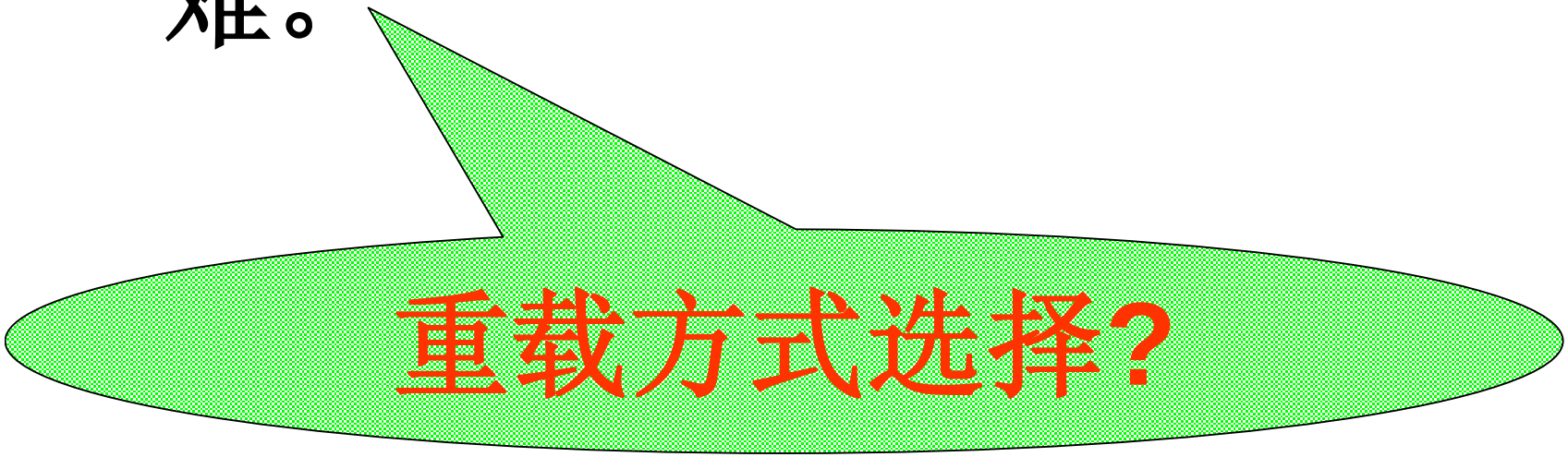
计数器超载运算符

不能够随使重载

CounterOverload.cpp

- 由于能够对旧运算符定义新含义，有可能使编出来的程序几乎不可理解。
- 例如，设想如果在某一个程序中，运算符“+”用来表示减运算，而“*”用来表示除运算，那么该程序的阅读者将面临什么样的问题?

- 因此，运算符超载主要用来模仿运算符的习惯用法，这可能是明智的，如果不能建立运算符的这种习惯用法，应该采用函数调用法，以免造成阅读程序时的困难。



重载方式的选择

（1）若运算符的操作要修改对象的状态，或需要左值运算数（如=，*=，++），选择成员函数

（2）若运算符的操作数（特别是第一个操作数）希望有隐藏类型转换，必须选择友元函数

既要声明、也要定义

一、用成员函数重载运算符

重载运算符的成员函数称为运算符重载函数，有this指针,是一种特殊的成员函数

1.声明格式

```
class class_name
{ type operator@(arg-list); }
```

2.定义格式

```
type class_name::operator@(arg-list){ ......}
```

其中：type为返回类型，@为要重载的运算符

operator为关键字，arg-list为该运算所需要的操作数

若@为一元的，则arg-list为空，当前对象作为单操作数

若@为二元的，则arg-list中有一个操作数，当前对象为@的左操作数，arg-list中的操作数为@的右操作数

	隐式调用	显式调用
一元	aa@或@aa	aa.operator@()

无参数，隐式传递

	隐式调用	显式调用
二元	aa@bb	aa.operator@(bb)

⇒ 一个参数，一隐一显

⇒

二、用友元函数重载运算符

重载运算符的友元函数也称为运算符重载函数

1.声明格式

```
class class_name
{ friend type operator@(arg-list);}
```

2.定义格式

```
type operator@(arg-list){.....}
```

若@为一元的，则arg-list中有一个操作数，作为唯一的操作数

若@为二元的，则arg-list中有二个操作数，作为运算的两个操作数

	隐式调用	显式调用
一元	aa@或@aa	operator@(aa) 一个参数，显式传递
二元	aa@bb	operator@(aa, bb) 二个参数，显式传递

⇒

⇒

注　　意

1.为什么要用友元函数重载运算符

2.=，（），[]，→只能用成员函数重载

3.<<，　>>只能用友元函数重载

4.友元函数重载单目运算符时要特别小心

若单目运算符要改变对象的状态，且用友元函数重载该运算符，则应使用引用参数

4.1.2 一元和二元运算符

- 超载运算符的函数一般定义为类的公有非static成员函数或类的友元函数。
- 虽然也可以使用static成员函数超载运算符，但不能隐式调用，失去了运算符超载函数的意义。

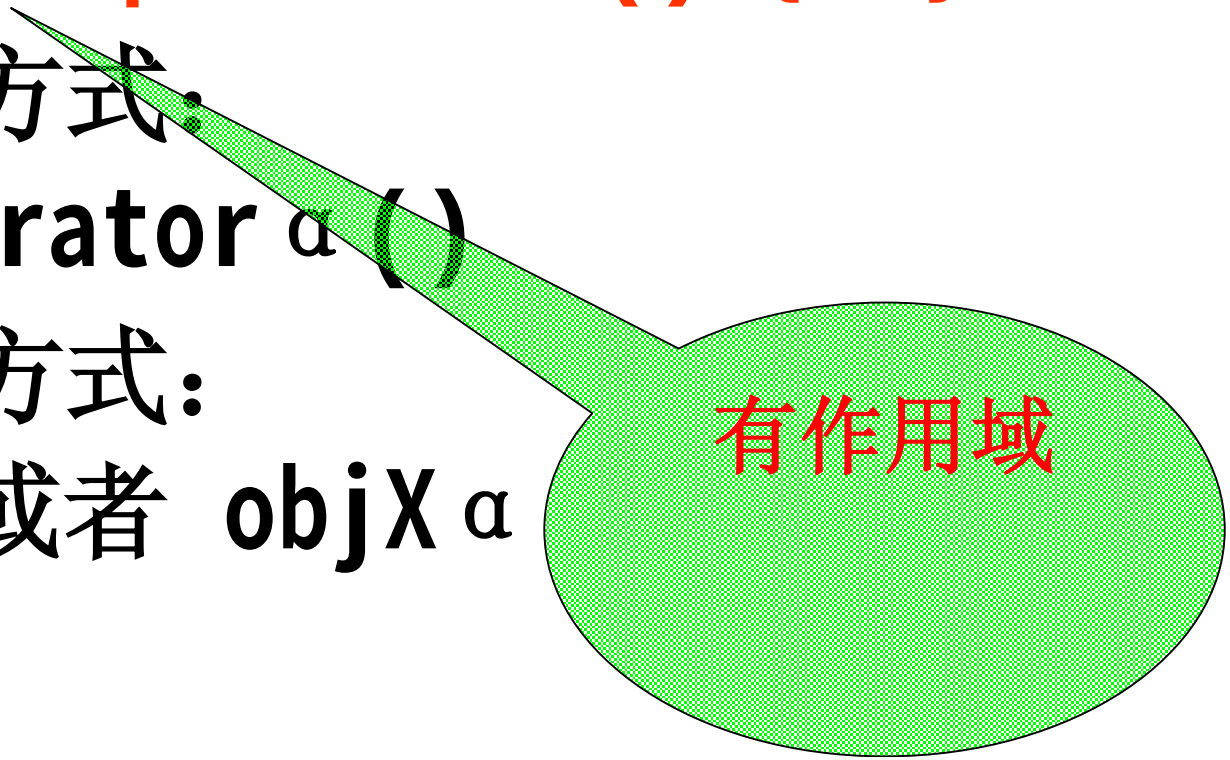
- 如果使用类外的非成员函数（也非友元函数）超载运算符，一般只能对必须包含有公有数据的类进行，或者，要受到某些条件的限制，所以一般不采用。

- 运算符超载时，运算符超载函数一般定义为两种方式，或者作为一个类的成员函数，或者作为友元函数。
- 这两种方式非常类似，但还有许多差别，关键原因在于成员函数具有this指针 ，而友元函数没有this指针。

- 1.一元运算符
- 一元运算符，不论是前缀还是后缀，都需要一个操作数。

对任意一元运算符α

- (1)成员函数超载运算符
- type X::operatorα() {...}
- 显式调用方式:
  objX.operatorα()
- 隐式调用方式:
  αobjX 或者 objXα



- 一元运算符超载函数

  operatorα所需的一个操作数通过this指针隐含地传递。因此，它的参数表为空。

(2)友元函数超载运算符

- type operatorα(X obj){ … }
- 显式调用方式:
- operatorα(objX)
- 隐式调用方式:
- αobjX 或者 objXα

- 一元运算符超载函数 operatorα 所需的一个操作数在参数表中(友元中)。

2.二元运算符

- 二元运算符，需要两个操作数。
- 对任意二元运算符β：

(1)成员函数超载运算符

- type X::operatorβ(X obj){...}
- 显式调用方式：

  objX1.operatorβ(objX2)
- 隐式调用方式：

  objX1βobjX2

- 二元运算符超载函数operator β所需的第一个操作数通过this指针隐含地传递。
- 第二个操作数通过参数提供，因此，它只有一个参数。

(2)友元函数超载运算符

```
type operator β (X obj1,X obj2)
   {   ...   }
```

- 显式调用方式：

  `operator β (objX1,objX2)`

- 隐式调用方式：

  `objX1 β objX2`

- 二元运算符超载函数operator β所需的两个操作数都通过参数提供，因此，它有两个参数。
- 不管是用成员函数还是友元函数超载运算符，该运算符的隐式调用方法是相同的。

- 一元运算符：

  α obj 或者 **obj** α

- 二元运算符：

  objX1 β objX2

- 用成员函数超载运算符和用友元函数超载运算符，它们传递参数的方法不一样，也就导致了它们的实现代码不相同。

4.1.3 用成员函数超载运算符

- 例4-4 创建一个三维坐标系内点的类three_d，含有三维空间中对象的坐标。
- 该程序相对于three_d类对“+”和“=”运算符进行超载。
- ThreedOverLoad.cpp

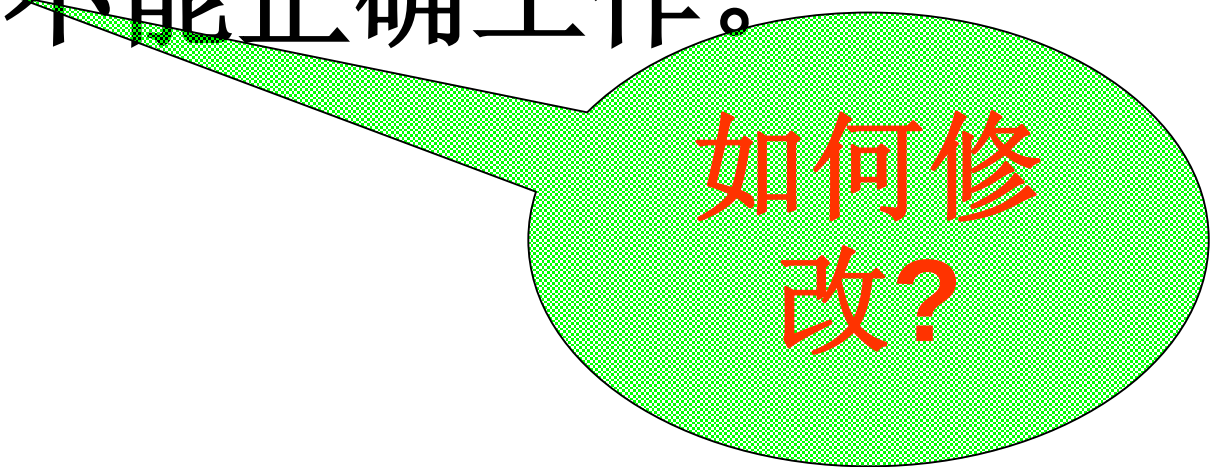
4.1.4 用友元函数超载运算符

- 有些时候，用成员函数超载运算符会碰到麻烦。
- 比如：CompOverLoad.cpp
- 表达式Z+27可被解释为：

  Z.operator+(27)
- Z是一个复数类的对象，系统通过构造函数Comp(int a) 将整型常量27变为Comp类类型常量Comp(27)。因此该表达式能正确工作。

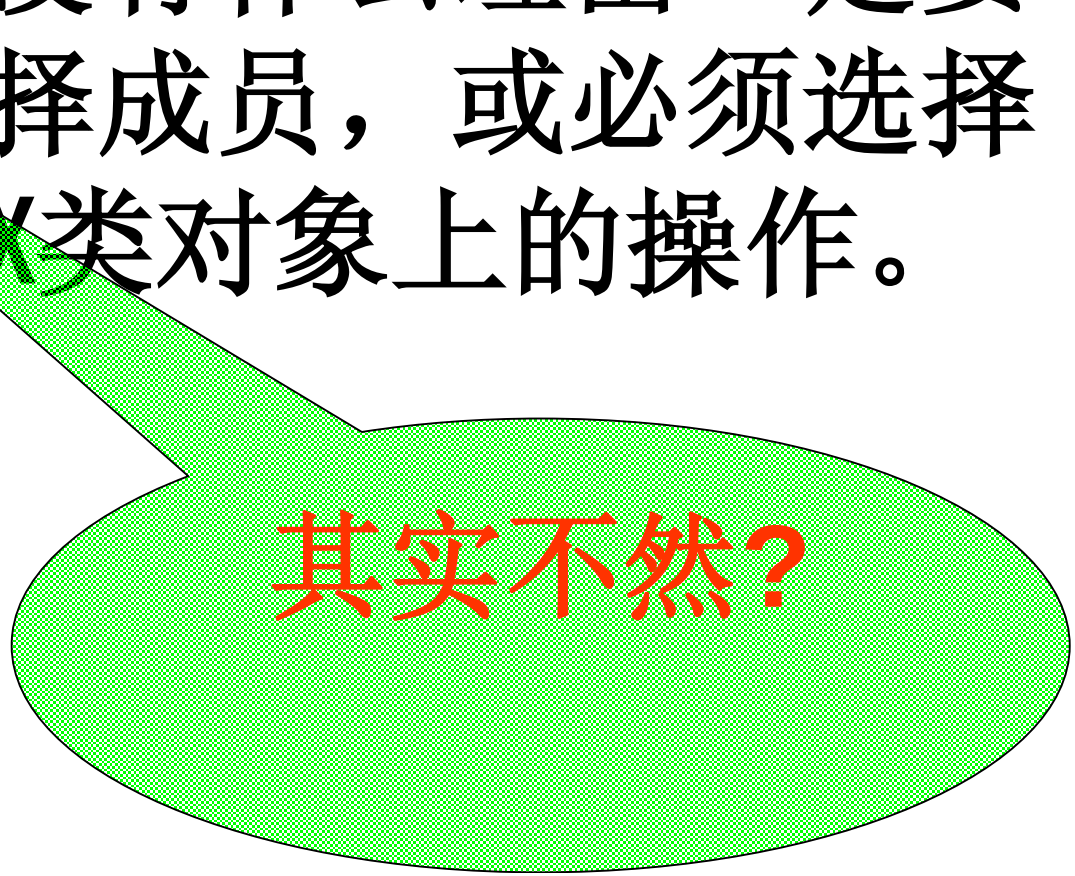
- 但是，表达式 27+Z被解释为
- ```
  27.operator+(Z);
  ```
- 这个式子毫无意义。
- 因为27不是用户定义的对象，该表达式不能工作。
  ```

- 究其原因，在于运算符”+”被超载为成员函数，成员函数仅能被一个“实际对象”所调用 。
- 如果引起成员函数调用的是一个值(这里为27)，则该成员函数无所适从，不能正确工作。



- 在第一个参数需要隐式转换的情形下，用友元函数超载运算符。
- 由于友元函数没有隐含的this指针，因此，用友元超载运算符时，该运算符所需的操作数都必须在参数表中明确地声明，很容易地实现隐式类型转换。
- 修改如下即可：CompOverLoad1.cpp

- 何时用成员函数，何时用友元函数来访问类的私有部分呢?
- 表面上看，没有什么理由一定要用户必须选择成员，或必须选择友元来实现X类对象上的操作。



- 然而，成员函数仅仅能为一个“实际对象”所调用，友元无此限制。
- 因此，若运算符的操作需要修改类对象的状态，则它应该是成员函数，而不应是友元。
- 需要左值操作数的运算符(如=，+=，++)的超载最好用成员函数。

- 相反，如果运算符所需的操作数(尤其是第一个操作数)希望有隐式类型转换，则该运算符超载必须用友元， 而不是成员函数。

- 成员函数能隐含this指针，而友元必须至少带一个显式参数 ，成员函数方式表达较为简洁。
- 另外，友元函数破坏了封装机制，建议尽量使用成员函数实现操作。

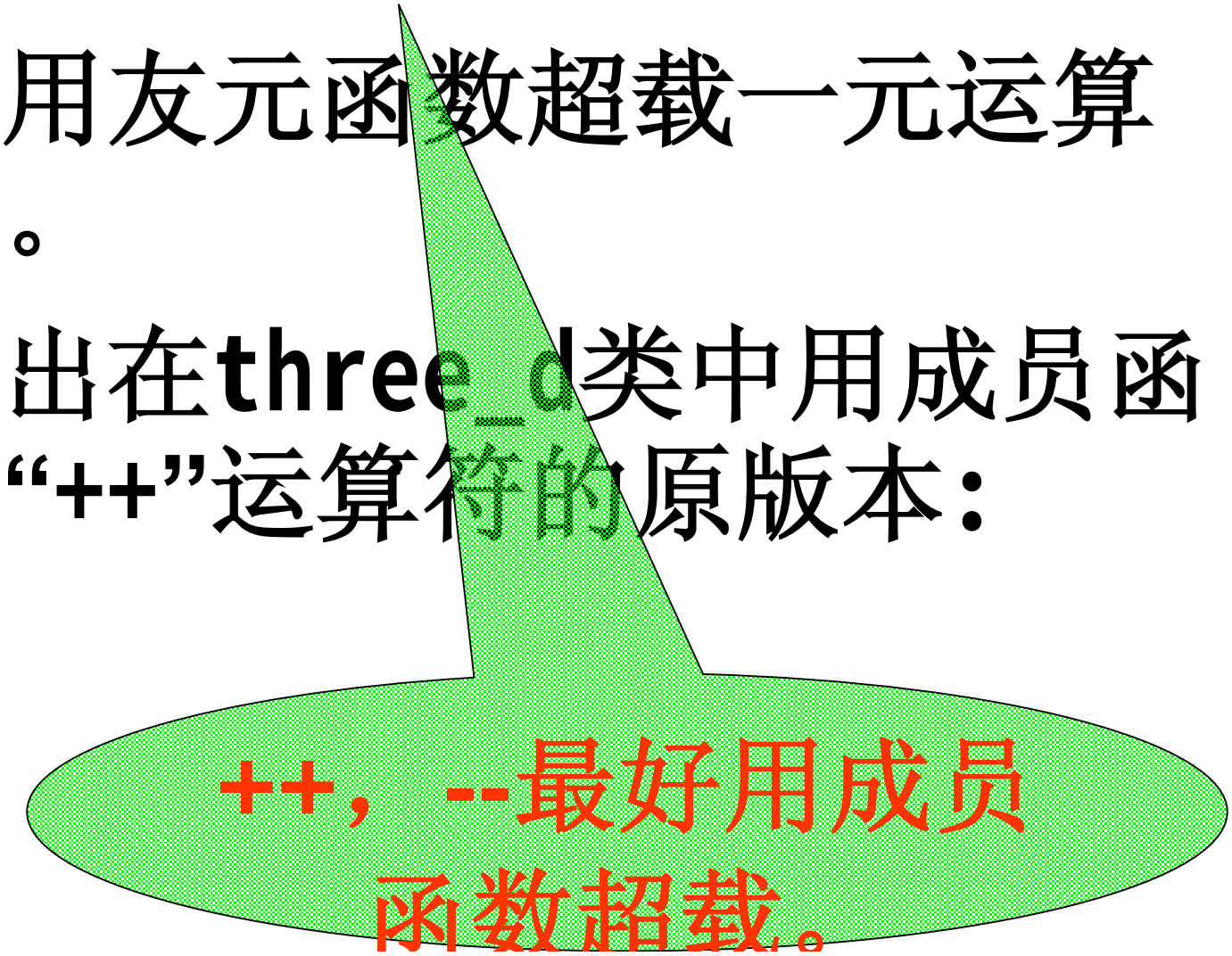
- 注意：
- =，( )，[]，-> 不能使用友元函数进行超载。
- <<和>>只能用友元函数超载.
- 其余的运算符都可以使用友元函数来实现超载。
- 友元函数重载单目运算符时要特别小心 若单目运算符要改变对象的状态，且用 友元函数重载该运算符，则应使用引用 参数

- 比如：FriendOverLoad1.cpp
- 用友元函数超载了“+”运算符，使得

  10+obj 和 obj1+12

- 都正确。

- 用友元函数超载像“++”这样的运算符，会碰到一些问题。
- 下面使用友元函数超载一元运算符“++”。
- 下面给出在three_d类中用成员函数超载“++”运算符的原版本：

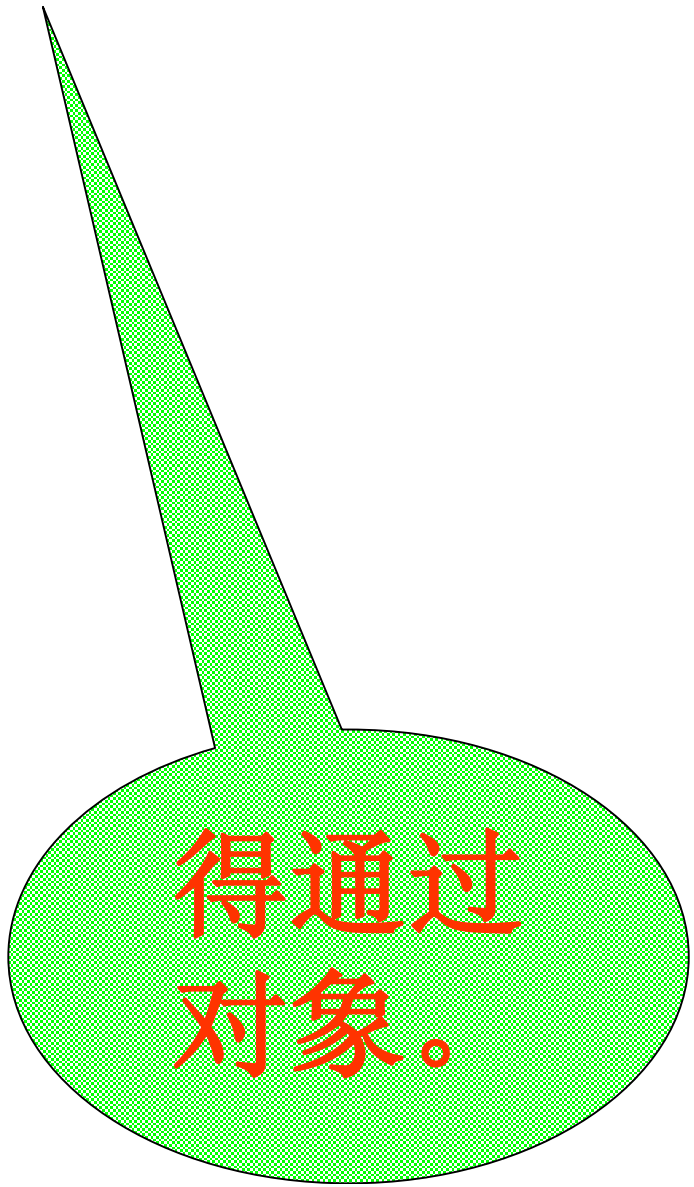


```
td td::operator++( )
{
    ++x;
    ++y;
    ++z;
    return *this;
}
```

- 所有成员函数都有一个**this**指针。
- 在用成员函数超载一元运算符时，所需要的惟一变元通过**this**指针传递，**this**指针指向激活运算符函数的对象。

- 对this所指向的数据的任何改变都会影响到激活运算符函数的对象。
- 与成员函数不同，友元函数没有this指针。

```
td operator++(td opl)
{
    ++opl.x;
    ++opl.y
    ++opl.z;
    return opl;
}
```



- 原因在于C++的函数是传值的，函数体内对opl所有修改都无法传到函数体外。
- 因此，在operator++( )函数中任何内容的改变不会影响实参对象。

- 在用友元超载一元运算符++或--时，采用的方法是使用引用参数。
- 这就避免了上述的二义性 。

```
td opator++(td & opl)
{
    ++opl.x;
    ++opl.y;
    ++opl.z;
    return opl;
}
```

- 可见，如果一个运算符的操作要修改类对象的状态，例如：
- ob++;
- 则需要将对象ob递加，当超载为友元时，应该使用引用参数。

使用友元超载运算符的td程序

- FriendOverLoad2.cpp

- 不管是用成员函数还是友元函数超载运算符，该运算符的隐式调用方法是相同的。
- 一元运算符： α obj或者obj α
- 二元运算符：objX1 β objX2

几个特殊运算符的重载

1．++和--　前缀方式 ++i, --i

后缀方式 i++, i--

(1)**前缀方式** aa.operator++()　　成员函数重载

operator++(X &aa)　　友元函数重载

aa.operator--()　　成员函数重载

operator--(X &aa)　　友元函数重载

(2)**后缀方式** aa.operator++(int)　　成员函数重载

operator++(X &aa, int)　友元函数重载

aa.operator--(int)　　成员函数重载

operator--(X &aa, int)　友元函数重载

隐式调用　a++　　但 a++3;//error

显式调用　a.operator++(0) a.operator++(3);//ok

超载++和--的前缀和后缀方式

- 早期的C++在超载运算符++(或--)时，不能显式地区分是前缀形式还是后缀形式。
- 目前的C++标准中，对此作了一些约定。

- ① 对于前缀方式++obj:
- 成员函数超载

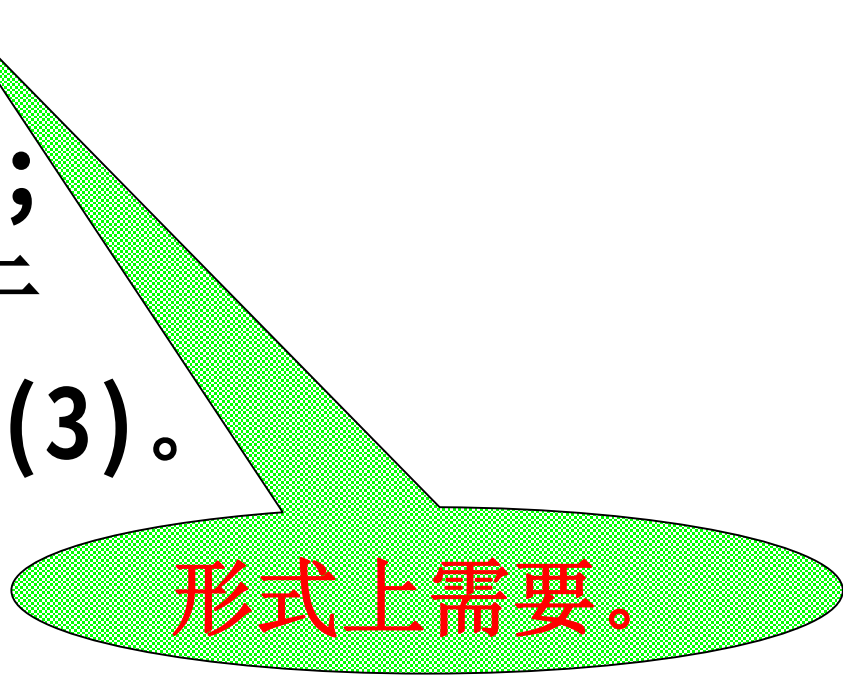
  X X::operator++( );
- 友元函数超载

  X operator++(X&);
- ② 对于后缀方式obj++:
- 成员函数超载
- X X::operator++(int);
- 友元函数超载
- X operator++(X&,int);

- 这时，第二个参数(int)一般设置为0，
- 例如：
- obj++ 等价于
- obj++(0) 或者 obj++=0。

- 区分前缀和后缀（使用成员函数超载）
- PrefixOverLoad2.cpp
- 区分前缀和后缀（使用友元函数超载）
- PrefixOverLoad1.cpp
- 超载运算符“--”也用类似的方法。
- 超载运算符++(或--)后，可以用两种方法来使用：
- 隐式调用和显式调用。

- 对于后缀方式，也可以有非0的实参。
- 如：显式调用
- `obj.operator++(3);`
- 或
- `operator++(obj,3);`
- 是合法的，它们等价于
- `obj++=3;` 或 `obj++(3)`。



- 4.1.6 超载赋值运算符
- 赋值运算符“=”可以被超载，用户可以定义自己需要的超载“=”的运算符超载函数。
- 超载了的运算符函数operator=不能被继承，而且它必须被超载为成员函数。

- 一般超载格式为：
- `X X::operator=(const X & fm)`
- `{`
- `        //复制X的成员`
- `}`

- 当用户在一个类中显式地超载了运算符“=”时，称用户定义了类赋值运算。
- 它将一个X类对象fm逐域拷贝到赋值号左端的类对象中。

- 类的赋值运算符跟其他的运算符略有不同。
- 如果用户没有为一个类超载赋值运算符，编译程序将生成一个缺省的赋值运算符。
- 赋值运算把源对象逐域地拷贝到目的对象。

- 既然拷贝构造函数和赋值运算符都是把一个对象的数据成员拷贝到另一对象，两者都配备看来有点奇怪。
- 它们的区别是，拷贝构造函数要创建一个新对象，而赋值运算符则是改变一个已存在的对象的值。

超载运算符( )和[]

- 运算符“( )”和运算符“[]”不能用友元函数超载，只能采用成员函数超载。

超载函数调用运算符( )

- 对应的运算符超载函数为 operator( )(...)
- 设xobj为类X的一个对象，则
- xobj(arg1,arg2)
- 可被解释为
- xobj.operator( )(arg1,arg2)

超载下标运算符[]

- 相应的运算符超载函数为 operator[](...)
- 设xobj为类X的对象，则
- xobj[arg]
- 解释为
- xobj.operator[](arg)

- 考虑一个数组，数组的大小在定义时初始化，而且其大小在运行时可以改变。
- 超载下标：
  SubindexOverLoad.cpp

超载输入和输出运算符

- 对于预定义类型，用户可以方便地使用运算符“>>”和“<<”进行输入和输出。
- 对于类类型，用户可以超载运算符“>>”和“<<”以满足自己的需要。

- 输出运算符“<<”的运算符的第一个操作数是cout，它实际上是标准类类型ostream的对象的引用。
- 若在程序中，用户自己定义一个ostream的对象的引用s，则s也可以使用运算符“<<”。

例如：

```
#include <iostream.h>
void main( )
{
    int num=10;
    ostream & scout=cout;
    scout<<num<<endl;
}
```

- 对某个用户定义的类类型X，超载输出运算符“<<”，超载函数的函数名为operator<<,而且只能使用友元函数进行超载。
- 为了保证输出运算符“<<”的连用性，超载函数的返回应该为ostream &。
- 超载函数有两个参数：
- ostream & stream和 X obj，
- 分别对应实参：
- cout 和 X_obj。

- 利用友元函数超载了输出运算符“<<”，就可以直接输出一个对象。
- 隐含调用：
- **cout<<obj;**
- 类似地，可以利用友元函数超载输入运算符“>>”，但要注意第二个参数必须是对象的引用 。

three_d 类超载I/O运算符

- overLoad1.CPP

4.2 new和delete的超载

- 自由存储主要涉及C++语言的动态存储分配系统。它允许在用户程序执行过程中动态地为对象分配存储空间。

- C语言用函数malloc( )和free( )动态分配存储空间及动态释放已分配的存储空间。
- C++提供了两个运算符new和delete，它们能完成malloc( )和free( )的功能，更为方便和优越。

- (1)new自动计算要分配存储区的大小(字节数)；
- (2)它自动返回正确的指针类型，不必对返回指针进行类型转换；
- (3)可以用new将分配的存储空间进行初始化；
- (4)可以用new分配一个对象或对象数组的存储空间；
- (5)可以超载与一个类相关的new和delete。

- new和delete典型用法有5种：
- 1)动态分配和释放单个数据的存储区
- 2)动态分配并初始化单个数据存储区
- 3)动态分配数组的存储空间
- 4)为对象动态分配存储区
- 5)为对象数组动态分配存储区

- 可以为任何合法类型（void除外）数据动态分配存储空间;当为对象对象分配时，实际上是通过调用构造函数实现的。
- 语法为：

```
Class_Name * pobj;
pobj=new Class_Name;
                    //构造函数没有参数
...
delete pobj;    //调用析构函数
```

- 或
- ```
  Class_Name * pobj;
  pobj=new Class_Name(...);
    //构造函数有参数
  ...
  delete pobj;   //调用析构函数
  ```

- newObject1.cpp
  ```

- 为动态数组动态分配存储区：

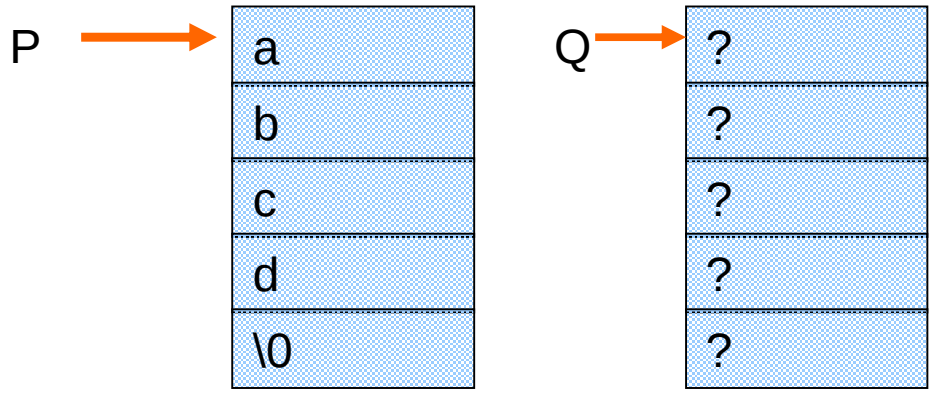
- **newObjectSet.cpp**

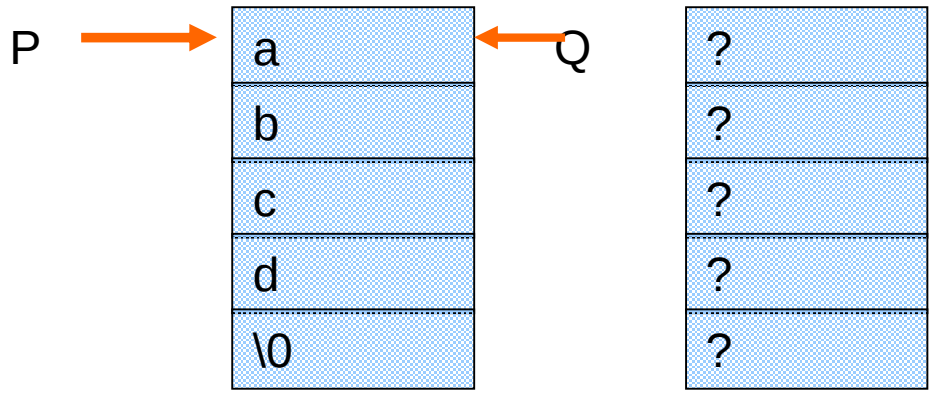
4.2.3 指针悬挂问题

- 当构造函数和析构函数中包含有 new 和 delete 时，可能会产生指针悬挂问题。
- 下面讲述指针悬挂问题的产生及其如何避免。

- 指针悬挂就是指：使用new申请的存储空间无法访问，也无法释放。
- 造成指针悬挂原因是对指向new申请的存储空间的指针变量进行赋值修改。

```
• void main( )
• {
•    char * p,* q;
•    p=new char[5];
•    q=new char[5];
•    strcpy(p, "abcd");
•    q=p;
•    delete[] p;
•    delete[] q;
• }
```

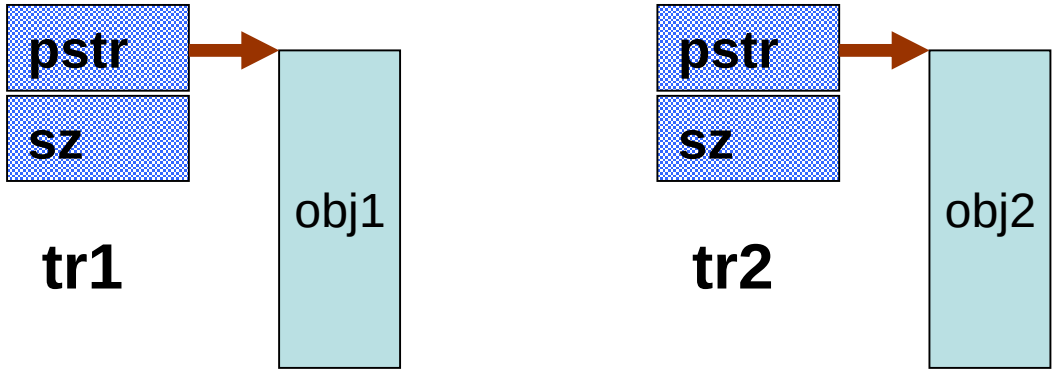


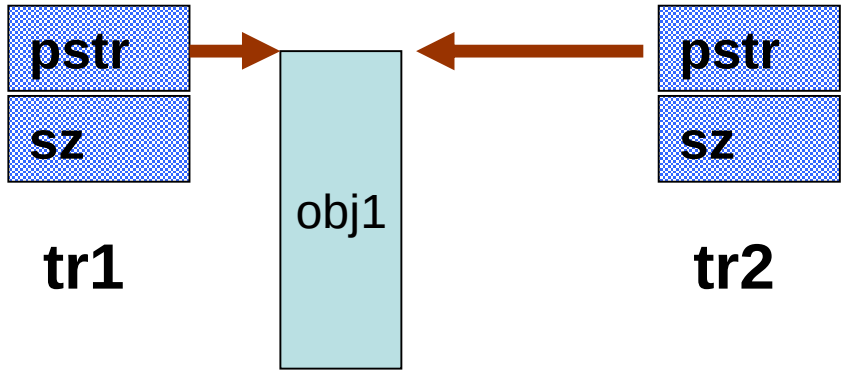


- 语句q=p;将q和p指向相同的地址，而q原来指向的空间再也无法访问，并且也不能够释放。
- 这就是指针悬挂。
- 语句delete [] q还会导致两次释放同一块存储空间(delete p 已经释放过一次)。

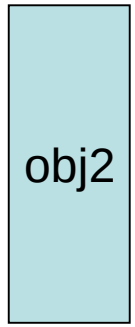
- 对象的拷贝有两种方式：初始化和赋值。
- 将一个对象的数据成员对应地赋值给另一个对象的数据成员；
- 如果一个类包含有指针类型的数据成员，那么，该类的两个对象的拷贝就可能会有问题，
- 因为对象包含的指针成员直接赋值，就可能导致指针悬挂。解决的方法为重新定义拷贝构造函数和超载“=”的函数。

- pointerSuspend.cpp
- 执行该程序，会出现运行错误。
- 缺省的类赋值函数str2=str1是将对象str1的数据成员(即字符指针pstr和整型变量sz)逐个拷贝到str2中，这样str2中原先的值就丢失了，它原先指向的内容被封锁起来而没办法再用，这就产生了指针悬挂问题。





我丢拉!没人能够管我了,呜呜！！



- 更为严重的是，由于str1和str2中的pstr都指向同一内存，
- 当str1和str2都退出其所在的作用域,即main( )结束时，这块内存被释放了两次(调用了两次析构函数)，这是一个非常严重的错误。

- 解决这一问题的方法是定义String类对象的赋值要恰当，需要超载“=”运算符为新的语义；还需要考虑拷贝构造函数的情况。

- 原来缺省的拷贝构造函数为：

```
String:: String(const String & str)
 {
    pstr =str.pstr;
    sz=str.sz ;
 }
```

- 原来缺省的超载”=”函数为：

```
String String ::operator=(
            const String & str)
{
    pstr =str.pstr;
    sz=str.sz ;
}
```

修改为:pointerSuspend1.cpp

集合的实现

- 语言C和C++中没有集合类型，可以定义一个集合类来实现。
- 与集合相关的操作有：
- 加入一个元素到集合中
- 判断一个元素是否在集合中
- 可以取两个集合的交集和并集

删除集合中的一个元素

- 可以扩充集合
- 可以输出集合的所有元素
- 集合可以互相复制
- 可以清空一个集合
- 等。
- Set1.cpp

4.2.4 new和delete的超载

- 超载new和delete是可能的。

（1）new和delete是运算符，可以重载
（2）申请空间可以是基本内存，扩充内存、扩展内存、硬盘

重载类型

局部重载：在某一个类中进行，用成员函数重载运算符函数

全局重载：在任何类外进行，用普通函数重载运算符函数

new	可重载	malloc()	不可重载
delete		free()	

```
void *operator new(size-t size, ......)
   {  // 完成分配工作
      return pointer-to-memory;
   }
   void operator delete(void *p, ......)
  {  // 释放由P指向的存贮空间
  }
```

（1）对operator new()来说，它的第一个参数必须是size-t类型，这是一个在标准头文件<stddef.h>中定义的与实现有关的整型类型，必须返回void *类型

（2）对operator delete来说，它的第一个参数必须是void *类型，第二个参数的类型是size-t（可选，删除对象的大小），必须返回void类型

局部重载

（1）类x的x::operator new()
类x的x::operator delete()
它们不是针对x类的对象操作的是静态成员函 数（无论是否有static）

（2）new在构造函数之前调用，分配构造函数建立X类对象的内存
delete在析构函数之后调用，释放析构函数刚刚破坏的X类对象的内存

（3）一个类只能声明一个operator delete()，故operator delete()不能进行函数重载
一个类可以声明多个operator new()，故operator new()可以进行函数重载

（4）重载的new()和delete()是可以继承的

- 重载的原因是，有时希望使用某种特殊的动态内存分配方法 。

- 例如，可能希望有一些分配子程序，它们能在堆(heap)已耗尽时，自动开始把一个磁盘文件当作虚存储使用，或者用户希望控制某一片存储空间的分配等等。
- 不管是什么原因，超载 new、delete是件简单的事。

- 超载函数new和delete的框架：
- void * operator new(size_t s)
- {
- return malloc(s);
- // 完成分配工作
- }

```
void operator delete(void *p)
{
    free(p);
    // 释放由p指向的存储空间
}
```

- 超载new和delete与超载一般的运算符的方法是一样的。
- 其中类型size_t由C++定义为能容纳可分配的单一的存储块的最大值。它往往是个unsigned的整数类型。参数s给出待创建对象的大小。

- 超载new函数必须返回一个指向分配存储空间的指针，若分配出错，返回零。
- delete函数接收一个指向存储区的指针，并将其释放。
- 超载new、delete有局部超载和全局超载两种。

- 注意：虽然超载new的函数有参数要求，但调用该函数，不必给实参，系统自动计算出所需要分配存空间的大小。

1 局部超载new与delete

- 要超载一个与类相关的new和delete 函数，只需使超载运算符函数成为该类的成员函数。
- 可以使用成员函数或友元函数进行。

- 使用new分配某个超载了new的类的对象空间时，先调用new的超载函数，再调用该类的构造函数，如果该类的构造函数有参数要求，还必须给出对应的实参。

- 使用delete释放某个超载了delete的类的对象空间时，先调用类的析构函数，再调用超载delete的函数。

- 与一般的运算符超载类似，当new与delete相对于特定的类进行超载时，超载的运算符new和delete仅用于这一特定的类，而在其他数据类型上还使用new与delete 的原始版本。

- 换句话说，当遇到new或delete时，编译程序首先检查正在分配(或删除)的对象所属的类是否超载了new或delete，如果是，就使用这个超载版本。
- 否则，C++将使用原来定义的new与delete。
- reloadNew.cpp

2.全局超载new和delete

- 可以在任何类说明之外超载new与delete，使它们成为全局的。
- 当new与delete被全局超载时，C++原来的new与delete被忽略，并且超载的运算符用于所有的分配要求。

- 超载new、delete的功能为用户提供了使用存储区很大的灵活性，但全局超载有时会带来某些不可预知的问题，建议尽量少使用全局超载。
- reloadNewG.cpp

4.3 类 型 转 换

- 类型转换是将一种类型的值转换为另一种类型的值。
- 标准类型除class、struct和union类型（即所有的类类型）外的其他所有类型。
- 由构造函数和转换函数说明。

- C++语言允许的类型转换有4种：
- 标准类型->标准类型
- 标准类型->类类型
- 类类型->标准类型
- 类类型->类类型

标准类型的转换

- C++提供了两种类型转换：

  隐式类型转换（混合运算、表达式赋值给变量、实参向形参传值、函数返回结果）

  显式类型转换（强制法、函数法）

4.3.1 标准类型转换为类类型

- 可以通过自定义的超载赋值号“=”的函数和构造函数实现转换：
- 标准类型->类类型；
- 它们都需要有标准类型的参数。
- 具有标准类型参数的构造函数说明了一种从参数类型到该类类型的转换。
- ClassInvert.cpp

构造函数放在私有段的类的对象

- 当程序员想保证不发生不希望的隐式转换时，可以：
- 将构造函数放在私有段
- PrivateConstrutor.cpp

- 构造函数vector(int s)放在私有段，并在公有段设置一个static函数或设置类外的一个友元函数来调用这个构造函数。
- 当出现语句v1=10时，它需要用构造函数隐式地转换，但构造函数是私有的(避免了不期望的隐式转换)。

4.3.2 类类型转换函数

- 带参数的构造函数可以进行类型转换，但转换功能很受限制。
- 比如：将一个类转换为基本类型,需要引入一种特殊的成员函数：类型转换函数，它在类对象之间提供一种类似显式类型转换的机制。

类型转换函数语法:

```
Class_Name::operator type( )
{
    …
    return (type类型的实例);
}
```

- 类型转换函数没有参数，没有返回类型，但这个函数体内必须有一条返回语句，返回一个type类型的实例。
- 类型转换函数不能被超载，因为它没有参数。

- 类型转换函数的功能是将Class_Name类型的对象转换为类型为type的实例。
- type可以是一个预定义类型，也可以是一个用户定义的类型。
- 类型转换函数只能定义为一个类的成员函数，而不能定义为类的友元。

类类型转换为整型

```
class INT
{    int num;
   public:
     INT(int a=0){num=a;}
     operator int( )
     {return num;}
     …
};
```

```
void main( )
{
    INT obj(12);
    int anint=int(obj);
    anint=(int)obj;
    anint=obj;
    //3次anint=obj.operator int( );
}
```

- 如果某个类有operator type( )，则该类的对象可以直接作为type类型的数据使用。会进行隐式类型转换。

- 类型转换函数有两种使用方式：
- 隐式使用 `anint=obj;`
- 显式使用 `anint=obj.operator int( );`
- 一般用隐式方式，
- 当需要明确指出用哪一个类型转换函数的场合，才使用显式方式。

- int anint=obj;
- 用类型转换函数进行转换
- INT obj=anint;
- 用构造函数进行转换。

- 问题：如果一个类既有用于转换的构造函数，又拥有类型转换函数
- `INT(int);`
- `operator int( );`
- 语句obj=obj+anint;该如何解释?

- 可以将表达式a+i解释为
- `obj.operator int( )+anint`
- 也可以这样解释
- `obj+INT(int)`
- 存在二义性，用户在这种情况下必须显式地使用类型转换函数。

- INT obj1=anint;
- obj=obj+obj1; //两个对象加
- 或
- obj=(int)obj+anint;//两个整数加
- 用户定义的类型转换函数只有在它们无二义性时才能隐式地使用。

- 下面看看二义性的处理：

- 定义一个类INT，它可以处理32位的整数;同时定义另一个类real，它可以处理32位的实数。
- SecondMeaning.cpp

- **integer i1;**
- **real r1;**

- **i1+r1**

使“+”运算产生了二义性，

- 编译器将不知道是使用
- **operator+(integer,integer)**
- 还是使用
- **operator+(real,real)**

- 因此“+”有二义性，必须使用显式的类型转换方式来消除二义性。

- 例如：
- real(i1)+r1;
- i1+(integer)r1;

- **i1-r3;**
- 因为“-”运算符被定义为一个成员函数，它只能被解释为
- **i1.operator-(r1)**
- 所以“-”不具有二义性。

- 将上述例子改为友元类的情况。
- **SecondMeaningF.CPP**

- 以下：
- 实现一个Point类（数据成员为一个点在两维直角坐标系内的坐标）；
- 实现一个Vector类（数据成员为一个点在两维极坐标系内的坐标）；
- 要求两个类的对象能互相赋值。
- PointVector.cpp

- 关于用户定义类型转换，还想再强调说明两点：
- (1)编译器在进行类型转换时，总试图使用用户定义的类型转换函数进行类型转换，
- 如果这样不能成功，则使用标准的类型转换机制。
- 如果都不成功，则转换失败。

- **(2)**在类型转换具有二义性的情况下，必须使用显式类型转换。即
- **(type) obj** 或 **type (obj)**
- 将对象**obj**转换为具有**type**类型的一个对象。

- 总之：类类型的对象，需要使用某个运算符，可以有两种方法：
- 超载该运算符；
- 将类类型转换为可以使用该运算符的类型。