# 第5章　多态性和虚函数

面向对象编程语言C++最重要的概念之一是多态性及其应用。多态性是指C++的代码可以根据运行情况不同执行不同的操作。在实际程序中，对象之间相互作用的方法可以任意组合，即便是那些简单的类层次结构，也会产生数量巨大的组合方式。因此常常必须依靠对象的多态性来开发中、大型软件。





## 5.1 C++的多态性

派生一个类的原因常常不仅是为了新添加一些描述新事例的变量和操作函数，而是为了重新定义基类的成员函数。当基类的成员函数在派生类中重新定义时，其结果是使对象呈现多态性。因此不是整个类都具有多态性，而是只有类成员函数具有多态性。

这种实现方式与自然语言中对动词的使用很类似。此时动词等价于C++的成员函数。例如一个对象若用“它”来表示，则在现实生活中可以这样使用：“清洗它，移动它，分解它，修理它”。这些动词仅代表了一般性动作，因为不知道发生在哪种对象上。

例如，移动铅笔所需的操作完全不同于移动机床所需的操作，尽管这两个概念是相似的，只有知道“move移动操作”作用的对象，才能将它与一系列特殊操作联系起来。显然对于不同的对象，“move移动操作“的具体内容却大不一样。





例如，假设我们要把draw()函数添加到Point和Circle类中，Point的draw()函数只是把一个点绘制在屏幕上，而Circle的draw()函数是以圆的形式画一连串点.

```
Point p;
Circle c;

p.draw(10, 10);
c.draw(10, 10, 4);
```

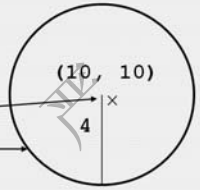


图5.1　成员函数draw()的多态性





由于Circle类是Point类的派生类，所以Point和Circle类的对象p和c要进行相同的操作时，需调用各自成员函数draw()，实现“画图“的操作，其结果是画出不同形状的图，这就是这类对象被称为多态性的原因。

由此可知，从基类Point继承而得派生类Circle，不仅能给派生类添加新的成员(数据成员和成员函数)，而且更重要的是可以重新定义基类中的成员函数draw()，即在基类和派生类中有同名成员函数。其结果是使得它们的对象呈现出多态性。

C++的多态性表现了它为编程者提供了运算符重载，函数名重载和虚函数的运行机制。

5.2运算符重载

操作符重载并不是新概念，自然语言中已使用了几千年，例如一般的加法操作可以认识。两数相加，将桔子和苹果加起来，给故事加上悬念。加法可用于抽象数据相加，也可用于可数的但相互没有关系的对象，如苹果和桔子相加，也可用于抽象事物，例如悬念，因此加法概念很直观，可表示数目增多，性质加强，但若不与上下文相联系，加法的概念相当模糊。

C++的运算符也类似于这种情况，C++语言原来定义的运算符是用来执行对基本数据类型的运算操作，但编程者定义了新的类型后，对已有的运算符必须赋予新的运算操作含义，使得它们适用于用户定义的类型。即C++语言本身提供的标准运算符可以重新在类中定义，使标准运算可作用于新定义的类对象，从而使运算操作表现得自然而又合乎常规。





```
#include <iostream>
using namespace std;
class Complex{
    double  real, imag;
public:
Complex( ) { real = imag = 0; }
//无参数构造函数
Complex(double r, double i)
{ real = r; imag = i; }//一般构造函数
void operator = (const Complex &c);   //重载赋值运算符
Complex operator +(const Complex & c);//重载加法运算符
Complex operator -(const Complex & c);//重载减法运算符
friend ostream& operator<<(ostream&,const Complex&)
   //重载<<运算符
};
```





```
void Complex::operator = (const
                          Complex &c)
{  static int i = 0;
 real = c.real ; imag = c.imag;
 cout << "(" << ++i << ") operator =( )
          called !";
}
inline Complex Complex::operator +
                        (const Complex &c)
{ return Complex(real + c.real ,
                    imag + c.imag);     }
```

```
inline Complex Complex::operator +(const
                                   Complex &c)
{  Complex  result;
   result.real = real + c.real;
   result.imag = imag + c.imag;
   return result;
}
```





```
inline Complex Complex::operator -
                        (const Complex &c)
{  return Complex(real - c.real ,
                    imag - c.imag);}

ostream& operator<<(ostream &output,  const Complex &t)
{
      if(t.imag>0)
            output<<t.real<<"+"<<t.imag<<"i"<<endl;
      else if(t.imag==0)
            output<<t.real<<endl;
      else if(t.imag<0)
            output<<t.real<<t.imag<<"i"<<endl;
      return output;
}
```

```
void main( )
{
 Complex a(2.0 , 3.0), b(4.0,-2.0), c;

 c = a + b;
 cout << " a + b = "<<c;

 c = a - b;
 cout << " a - b = "<<c;
}
```

如果要实现 c = a + 3; 需要什么条件？





该程序的输出结果：

(1)operator =( ) called ! a + b = 6+1i

(2)operator =( ) called ! a - b = -2+5i

这时两个复数对象a和b相加就可以像整数加法一样写成为”a+b;”，即赋予运算符’+’新的运算操作含义，可实现复数的加法运算。

显然要解决这一问题，必须在Complex类中，对标准运算符“+”重新定义，使它可以作用于复数对象，实现复数的加法运算，同理，如果编程者重新定义’-’、’*’、’/’，使它能作用于复数对象，按复数对象类所定义的规则完成复数的操作。





1.运算符函数的定义

运算符重载是由定义运算符函数来实现的，运算符函数名由关键字operator和重载的运算符组成记为operator <op>，C++语言的绝大多数标准运算符都能重载。OP代表+，-，*，/，%，^，&，|，~，！，=，...，++，--，...，[]，()，->，new，delete等。

定义运算符函数的关键是如何理解运算符表达式，并怎样转换成运算符函数调用的形式。这种转换取决于运算符是单目的，还是双目的，且运算符函数可以定义为类的成员函数，或者友元(非成员)函数。

(1) 运算符函数重载为类的成员函数

此时，由于成员函数隐含有该类对象的this指针作为隐含的参数传递给运算符函数。所以，对于单目运算符不带参数，对双目运算符仅带一个参数。





```
<返回类型> <类名>::operator <op>
                              (<类型>  参数)
{                  <函数体>                  }
```

例如：

```
Complex operator +(const Complex &c);
```

转换 this -> operator +(b);

```
c = a + b;   a.operator +(b);
```

剖析上式，运算表达式的第一个运算量复数对象a为隐含对象，通过该对象a调用运算符函数operator +(Complex & c)。表达式中的第二个运算量为显式对象b，赋给了参数Complex &c，在operator+( )函数体内将隐含对象a和形参b的值相加的结果作为该函数的返回值（右值），在main（）里赋给了c（左值）。

(2)运算符函数重载为友元函数：由于友元函数没有this指针，因此对于单目运算符友元函数必须带一个参数，对双目运算符应带两个参数，其格式为：

```
friend  <返回类型>  operator <op>
                (<类型> 参数1，<类型> 参数2)
{                   <函数体>                    }
```

例如：

```
friend Complex operator +
(const Complex &c1, const Complex &c2)
{ return Complex(c1.real + c2.real ,
                 c1.imag + c2.imag);
}
```

若运算符表达式为：c = a + b; 则编译系统解释为调用运算符函数 operator +(a, b); 的语句，求得复数对象a和b相加之和，然后赋给c。综上所述，如表5.1和5.2所示，将运算符表达式怎样转换成运算符函数调用的形式，其中的X和Y为对象名。





表5.1 双目运算符函数

| 成员函数 | 表达式 | 运算符函数调用 |
|---|---|---|
| 是 | X op Y | X.operator op(Y) |
| 否 | X op Y | operator op(X,Y) |

从表5.1可知，对于重载为成员函数的情况，运算符表达式的第一个运算量始终转换成对象，通过该对象调用运算符函数

表5.2 单目运算符函数

| 成员函数 | 表达式 | 运算符函数调用 |
|---|---|---|
| 是 | op X | X.operator op( ) |
| 是 | X op | X.operator op(int) |
| 否 | op X | operator op(X) |
| 否 | X op | operator op(X,int) |





运算符重载的事项：

（1）通常单目运算符最好重载为成员函数；而双目运算符最好重载为友元函数，这是因为有些情况不能重载为成员函数，例如有一个复数对象c，若程序中写有表达式：

6.8 + c，假如复数类中将’+’重载为成员函数，则编译系统解释为：6.8.operator +(c)，这显然是错误的。若将’+’重载为友元函数，则编译系统解释为：operator +(Complex(6.8),c)，从而可实现表达式6.8 + c的运算，这时Complex类中一定要有一个类型转换构造函数 Complex(double r){ real = r; imag = 0; }

然而双目运算符中只有赋值运算符重载为成员函数为好外，其它都重载为友元函数。





(2)不能改变C++中运算符的优先级和运算量的个数，例如赋值号是双目运算符，因此不能将它重新定义成单目运算符函数，也不能改变优先级，即表达式x =a + b;永远表示求出a+b的和赋给x，而不能定义成：  (x = a) + b;

(3)若运算符是一个非成员函数，其参数中至少有一个必须是类的对象。例如，不能重新定义’+’运算符去连接两个字符串：

```
friend char * operator +(char * a,
                         char * b)
{          ...             }
```

为了防止编译混乱，该条规则禁止编程者去修改C++中基本数据类型的运算符操作。

(4) 不能将两个运算符合并去建立一个新的运算符。例如可以直接重定义运算符函数operator +=()。但是有人认为，如果已经定义了operator +()运算符函数，再加上一个operator =()运算符函数把它们合并，以实现表达式a += b;的操作，这显然是错误的。

(5) C++ 本身并不知道运算符重载表示什么操作含义，它是编程者自行定义的，因此也没有相应的检查措施，只有靠用户自己检查，应用时应慎重选择合适的运算符。





(6) 下列标准运算符不能重载，避免与C++语法相矛盾：

- ? : 三项运算符
- . 取成员运算符
- :: 作用域运算符
- * 取内容运算符

(7) 应保持运算符原来的操作含义，例如不要将“+”运算符定义成两值相减。将按位异或运算定义成指数运算等。





### 3.重载赋值运算符(operator =())

可重载的最重要的运算符是赋值运算符，它是双目运算符，如a = b;编译系统解释为a.operator =(b);其运算符函数有如下限制。

(1) 如前所述，operator =()必须重载为非静态成员函数，可保证重载的operator =()函数所控制的赋值目标运算量(赋值号左边的运算量)，始终是该类的一个对象，这与赋值表达式的语义是一致的。

(2) 为了实现多重赋值，将赋值运算符函数返回值定义为目标对象的类型，即返回到目标对象。如例5.1复数类赋值运算符函数它只能实现一次赋值，例如：





```
void Complex::operator = (const Complex c)
{                ...
   real = c.real ; imag = c.imag;
                 ...
}
```

它只能实现一次赋值，例如：

```
void main( )
{ Complex    a(4.0,8.0),b,c;
   b = a;    c = b;
目标对象 源对象    ...
}
```

而不能实现多重赋值，例如：c = (b = a);

为了实现多重赋值，可将它定义为：

```
Complex Complex::operator =
                    (const Complex  c)
{  real = c.real;    imag = c.imag;
   return  * this;
}
```

该函数的最末端必须有return * this; 语句，表示它返回到this指针所指的对象，按照上述表达式转换成调用运算符函数的规则应为：

目标对象 源对象 源对象 目标对象

b = a; → b.operator =(a); → Complex b

在赋值表达式中，赋值号左边的运算量称为目标对象，赋值号右边的运算量称为源对象。由于是对象b调用运算符函数(非静态成员函数)operator =(a)，因此this指向对象b，即*this就是对象b。所以在多重赋值语句中，当将赋值运算符函数的返回值定义为目标对象的类型时，每次赋值返回值就是目标对象(赋值号左边的运算量)，作为下一次赋值时的源对象（赋值号右边的运算量）。

如前所述，由于引用是地址传递方式，书写简便，执行速度又快，因此通常将函数的参数和返回值定义成“类型&”的形式，即函数的参数是源对象的引用，而函数返回值为目标对象的引用。





```
Complex & Complex::operator =
                        (const Complex & c)
{   real = c.real;
    imag = c.imag;
    return * this;
}
```

Complex &c = a;

b = a; → b.operator =(a)
→ Complex &x = b;

这里是假想有个Complex类的对象引用x接受赋值运算符函数的返回值，x对目标变量b引用，它就是目标变量b，所以，该运算符函数返回到目标变量b。并且，采用这种形式由于调用赋值运算符函数和返回时均采用地址传递方式执行速度快。





(3) 没有重载赋值运算符函数时，C++自动提供一个默认的赋值运算符函数，但它只能完成“位模式拷贝”的任务，而不能为目标对象分配内存空间。对于含有指针类型数据成员的类，且该类的构造函数又含有用new为其对象的数据成员分配堆(Heap)中的内存空间时，与复制构造函数一样，必须自行编写重载赋值运算符函数。

```
#include <iostream>
using namespace std;
#include <string.h>     //库函数strcpy( )原型在其中
class Person {
  char * pName;
public :
Person(char * pN)  {
   static int j = 0;
   cout<< "第" << ++j
       << "次调用类型转换构造函数 !\n";
   pName = new char[strlen(pN) + 1];
   if(pName)     strcpy(pName, pN);
}
```





```
Person(Person & p);
Person & operator = (Person & s);
~Person( )
{ static int k = 0;
  cout << "第" << ++k
       << "次调用析构函数 !\n";
  pName[0] = '\0';
  delete  pName;
 }
void Display( )
   {  cout << pName << endl; }
};
Person::Person(Person & p)
{ cout << " 调用复制构造函数 !\n";
 pName=new char[strlen(p.pName)+ 1];
 if(pName)  strcpy(pName, p.pName);
}
```

```
Person& Person::operator = (Person & s)
{ cout << " 调用赋值运算符函数 !\n";
if(pName) {
//若字符串不为空，撤消目标对象已经占有的内存空间
  delete pName;//硬件电路也是这样，先清除再写入(flash)
   pName = '\0';
}  //给目标对象重新分配新的内存空间
pName=new char[strlen(s.pName) + 1];
if(pName)   strcpy(pName , s.pName);
return * this;
}
```





```
void main( )
{
  Person p1("OldObject");
   //第1次调用类型转换构造函数
  Person p2 = p1;     //调用复制构造函数。
  p1.Display( );
  //输出显示对象p1的字符串"OldObject"
  p2.Display( );
   //输出显示对象p2的字符串"OldObject"
  Person p3("NewObject");
  //第2次调用类型转换构造函数
  p3.Display( );
   //输出显示对象p3的字符串"NewObject"
  p3 = p1;              //调用赋值运算符函数
  p3.Display( );
  //输出显示对象p3的字符串变成"OldObject"
}
```





该程序的输出结果为：

```
第1次调用类型转换构造函数 !
 调用复制构造函数 !
OldObject                    (对象p1的字符串)
OldObject                    (对象p2的字符串)
第2次调用类型转换构造函数 !
NewObject              (对象p3赋值操作前的字符串)
 调用赋值运算符函数 !
OldObject              (对象p3赋值操作后的字符串)
第1次调用析构函数 !              (撤消对象p3)
第2次调用析构函数 !              (撤消对象p2)
第3次调用析构函数 !              (撤消对象p1)
```





如例5.2的Person类，其私有数据成员pName为字符型指针，且构造函数又含有用new为其对象的数据成员分配堆中的内存空间，必须自定义赋值运算符函数和复制构造函数，这种自定义赋值运算符函数通常包含两部分，第一部分用来撤消目标对象已经占有的内存空间，第二部分再给目标对象重新分配新的内存空间。对象p3在创建时，在堆中保存的字符串是“Newobject”，在执行”p3 = p1;”语句的过程中，则调用重载赋值运算符函数，将表达式转换成运算符函数的调用格式有：

```
p3.operator = (Person & p1);
```

因此首先撤消的对象应该是目标对象p3，即将原先保存的字符串"Newobject"所占用的空间还给堆，然后再给p3从堆中分配新的空间存储由源对象p1“位模式拷贝”而得的字符串"Oldobject"。

(4)赋值运算符函数和复制构造函数的区别：C++会自动提供默认的赋值运算符函数和复制构造函数以方便用户，但是对于含有指针类型数据成员的类，且该类的构造函数又含有用new为其对象的数据成员分配堆(Heap)中的内存空间时，用户都必须自行编写赋值运算符函数和复制构造函数，此时应注意两者的区别。

①当用已存在的对象去初始化同类的新对象时，调用复制构造函数。如例5.2中，

```
Person p2 = p1;
```

在执行复制构造函数时，对象p2还不存在，复制构造函数执行初始化操作。而对于”p3 = p1;”语句，则调用赋值运算符函数将源对象p1赋值给目标对象p3，此时p1和p3都是已存在的对象。





②由于复制构造函数是用已存在的对象去初始化新对象，因此先编写用new运算符为新对象分配内存空间的语句，然后对新对象进行“位模式拷贝”即可。而赋值运算符函数源对象和目标对象都已存在，因此应包含两部分，第一部分类似与析构函数的功能用来目标对象，第二部分类似与复制构造函数的功能。





```
#include <iostream>
using namespace std;
class Point {
public :
      Point(int i, int j);
      Point(Point & p);
      ~Point(void);
          int ReadX( ) { return x; }
      int ReadY( ) { return y; }
          Point operator =(Point p);
private :
      int x, y;
};
Point::Point(int i, int j)
{ static int k = 0;
       x = i; y = j;
       cout << "一般构造函数第" << ++k
         << "次被调用 !\n";
}
```





```
Point::Point(Point & p)
{ static int i = 0;
  x = p.x;      y = p.y;
  cout << "复制构造函数第" << ++i
       << "次被调用 !\n";
}
Point::~Point(void)
{ static int j = 0;
      cout << "析构函数第" << ++j
        << "次被调用 !\n";
}
Point Point::operator =(Point p)
{     static int l = 0;
  x = p.x;      y = p.y;
  cout << "重载赋值运算符函数第" << ++l
       << "次被调用 !\n";
      return * this;
}
```

```
Point func(Point Q)
{ cout << "\n已经进入到func( )的函数体内 !\n";
  int x, y;
  x = Q.ReadX( ) + 10;
  y = Q.ReadY( ) + 20;
  cout << "定义对象R, 则";
      Point  R(x, y);
  return R;
}
void main( )
{ Point M(20, 35), P(0, 0);
  Point N(M);
      cout << "下一步将调用func( )函数,
       请注意各种构造函数的调用次数 !\n\n";
  P = func(N);
  cout << "P = " << P.ReadX( )<< ", "
     << P.ReadY( ) << endl;
}
```





若将main( )改成:

```
void main( )
{ Point M(20, 35);
  Point N(M);

  cout << "下一步将调用func( )函数,
       请注意各种构造函数的调用次数 !\n\n";
  Point P = func(N);
  /* 这是初始化操作, 与执行语句 "P = func(N);"有区别, 不产生
匿名对象 x, 直接调用复制构造函数, 用func(N)返回值初始化新创建的对象P
*/
  cout << "P = " << P.ReadX( )
    << ", " << P.ReadY( ) << endl;
}
```





5.3其它运算符的重载

1.重载增量、减量运算符：增量、减量运算符是单目运算符，但它们有前缀运算和后缀运算之分，为了区分前、后缀运算，C++约定把前缀运算符仍重载为单目运算符函数，即表达式 ++a;可解释为

```
a.operator ++( );(重载为成员函数)
```

而把后缀运算符看成双目的，重载为双目运算符函数，在其参数表内放上一个整数形参，该形参只是用来区分前、后缀运算，此外无任何作用，所以形参名在函数头中省略，即：重载为成员函数的格式：

```
<返回类型> <类名>::operator <op>
                                  (int)
{                 <函数体>                }
```





对于表达式 a++;可等价看成a ++(0); 这解释为: a.operator ++(0);(重载为成员函数)

例5.3说明了因前、后增量操作的含义不同而决定其返回类型，前置增量运算符返回对象引用，后置增量运算符返回对象的值。

```
#include <iostream>
using namespace std;
class Counter {
  unsigned value;
public :
    Counter( ) { value = 0; }
    Counter & operator ++( );
//前置增量运算符函数。
    Counter operator ++(int);
//后置增量运算符函数。
    void Print( );
};
```

```
Counter & Counter::operator ++( )
{ value++;
   //把Counter类对象的value值增1
  cout << " 调用前置增量运算符函数 !\n";
  return * this;
  //该函数的返回值是value值增1后的对象
}
Counter Counter::operator ++(int)
{ Counter t;     //定义一个暂存对象t
  t.value = value++;
  //把增1前的对象value值保存在暂存对象t中
  cout << " 调用后置增量运算符函数 !\n";
  return t;
  //该函数的返回值是value值增1前的对象
}
```





```
void Counter::Print( )
{ static int i = 0;
      ++i;
  cout << "(" << i << ") 对象的
          value值 = " << value << endl;
}
```





```
void main( )
{ Counter c , d;
      cout<<"一. 检测后置增量运算符函数:\n";
    for(int i = 0; i < 3; i++)
      c++;
    c.Print( );
    d = c++;
    c.Print( );
    d.Print( );
    cout<<"二. 检测前置增量运算符函数:\n";
    for(i = 0; i < 3; i++)
      ++c;
    c.Print( );
    d = ++c;
    c.Print( );
    d.Print( );
}
```





```
    该程序的输出结果:
    一. 检测后置增量运算符函数 :
     调用后置增量运算符函数 !
  调用后置增量运算符函数 !
  调用后置增量运算符函数 !
    (1) 对象的value值 = 3 (对象c的value值)
     调用后置增量运算符函数 !
    (2) 对象的value值 = 4 (对象c的value值)
    (3) 对象的value值 = 3 (对象d的value值)
    二. 检测前置增量运算符函数 :
     调用前置增量运算符函数 !
     调用前置增量运算符函数 !
     调用前置增量运算符函数 !
    (4) 对象的value值 = 7 (对象c的value值)
     调用前置增量运算符函数 !
    (5) 对象的value值 = 8(对象c的value值)
    (6) 对象的value值 = 8(对象d的value值)
```

2.函数调用运算符( )的重载：函数调用运算符不是双目运算符。而是“多目”运算符，可适用于多个参数的情况，例如用于多维矩阵，作为接收多个参数的下标运算符使用，而下标运算符[ ]只能有一个参数。例如：

```
#include <iostream>
using namespace std;
class A
{
  int value[10][10];
public:
  int & operator( )(int x, int y)
  { return value[x][y]; }
};
```





```
void main( )
{ A a;
  int i, j;
  for( i = 0; i < 3; i++)
        for( j = 0; j < 3; j++)
            a(i, j) = i * 10 + j;
  for(i = 0; i < 3; i++){
        cout << "\n value[" <<i << "] = ";
        for(j = 0; j < 3; j++){
             cout << a(i, j);
             if(i == 0) cout << "   ";
             else cout << "  ";
         }
}
```





```
cout << "\n";
int x = 1, y = 1;
a(x, y) = a(x + 1, y) + a(x, y + 1);
cout << " a(" << x + 1 << ", "
      << y << ") = " << a(x + 1, y);
 cout << "\n a(" << x << ", "
      << y + 1 << ") = " << a(x, y + 1);
 cout << "\n a(" << x << ", " << y
       << ") = " << a(x, y) << endl;
}
```





该程序的输出结果

```
value[0] = 0   1   2
value[1] = 10  11  12
value[2] = 20  21  22
a(2, 1) = 21
a(1, 2) = 12
a(1, 1) = 33
```

说明：

(1)通过重载函数调用运算符可实现如下数学函数的抽象：a(i,j) = a * 10 + j。它用来对int型二维数组value[10][10]的每个元素初始化。

(2)在执行main( )函数内如下语句的过程中

a(x, y) = a(x + 1, y) + a(x, y + 1);函数调用运算符函数被调用了三次，分别为：

a(2,1) → a.operator()(2,1)
→ 读value[2][1]( = 21)的值
a(1,2) → a.operator( )(1, 2)
→ 读value[1][2]( = 12)的值
a(1,1) → a.operator( )(1, 1)
→ 读value[1][1]( = 33)的值





(3)类A并没有重载’+’运算符，由于类A的函数调用运算符函数返回一个int类型的引用，所以编译系统可以采用为int类型定义的内部’+’操作实现类A的加法。

(4)函数调用运算符只能定义成非静态成员函数而不能是友元函数。





3.下标运算符[ ]的重载：下标运算符[]用于数组，在定义数组的说明语句中，方括号只是用来指定数组的元素个数，例如：

```
char ch, str[10] = “abcdefghi”;
ch = str[6];
```

这后一条语句才是使用了下标运算符访问第6号元素，由于编译系统不进行下标范围的检查，无法确保访问数组元素不会越界。采用类可定义一种更安全、功能更强的数组类型，为此应对该类重载下标运算符[]。例如：





```
#include <iostream>
using namespace std;
class CharArray {
  char * text;
  int  size;
public:
  CharArray(int sz)
  { size = sz; text = new char[size]; }
  ~CharArray( ) { delete [ ] text; }
  char & operator[ ](int i);
  int GetSize( ) { return size; }
};
char & CharArray::operator[ ](int i)
{
  static int k = 0;
  if(i > size - 1) {
       i = size - 1;
       cout << "\n(" << ++k << ")<<"Index
   out of range , return last element !";  }
  return text[i];
}
```

```
void main( )
{
  int cnt;
  CharArray str1(6);
  char * str2 = "string";
  for(cnt = 0; cnt < 8; cnt++)
        str1[cnt] = str2[cnt];
  cout << "\n";
  for(cnt = 0; cnt < 8; cnt++)
        cout << str1[cnt];
  cout << "\nSize of str1 = "
       << str1.GetSize( ) << endl;
}
```





该程序的输出结果：

```
(1)Index out of range , return last element !
(2)Index out of range , return last element !
strin
(3)Index out of range , return last element !
(4)Index out of range , return last element !
Size of str1 = 6
```

(1)下标运算符被认为是双目运算符，它只能有一个参数，不可带多个参数，即只能重载一维数组的下标运算符。例如：

```
a = b[20];
a = b[20][10];   //error
```

在重载下标运算符中不会出现一个以上的方括号，若确实需要这种表示时，可采用函数调用运算符a = b(20, 10);





(2)下标运算符函数只能声明为非静态成员函数，不能是友员函数。如例程reloaxb.cpp中

```
char & CharArray::operator[ ](int i)
{        。。。
     return text[i];
}
void main( )
{     。。。
   str1[4] = 'n'; → str1.operator[ ](4)
}
```

与所有双目运算符函数一样，第1运算量(或称操作数)是对象(str1)本身，第2运算量是函数的显式参数，即下标变量int i = 4。





(2)可以将下标运算符函数的返回值类型指定为某个类的对象引用，从而可以将该下标表达式写在赋值号的左边(即用于被赋值)。如例程reloaxb.cpp中：

```
char & CharArray::operator[ ](int i)
{ static int k = 0;
  if(i > size - 1) {
       i = size - 1;
       cout << "\n(" << ++k << ")Index
   out of range , return last element !";
  }
  return text[i]; //执行字符数组的下标操作。
}
```

下标运算符函数operator[ ]( )是返回到第i个元素的引用。若i越界，则返回到最后一个元素，这种方法是迫使下标在界内，这是由函数体内的条件语句来完成的。另一种方法是输出报错信息并立即终止程序，可写成：

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

char & CharArray::operator[ ](int i)
{  if(i > size - 1)
   {
     cout << "\nIndex out of range ! ";
     exit(1);
   }
   return text[i]; //执行字符数组的下标操作。
}
```





(3)请注意CharArray类下标运算符函数的返回语句return text[i];它虽然执行数组的下标操作，但不会发生递归调用，因为text[i]是字符型为基本数据类型，而不是class类型，编译系统将text[i]解释为*( text + i)，不会再次调用CharArray类的下标运算符函数operator[ ]( )导致递归调用。

(4)当main( )函数内的for语句循环变量cnt变成6以后，将一个越界下标(CharArray 类对象str1的下标范围为0 ~ 5)传递给下标运算符函数operator[ ]( )，它将检查出越界下标并迫使下标在界内(最大下标为5，使得第2个for语句只输出" strin"5个字符)，接着向CRT屏幕输出越界信息，静态变量k记录了越界次数，读者可自行分析程序的输出结果。