(2) 拷贝初始化构造函数 (或称复制构造函数)

①功能：当创建一个新对象时，可用同“类”的一个已存在对象去初始化新对象。例如：

```
Complex a(3.2 , 6.18);
Complex b = a;
```

它将已存在对象的每个数据成员 (如Complex类中的real、imag) 的值都复制给新对象的对应数据成员，即按每个数据成员在类中说明的顺序，每次复制一个数据成员，称为实现“同类对象的位模式拷贝”，编程者应在类中设计如下复制构造函数。

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


②复制构造函数特点：

•该函数的第1个参数 (通常只有这一个参数) 是对同类对象的引用，且多数采用对同类常量类型的引用，这样可以确保被复制对象 (例中的对象a) 的所有数据成员值在复制构造函数体内不会被改变。例如，b通过复制构造函数初始化为与a相同的复数值。

•设X为类名，形式为：

```
X::X(const X & 引用名, ...);
```

或

```
X::X(X & 引用名, ...);
```

形参

称为拷贝初始化构造函数，或称复制构造函数。

•若某个类没有定义复制构造函数，而编程者又用同类的已存在对象去初始化新对象，则编译系统将自动地生成一个如下形式的缺省 (默认) 的复制构造函数：

```
X::X(const X & 引用名)
{  数据成员名1 =  引用名.数据成员名1;
   数据成员名2 =  引用名.数据成员名2;
          ...
}
```

作为该类的公有成员函数，来实现“同类对象的位模式拷贝”，它的拷贝策略是逐个数据成员依次拷贝，而不完成其他任何任务。由于这种缺省复制构造函数X::X(const X & 引用名);是一种通用格式，编程者应该谨慎使用，在某些场合会带来程序隐患。为此，首先应该搞清编译系统在哪些情况下会自动调用缺省复制构造函数呢？


```
#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; }
private :
      int x, y;
};
```

C++ 程序设计

```
Point::Point(int i, int j)
{  static int k = 0;
//内部静态变量k用来记录调用一般构造函数的次数
    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;
//内部静态变量i用来记录调用复制构造函数的次数
    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;
//内部静态变量j用来记录调用析构函数的次数
    cout << "析构函数第" << ++j
         << "次被调用 !\n";
}
```


C++ 程序设计

```
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;
}
```


C++ 程序设计

```
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;
}
```


C++ 程序设计

该程序的输出结果：

```
一般构造函数第1次被调用 !（创建对象M）
一般构造函数第2次被调用 !（创建对象P）
复制构造函数第1次被调用 !（创建对象N）
下一步将调用func( )函数，请注意各种构造函数的调用次数 !

复制构造函数第2次被调用 !（执行Point Q = N ; ）

已经进入到func( )的函数体内 !
定义对象R，则一般构造函数第3次被调用 !（创建对象R）
复制构造函数第3次被调用 !（返回时创建匿名对象x）
析构函数第1次被调用 !
析构函数第2次被调用 !  (撤消匿名对象X、对象R和形参Q)
析构函数第3次被调用 !
P = 30, 55
析构函数第4次被调用 !
析构函数第5次被调用 !  (撤消对象N、P和M)
析构函数第6次被调用 !
```

①用已存在的对象(如M)显式初始化新建对象(如N)。如上例中在main()函数内如下语句:

Point N(M); 它完全等价于 Point N = M;

这时需要调用复制构造函数，如果该类未定义复制构造函数则系统将调用默认复制构造函数。

②在C++中对象可作为函数实参传递给形参，称为“传值调用”。如上例中在main()函数内的语句P = func(N); 便是属于这种情况。

相当于执行了Point Q = N;

Point func(Point Q);

在调用func( )函数时，对象N作为实参用来初始化被调用函数func( )的形参Q，相当于执行了”Point Q = N;”语句，这时需要调用复制构造函数，如果该类未定义复制构造函数则系统将调用默认复制构造函数。

③当对象作为函数返回值时，如上例中在f( )函数内的语句return R;便是属于这种情况。执行返回语句“return R;”时，系统将用对象R来初始化一个临时的匿名对象x，即Point x = R;

这时需要调用复制构造函数，如果该类未定义复制构造函数则系统将调用缺省复制构造函数。

显然在执行“P = func(N);”时，将两次调用复制构造函数，特别是当某类未定义复制构造函数时，则系统将调用默认复制构造函数，从而自动创建了该类的临时对象。如果编程者不想利用该函数的返回值，例如只执行“func(N);”而防止第2次调用默认复制构造函数，可将该函数的返回类型定义成类对象的引用，即:

类型名 &函数名(参数表);

也可把函数的形参定义成类对象的引用，如:

类型名 &函数名(类型名 &,...);

用以避免调用默认复制构造函数。

如上例可改成:

```
Point & func(Point & Q)
{       ...
  static Point  R(x ,y);
  return R;
}
```


后其输出结果为:

```
    一般构造函数第1次被调用 !
    一般构造函数第2次被调用 !
    复制构造函数第1次被调用 !
    下一步将调用func( )函数，请注意各种构造函数的
调用次数 !


    已经进入到func( )的函数体内 !
    定义对象R，则一般构造函数第3次被调用 !
    P = 30, 55
    析构函数第1次被调用 !
    析构函数第2次被调用 !
    析构函数第3次被调用 !
    析构函数第4次被调用 !
```

C++ 程序设计

如前所述，这种返回引用的函数，只能用全局对象或静态对象作返回值，而不能用该函数的自动对象作为返回值，因此将对象R定义成静态型。这时仅在执行“Point N(M);”语句时，调用了复制构造函数。因为只有当对象作为值传递时，编译系统才会调用复制构造函数(包括默认复制构造函数)，而在传递对象的引用或者返回对象的引用时并不调用它，这时实际上只是传递了指向对象的指针。所以编程者常常把函数的参数和它的返回值定义成“类型名 &”的形式，以防止编译系统自动调用默认复制构造函数。但是最好的办法是自行设计复制构造函数。

由于默认复制构造函数只完成一个成员一个成员的拷贝任务，当某类的构造函数含有用new为其对象的数据成员分配堆(Heap)中的内存空间时，将使新建对象和被复制对象两者占有同一内存空间(资源)，当对象撤消时该内存空间将经历两次资源回收操作。

C++ 程序设计

```
#include <iostream>
using namespace std;
#include <string.h>

class Person {
public :
    Person(char * pN);
    Person(Person & p); //默认复制构造函数，只做位模式拷贝
      {  pName = p.pName;   //使两个字符串指针指向同一地址位置
       }

    void Show(void)          //输出显示指针pName所指的字符串。
    {  cout << pName << " .\n"; }
    ~ Person( );
private :
    char * pName;
};
```


C++ 程序设计

```
Person::Person(char * pN)
{  cout << "一般构造函数被调用 !\n";
   pName = new char[strlen(pN) + 1];
   //在堆中开辟一个内存块存放pN所指的字符串
   if(pName != NULL)
        strcpy(pName ,pN);
   //如果pName不是空指针，则把形参指针pN所指的字符串复制给它
}
Person::~ Person( )
{
   static int k = 0;

    cout << "析构造函数第" << ++k
         << "次被调用 !\n";
    pName[0] = '\0';
    delete pName;
}
```


C++ 程序设计

```
void main( )
{
    Person p1("WillianFord");
    Person p2 = p1;
    /* 用已存在的对象p1去初始化新创建的对象p2，必将调用复制构造函数，若所
属类没有定义复制构造函数，则调用默认的复制构造函数。*/
    cout << "这本书作者的英文名字是";
    p1.Show( );
        cout << "另一本书作者的英文名字也是";
        p2.Show( );
        cout << "程序执行完 !\n";
}
```

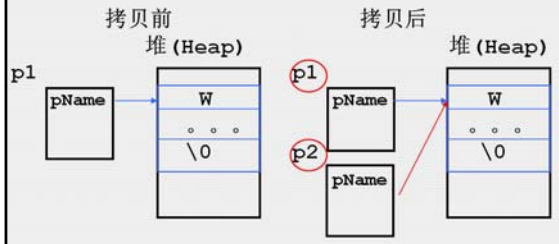

图3.6　对象p1和p2的拷贝

程序开始执行时，创建对象p1调用构造函数Person(char * pN)，用new从堆中为字符串"BillGats"分配内存空间，并将其返回地址值赋给数据成员pName。执行“Person p2 = p1;”时，因为该类没有定义复制构造函数，于是就调用默认复制构造函数，由于它没有给新对象p2分配堆空间，而只是完成了数据成员pName的拷贝任务。如图3.6所示，使得p1和p2的pName都指向了堆中的同一内存位置。当main( )函数结时，调用析构函数逐个撤消对象，必然导致用delete对同一内存空间进行了两次资源回收操作，为非法操作。为此自行设计复制构造函数以解决这个问题。


```
#include <iostream>
using namespace std;
#include <string.h>
class Person {
public :
     Person(char * pN);
     Person(Person & p);
//自行设计复制构造函数
     void Show(void)
     {    cout << pName << " .\n";    }
     ~ Person(void);
private :
     char * pName;
};
```



```
Person::Person(char * pN)
{   cout << "一般构造函数被调用 !\n";
    pName = new char[strlen(pN) + 1];
    if(pName) strcpy(pName ,pN);
    //复制非空字符串
}
Person::Person(Person & p)
//自行设计复制构造函数
{   cout << "复制构造函数被调用 !\n";
    pName=new char[strlen(p.pName)+ 1];
    /* 用运算符new为新对象的数据成员，即指针pName
       所指的字符串在堆中分配内存空间。*/
if(pName)
     strcpy(pName ,p.pName);
//复制非空字符串
}
```

```
Person::~ Person( )
{  static int k = 0;
   //内部静态变量k用来记录调用析构函数的次数
   cout << "析构造函数第" << ++k
        << "次被调用 !\n";
   pName[0] = '\0';
   /*把被撤消对象的数据成员，即指针pName所指的字符
     串设置为空串*/
   delete pName;
}
```



```
void main( )
{
   Person p1("WillianFord");
   Person p2 = p1;
   cout << "这本书作者的英文名字是";
   p1.Show( );
   cout << "另一本书作者的英文名字也是";
   p2.Show( );
   cout << "程序执行完 !\n";
}
```

因此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，自行编写复制构造函数是消除程序隐患的最好办法，它应具有用new运算符为新对象分配内存空间的功能。除此之外，都可以借助默认复制构造函数使程序简化。

(3)类型转换构造函数

X为类名，T表示不同于X的类型，则形式为：

```
X(const T &, ...); 或  X(T &, ...);
```

的构造函数具有类型转换功能，称为类型转换构造函数。

例如：

```
Complex::Complex(const double & r)
{  real = r;   imag = 0.0;   }
```

complex类在定义中提供了将一个double型的单个参数r转换为用户所定义的complex类的方法，把double型单个参数r赋给complex类的real成员，而把它的imag置为0.0。

(4)一般构造函数：

X为类名，T_1 、T_2为不同于X的类型，则形式为：

$X(T_1, T_2, ...)$；的构造函数，称为一般构造函数。

表3.1

构造函数分类	原　　型	备　注
一般构造函数	X(T_1, T_2, ...);	X 为类名，T_1、T_2为不同于X的类型。T为不同X的类型
无参数构造函数	X(void);	
复制构造函数	X(X &, ...); X(const X &, ...);	
类型转换构造函数	X(T); X(T &); X(const T); X(const T &);	

4.C++的结构体：

在C++中结构体是class类型的一种特殊形式，也可以有私有部分、公有部分、保护部分，也可有数据成员、成员函数(其中包括构造函数和析构函数)。例如如下说明语句，它说明了一个结构名为Point的结构类型：

```
struct Point {
    float x, y;
};
```

C和C++中的结构体具有如下不同点：

(1)在C语言中用Point说明结构变量时，必须使用关键字struct。

```
struct Point p;
```

而C++中，Point一经定义可象基本数据类型名一样使用，不必写关键字struct。如：

```
Point  p;
```


(2) C++扩充了结构体的功能,它也可以有成员函数.例如:

```
struct Point {
[public :]
    float x, y;
    void set(float xi, float yi)
    {  x = xi;   y = yi;    }
};
```

也可以把它定义成类:

```
 class Point {
[private :]
float x, y;
public :
Point(void)
{ x = 0.0;  y = 0.0; }
void set(float xi, float yi)
{  x = xi;   y = yi;    }
};
```


(3) C++中struct 和class的区别:

①struct中访问限制符public:可以缺省，而class中private:可缺省。

②struct对其对象(结构变量)的初始化可采用大括号包围的初始化列表。例如:

```
Point p = {6.6 , 8.6};//struct Point
```

也可通过调用自身的成员函数或构造函数对结构变量进行初始化。而通常class对其对象的初始化，只能通过调用自身的成员函数或构造函数进行，不能使用初始化列表。

```
Point p;//class Point调用Point(void)
```

只有当该类没有私有数据成员，没有构造函数，没有虚函数，且不是派生类时才可以用初始化列表。如: Point p = {6.6,8.6};

3.4 对象指针和对象引用

1．对象指针和对象引用作为函数的参数：如前所述，函数调用时有“传值”和“传地址”两种参数传递方式，前者称为“传值调用”，后者称为“传址调用”。但“传址调用”比“传值调用”应用更普遍，它有如下优点：

(1)实现“传址调用”，可在被调用函数体内改变作为函数实参的对象，即改变对象数据成员的值，实现函数间信息的双向传递。

(2)“传值调用”需要将整个对象进行位模式拷贝生成副本，而“传址调用”仅将对象的地址作为实参传递给形参对象指针，不仅执行速度快且占用内存空间小，减少时空开销。


```
#include <iostream>
using namespace std;
class M {
    int x, y;
public :
    M( ) { x = y = 0; }
    M(int i, int j)
    { x = i; y = j; }
    void Copy(M * m);
    void SetXY(int i, int j)
    { x = i; y = j; }
    void Print( )
      { cout << "x = " << x
             << " , y = "<< y << endl; }
};
```



```
void M::Copy(M * m)
{   x = m -> x;     y = m -> y;  }
    ①          //①相当于执行了M * m = &p;
void fun(M m1, M * m2)   //形参m1是传值方式，m2是传地址方式。
{
    m1.SetXY(12, 15);
    m2 -> SetXY(22, 25);
}
          ②      ③
void main( )
{
    M p(5, 7), q;
    q.Copy(&p);
    fun(p, &q); //②相当于执行了 M m1 = p;
                //③相当于执行了 M * m2 =  &q;
    p.Print( );
    q.Print( );
}
```


该程序的输出结果：

```
x = 5 , y = 7
x = 22 , y = 25
```

C++ 程序设计

•该程序中有两个指向对象的指针，一个是成员函数copy( )的形参M * m，另一个是用作一般函数fun( )的形参M * m2。当形参是指向对象的指针时，调用函数所对应的实参应该是某对象的地址值，采用“&对象名”的格式。

•fun( )函数有两个形参，一个是对象名m1进行值传递，另一个是指向对象的指针名(M *)m2采用地址传递方式。因此前者的信息传递是单方向，即作为实参的对象p，其数据成员x、y的值确实传递给作为形参的对象m1，但是在fun( )函数体内，对形参对象m1的操作，使其数据成员x、y的值发生变化却不能改变作为实参对象p的x(仍等于5)、y(仍为7)值。而采用地址传递方式的形参m2却能实现数据的双向传递，当调用fun( )函数时，在实参传递给形参的过程中使得对象指针m2指向了作为实参的对象q，那么在该函数体内，对形参对象m2的操作会使实参对象q发生相同的变化，所以对象q的数据成员值变成为x = 22，y = 25。显然成员函数copy( )的形参M * m也是地址传递方式，因此执行了“q.copy(&p);”语句后，可将对象p拷贝给q。

C++ 程序设计

2.对象引用作函数参数：若将2.6中所述的规则，从变量引伸到对象则变成：凡是用指针作形参进行地址传递的函数，都可以将作为形参的对象指针改为“对象引用”，调用函数的实参可直接写对象名；被调用函数体内的形参可象对象一样使用，而不必使用取地址运算和取内容运算。显然对象引用作函数参数更简单、更方便而更直观，被普遍采用。

C++ 程序设计

```
void M::Copy(M & m)
{   x = m.x;
    y = m.y;          M & m = p;
}
void fun(M m1, M & m2);
void main( )
{ M p(5, 7), q;
  q.Copy(p);
  fun(p, q);
  p.print( );
  q.print( );        M & m2 = q;
} M m1 = p;
void fun(M m1, M & m2)
{     m1.setxy(12, 15);
      m2 -> setxy(22, 25);
}
```


C++ 程序设计

3.this指针：

C++为每个类的成员函数都隐含(编程者不必写)定义了一个特殊的指针this指针，其格式为：

```
类名 * const this;
```

this指针是作为隐含的参数自动传递给成员函数，哪一个对象调用成员函数，this指针就指向那个对象，因此所有成员函数都拥有this指针。

这是因为每当对象调用成员函数时，编译系统就初始化this指针把对象的地址赋给它，即给this指针定向指向了调用成员函数的对象。

由于this指针说明为*const，即常量指针，因此在成员函数体内不能被修改而重新定向。使用时应注意：

C++ 程序设计

(1) this指针是成员函数的一个隐含参数，在成员函数体内，可以隐含使用this指针访问数据成员，不管它们是公有的、私有的还是保护的。例如，Counter类的成员函数increment()也可写成：

```
void Counter::
      increment(Counter * const this)
      {  this -> value++;  }
```

若程序中写有

```
   Counter c1;
   c1.increment();  //this指向对象c1
```

当执行到该语句，调用成员函数increment()，this指针就指向了对象c1，函数体内的”this ->value++;”语句使c1的value值增1。

由于this指针是隐含参数不必显式地写出来，同样可以达到“用this指针代替它所指的对象c1来访问成员”的目的。所以”this -> value++;”语句与”value++;”语句等价。

C++ 程序设计

(2) this指针是一个自动型变量，它在任何非静态成员函数体内都是可见的，即都可以隐含使用this指针访问数据成员。

(3) 通常虽然不去显式地使用this指针，但在某些场合需要显式地使用this指针来操作调用成员函数的对象。例如用*this来标识该对象。因为this指针指向了调用它的对象，则对this指针进行取内容运算的结果也就是该对象。

C++ 程序设计

```
#include <iostream>
using namespace std;
class A {
public :
      A( ) { a = b = 0; }
   A(int i ,int j) { a = i; b = j; }
   void Copy(A &aa);
      void Print( )
      { cout << "a = " << a
           <<  " , b = " << b << endl; }
      int Geta( ) { return a; }
private :
      int a ,b;
};
```


C++ 程序设计

```
void A::Copy(A & aa)
{   if(this == &aa) return;
    *this = aa;       //*this就是a1
}this = &a1;    A & aa = a2;
void main( )
{   A a1 ,a2(3 ,4);
    a1.Copy(a2);    //实现了a2拷贝到a1
       a1.Print( );
}
```

(4) this指针只有在对象调用成员函数时才被初始化重新定向，指向调用成员函数的对象，并作为隐式参数传递给成员函数，随后进入到成员函数体内就不能再修改。

(5) this指针是一个常量指针，但也可以将它定义成指向常量的常量指针，其方法是在定义成员函数时，或者在类体内声明成员函数时，在函数头后面加上关键字const


```
class Counter{
public :
    Counter( );
    //Counter类的无参数构造函数。
            ...
    unsigned ReadValue( ) const
    {  return value; }
~ Counter( );
//Counter类的析构函数。
private :
     unsigned value;
    //Counter类的私有数据成员。
};
```


这实质上是把this指针在ReadValue( )函数体内说明为如下类型：

```
const  Counter * const this ;
```

为了与一般成员函数相区别，把成员函数ReadValue( )称为“const成员函数”。那么，在该成员函数体内，不仅this指针不能重新定向，并且this指针所指对象的数据成员也不能再修改其值。

例如：

```
unsigned ReadValue( ) const
{
    return value ++ ;
    //出错，在const成员函数体内，this
      指针所指对象的数据成员不能修改。
}
```


3.5静态成员：

静态成员包含静态数据成员和静态成员函数。

1.静态数据成员：在定义一个类时，若将一个数据成员指明为static型，则称该成员为静态数据成员。不管该类创建了多少个对象，而静态数据成员只有一个静态数据值。因此静态数据成员是该类的所有对象共享的成员，也是连接该类中不同对象的桥梁。

```
#include <iostream>
using namespace std;
class Counter{
public :
Counter(char c){ count++; ch = c; }
//构造函数，使静态数据成员count的值增1
static int HM( ) { return count; }
//静态成员函数，读静态数据成员count的值
int ReadCh( ){ return ch; }
//普通成员函数，读数据成员ch的值
~Counter( ){ count--; ch = 0; }
//析构函数，使静态数据成员count的值减1
private :
    static int count;   //静态数据成员count
    char ch;            //普通数据成员ch ，记录每个对象的序号
};
```



```
int Counter::count = 100;//静态数据成员的初始化
void main( )
{ Counter c1(1) ,c2(2) ,c3(3),c4(4);

cout << "Value of Counter is
  currently " << Counter::HM( )
     << endl;
cout << "Object c1 NO : "
      << c1.ReadCh( ) << endl;
cout << "Object c2 NO : "
      << c2.ReadCh( ) << endl;
cout << "Object c3 NO : "
     << c3.ReadCh( ) << endl;
cout << "Object c4 NO : "
      << c4.ReadCh( ) << endl;
}
```


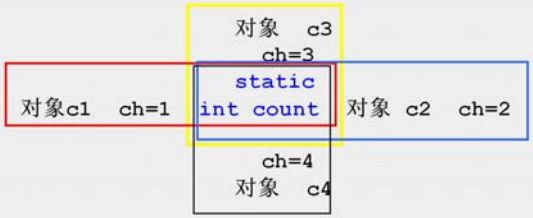

图3.9 Counter类的静态数据成员

该程序的输出结果：

```
Value of count is currently 104
Object c1 NO : 1
Object c2 NO : 2
Object c3 NO : 3
Object c4 NO : 4
```

每创建一个Counter类的对象，不管这些对象是auto型、extern型和static型都调用一次构造函数Counter()，静态数据成员count的值都增1。因此静态数据成员经常用来记录某个类所创建的对象个数。

C++ 程序设计

(1)静态数据成员可说明为公有的、私有的或保护的。说明为公有的可直接访问它，但必须用”类名::”加以指定，其格式为：

类名:: 静态数据成员名

C++ 程序设计

```
#include <iostream>
using namespace std;
class Counter{
public :
    static long count;//公有静态数据成员count
Counter( ){ count++; }
//构造函数，静态数据成员count的值增1
    long GetCount( )
    { return count; }
~Counter( ){ count--; }
//析构函数，静态数据成员count的值减1
};
Counter c1, c2, c3;
//创建Counter类的3个对象
long Counter::count = 5;
//静态数据成员count的初始化操作
```


C++ 程序设计

```
void main( )
{
   cout << "(1)The object count is "
        << Counter::count << endl;
   Counter c4;
   cout << "(2)The object count is "
        << Counter::count << endl;

}
```

说明为私有的或保护的静态数据成员与普通数据成员一样，只有通过调用公有部分的成员函数才能访问它们。

C++ 程序设计

```
#include <iostream>
using namespace std;
class Counter{

static long count; //私有静态数据成员count

public :
    Counter( ){ count++; }
    long GetCount( )
  { return count; }
    ~Counter( ){ count--; }
};
long Counter::count = 5; → 0
Counter c1, c2, c3;
```

```
void main( )
{
cout << "(1)The object count is "
      << c3.GetCount( ) << endl;
  /* 对象c3通过调用公有成员函数GetCount( )间接
   访问静态数据成员count，输出显示count的值。*/
     Counter c4;
cout << "(2)The object count is "
      << c3.GetCount( ) << endl;
//通过对象c4，调用公有成员函数GetCount( )输出显示count的值
cout << "(3)The object count is "
      << c4.GetCount( ) << endl;
}
```


(2)静态数据成员是class类型中一种特殊的数据成员，它存放在内存空间中的值始终保留。只要程序一开始运行它就存在，程序结束时才消失。其空间不会随着对象的创建而分配，或随着对象的消失而回收，所以它的空间分配不在类的构造函数里完成，且空间回收也不在类的析构函数里完成。因此不管它是公有的、私有的还是保护的，都必须在所有函数以外用如下格式对它进行初始化，否则将产生连接(linking)错误。

<类型> 类名:: 静态数据成员名 = 初值;

显然将sta.cpp中的counter设置为0时，则可用来记录Counter类所创建的对象个数。但是日常生活中不是以零为初值的例子很多，如出租汽车里程费就不是以零为初始值。在编写这类程序时，应先将counter初值(基准值)设为5，然后再创建3个对象其结果为8，明确地表达出编程者的意图，若将两语句对调虽然结果一样，但编程者的意图含糊不清程序可读性差。

(3)静态成员(静态数据成员和静态成员函数)封装在类的作用域内，它不是全局性的。因此静态数据成员，在安全性、消除模块间耦合方面比全局变量好得多。(类作用域下的全局量)

(4)由于普通成员函数都具有this指针，所以在普通成员函数体内都可以直呼其名地访问静态数据成员，如在GetCount( )成员函数体内，可直接写“return count;”语句。

2.静态成员函数：

由于类的静态数据成员是所有对象共享的数据，其中任一对象通过成员函数修改了静态数据成员的值，必然影响其他所有对象，从而导致对象间的强耦合，引起混乱不好控制。为此C++还提供了静态成员函数，用它来访问静态数据成员则安全多了。这只需将关键字”static”加到成员函数头前，即指定该成员函数为静态成员函数。现以例程simpl.cpp说明它的特点：

(1)静态成员函数没有隐含的this指针，这意味着它属于一个类而不是属于某一对象。因此在直接调用静态成员函数时，必须明确指出该静态成员函数在哪个类的对象上操作，通常采用如下格式：

类名::静态成员函数名(实参表);

例如，Simple::sum2(obj2);这是调用静态成员函数的一种方法，称为“直接调用法”。


```
#include <iostream>
using namespace std;
class Simple {
public :
  Simple(int x, int y){ v1 = x; v2= y;}
  static void Sum1(Simple * s)
  { s -> v3 = s -> v1 + s -> v2; }
  int Get( ) { return v3; }
  static void Sum2(Simple & r)
  { r.v3 = r.v1 + r.v2;  }
  int GetV3( ) { return  v3;  }
private :
  static long v1, v2;
         long v3;
};              Simple * s = &obj1;
long Simple::v1 = 6;
long Simple::v2 = 8;
```



```
void main( )          Simple & r = obj2;
{ Simple obj1(3, 4);
  obj1.sum1(&obj1);//用对象调用公有成员函数
  cout << "(1)The v3 of obj1 is
    currently " << obj1.get( ) << endl;
  Simple obj2(6, 8);
  Simple::sum2(obj2);     // 直接调用法
  cout << "(2)The v3 of obj2 is
    currently " << obj2.get( ) << endl;
}
```

该程序的输出结果：

```
(1)The v3 of obj1 is currently 7
(2)The v3 of obj2 is currently 14
```


(2)调用公有静态成员函数的另一种方法是通过一个对象来调用静态成员函数，则必须传递给它一个指向该类某对象的指针来代替this指针。如例程中，用指向Simple类的指针s作为静态成员函数sum1(Simple * s)的形参，当执行

“obj1.sum1(&obj1);”语句时，进行实参传递给形参的操作，使得对象指针s指向了对象obj1，在其函数体内，用对象指针s既可以访问非静态数据成员，又可以访问静态数据成员。显然只有公有静态成员函数才能通过对象调用，而私有静态成员函数不能通过对象调用，也不能用“直接调用法”。

(3)静态成员函数由于没有this指针，它只能访问类中的静态数据成员，而不能访问类中的非静态数据成员，这就是静态成员的安全机制。

即静态成员函数专门用来处理静态数据成员，如果实际却有需要希望静态成员函数去访问类中的非静态数据成员，方法之一是把该类的一个对象或者该类的对象指针，或者该类的对象引用作为静态成员函数的一个参数代替this指针传递给它。则在该静态成员函数体内，不管是私有的静态数据成员还是私有的非静态数据成员该参数都能访问。


```
static void Sum2(Simple & r)
{ r.v3 = r.v1 + r.v2;  }
```

如果把该函数写成：

```
static void Sum2( )
{ v3 = v1 + v2;  }
```

这时v3、v1和v2必须是静态数据成员，否则不能访问。这种方法仅是在静态成员函数的作用域内，使得私有的静态数据成员和私有的非静态数据成员都能访问，虽然牺牲了本作用域内静态成员的安全机制，但不会影响其它程序部分，所以这种隐藏技巧是可取的。顺便指出，非静态成员函数是可以访问静态数据成员的。

(4)静态成员函数与静态数据成员一样都存在于类作用域内。

但是静态数据成员的生存周期是与对象变量无关。

不随对象变量的生成而生成，不随其消亡而消亡。单独存储，一直存在。