3.8对象数组和成员对象

1.对象数组：在ANSI C中，把具有相同结构类型的结构变量，有序地集合起来便组成了结构数组。在ANSI C++中，与此类似将具有相同class类型的对象有序地集合在一起便组成了对象数组，对于一维对象数组也称为“对象向量”，因此对象数组的每个元素都是同种class类型的对象。

(1)对象数组的定义：其定义格式为：

```
<存储类> <类名> 对象数组名[元素个数]。。。
                          [ = {初始化列表}];
```

其中<存储类>是对象数组元素的存储类型，与变量一样有extern型、static型和auto型等，该对象数组元素由<类名>指明所属类，与普通数组类似，方括号内给出某一维的元素个数。对象向量只有一个方括号，二维对象数组有两个方括号，如此类推。


```
#include <iostream>
using namespace std;
class Point {
  int x, y;
public :
      Point(void) { x = y = 0; }
      Point(int xi, int yi)
  { x = xi; y = yi; }
  Point(int c) { x =  y = c; }
  void Print( )
      { static int i = 0 ;
        cout << "P" << i++ << "(" << x
           << " , " << y << ")\n";
      }
};
```



```
void main( )
{     Point Triangle[3] = {Point(0, 0),
            Point(5, 5), Point(10, 0)};
  int k = 0;
  cout << "输出显示第" << ++k
       << "个三角形的三顶点 : \n";
      for(int i = 0; i < 3; i++)
           Triangle[i].Print( );
           Triangle[0] = Point(1);
           Triangle[1] = 6;   //Call Point(6)
           Triangle[2] = Point(11, 1);
      cout << "输出显示第" << ++k
         << "个三角形的三顶点 : \n";
  for(i = 0; i < 3; i++)
           Triangle[i].Print( );
```



```
Point Rectangle[2][2] = {Point(0, 0)
          ,Point(0,6),Point(16,6) ,
           Point(16,0)};
cout << "输出显示一个矩形的四顶点 : \n";
for(i = 0; i < 2 ; i++)
      for(int j = 0; j< 2; j++)
           Rectangle[i][j].Print( );
cout<<"输出显示45度直线上的三点 : \n";
Point Line45[3] = {0, 1, 2};
      for(i = 0; i < 3; i++)
         Line45[i].Print( );
Point PtArray[3];
cout << "输出显示对象向量PtArray的三元素 : \n";
      for(i = 0; i < 3; i++)
           PtArray[i].Print( );
}
```

该程序的输出结果：

输出显示第1个三角形的三顶点 ：
P0(0 , 0)
P1(5 , 5)
P2(10 , 0)

输出显示第2个三角形的三顶点 ：
P3(1 , 1)
P4(6 , 6)
P5(11 , 1)

输出显示一个矩形的四顶点 ：
P6(0 , 0)
P7(0 , 6)
P8(16 , 6)
P9(16 , 0)

输出显示45度直线上的三点 ：
P10(0 , 0)
P11(1 , 1)
P12(2 , 2)

输出显示对象向量PtArray的三元素 ：
P13(0 , 0)
P14(0 , 0)
P15(0 , 0)

(2)对象数组的初始化

①当对象数组所属类含有带参数的构造函数时，可用初始化列表按顺序调用构造函数初始化对象数组的每个元素。如上例中：

```
Point Triangle[3] = {Point(0, 0),
          Point(5, 5),Point(10, 0)};
Point Rectangle[2][2] = {Point(0, 0),
    Point(0, 6),Point(16,6),Point(16,0)};
```

也可以先定义后给每个元素赋值，其赋值格式为：

对象数组名[行下标] [列下标] = 构造函数名(实参表);

例如：

```
Rectangle[0][0] = Point(0, 0);
Rectangle[0][1] = Point(0, 6);
Rectangle[1][0] = Point(16, 6);
Rectangle[1][1] = Point(16, 0);
```

若对象数组所属类含有单个参数的构造函数时；

②如上例中”Point(int c);”，该构造函数置x和y为相同的值。那么对象数组的初始化可简写为：

```
Point Line45[3] = {0, 1, 2};
Point Triangle[3] =  {0,//Call Point(0)
                      5,//Call Point(5)
                      Point(10, 0)};
```

③对象数组创建时若没有初始化列表，其所属类中必须定义无参数的构造函数，在创建对象数组的每个元素时自动调用它。如上例中在执行 “Point PtArray[3];”语句时，调用Point(void)，初始化对象数组PtArray[ ]的每个对象为(0,0)。

④如果对象数组所属类含有析构函数，那末每当建立对象数组时，按每个元素的排列顺序调用构造函数，每当撤消数组时，按相反的顺序调用析构函数。


```
#include <iostream>
using namespace std;
#include <string.h>
class Personal {
      char name[20];
public :
      Personal(char * n)
      { strcpy(name , n);
        cout << name << " says hello !\n";
      }
      ~Personal(void)
      { cout << name << " says goodbye !\n"; }
};
void main( )
{     cout << "创建对象数组，调用构造函数 : \n";
      Personal people[3] = {"Wang", "Li", "Zhang"};
      cout << "撤消对象数组，调用析构函数 : \n";
}
```


该程序的输出结果为：

```
创建对象数组，调用构造函数 :
Wang says hello !
Li says hello !
Zhang says hello !
撤消对象数组，调用析构函数 :
Zhang says goodbye !
Li says goodbye !
Wang says goodbye !
```


2.成员对象和容器类(Container Class)：

当一个类的对象作为另一个类的成员时，该对象称为成员对象或子对象，这另一个类称为容器类(Container Class)，这种方法称为组合技术。当创建容器类的对象时，作为成员对象所需的参数初始化机制，应由容器类的构造函数提供，这是在它的构造函数头内，为成员对象指定一个初始化参数表来完成的，其格式为：

```
容器类(构造函数)名(参数表) :
        成员对象名1(参数表1),
        成员对象名2(参数表2), ...
```

其中参数表1，参数表2，...为成员对象所属类相对应的构造函数参数表。

```
#include <iostream>
using namespace std;
class foo {
  int  i;
public :
  foo( ) { i = 0; }
};
class bar {
  int  i;
public :
  bar(int x) { i = x; }
  int get( ) { return i; }
};
```



```
class Snafu {
  foo f;
  bar b1;
  bar b2;
public :                                    //空函数
  Snafu( ) :[ f( ),] b1(1), b2(2) { }
  void read1( )
  { cout << "b1 of obj = "
         << b1.bar::get( ) << endl; }
  void read2( )
  { cout << "b2 of obj = "
         << b2.bar::get( ) << endl; }
};
```



```
void main( )
{ Snafu obj;
  obj.read1( );
  obj.read2( );
}
```

该程序的输出结果：

```
b1 of obj = 1
b2 of obj = 2
```


(1) 成员对象b1和b2的初始化参数表应放在所属类的构造函数Snafu( )的头部，并用冒号隔开，各成员对象的初始化参数表b1(1)和b2(2)间用逗号隔开，每个成员对象只能在初始化参数表中出现一次。而成员对象f的构造函数foo( )没有参数，所以成员对象f不必写到初始化参数表中，可以缺省。

(2) 当创建容器类的对象时，例如：

```
Snafu obj;
```

编译系统自动地先调用成员对象所属类相应的构造函数，执行初始化参数表b1(1)和b2(2)所规定的任务，然后再调用容器类的构造函数Snafu( )。而调用成员对象构造函数的顺序取决于成员对象在容器类中定义的先后顺序，而不按初始化参数表的排列顺序。

(3) 撤消容器类对象时，先为容器类对象调用析构函数，然后再调用成员对象的析构函数。

(4) 容器类必须有一个构造函数，以便提供一个成员对象的初始化参数表b1(1)和b2(2)，尽管该构造函数体为空函数。编译系统一般不为容器类提供默认的构造函数（在容器类有无参构函的时候可以）。

(5) 必须在容器类中定义一些公有成员函数，用来访问成员对象中的私有数据成员，在容器类的这些成员函数体内，调用成员对象所属类的公有成员函数去访问成员对象的私有数据成员，在访问表达式中用成员对象所属的类名加作用运算符指明该成员函数，其格式为：

```
容器类成员对象名.[成员对象所属的类名::]
                        成员函数名(实参表);
```

或

```
容器类的对象指针成员->[成员对象所属的类名::]
                        成员函数名(实参表);
```



```
#include <iostream>
using namespace std;
class Point {
public :
  Point(int xi, int  yi)
  { x = xi;  y = yi; }
  Point(Point & p)
  {   x = p.x;    y = p.y; }
  ~Point( ) {  }
  int Xcoord( ) { return x; }
  int Ycoord( ) { return y; }
private :
  int x ,y;
};
```



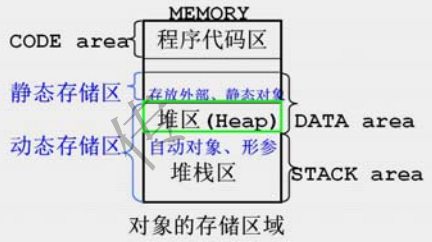
```
class Circle {
public :
  Circle(int x1, int y1, int r) :
          center(x1, y1)
  { radius = r; }
  int GetRad( ) { return radius; }
  int GetcenterX( )
  { return center.Point::Xcoord( ); }
  int GetcenterY( )
  { return center.Point::Ycoord( ); }
private :
  Point center;
  int  radius;
};
```



```
void main( )
{
  Point pnt(160, 180), * bp;
  Circle spotlight(320, 240, 40), * dp;
  bp = & pnt;
  cout<<"(1)pnt's X = "<<bp-> Xcoord( )
  << "\t Y = " << bp -> Ycoord( ) << endl;
  dp = & spotlight;
  cout << "(2)spotlight's X = "
     << dp -> GetcenterX( )<< "\t Y = "
     << spotlight.GetcenterY( );
  cout << "\t radius = "<< dp -> GetRad( )
       << endl;
} 该程序的输出结果：
(1)pnt's X = 160        Y = 180
(2)spotlight's X = 320  Y = 240
   radius = 40
```

3.9对象的存储类型：

与基本数据类型的变量一样，对象在定义时也必须指明它的存储类。C++源程序中，可定义自动(auto型)对象、外部(extern型)对象、静态(static)对象和动态对象(又称堆对象，即使用new和delete运算符创建和撤消的对象)。不同存储类的对象,其作用域和生存期也不相同。生存期与对象所在的存储区域密切相关，存储区域如下图所示，有程序代码区(CODE area)、数据区(DATA area)、堆区(HEAP area)和堆栈区(STACK area)等。



对应的生存期为静态生存期、动态生存期和局部生存期等。

•静态生存期：具有静态生存期的对象，只要程序一开始运行或者运行到其定义点时它就存在，程序结束时才消失。它存放在数据区中分配固定的内存空间，当没有初始化时，编译系统自动将其数据成员设置为零(对数值型)或空(字符串或指针)。象外部(extern)型对象、静态(static)型对象都具有静态生存期。

•动态生存期：使用new和delete运算符创建和撤消的对象(包括变量)，以及调用ANSI C标准函数库中的malloc()和free()函数创建和撤消的变量。具有动态生存期。它们存放在内存的堆中，由new运算符(或malloc()函数所创建的变量)为其分配内存空间则生存期开始，当用delete运算符(或free()函数撤消该变量)撤消它时，或者程序结束时生存期结束。

•局部生存期：自动对象和函数的形参具有局部生存期，它们存放在内存的堆栈区，若没有初始化，其数据成员为随机值。对象与基本数据类型的变量不同的是，若类提供了构造函数，每当创建该类的对象时，都将自动调用构造函数来实现每个新对象的初始化，每当撤消该类对象时都将自动调用析构函数(若该类提供了析构函数)或默认的析构函数，以完成对象存储空间的自动回收。

1.自动对象：与自动变量类同在函数体或程序块内定义的对象称为自动对象。定义它时就创建并自动调用构造函数来实现初始化，当程序退出定义它的作用域时，便自动调用析构函数或默认的析构函数撤消它。

2.内部静态对象：与内部静态变量类同在函数体或程序块内定义的静态对象称为内部静态对象。当程序执行到它的定义点时便创建它，并自动调用构造函数来实现初始化，当程序结束时便自动调用析构函数或默认的析构函数撤消它。它的作用域和可见性一致，但与存在期不一致。

3.外部对象和外部静态对象：它们都是全局型的对象，一旦程序启动，按它们定义的先后次序创建，并依次调用构造函数进行初始化，当程序结束时按相反的次序调用析构函数撤消它们。


```
#include <iostream>
using namespace std;
#include <string.h>
class A {
  char string[50];
public :
  A(char * st);
  ~A( );
};
A::A(char * st)
{ strcpy(string, st);
   cout << string
   << "被创建时调用构造函数 ! " << endl;
}
A::~A( )
{  cout << string <<
    "被撤消时调用析构函数 ! " << endl; }
```



```
void fun( ){
   cout << "在fun( )函数体内 : \n" << endl;

   A FunObj("fun( )函数体内的自动对象
              FunObj");

   static A InStaObj("内部静态对象
                       InStaObj");
}
```



```
void main( )
{ A MainObj("主函数体内的自动对象
                MainObj");
   cout<<"主函数体内，调用fun()函数前: \n";
   fun( );
   cout << "\n主函数体内,
            调用fun()函数后:\n";
}
static A ExStaObj("外部静态对象
                    ExStaObj");
A GblObj("外部对象GblObj");
```

该程序的输出结果：

```
外部静态对象ExStaObj被创建时调用构造函数 !
外部对象GblObj被创建时调用构造函数 !
主函数体内的自动对象MainObj被创建时调用构造函数 !
主函数体内，调用fun( )函数前 :
在fun( )函数体内 :

fun( )函数体内的自动对象FunObj被创建时调用构造函数 !
内部静态对象InStaObj被创建时调用构造函数 !
fun( )函数体内的自动对象FunObj被撤消时调用析构函数 !

主函数体内，调用fun( )函数后 :
主函数体内的自动对象MainObj被撤消时调用析构函数 !
内部静态对象InStaObj被撤消时调用析构函数 !
外部对象GblObj被撤消时调用析构函数 !
外部静态对象ExStaObj被撤消时调用析构函数 !
```


- **自动对象**的作用域仅在定义它的函数体或程序块内，其作用域范围小，生存期也短。
- **内部静态对象**的作用域虽然在定义它的函数体或程序块内，其作用域范围小，但生存期与作用域不一致却较长，从定义点开始一直到程序结束，在作用域以外虽然存在但不可见。
- **外部静态对象**的作用域与外部静态变量类同，是定义该对象所在的整个源文件，从定义点开始到文件结束。其生存期与作用域一致比较长。
- **外部对象**是在某个源文件中定义，而它的作用域却是包含该源文件的整个源程序，其生存期与作用域一致是最长的。

4.动态对象：

如前所述，全局变量、静态数据、常量存放在全局数据区(Data Area)，所有类成员函数和非成员函数的代码存放在代码区(Code Area)，为调用函数而分配的局部变量、函数参数、返回数据、返回地址等存放在堆栈区(Stack Area)，余下的内存空间是自由存储区称之为堆区(Heap Area)。

(1) 动态对象的创建：在C++中，可用new运算符动态地创建对象，并为该对象在内存堆中分配存储空间，取代了ANSI C中的malloc( )函数，这类对象称为动态对象，其定义格式为：

```
new <类型>(初值表)        ①
```

或

```
new <类型>                ②
```


它表明在堆(heap)中动态建立了一个由<类型>所指定的对象。第①种形式由圆括号包围的“初值表”给出被创建对象的初始值。也可采用不赋初始值的第②种形式，但必须给它赋值后才能参加运算和操作.这种形式经常用来定义动态对象数组。这里所说的<类型>包含所有的基本数据类型和派生类型(或称复杂的数据类型、构造类型)，以及用户定义的class类型。new为任意类型的对象在堆(Heap)中分配一块所需的存储空间，并返回该存储空间的首地址。如果堆中没有足够的存储空间或分配出错时返回空(NULL)指针。因此与使用malloc( )函数一样，必须检测其返回值不为空指针，在预先定义了一个同类型的指针后，这种形式便于应用if语句，检测其返回值是否为空指针，可写为：

```
if((指针名 = new 类型) == NULL){ ...}
```

例中：`if((p = new Point)==NULL){ ...}`

顺便指出，在以后程序中该指针名用来代替所创建的动态对象名，该动态对象本身是匿名的。

由于：　　不建议简化

if(p == NULL)（即if(p == 0)）→ if(!p)

简化

if(p != NULL)（即if(p != 0)）→ if(p)

一般格式可写为：

```
if(! (指针名 = new 类型)){出错处理操作；}
if(指针名 = new 类型){创建成功的处理操作；}
```

有时，也可直接采用初始化操作创建新的动态对象，其格式为：

```
类型 * 指针名 = new 类型（初值表）；
```

例如：int * pi = new int;

Date * pd = new Date;

用new也可定义一个动态对象数组，new的返回值是数组第1个元素的地址，其格式为：

```
<类型> *指针变量名= new <类型>[元素个数];
```

例如：Point * pt = new Point[3];

即创建了具有3个元素的Point类型的(动态)数组pt，对单个对象也可以采用这种格式，只不过元素个数为1。例如：

```
Point *px = new  Point[1];
```

用这种格式创建对象时，只是给对象分配了内存空间，不能对它赋初值。因此在使用前，还必须用赋值操作对其赋值。


```
#include <iostream>
using namespace std;
#include <string.h>
class B {
  char name[80];
  double b;
public :
  static int count;
  B(char * s, double n)
  {  strcpy(name, s);   b = n;
     cout << "调用一般构造函数, count : "
          <<  ++count << endl; }
  B( )

  {  cout << "调用无参数构造函数,count: "
          << ++count << endl;
  }
  ~B( )
 { cout << "调用析构函数撤消对象 " << name
   << "后, count : " << --count << endl;
 }
 void Getb(char * s, double & n)
 {  strcpy(s, name);  n = b; }
};
```



```
int B::count = 0;
void main( )
{ B * p;     double d;   char str[80];
  p = new B[3];
   cout << "初始化对象数组p[ ] !\n";
  p[0] = B("ma", 4.8);   //调用一般构造函数
  p[1] = B("wang", 3.6); //初始化每个元素
  p[2] = B("li", 3.1);
  for(int i = 0; i < 3; i++) {
     p[i].Getb(str, d);
//两个参数均为地址传递方式，在getb()函数体内，对n的
  操作就是对d的操作，即将b赋给了d。
     cout << str << " , " << d << endl;
  }
  cout << "\n";
  delete [ ]p;
}
```

该程序的输出结果：

```
调用无参数构造函数, count : 1
调用无参数构造函数, count : 2
调用无参数构造函数, count : 3
初始化对象数组p[ ] !
调用一般构造函数, count : 4   } 执行p[0] =B("ma",4.8);
调用析构函数撤消对象 ma后, count : 3
调用一般构造函数, count : 4   } 执行p[1] = B("wang", 3.6);
调用析构函数撤消对象 wang后, count : 3
调用一般构造函数, count : 4   } 执行p[2] = B("li", 3.1);
调用析构函数撤消对象 li后, count : 3
ma , 4.8
wang , 3.6
li , 3.1

调用析构函数撤消对象 li后, count : 2
调用析构函数撤消对象 wang后, count : 1
调用析构函数撤消对象 ma后, count : 0
```


(2)动态对象的撤消：

用delete运算符来撤消由new运算符创建的动态对象，即delete运算符只能作用于new返回的指针。其格式为：　delete 指针名;

该指针保存的地址必须是new所分配的内存空间首地址。例如：Date * pd = new Date;

```
...
delete pd;
```

顺便指出，delete运算符可作用于NULL型指针，如上例可改写：

```
pd = NULL;
delete pd;
```

用delete运算符撤消由new创建的数组时采用如下格式：

```
delete [ ] 指向数组的指针名;
```

这里所说的数组可以是基本数据类型的数组，或者是用户定义的class类型的对象数组。指针名前只用一对空的方括号，可以忽略方括号内的数字(元素个数)。例如，


```
B * p;
double d;
char s[80];
p = new B[3];
  ...
delete [ ]p;
```

对一个(用new创建的)对象只能使用一次delete操作，否则将出现非法操作。

(3)必须使用new和delete的原因：由于new能自动调用类的构造函数初始化新创建的对象，delete也能自动调用类的析构函数撤消对象，而malloc()和free()则不能。并且new优于malloc()之处还有，它具有自动检测每种对象所需内存空间大小的功能，且返回的void型指针会自动转换成与赋值号左边类型相同的指针。

(4)使用delete和delete[]：

都可以释放new申请分配的内存空间

detele只能触发首个数组元素调用析构函数

delete[]触发所有对象数组元素的析构函数的调用

(5) delete和delete[]之后，要将指针单独赋NULL