# 第13章　数组、指针、引用和动态分配运算符

在第一部分中，就指针和数组与C++内嵌类型的关系进行了讨论。这里，讨论它们与对象的关系。本章也将讨论与指针有关的一个特征，称为引用。最后讨论C++的动态分配运算符。

## 13.1　对象数组

在C++中，可能存在对象数组。声明和使用对象数组的语法与使用其他任何类型的数组完全相同。例如，下面的程序使用了一个含三个元素的对象数组：

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

class cl {
  int i;
public:
  void set_i(int j) {  i=j; }
  int get_i( ) {  return i; }
};

int main( )
{
  cl ob[ 3] ;
  int i;

  for(i=0; i<3; i++) ob[ i] .set_i(i+1);

  for(i=0; i<3; i++)
    cout << ob[ i] .get_i( ) << "\n";

  return 0;
}
```

这个程序在屏幕上显示数字1，2和3。

如果一个类定义了一个带参数的构造函数，则可以像对其他类型的数组一样，通过指定一个初始列表来初始化数组中的每个对象，但初始列表的准确形式则由对象的构造函数所需的参数个数决定。对于其构造函数只带一个参数的对象，可利用通常的数组初始化语法指定一个初值列表。当创建数组中的每个元素时，该列表中的每个值被传给构造函数。例如，下面的程序使用了初始化，和前面的程序稍微有些区别：

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

class cl {
  int i;
public:
  cl(int j) {  i=j; }  // constructor
```

```
  int get_i( ) { return i; }
};

int main( )
{
  c1 ob[ 3] = { 1, 2, 3}; // initializers
  int i;

  for(i=0; i<3; i++)
    cout << ob[ i] .get_i( ) << "\n";

  return 0;
}
```

这个程序在屏幕上也显示数字1，2和3。

实际上，前面程序中所示的初始化语法是下面形式的简写：

```
c1 ob[ 3] = { c1(1), c1(2), c1(3) };
```

这里，显式调用c1的构造函数。当然，程序中所用的简写形式更常见。简写形式之所以工作，是因为应用于只带一个变元的构造函数的自动转换的缘故（参见第12章）。因此，只能使用简写形式来初始化其构造函数只带一个变元的对象数组。

如果对象的构造函数需要两个或多个变元，则必须使用非简写的初始化形式，例如：

```
#include <iostream>
using namespace std;
class c1 {
  int h;
  int i;
public:
  c1(int j, int k) { h=j; i=k; } // constructor with 2 parameters
  int get_i( ) {return i;}
  int get_h( ) {return h;}
};

int main( )
{
  c1 ob[ 3] = {
    c1(1, 2), // initialize
    c1(3, 4),
    c1(5, 6)
  };

  int i;

  for(i=0; i<3; i++) {
    cout << ob[ i] .get_h( );
    cout << ", ";
    cout << ob[ i] .get_i( ) << "\n";
  }

  return 0;
}
```

其中，c1的构造函数有两个参数，因此需要两个变元。也就是说，不能使用简写初始化格

式，而是使用上例所示的“长格式”。

### 13.1.1 创建初始化与未初始化数组

如果要创建初始化和未初始化对象数组，就会发生一种特殊的情况，考虑下面的类：

```
class cl {
  int i;
public:
  cl(int j) { i=j; }
  int get_i( ) { return i; }
};
```

其中，在cl中定义的构造函数需要一个参数。这暗示所声明的这种类型的任何数组都将初始化，也就是说，不允许有如下的数组声明：

```
cl a[ 9]; // error, constructor requires initializers
```

这个语句（正如当前定义的cl）是不合法的，其原因是它暗示cl有一个无参数的构造函数，因为没有指定初值。但是，cl并没有无参数的构造函数。由于没有对应于这个声明的有效构造函数，所以编译器会报出一个错误。要解决这个问题，需要重载一个不带参数的构造函数，如下所示。这样，就允许某些数组初始化，而某些不初始化。

```
class cl {
  int i;
public:
  cl( ) { i=0; } // called for non-initialized arrays
  cl(int j) { i=j; } // called for initialized arrays
  int get_i( ) { return i; }
};
```

根据上面的类，下面的两条语句都是允许的：

```
cl a1[ 3] = { 3, 5, 6}; // initialized
cl a2[ 34]; // uninitialized
```

## 13.2 指向对象的指针

像可以有指向其他类型变量的指针一样，也可以有指向对象的指针。当访问一个给出指向对象的指针的类成员时，使用箭头（->）运算符而不用点（.）运算符。下面的程序说明如何访问一个给出指向对象的指针的对象：

```
#include <iostream>
using namespace std;
class cl {
  int i;
public:
  cl(int j) { i=j; }
  int get_i( ) { return i; }
};
```

```
int main( )
{
  c1 ob(88), *p;

  p = &ob; // get address of ob

  cout << p->get_i( ); // use -> to call get_i( )

  return 0;
}
```

我们知道，当指针递增时，它指向它所指类型的下一个元素。例如，整数指针指向下一个整数。通常，所有指针运算与指针的基本类型有关（即与正在声明的指针所指向的数据类型有关）。指向对象的指针也是如此。例如，下面的程序在被赋予了ob的起始地址后，使用一个指针来访问数组ob的所有三个元素。

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

class c1 {
  int i;
public:
  c1( ) { i=0; }
  c1(int j) { i=j; }
  int get_i( ) { return i; }
};

int main( )
{
  c1 ob[3] = {1, 2, 3};
  c1 *p;
  int i;

  p = ob; // get start of array
  for(i=0; i<3; i++) {
    cout << p->get_i( ) << "\n";
    p++; // point to next object
  }

  return 0;
}
```

可以把一个对象的公有成员的地址赋给一个指针，然后通过这个指针访问该成员。例如，下面是一段有效的C++程序，它在屏幕上显示数字1：

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

class c1 {
public:
  int i;
  c1(int j) { i=j; }
};

int main( )
```

```
{
  cl ob(1);
  int *p;

  p = &ob.i; // get address of ob.i

  cout << *p; // access ob.i via p

  return 0;
}
```

由于p指向一个整数，p被声明为一个整型指针。在这种情况下，i是否是ob的成员是不相关的。

## 13.3 C++指针的类型检查

理解C++语言中的指针时有一点很重要：只有两个指针类型兼容时，才可以将一个指针赋给另一个指针。例如：

```
int *pi;
float *pf;
```

在C++中，下面的赋值是非法的：

```
pi = pf; // error -- type mismatch
```

当然，也可以使用强制类型转换解决所有类型不兼容性问题，但这样做跳过了C++的类型检查机制。

## 13.4 this指针

当调用一个成员函数时，该成员函数被自动传入一个隐含的变元，这个变元是一个指向调用对象（即在其上调用函数的对象）的指针。我们把这个指针称为this。为便于理解this，首先给出一段程序，该程序创建一个称为pwr的类来计算一个数的幂：

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

class pwr {
  double b;
  int e;
  double val;
public:
  pwr(double base, int exp);
  double get_pwr( ) { return val; }
};

pwr::pwr(double base, int exp)
{
  b = base;
  e = exp;
  val = 1;
  if(exp==0) return;
```

```
  for( ; exp>0; exp--) val = val * b;
}

int main( )
{
  pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);

  cout << x.get_pwr( ) << " ";
  cout << y.get_pwr( ) << " ";
  cout << z.get_pwr( ) << "\n";

  return 0;
}
```

在成员函数内，可以直接访问类的成员，而无需任何对象或类。所以，在pwr( )中，语句：

```
b = base;
```

意味着与调用对象相关的b的副本将被赋予包含在base里的值，然而，同样的语句可以改写如下：

```
this->b = base;
```

this 指针指向调用 pwr( )的对象，所以 this->b 指该对象的 b 的副本。例如，如果通过 x 调用 pwr( )[ 如在 x（4.0，2）中 ]，则前面语句中的 this 指向 x。事实上，不使用 this 的语句只是缩写形式。

下面是用 this 指针改写的完整的 pwr( )构造函数：

```
pwr::pwr(double base, int exp)
{
  this->b = base;
  this->e = exp;
  this->val = 1;
  if(exp==0) return;
  for( ; exp>0; exp--)
    this->val = this->val * this->b;
}
```

实际上，C++程序员不会像这样写pwr( )，因为没有任何好处，而且标准形式来得更容易。但是，在重载运算符和成员函数必须使用指向调用它的对象的指针时，this指针是非常重要的。

记住，this 指针被自动传递给所有的成员函数，因此，get_pwr( )可以改成为如下格式：

```
double get_pwr( ) { return this->val; }
```

这样，如果像下面这样调用 get_pwr( )：

```
y.get_pwr( );
```

则 this 指向对象 y。

关于 this 的最后两点：第一，友元函数不是类的成员，所以没有传以 this 指针。第二，静态成员函数没有 this 指针。

## 13.5 指向派生类型的指针

通常，一种类型的指针不能指向另一类型的对象。但对于派生类，这个规则有一个重要的例外。假设有两个类分别为B和D，并且假设D是从基类B派生来的。在这种情况下，类型B的指针*也可以指向类型D的对象。通常，基类的指针可以用做指向任何派生类的指针。

虽然基类指针可用于指向派生类，反过来却不成立。类型D的指针*不能指向类型B的对象。进一步讲，尽管可以用一个基类指针指向一个派生的对象，但只能访问从基类继承的派生类的成员，也就是说，不能访问任何由派生类增加的成员（然而，可以把一个基类指针强制转换为派生指针并访问整个派生类）。

下面这段短程序说明了使用一个基类指针来访问派生对象。

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

class base {
  int i;
public:
  void set_i(int num) { i=num; }
  int get_i( ) { return i; }
};
class derived: public base {
  int j;
public:
  void set_j(int num) { j=num; }
  int get_j( ) { return j; }
};

int main( )
{
  base *bp;
  derived d;

  bp = &d; // base pointer points to derived object

  // access derived object using base pointer
  bp->set_i(10);
  cout << bp->get_i( ) << " ";

/* The following won't work. You can't access elements of
   a derived class using a base class pointer.

   bp->set_j(88); // error
   cout << bp->get_j( ); // error

*/
  return 0;
}
```

可以看出，一个基类指针可以用来访问派生类的对象。

尽管必须小心，但是把基类指针强制转换为派生类的指针，以通过基类指针访问派生类的成员是可能的。例如，下面是有效的C++代码：

```
// access now allowed because of cast
((derived *)bp)->set_j(88);
cout << ((derived *)bp)->get_j( );
```

重要的是记住，指针运算与指针的基类型有关。因此，当一个基类指针指向一个派生对象时，递增指针不能使之指向派生类型的下一个对象。相反，它将指向基类型的下一个对象。这当然会引发错误。例如，下面的程序从语法上讲是正确的，但却含有这种错误：

```
// This program contains an error.
#include <iostream>
using namespace std;

class base {
   int i;
public:
   void set_i(int num) { i=num; }
   int get_i( ) { return i; }
};

class derived: public base {
   int j;
public:
   void set_j(int num) {j=num;}
   int get_j( ) {return j;}
};

int main( )
{
   base *bp;
   derived d[2];

   bp = d;

   d[0].set_i(1);
   d[1].set_i(2);

   cout << bp->get_i( ) << " ";
   bp++; // relative to base, not derived
   cout << bp->get_i( ); // garbage value displayed

   return 0;
}
```

在通过虚函数机制建立运行时多态性时，用基类指针指向派生类型是最有用的（参见第17章）。

## 13.6 指向类成员的指针

C++允许产生一种特殊类型的指针，它一般指向类的成员，而不是指向对象中该成员的特定实例。这种指针称为指向类成员的指针或简称为指向成员的指针。指向成员的指针不同于普通的C++指针，它只提供一个在成员所属类中对象的偏移量。由于成员指针不是真正的指针，所以运算符.和->不适用于它们。为了访问有指针指向的类的成员，必须使用特殊的指向成员的指针运算符.*和->*。它们的作用是允许访问有指针指向那个成员的类成员。

下面是一个例子：

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

class cl {
public:
  cl(int i) { val=i; }
  int val;
  int double_val( ) { return val+val; }
};

int main( )
{
  int cl::*data; // data member pointer
  int (cl::*func)( ); // function member pointer
  cl ob1(1), ob2(2); // create objects

  data = &cl::val; // get offset of val
  func = &cl::double_val; // get offset of double_val( )

  cout << "Here are values: ";
  cout << ob1.*data << " " << ob2.*data << "\n";

  cout << "Here they are doubled: ";
  cout << (ob1.*func)( ) << " ";
  cout << (ob2.*func)( ) << "\n";

  return 0;
}
```

在main( )中，这个程序创建了两个成员指针：data和func。注意每个声明的语法。当声明指向成员的指针时，必须指定类并使用作用域分辨符（scope resolution operator）。程序同时也创建了cl的对象ob1和ob2。正如程序所表明的，程序指针既可以指向函数、也可以指向数据。然后，程序获得val和double_val( )的地址。如前所述，这些地址实际上只是在类型cl的对象中的相对位移，通过它们可以找到val和double_val( )。接着，要显示每个对象的val值，则通过data访问它们每一个。最后，程序用func调用函数double_val( )。为了正确地关联运算符.*，附加的括号是必要的。

当通过使用对象或引用访问对象的成员时，必须使用运算符.*（在本章后面讨论）。如果使用指向对象的指针，则应使用运算符->*。正如下面的程序所示的那样，下面的程序是前一个程序修改后的版本。

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

class cl {
public:
  cl(int i) { val=i; }
  int val;
  int double_val( ) { return val+val; }
};

int main( )
```

```
{
  int cl::*data; // data member pointer
  int (cl::*func)( ); // function member pointer
  cl ob1(1), ob2(2); // create objects
  cl *p1, *p2;

  p1 = &ob1; // access objects through a pointer
  p2 = &ob2;

  data = &cl::val; // get offset of val
  func = &cl::double_val; // get offset of double_val( )

  cout << "Here are values: ";
  cout << p1->*data << " " << p2->*data << "\n";

  cout << "Here they are doubled: ";
  cout << (p1->*func)( ) << " ";
  cout << (p2->*func)( ) << "\n";

  return 0;
}
```

在这个程序中，p1和p2是指向类型cl的对象的指针，所以，使用运算符->*访问val和double_val( )。

记住，指向成员的指针和指向对象元素的特定实例的指针是不同的。例如，考虑下面的程序段（假设cl是按上面的程序声明的）：

```
int cl::*o;
int *p;
cl o;

p = &o.val // this is address of a specific val

d = &cl::val // this is offset of generic val
```

其中，p是指向特定对象中一个整数的指针，而d只是一个偏移量，该偏移量表明val将在类型cl的任何对象中的哪个位置出现。

通常，成员指针运算符应用于特殊情况，不用于日常的程序设计中。

## 13.7 引用

C++包含一个与指针有关的特征，称为“引用”。实质上，引用是一个隐含的指针。使用引用的方式有三种：作为函数参数，作为函数返回值，或作为单个引用。下面依次讨论它们。

### 13.7.1 引用参数

或许引用的一个最重要用途是，允许用户创建自动使用按引用调用方式进行参数传递的函数。正如在第6章中解释的，变元可以按两种方式之一传递给函数：使用按值调用或按引用调用。当使用按值调用时，变元的副本传递给函数。按引用调用把变元的地址传给函数。默认时，C++使用按值调用，但是它提供了两种方法来得到按引用调用的参数传递。首先，可以显式地把一个指针传递给变元。第二，可以使用引用参数。大多数情况下，最好使用引用参数。

要理解什么是引用参数及为什么它是有价值的，让我们通过回顾如何使用指针参数生成按引用调用开始。下面的程序在neg( )函数中使用一个指针手工创建一个按引用调用的参数，函数neg( )使变元所指的整型变量的符号颠倒。

```
// Manually create a call-by-reference using a pointer.
#include <iostream>
using namespace std;

voic neg(int *i);

int main( )
{
  int x;

  x = 10;
  cout << x << " negated is ";

  neg(&x);
  cout << x << "\n";

  return 0;
}

void neg(int *i)
{
  *i = -*i;
}
```

在这个程序中，neg( )有一个指向整数的指针作为参数，其中的整数符号要求反。所以，neg( )必须用x的地址明确地调用。此外，在neg( )内，必须使用运算符*来访问i所指的变量。这就是在C++中生成一个“手工的”按引用调用的方式，也是使用C子集获得按引用调用的惟一方式。幸运的是，在C++中，可以通过使用一个引用参数自动实现这个特征。

为了创建一个引用参数，在参数的名字前加上&即可。下面是如何引用声明neg( )的例子，其中i被声明为一个引用参数：

```
void neg(int &i);
```

实际上，这导致在用任何变元调用neg( )时i变成了另一个名字。任何适用于i的操作实际上影响调用变元。从技术上讲，i是一个隐含指针，自动引用调用neg( )时所使用的变元。一旦i成为一个引用，就没有必要再使用运算符*了（即使是合法的）。相反，每次使用i时，它都隐含着对变元的一个引用，对i所做的任何改变都会影响这个变元。还有，在调用neg( )时，就不必（即使合法）在变元名前加运算符&了，编译器会自动完成这项工作。下面是前一个程序的引用形式：

```
// Use a reference parameter.
#include <iostream>
using namespace std;

void neg(int &i); // i now a reference

int main( )
{
  int x;
```

```
  x = 10;
  cout << x << " negated is ";

  neg(x); // no longer need the & operator
  cout << x << "\n";

  return 0;
}

void neg(int &i)
{
  i = -i; // i is now a reference, don't need *
}
```

总之，当创建一个引用参数时，这个参数将自动引用（隐式地指向）用于调用函数的变元。所以，在前面的程序中，语句

```
i = -i ;
```

实际上是对x操作，而不是对x的副本。没有必要再把运算符&用于变元了。同时，在函数中，引用参数无需用运算符*就可以直接使用。通常，当把一个值赋给一个引用时，实际上是把该值赋给引用所指的变量。

在函数里，改变引用参数所指的内容是不可能的，也就是说，像如下 neg( )中的语句

```
i++;
```

递增在调用中使用的变量的值，它不能使i指向某个新的位置。

下面是另一个例子，程序用引用参数交换调用时所带变量的值。swap( )函数是典型的按引用调用方式进行参数传递的例子。

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

void swap(int &i, int &j);

int main( )
{
  int a, b, c, d;

  a = 1;
  b = 2;
  c = 3;
  d = 4;

  cout << "a and b: " << a << " " << b << "\n";
  swap(a, b); // no & operator needed
  cout << "a and b: " << a << " " << b << "\n";

  cout << "c and d: " << c << " " << d << "\n";
  swap(c, d);
  cout << "c and d: " << c << " " << d << "\n";

  return 0;
}

void swap(int &i, int &j)
```

```
{
  int t;

  t = i; // no * operator needed
  i = j;
  j = t;
}
```

程序显示如下：

```
a and b: 1 2
a and b: 2 1
c and d: 3 4
c and d: 4 3
```

### 13.7.2 向对象传递引用

在第12章中，我们讲述过，当对象作为变元传给函数时，就创建了该对象的一个副本。当函数结束时，要调用副本的析构函数。然而，通过引用传递时，不创建对象的副本。这意味着当函数结束时，没有作为参数的对象被撤销，也不调用参数的析构函数。例如，试试下面的程序：

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

class cl {
  int id;
public:
  int i;
  cl(int i);
  ~cl( );
  void neg(cl &o) { o.i = -o.i; } // no temporary created
};

cl::cl(int num)
{
  cout << "Constructing " << num << "\n";
  id = num;
}

cl::~cl( )
{
  cout << "Destructing " << id << "\n";
}

int main( )
{
  cl o(1);

  o.i = 10;
  o.neg(o);

  cout << o.i << "\n";

  return 0;
}
```

下面是这个程序的输出：

```
Constructing 1
-10
Destructing 1
```

可以看到，c1 的析构函数只被调用了一次。如果 o 是通过值传递的，在 neg( )里就会再创建一个对象，并且在 neg( )结束、撤销对象时，将再一次调用析构函数。

正如 neg( )中的代码所演示的，当通过一个引用访问类的成员时，使用点号运算符。只有指针才使用箭头运算符。

当通过引用传递参数时，对函数内部对象的改变将影响调用的对象。

还有，通过引用传递除了最小的对象以外的所有对象要比按值传递它们要快。变元通常传递到堆栈上。因此，要把大对象推进堆栈或从堆栈中弹出，需要花费相当多的 CPU 周期。

### 13.7.3　返回引用

函数可以返回引用。这对于允许函数用在赋值语句的左边有令人吃惊的效果。例如，考虑这个简单程序：

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

char &replace(int i); // return a reference

char s[80] = "Hello There";

int main( )
{
  replace(5) = 'X'; // assign X to space after Hello

  cout << s;

  return 0;
}

char &replace(int i)
{
  return s[i];
}
```

这个程序把 Hello 和 There 之间的空格用 X 替换，即程序显示 HelloXthere。看一看它是如何实现的。首先，replace( )被声明为返回一个字符的引用。replace( )编码时，它返回一个通过变元 i 指定的 s 元素的引用，然后在 main( )中使用 replace( )返回的引用把字符 X 赋给那个元素。

### 13.7.4　独立引用

虽然引用通常的用途是使用按引用调用来传递变元和用做函数的返回值，但也可以声明作为一个简单变量的引用，这种类型的引用称为独立引用。

当创建独立引用时，创建的只是对象的另一个名字。所有的独立引用必须在创建时初始化，其原因很容易理解。除初始化以外，用户不能改变引用变量所指的对象，所以，它在声明时就必须初始化（在 C++ 中，初始化和赋值是两个完全不同的操作）。

下面的程序说明了一个独立引用：

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

int main( )
{
  int a;
  int &ref = a; // independent reference

  a = 10;
  cout << a << " " << ref << "\n";

  ref = 100;
  cout << a << " " << ref << "\n";

  int b = 19;
  ref = b; // this puts b's value into a
  cout << a << " " << ref << "\n";

  ref--; // this decrements a
         // it does not affect what ref refers to

  cout << a << " " << ref << "\n";

  return 0;
}
```

该程序显示如下输出：

```
10 10
100 100
19 19
18 18
```

由于每一个独立引用其实就是另一个变量的另一个名字，所以独立引用实际上没有什么价值。如果用两个名字来描述同一个对象，很可能程序就会变得杂乱无章。

### 13.7.5 派生类型的引用

类似于前面描述指针时的情形，可以使用基类引用来引用派生类的对象，其最常见的应用出现于函数参数中。一个基类引用参数可以接受基类对象和从那个基类派生出的任何其他类型。

### 13.7.6 对引用的限制

引用有几个限制。不能引用别的引用，换句话说，不能获得引用的地址，不能创建引用数组，不能创建指向引用的指针，不能引用位域。

如果引用变量不是类的成员、函数参数或返回值，在声明引用变量时必须初始化它。此外，要禁止空引用。

## 13.8 格式问题

在声明指针和引用变量时，有些C++程序员使用一种特殊的编码形式，即将*或&和类型名而不是变量联系在一起。例如，下面是两个功能一样的声明。

```
int& p; // & associated with type
int &p; // & associated with variable
```

把*或&和类型名联系起来，反映了一些C++程序员想得到一种单独的指针类型的愿望。但把&或*和类型名而不是变量联在一起带来的烦恼是，根据通常的C++语法，&或*都不能分配给变量表。所以，很容易创建错误的声明。例如，下面的声明创建了一个而不是两个整型指针。

```
int* a, b;
```

这里，b被声明为一个整型数（不是整型指针），因为根据C++语法的规定，当*（或&）用在声明中时，它只和其后的单个变量相联，而与其他变量无关。这个声明的麻烦之处是，尽管a和b给人的感觉都是指针类型，但实际上只有a是指针。这不仅使初学C++的程序员感到困惑，而且老手偶尔也会这样。

重要的是要理解，只要在C++编译器设计的范围之内，写int*p和int*p是无关紧要的。所以，如果愿意把*或&和类型名而不是变量联系起来，就只管这样做好了。但为避免误解，本书将继续把&和*与它们修饰的变量而不是类型名联在一起。

## 13.9 C++的动态分配运算符

C++提供了两个动态分配运算符：new和delete。使用这两个运算符来在运行时分配和释放内存。动态分配是几乎所有实际程序的重要组成部分。正如在第一部分中解释的，C++也支持动态内存分配函数malloc( )和free( )。这是为了保持与C语言的兼容性而包括的。然而，对C++代码来说，应该使用new和delete运算符，因为它们有几个优点。

运算符new分配内存并返回指向分配的内存开始处的指针。delete运算符释放new分配的内存。new和delete的一般格式如下：

```
p_var = new type;
delete p_var;
```

其中，p_var是一个指针变量，它接受指向足够能容纳类型type的内存的指针。

因为堆是有限的，所以可能耗尽。如果没有足够的内存来满足分配请求，那么new将失败并生成bad_alloc异常。这个异常是在头文件< new>中定义的。程序应该处理这个异常，并且如果出现错误，应采取适当的行动（异常处理将在第19章描述）。如果程序没有处理这个异常，那么程序将会终止运行。

刚才所描述的new在失败时所采取的行动是由标准C++规定的。麻烦之处是，并不是所有的编译器，尤其是早期的那些，会实现与标准C++一致的new。当刚发明C++时，失败时new返回空值。后来，改变了这一点，以便在失败时new引发一个异常。最后，虽然规定默认时new的失败将生成一个异常，但是可以选择返回一个空指针。因此，在不同时间，不同的编译器制造商所实现的new是不同的。尽管所有的编译器最终都会实现与标准C++一致的new，现在知道失败时new所采取的行动的惟一途径是查看编译器文档。因为标准C++规定new在失败时生成一个异常，所以本书中代码在编写时也采用这种方式。如果你的编译器处理失败的方式不同于此，需要做出相应的改变。

下面是一个分配容纳一个整数的内存的程序：

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

int main( )
{
  int *p;

  try {
    p = new int; // allocate space for an int
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  *p = 100;

  cout << "At " << p << " ";
  cout << "is the value " << *p << "\n";

  delete p;

  return 0;
}
```

这个程序把大得足以容纳一个整数的堆中的地址赋给p。然后，把值100赋给那个内存并在屏幕上显示内存的内容。最后，释放动态分配的内存。记住，如果编译器实现new以便它在失败时返回空值，那么必须对前面的程序做出相应的修改。

运算符delete必须和先前用new分配的有效指针一起使用。和delete一起使用其他任何类型的指针并不受到支持，几乎总是带来严重问题，如系统崩溃等。

虽然new和delete完成和malloc( )和free( )相似的功能，但它们有几个优点。第一，new自动分配足够的空间以容纳一个指定类型的对象，而不必使用sizeof运算符。由于容量是自动计算的，所以消除了在这方面发生错误的可能性。第二，new自动返回指定类型的指针，不必像用malloc( )分配内存时需要显式地使用强制类型转换。第三，new和delete都可以重载，允许创建自定义的分配系统。

尽管没有正式的规则来说明这一点，在同一个程序中最好不要把new和delete与malloc( )和free( )混合使用，因为并没有保证它们是互相兼容的。

### 13.9.1 初始化动态分配的内存

可以通过在new语句中类型名后面放一个初值来把分配的内存初始化为某一可知的值。下面是包括初始化的new的一般形式：

*p_var* = new *var_type (initializer);*

当然，初始化部分的类型必须与内存分配的数据类型兼容。

下面的程序赋给分配的整数一个初值87：

```
#include <iostream>
#include <new>
```

```
using namespace std;

int main( )
{
  int *p;

  try {
    p = new int (87); // initialize to 87
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  cout << "At " << p << " ";
  cout << "is the value " << *p << "\n";

  delete p;

  return 0;
}
```

### 13.9.2　分配数组

可以用 new 以如下形式分配数组：

*p_var* = new *array_type [size];*

其中，size 规定了数组中的元素个数。

要释放数组，用下列 delete 形式：

delete [ ] *p_var;*

其中，[ ]通知 delete 要释放一个数组。

例如，下面的程序分配一个有 10 个元素的整数数组。

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

int main( )
{
  int *p, i;

  try {
    p = new int [10]; // allocate 10 integer array
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  for(i=0; i<10; i++ )
    p[ i] = i;

  for(i=0; i<10; i++)
    cout << p[ i] << " ";

  delete [] p; // release the array
```

```
  return 0;
}
```

注意delete语句。当释放用new分配的数组时，必须告诉delete用[ ]释放一个数组（在下一节可以看出，在分配对象数组时这一点尤其重要）。

分配数组时，有一个限制：不能赋给初值，即分配数组时不能指定初值。

### 13.9.3 分配对象

可以用new动态地分配对象。这样做之后，就创建了一个对象并返回指向它的指针。动态创建的对象和任何其他对象是一样的。创建对象时，调用它的构造函数（如果有的话）；释放对象时，执行它的析构函数。

下面的程序创建了一个称为balance的类，它把一个人的姓名和他的账目收支情况联系起来。在main( )中，动态创建了一个类型为balance的对象。

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

class balance {
  double cur_bal;
  char name[80];
public:
  void set(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }

  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main( )
{
  balance *p;
  char s[80];
  double n;

  try {
    p = new balance;
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  p->set(12387.87, "Ralph Wilson");

  p->get_bal(n, s);

  cout << s << "'s balance is: " << n;
  cout << "\n";
```

```
  delete p;

  return 0;
}
```

由于 p 包含指向对象的指针，所以要用箭头运算符 ->访问对象的成员。

前面已经提到，动态分配的对象可以有构造函数和析构函数。同时，构造函数也可以带参数。下面是前一个程序的另一版本。

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

class balance {
  double cur_bal;
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  ~balance( ) {
    cout << "Destructing ";
    cout << name << "\n";
  }
    void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main( )
{
  balance *p;
  char s[80];
  double n;
  // this version uses an initializer
  try {
    p = new balance (12387.87, "Ralph Wilson");
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
  }

  p->get_bal(n, s);

  cout << s << "'s balance is: " << n;
  cout << "\n";

  delete p;

  return 0;
}
```

注意对象的构造函数的参数放在类型名后面，就像其他类型的初始化一样。

可以分配对象数组，但有一点值得注意。由于用new分配的数组不能带初始值，所以必须确保如果类包含构造函数，有一个将没有参数。如果不这样，当尝试分配数组时，C++编译器就找不到与之匹配的构造函数，也不编译用户程序。

下面是前一个程序的另一版本，它分配了balance对象的数组，并调用了无参数的构造函数：

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

class balance {
  double cur_bal;
  char name[80];
public:
  balance(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  balance( ) {} // parameterless constructor
    ~balance( ) {
    cout << "Destructing ";
    cout << name << "\n";
  }
  void set(double n, char *s) {
    cur_bal = n;
    strcpy(name, s);
  }
  void get_bal(double &n, char *s) {
    n = cur_bal;
    strcpy(s, name);
  }
};

int main( )
{
  balance *p;
  char s[80];
  double n;
  int i;

  try {
    p = new balance [3]; // allocate entire array
  } catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
}

  // note use of dot, not arrow operators
  p[0].set(12387.87, "Ralph Wilson");
```

```
  p[1].set(144.00, "A. C. Conners");
  p[2].set(-11.23, "I. M. Overdrawn");

  for(i=0; i<3; i++) {
    p[i].get_bal(n, s);

    cout << s << "'s balance is: " << n;
    cout << "\n";
  }

  delete [] p;
  return 0;
}
```

这个程序的输出如下所示：

```
Ralph Wilson's balance is: 12387.9
A. C. Conners's balance is: 144
I. M. Overdrawn's balance is: -11.23
Destructing I. M. Overdrawn
Destructing A. C. Conners
Destructing Ralph Wilson
```

在删除动态分配对象的数组时，要使用delete[ ]形式，目的是使数组中每个对象均可以调用析构函数。

### 13.9.4 nothrow替代形式

在标准C++中，当分配失败时，可以让new返回空值而不是抛出一个异常。在使用现代C++编译器编译老的代码时，这种形式的new是更有用的。当用new代替对malloc( )的调用时，它也是很有价值的（当把C代码升级为C++代码时，这是很常见的）。这种形式的new如下所示：

*p_var* = new(nothrow) *type*;

其中，p_var是type类型的指针变量。new的nothrow形式的工作方式与几年前new的最初版本的工作方式相同。因为它在失败时返回空值，所以可以把它“放到”老代码中，而无需添加异常处理。然而，对于新代码，异常是更好的方式。要使用nothrow选项，必须包括头文件<new>。

下面的程序显示了如何使用new(nothrow)替代形式。

```
// Demonstrate nothrow version of new.
#include <iostream>
#include <new>
using namespace std;

int main( )
{
  int *p, i;

  p = new(nothrow) int[32]; // use nothrow option
  if(!p) {
    cout << "Allocation failure.\n";
    return 1;
```

```
    }
    for(i=0; i<32; i++) p[i] = i;
    for(i=0; i<32; i++) cout << p[i] << " ";
    delete [] p; // free the memory
    return 0;
}
```

如这个程序所示，当使用nothrow方法时，在每次分配请求后，必须检查由new返回的指针。

### 13.9.5 new的放置形式

有一种特殊形式的new，称为放置形式（placement form），可用于指定分配内存的另一种方法。在某些特殊场合，当重载new运算符时，它是特别有用的。下面是它的一般形式：

*p_var = new (arg-list) type;*

其中，arg-list是一个用逗号分隔的值列表，用于传递给new的重载形式。