第 2 周课程简介

至此，读者已经学完了有关如何使用 C++进行编程的第 1 周课程，应该能够熟练地输入程序和使用编译器，并熟悉对象、类和程序流程。

本周内容简介

本周首先介绍指针。对于 C++新手来说，指针通常是一个难以理解的主题，但第 8 章对其进行了全面、清晰的讨论，它们不再是你的绊脚石；第 9 章介绍与指针非常相似的引用；第 10 章将讨论如何重载函数。

第 11 章是一个分水岭：不再将重点放在 C++语言的语法上，而讨论面向对象分析与设计；第 12 章介绍一个面向对象编程的基本概念：继承；第 13 章介绍如何使用数组和集合；第 14 章介绍多态，对第 12 章的内容进行拓展。

第 8 章　理解指针

对于 C++程序员来说，一个强大而低级的工具是，可以使用指针来直接操纵计算机内存。这是 C++相对于诸如 C#和 Visual Basic 等其他语言的优点之一。

本章介绍如下内容：

- 什么是指针？
- 如何声明和使用指针？
- 什么是自由存储（free store）以及如何操纵内存？

学习 C++时，指针带来了两方面的挑战：它们令人迷惑；为何需要它们的原因不那么明显。本章将循序渐进地介绍指针的工作原理。然而，只有通过继续阅读本书，读者才能完全地理解为什么需要使用指针。

注意：能够使用指针以及在低层操纵内存，是 C++被选择用于编写嵌入式和实时应用程序的原因之一。

8.1　什么是指针

指针是存储内存地址的变量，就这么简单。如果读者明白了这句话，就理解了有关指针的核心知识。

8.1.1　内存简介

要理解指针，必须对计算机内存有一定的了解。计算机内存被划分成按顺序编号的内存单元。每个变量都位于某个独特的内存单元中，这被称为地址。图 8.1 说明了名为 theAge 的 unsigned long 变量在内存中的存储情况。

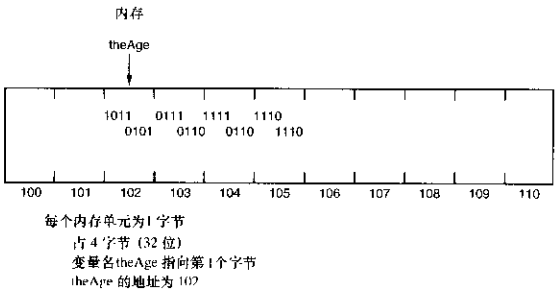

图 8.1　theAge 的存储情况

8.1.2　获取变量的内存地址

不同的计算机使用不同的复杂方案对内存进行编号。通常，程序员不需要知道变量的具体地址，因为编

译器负责处理细节。然而，要获得变量的内存地址，可以使用地址运算符&，它返回对象在内存中的地址。程序清单 8.1 演示了这种运算符的用法。

程序清单 8.1　地址运算符

```
1:  // Listing 8.1 Demonstrates address-of operator
2:  // and addresses of local variables
3:  #include <iostream>
4:
5:  int main ()
6:  {
7:     using namespace std;
8:     unsigned short shortVar=5;
9:     unsigned long  longVar=65535;
10:    long sVar = -65535;
11:
12:    cout << "shortVar:\t" << shortVar;
13:    cout << "\tAddress of shortVar:\t";
14:    cout <<  &shortVar  << endl;
15:
16:    cout << "longVar:\t"  << longVar;
17:    cout  << "\tAddress of longVar:\t" ;
18:    cout <<  &longVar  << endl;
19:
20:    cout << "sVar:\t\t"    << sVar;
21:    cout << "\tAddress of sVar:\t" ;
22:    cout <<  &sVar    << endl;
23:
24:    return 0;
25: }
```

输出：

```
shortVar:       5       Address of shortVar:    0012FF7C
longVar:        65535   Address of longVar:     0012FF78
sVar:           -65535  Address of sVar:        0012FF74
```

读者运行该程序时，输出可能与此不同，尤其是最后一列。

分析：

声明并初始化了 3 个变量：第 8 行的 unsigned short 变量，第 9 行的 unsigned long 变量，第 10 行的 long 变量。第 12～22 行打印它们的值和地址。在第 14、18 和 22 行，使用地址运算符&来获得变量的地址。只需将该运算符放在变量名的前面，就可以返回变量的地址。

第 12 行打印 shortVar 的值，结果为 5，与预期的相同。从输出中的第 1 行可知，该变量的地址为 0012FF7C，这是在一台奔腾（32 位）计算机上运行该程序得到的。该地址随计算机而异，且每次运行该程序时都可能有所不同。读者的结果将与此不同。

当你声明变量时，编译器将根据其类型决定为它分配多少内存。编译器负责为变量分配内存，并自动给它们指定地址。例如，long 变量通常占用 4 字节，因此使用一个指向 4 字节内存的地址。

注意：你的编译器可能总是给新变量分配 4 字节整数倍的内存，因此 longVar 的地址可能比 shortVar 的地址大 4，虽然 shortVar 只需要 2 字节。

8.1.3　将变量的地址存储到指针中

每个变量都有地址。即使不知道变量的具体地址，也可以将其地址存储到指针中。

例如，假设 howOld 是一个整型变量。要声明一个名为 pAge 的指针来存储它的地址，可以这样做：

```
int *pAge=0;
```

该语句将 pAge 声明为一个 int 指针。也就是说，pAge 被声明为用于存储整型变量的地址。

注意，pAge 是一个变量。当你声明（类型为 int）整型变量时，编译器将分配足以存储一个整数的内存。当你声明诸如 pAge 等指针变量时，编译器将分配足以存储一个地址的内存（在大多数计算机上为 4 字节）。指针（如 pAge）只不过是另一种类型的变量而已。

8.1.4 指针名

由于指针也是变量，因此可以使用任何合法的变量名，有关变量的命名规则和建议也适用于指针名。很多程序员采用这样的命名规则：所有指针名都以 p 打头，如 pAge 和 pNumber。

在下面的例子中：

```
int *pAge = 0;
```

pAge 被初始化为 0。值为 0 的指针被称为空指针。所有指针在创建时都应该进行初始化。如果不知道要将什么值赋给指针，将 0 赋给它。没有被初始化的指针被称为失控指针（wild pointer），因为你不知道它指向的什么——可能指向任何东西！失控指针是非常危险的。

注意：一定要初始化所有的指针。

要让指针存储一个地址，必须将该地址赋给它。在前面的范例中，必须将 howOld 的地址赋给 pAge，如下所示：

```
unsigned short int howOld = 50;     // make a variable
unsigned short int * pAge = 0;      // make a pointer
pAge = &howOld;                     // put howOld's address in pAge
```

第 1 行创建了一个名为 howOld 的变量（其类型为 unsigned short int），并将其初始化为 50；第 2 行将 pAge 声明为一个 unsigned short int 指针，并将其初始化为 0。由于变量类型后和变量名之间有一个星号（*），你知道 pAge 是一个指针。

第 3 行将 howOld 的地址赋给指针 pAge。你之所以知道赋给 pAge 的是 howOld 的地址，是因为其中使用了地址运算符&。如果不使用地址运算符，赋给的将是 howOld 的值。这个值可能是有效的地址，也可能是无效地址。

现在，pAge 的值为 howOld 的地址，而 howOld 的值为 50。可以采取更少的步骤来完成上述工作，如下所示：

```
unsigned short int howOld = 50;          // make a variable
unsigned short int * pAge = &howOld;     // make pointer to howOld
```

8.1.5 获取指针指向的变量的值

使用 pAge，可以获取 howOld 的值（这里为 50）。通过指针 pAge 访问变量的值被称为间接访问，因为这是通过指针间接地访问变量。例如，可以通过指针 pAge 间接地访问 howOld 的值。

间接访问意味着访问位于指针存储的地址处的值。指针提供了一种间接方式来获取存储在地址处的值。

注意：常规变量的类型告诉编译器，需要多少内存来存储这个变量的值；而指针的类型没有这样的功能，所有指针的长度都相同：在使用 32 位处理器的计算机上为 4 字节，在使用 64 位处理器的计算机上为 8 字节。指针的类型告诉编译器，它存储的地址处的对象需要多少内存。

下面的声明将 pAge 声明为一个 unsigned short int 指针：

```
unsigned short int * pAge = 0;      // make a pointer
```

这告诉编辑器，该指针（为存储地址需要 4 字节）存储一个类型为 unsigned short int 的对象的地址，该

对象本身需要 2 字节。

8.1.6　使用间接运算符解除引用

间接运算符（*）也被称为解除引用（dereference）运算符。对指针解除引用时，将得到指针存储的地址处的值。

常规变量让你能够直接访问它的值。要创建一个名为 yourAge 的 unsigned short int 变量，并将 howOld 的值赋给它，可以这样做：

```
unsigned short int yourAge;
yourAge = howOld;
```

指针让你能够间接地访问其指向的变量的值。要通过指针将 howOld 的值赋给变量 yourAge，可以这样做：

```
unsigned short int yourAge;
yourAge = *pAge;
```

指针变量 pAge 前面的间接运算符（*）的含义是：存储在该地址处的值。这条赋值语句的含义是：将存储在 pAge 中的地址处的值赋给 yourAge。如果不使用间接运算符：

```
yourAge = pAge;  // bad!!
```

则是试图将 pAge 的值（一个内存地址）赋给 yourAge。编译器很可能会对此提出警告：指出你可能在犯错。

星号的其他用途

用于指针时，星号的用途有两种：声明指针和用作解除引用运算符。

声明指针时，*是声明的组成部分，位于被指向的对象的类型后面，例如：

```
// make a pointer to an unsigned short
unsigned short * pAge = 0;
```

对指针解除引用时，解除引用（间接）运算符指出要访问指针指向的内存单元中的值，而不是地址本身。

```
// assign 5 to the value at pAge
*pAge = 5;
```

另外，星号（*）还可用作乘法运算符。编译器能够根据你使用星号（上下文）知道该调用哪个运算符。

8.1.7　指针、地址和变量

对指针、其存储的地址以及其存储的地址处的值进行区分至关重要。这是人们对指针感到迷惑的主要根源。

请看下面的代码段：

```
int theVariable = 5;
int *pPointer = &theVariable;
```

theVariable 被声明为一个 int 变量，并被初始化为 5；pPointer 被声明为一个 int 指针，并被初始化为 theVariable 的地址。pPointer 是一个指针，存储的是 theVariable 的地址。pPointer 存储的地址处的值为 5。图 8.2 说明了 theVariable 和 pPointer。

在图 8.2 中，5 存储在地址为 101 的内存单元中。5 的二进制表示如下：

```
0000 0000 0000 0101
```

这个二进制数长两字节（16 位），对应的十进制数为 5。

指针变量位于存储在内存单元 106，其值如下：

```
000 0000 0000 0000 0000 0000 0110 0101
```

这是十进制数 101 的二进制表示，它是变量 theVariable 的地址，该变量的值为 5。
这里的内存布局是示意性的，但它说明了指针是如何存储地址的。

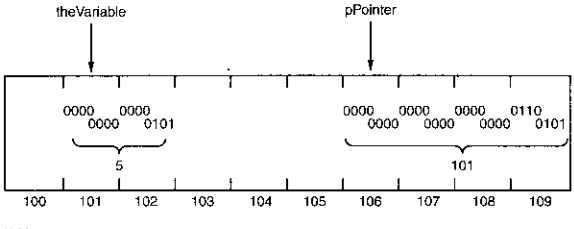

图 8.2 theVariable 和 pPointer 在内存中的情况

8.1.8 使用指针来操纵数据

除使用间接运算符来查看存储在指针指向的位置处的数据外，还可以对数据进行操纵。将变量的地址赋给指针后，可以使用该指针来访问它指向的变量中的数据。

程序清单 8.2 使用了刚才介绍的全部有关的指针，它演示了如何将一个局部变量的地址赋给指针以及如何使用指针和间接运算符来操纵该变量的值。

程序清单 8.2 使用指针来操纵数据

```
// Listing 8.2 Using pointers
#include <iostream>

typedef unsigned short int USHORT;

int main()
{

  using namespace std;

  USHORT myAge;        // a variable
  USHORT * pAge = 0;   // a pointer

  myAge = 5;

  cout << "myAge: " << myAge << endl;
  pAge = &myAge;     // assign address of myAge to pAge
  cout << "*pAge: " << *pAge << endl << endl;

  cout << "Setting *pAge = 7... " << endl;
  *pAge = 7;         // sets myAge to 7

  cout << "*pAge: " << *pAge << endl;
  cout << "myAge: " << myAge << endl << endl;

  cout << "Setting myAge = 9... " << endl;
  myAge = 9;

```

```
29:    cout << "myAge: " << myAge << endl;
30:    cout << "*pAge: " << *pAge << endl;
31:
32:    return 0;
33: }
```

输出：

```
myAge: 5
*pAge: 5

Setting *pAge = 7...
*pAge: 7
myAge: 7

Setting myAge = 9...
myAge: 9
*pAge: 9
```

分析：

该程序声明了两个变量：unsigned short 变量 myAge 和 unsigned short 指针 pAge。第 14 行将 5 赋给 myAge，第 16 行的打印输出验证了这一点。

第 17 行将 myAge 的地址赋给 pAge。第 18 行使用间接运算符（*）对 pAge 解除引用并打印，结果表明，pAge 存储的地址处的值是存储在 myAge 中的 5。

第 21 行将 7 赋给 pAge 指向的变量，这相当于将 myAge 的值设置为 7，第 23 和 24 行证明了这一点。间接访问变量是使用星号（间接运算符）实现的。

第 27 行将 9 赋给变量 myAge。第 29 行直接获取这个值，而第 30 行间接（通过对 pAge 解除引用）获得这个值。

8.1.9　查看地址

指针让你能够操纵地址，而无需知道其实际值。将变量的地址赋给指针时，指针的值将为变量的地址，阅读本章后，读者将对此深信不疑。程序清单 8.3 说明了这一点。

程序清单 8.3　确定指针中存储的内容

```
 1: // Listing 8.3
 2: // What is stored in a pointer.
 3: #include <iostream>
 4:
 5: int main()
 6: {
 7:    using namespace std;
 8:
 9:    unsigned short int myAge = 5, yourAge = 10;
10:
11:    // a pointer
12:    unsigned short int * pAge = &myAge;
13:
14:    cout << "myAge:\t" << myAge
15:       << "\t\tyourAge:\t" << yourAge << endl;
16:
17:    cout << "&myAge:\t" << &myAge
```

```
18:       << "\t&yourAge:\t" << &yourAge << endl;
19:
20:    cout << "pAge:\t" << pAge << endl;
21:    cout << "*pAge:\t" << *pAge << endl;
22:
23:
24:    cout << "\nReassigning: pAge = &yourAge..." << endl << endl;
25:    pAge = &yourAge;       // reassign the pointer
26:
27:    cout << "myAge:\t" << myAge <<
28:       "\t\tyourAge:\t" << yourAge << endl;
29:
30:    cout << "&myAge:\t" << &myAge
31:       << "\t&yourAge:\t" << &yourAge << endl;
32:
33:    cout << "pAge:\t" << pAge << endl;
34:    cout << "*pAge:\t" << *pAge << endl;
35:
36:    cout << "\n&pAge:\t" << &pAge << endl;
37:
38:    return 0;
39: }
```

输出：

```
myAge:  5              yourAge:       10
&myAge: 0012FF7C       &yourAge:      0012FF78
pAge:   0012FF7C
*pAge:  5

Reassigning: pAge = &yourAge...

myAge:  5              yourAge:       10
&myAge: 0012FF7C       &yourAge:      0012FF78
pAge:   0012FF78
*pAge:  10

&pAge:  0012FF74
```

读者的输出可能与此不同。

分析：

第 9 行将 myAge 和 yourAge 声明为 unsigned short int 变量；第 12 行将 pAge 声明为 unsigned short int 指针，并将其初始化为变量 myAge 的地址。

第 14～18 行打印了 myAge、yourAge 的值和地址。第 20 行打印 pAge 的内容：myAge 的地址。第 21 行打印对 pAge 解除引用的结果，即打印 pAge 指向的变量 myAge 的值 5。

这就是指针的实质。第 20 行表明，pAge 存储的是 myAge 的地址；第 21 行演示了如何通过对指针 pAge 解除引用来获得 myAge 的值。继续学习后面的内容之前，一定要完全了解这些概念。请仔细研究代码并查看输出。

第 25 行重新给 pAge 赋值，使之指向 yourAge。然后，再次打印值和地址。输出表明，pAge 现在存储的是变量 yourAge 的地址，而对 pAge 解除引用得到的是 yourAge 的值。

第 36 行打印 pAge 本身的地址。像其他任何变量一样，它也有地址，在该地址处可以存储一个指针。稍

后将讨论将指针的地址赋给另一个指针。

应该：

务必使用间接运算符（*）来访问存储在指针指向的地址处的值。

务必将任何指针初始化为合法的地址或空值（0）。

不应该：

不要将指针指向的地址与存储在该地址处的值混为一谈。

使用指针

要声明指针，首先指出指针将指向的变量或对象的类型，然后加上指针运算符（*）和指针名。例如：

```
unsigned short int * pPointer = 0;
```

要给指针赋值或初始化，在要将其地址赋给指针的变量名前加上地址运算符&。例如：

```
unsigned short int theVariable = 5;
unsigned short int * pPointer = & theVariable;
```

要对指针解除引用，在指针名前使用解除引用运算符（*）。例如：

```
unsigned short int theValue = *pPointer
```

8.2　为什么使用指针

至此，读者已经知道了将变量的地址赋给指针的步骤，然而，在实际编程中，你不会这样做。既然使用变量就可以访问数据，为什么还要使用指针呢？使用指针来操纵自动变量的惟一原因是，为了说明指针的工作原理。现在，读者已经熟悉了指针的语法，可以将其用于更好的用途。指针最常被用于完成下列 3 种任务：

- 管理自由存储区中的数据。
- 访问类的成员数据和函数。
- 按引用传递参数。

本章余下的内容将重点介绍管理自由存储区中的数据和访问类的成员数据和函数。下一章将介绍使用指针来传递变量，这被称为按引用传递。

8.3　栈和自由存储区（堆）

在第 5 章的“函数的工作原理”一节中，提到了 5 个内存区域：

- 全局名称空间。
- 自由存储区。
- 寄存器。
- 代码空间。
- 堆栈。

局部变量和函数参数位于堆栈中；当然，代码位于代码空间中；而全局变量位于全局名称空间中；寄存器用于内部管理工作，如记录栈顶指针和指令指针。余下的所有内存都被作为自由存储区，通常被称为堆。

局部变量不是永久性的，函数返回时，局部变量就被删除。这很好，因为这意味着根本不用为管理这种内存空间而劳神；也不好，因为这使得函数在不将堆中的对象复制到调用函数中的目标对象的情况下，将难以创建供其他对象或函数使用的对象。全局变量解决了这种问题，其代价是在整个程序中都可以访问它们，这导致创建了难以理解和维护的代码。如果管理得当，将数据存储在自由存储区可以解决这两种问题。

可以将自由存储区视为一块很大的内存，其中有数以千计的依次被编号的内存单元，可用于存储数据。与堆栈不同，你不能对这些单元进行标记，而必须先申请内存单元的地址，然后将它存储到指针中。

可以使用这样的类比：朋友给了你 Acme Mail Order 的 800 电话号码。你回到家中，将该电话号码与某个按钮绑定，然后扔掉记录电话号码的纸张。如果按下这个按钮，被拨打的电话将响铃，Acme Mail Order 公司的职员进行接听。你不需要记下这个电话号码，也不知道被拨打的电话在哪里，但只要按下绑定到的按钮就能致电 Acme Mail Order 公司。自由存储区中的数据就像 Acme Mail Order 公司一样。你不知道它在什么地方，但知道如何找到它。你使用地址（在这个例子中，地址为电话号码）访问它。你不必知道地址，只需将其放在一个指针（按钮）中。指针让你能够访问数据，而不必知道细节。

函数返回时，堆栈被自动清空。所有局部变量都不在作用域内，它们被从堆栈中删除。程序结束前，自由存储区不会被清空，程序员使用完自己分配的内存后，必须负责将其释放。

自由存储区的优点是，你从中分配的内存将一直可用，直到你明确地指出不再需要——将其释放。如果在函数中分配自由存储区中的内存，在函数返回后该内存仍可用。

这也是自由存储区的缺点，如果你忘记释放内存，被占据而没有使用的内存将随着时间的推移越来越多，导致系统崩溃。

采取这种内存访问方式而不是全局变量的优点是，只有能够访问指针的函数才能访问它指向的数据。这样，只有将包含指针的对象或指针本身传递给函数，函数才能修改指针指向的数据，从而减少了函数能够修改数据而又无法跟踪变更的情况发生。

要做到这一点，必须能够创建指向自由存储区中某个区域的指针，并能够在函数之间传递该指针。接下来的几节介绍这是如何实现的。

8.3.1 使用关键字 new 来分配内存

在 C++中，使用关键字 new 来分配自由存储区中的内存。在 new 后面跟上要为其分配内存的对象的类型，让编译器知道需要多少内存。因此，new unsigned short int 在自由存储区中分配 2 字节内存，而 new unsigned long 分配 4 字节内存；这里假设在你的系统中，unsigned short int 和 long 变量分别占用 2 字节和 4 字节。

new 的返回值是一个内存地址。读者知道，内存地址被存储在指针中，因此应将 new 的返回值赋给一个指针。要在自由存储区中创建一个 unsigned short 变量，可以这样做：

```
unsigned short int *pPointer;
pPointer = new unsigned short int ;
```

当然，可以在声明指针的同时对其进行初始化，从而在一行代码中完成上述工作：

```
unsigned short int *pPointer = new unsigned short int ;
```

无论采用哪种方式，pPointer 都将指向自由存储区中的一个 unsigned short int。可以像使用其他指向变量的指针那样使用它，将一个值赋给它指向的内存区域：

```
*pPointer = 72;
```

这条语句的含义是：将 72 放在 pPointer 指向的区域中，或者将 72 赋给 pPointer 指向的自由存储区中的区域。

注意：如果 new 不能在自由存储区分配内存（毕竟内存是有限的资源），将引发异常（参见第 20 章）。

8.3.2 使用关键字 delete 归还内存

使用完内存区域后，必须将其归还给系统。为此，可以将 delete 应用于指针。Delete 将内存归还给自由存储区。

请切记，使用 new 分配的内存不会被自动释放。如果指针变量指向自由存储区中的内存块，离开该指针的作用域时，该内存块不会被自动归还给自由存储区。相反，该内存块被视为已分配出去，同时由于该指针不再可用，你将无法访问该内存块。当指针为局部变量时将发生这样的情况。当函数返回时，在该函数中声

明的指针将不在作用域中，从而丢失。使用 new 分配的内存不会被释放，而是变得不可用。

这被称为内存泄漏，因为在程序结束前，该内存块再也无法使用，就像从计算机中泄漏掉了一样。

为防止内存泄漏，应将分配的内存归还给自由存储区。为此，可使用关键字 delete。例如：

```
delete pPointer ;
```

删除指针时，实际上是释放了其地址存储在指针中的内存。这相当于说：将该指针指向的内存归还给自由存储区。该指针仍然存在，可以重新给它赋值。程序清单 8.4 在堆中给一个变量分配内存，然后使用并删除它。

最常见的情况是，在构造函数中从堆中分配内存，在析构函数中释放这些内存。换句话说，你在构造函数中初始化指针，在对象被使用时为这些指针分配内存，并在析构函数中检查指针是否为空，如果不为空，则释放它们。

警告：将 delete 用于指针时，它指向的内存将被释放。如果再次对该指针使用 delete，程序将崩溃！因此，删除指针后，应将其值设置为 0（空指针）。将 delete 用于空指针是安全的，例如：

```
Animal *pDog = new Animal; // allocate memory
delete pDog;   //frees the memory
    pDog = 0;   //sets pointer to null
    //...
delete pDog;   //harmless
```

程序清单 8.4　分配、使用和删除指针

```
1:  // Listing 8.4
2:  // Allocating and deleting a pointer
3:  #include <iostream>
4:  int main()
5:  {
6:     using namespace std;
7:     int localVariable = 5;
8:     int * pLocal= &localVariable;
9:     int * pHeap = new int;
10:    *pHeap = 7;
11:    cout << "localVariable: " << localVariable << endl;
12:    cout << "*pLocal: " << *pLocal << endl;
13:    cout << "*pHeap: " << *pHeap << endl;
14:    delete pHeap;
15:    pHeap = new int;
16:    *pHeap = 9;
17:    cout << "*pHeap: " << *pHeap << endl;
18:    delete pHeap;
19:    return 0;
20: }
```

输出：

```
localVariable: 5
*pLocal: 5
*pHeap: 7
*pHeap: 9
```

分析：

第 7 行声明并初始化了一个名为 localVariable 的局部变量；第 8 行声明一个名为 pLocal 的指针，并将其初始化为局部变量的地址。第 9 行声明了另外一个名为 pHeap 的指针，但将其初始化为 new int 的返回值，

这将在从自由存储区中分配了可存储 int 值的内存块，该内存块可使用指针 pHeap 来访问。第 10 行将 7 存储到该内存块中。

第 11～13 行打印了几个值。第 11 行打印局部变量 localVariable 的值，第 12 行打印指针 pLocal 指向的值，第 13 行打印指针 pHeap 指向的值。正如所预期的，第 11 和 12 行打印的值相同。第 13 行表明，第 10 行所赋的值是可被访问的。

第 14 行通过调用 delete 将第 9 行分配的内存归还给自由存储区，这释放了内存，并将指针与内存脱离。现在，可以将 pHeap 指向任何其他内存。第 15 和 16 行重新给它赋值，第 17 行打印结果。第 18 行将内存归还给自由存储区。

虽然第 18 行是多余的（程序结束时将归还该内存块），但显式地释放是个不错的主意。如果程序需要修改或扩展，采取上述步骤将是有好处的。

8.4 再谈内存泄漏

内存泄漏是最严重的问题之一，也是人们指责指针的原因之一。前面介绍了一种可能发生内存泄漏的情形；另一种可能导致内存泄漏的情形是，重新给指针赋值之前没有释放它原来指向的内存。请看下面的代码段：

```
1:  unsigned short int * pPointer = new unsigned short int;
2:  *pPointer = 72;
3:  pPointer = new unsigned short int;
4:  *pPointer = 84;
```

第 1 行创建了指针 pPointer，并将自由存储区中一个内存块的地址赋给它。第 2 行将 72 存储到这个内存块中。第 3 行重新将另一个内存块的地址赋给该指针。第 4 行将 84 存储到新的内存块中。原来的内存块（现在存储的值为 72）将不可用，因为指向该内存块的指针已被重新赋值。现在无法访问原来的内存块，在程序结束前也无法将其释放。

上述代码应写成这样：

```
1: unsigned short int * pPointer = new unsigned short int;
2: *pPointer = 72;
3: delete pPointer;
4: pPointer = new unsigned short int;
5: *pPointer = 84;
```

这样，第 3 行将释放 pPointer 原来指向的内存块。

注意： *程序中的每个 new 都应该有对应的 delete。跟踪指针指向的内存区域并确保使用完毕后将其归还给自由存储区至关重要。*

8.5 在自由存储区上创建对象

就像可以创建指向整型的指针一样，也可以创建指向任何数据类型（包括类）的指针。如果声明了一个 Cat 类，则可以声明一个指向 Cat 类的指针，并在自由存储区中实例化一个 Cat 对象，就像可以在堆栈中创建 Cat 对象一样。创建 Cat 指针的语法与创建 int 指针相同：

```
Cat *pCat = new Cat ;
```

这将调用默认构造函数：不接受任何参数的构造函数。每当对象被创建（无论是在自由存储区还是堆栈中）都将调用构造函数。然而，需要注意的是，使用 new 创建对象时，不仅可以使用默认构造函数，也可以

使用任何构造函数。

8.6　删除自由存储区中的对象

将 delete 用于指向自由存储区中的对象的指针时，在释放内存之前将调用对象的析构函数。这给类提供了一个执行清理工作的机会（通常是释放从堆中分配而来的内存），就像从堆栈中删除对象一样。程序清单 8.5 演示了如何在自由存储区中创建和删除对象。

程序清单 8.5　在自由存储区中创建和删除对象

```
// Listing 8.5 - Creating objects on the free store
// using new and delete

#include <iostream>

using namespace std;

class SimpleCat
{
public:
   SimpleCat();
   ~SimpleCat();
private:
   int itsAge;
};

SimpleCat::SimpleCat()
{
   cout << "Constructor called. " << endl;
   itsAge = 1;
}

SimpleCat::~SimpleCat()
{
   cout << "Destructor called. " << endl;
}

int main()
{
   cout << "SimpleCat Frisky... " << endl;
   SimpleCat Frisky;
   cout << "SimpleCat *pRags = new SimpleCat..." << endl;
   SimpleCat * pRags = new SimpleCat;
   cout << "delete pRags... " << endl;
   delete pRags;
   cout << "Exiting, watch Frisky go... " << endl;
   return 0;
}
```

输出：

```
SimpleCat Frisky...
```

```
Constructor called.
SimpleCat *pRags = new SimpleCat...
Constructor called.
delete pRags...
Destructor called.
Exiting, watch Frisky go...
Destructor called.
```

分析：

第 8～15 行声明了一个简单类 simpleCat。第 11 行声明了 SimpleCat 的构造函数，第 17～21 行是其定义。第 12 行声明了 simpleCat 的析构函数，第 23～26 行是其定义。正如读者看到的，构造函数和析构函数都只是打印一条消息，让用户知道它们被调用了。

第 31 行将 Frisky 声明为一个常规局部变量，因此将在堆栈中创建它。这种创建导致构造函数被调用。第 33 行创建了一个被 pRags 指向的 SimpleCat 对象，然而，由于使用了指针，因此该对象将在堆中创建。同样，这将导致构造函数被调用。

第 35 行将 delete 用于指针 pRags。这将导致析构函数被调用，同时分配用于存储 pRags 指向的 SimpleCat 对象的内存被释放。当函数在第 38 行结束时，Frisky 不再在作用域中，因此其析构函数被调用。

8.7 访问数据成员

第 6 章介绍过，可以使用句点运算符（.）来访问类的数据成员和函数。读者应该知道，这种方法适用于在堆栈中创建的 Cat 对象。

使用指针来访问对象的成员要复杂些。要访问自由存储区中的 Cat 对象，必须先对指针解除引用，然后将句点运算符用于指针指向的对象。这里有必要重复一次：必须首先对指针解除引用，然后结合使用解除引用得到的值（指针指向的值）和句点运算符来访问对象的成员。因此，要访问 pRags 指向的对象的成员函数 GetAge，可以这样做：

```
(*pRags).GetAge();
```

使用括号可确保首先对 pRags 解除引用，然后再访问 GetAge()。别忘了，括号的优先级比任何运算符都高。

由于这样比较麻烦，C++为间接访问提供了一个简捷运算符：类成员访问运算符（->）。该运算符由短划线（-）和大于号（>）组成，在 C++中，这被视为一个符号。程序清单 8.6 演示了如何访问在自由存储区中创建的对象的成员变量和函数。

注意：由于类成员访问运算符（->）也可用于（通过指针）间接访问对象的成员，因此也被称为间接运算符。有些人也将它称为指向（points-to）运算符，因为其功能就是这样的。

程序清单 8.6 访问自由存储区中对象的成员数据

```
1:  // Listing 8.6 - Accessing data members of objects on the heap
2:  // using the -> operator
3:
4:  #include <iostream>
5:
6:  class SimpleCat
7:  {
8:    public:
9:      SimpleCat() {itsAge = 2; }
10:     ~SimpleCat() {}
```

```
11:     int GetAge() const { return itsAge; }
12:     void SetAge(int age) { itsAge = age; }
13:   private:
14:     int itsAge;
15: };
16:
17: int main()
18: {
19:    using namespace std;
20:    SimpleCat * Frisky = new SimpleCat;
21:    cout << "Frisky is " << Frisky->GetAge() << " years old " << endl;
22:    Frisky->SetAge(5);
23:    cout << "Frisky is " << Frisky->GetAge() << " years old " << endl;
24:    delete Frisky;
25:    return 0;
26: }
```

输出：

```
Frisky is 2 years old
Frisky is 5 years old
```

分析：

第 20 行在自由存储区中实例化了一个 SimpleCat 对象，并让指针 Frisky 指向它。默认构造函数将对象的年龄设置为 2，第 21 行调用了 GetAge( )方法。由于 Frisky 是一个指针，因此使用间接运算符来访问成员数据和函数。第 22 行调用了 SetAge( )方法，第 23 行再次调用了 GetAge( )。

8.8　在自由存储区中创建成员数据

除在自由存储区中创建对象外，还可以在自由存储区中创建对象的数据成员。类的数据成员可以是指向自由存储区中对象的指针。使用前面介绍过的知识，可以在自由存储区中分配内存供这些指针使用。内存分配可在类的构造函数或其他方法中进行；使用完成员后，可以（也应该）在析构函数或其他方法中将其占用的内存释放，如程序清单 8.7 所示。

程序清单 8.7　将指针用作成员数据

```
1:  // Listing 8.7 - Pointers as data members
2:  // accessed with -> operator
3:
4:  #include <iostream>
5:
6:  class SimpleCat
7:  {
8:    public:
9:      SimpleCat();
10:     ~SimpleCat();
11:     int GetAge() const { return *itsAge; }
12:     void SetAge(int age) { *itsAge = age; }
13:
14:     int GetWeight() const { return *itsWeight; }
15:     void setWeight (int weight) { *itsWeight = weight; }
```

```
16:
17:   private:
18:     int * itsAge;
19:     int * itsWeight;
20: };
21:
22: SimpleCat::SimpleCat()
23: {
24:    itsAge = new int(2);
25:    itsWeight = new int(5);
26: }
27:
28: SimpleCat::~SimpleCat()
29: {
30:    delete itsAge;
31:    delete itsWeight;
32: }
33:
34: int main()
35: {
36:    using namespace std;
37:    SimpleCat *Frisky = new SimpleCat;
38:    cout << "Frisky is " << Frisky->GetAge()
39:            << " years old " << endl;
40:    Frisky->SetAge(5);
41:    cout << "Frisky is " << Frisky->GetAge()
42:            << " years old " << endl;
43:    delete Frisky;
44:    return 0;
45: }
```

输出：

```
Frisky is 2 years old
Frisky is 5 years old
```

分析：

SimpleCat 类被声明为包含两个成员变量（第 18 和 19 行），它们都是 int 指针。构造函数（第 22～26 行）对这两个指针进行初始化，使之指向自由存储区中的内存块，并将默认值存储到内存块中。

注意，在第 24 和 25 行，创建 int 变量时调用了一个伪构造函数，并传递 int 变量的值。这将在堆中创建一个 int 变量，并将其初始化为传入的值（第 24 行初始化为 2，第 25 行初始化为 5）。

析构函数（第 28～33 行）释放分配的内存。由于这是一个析构函数，因此将这些指针设置为空没有任何意义，因为调用析构函数后，它们就不能被访问了。这是可以违反“对指针使用 delete 后应将其设置为空”这一规则的地方之一，虽然遵守这种规则也没有什么坏处。

调用函数（这里为 main ()）并不知道 itsAge 和 itsWeight 是指向自由存储区中内存块的指针。main( )函数只是调用 GetAge( )和 SetAge( )，而内存管理的细节隐藏在类的实现中——确实应该这样。

第 41 行删除 Frisky 时，将调用析构函数。析构函数删除每个成员指针，如果这些指针指向的是其他用户定义的类对象，这些对象的析构函数也将被调用。

理解要实现的目标

在实际程序中，像程序清单 8.7 那样使用指针是极其愚蠢的，除非有充分的理由要求 Cat 对象通过引用来存储成员。在这个例子中，没有理由将 intsAge 和 itsWeight 声明为指针，但在其他情况这样做可能是有道

理的。

这就提出了一个问题：你将数据成员声明为指针（而不是变量）想达到什么目的？为理解这一点，必须首先考虑设计。如果你设计的是将另一个对象作为数据成员的对象，但前者在后者被创建之前就已存在，且在后者消失后仍将存在，则后者必须通过引用来包含前者。

例如，后者可能是一个窗口，而前者可能是一个文档。窗口需要访问文档，但不能控制文档的生命周期。因此，窗口需要通过引用来包含文档。

在 C++中，这是通过使用指针或引用实现的。引用将在第 9 章介绍。

8.9　this 指针

每个类成员函数都有一个隐藏的参数：this 指针。this 指针指向当前对象。因此，在每次调用函数 GetAge( ) 或 SetAge( )时，函数都通过一个隐藏参数收到了一个 this 指针，该指针指向通过它调用函数的对象。

可以显式地使用 this 指针，如程序清单 8.8 所示。

程序清单 8.8　使用 this 指针

```
1:  // Listing 8.8
2:  // Using the this pointer
3:
4:  #include <iostream>
5:
6:  class Rectangle
7:  {
8:    public:
9:      Rectangle();
10:     ~Rectangle();
11:     void SetLength(int length)
12:       { this->itsLength = length; }
13:     int GetLength() const
14:       { return this->itsLength; }
15:
16:     void SetWidth(int width)
17:       { itsWidth = width; }
18:     int GetWidth() const
19:       { return itsWidth; }
20:
21:   private:
22:     int itsLength;
23:     int itsWidth;
24: };
25:
26: Rectangle::Rectangle()
27: {
28:   itsWidth = 5;
29:   itsLength = 10;
30: }
31: Rectangle::~Rectangle()
32: {}
33:
34: int main()
```

```
35: {
36:    using namespace std;
37:    Rectangle theRect;
38:    cout << "theRect is " << theRect.GetLength()
39:       << " feet long." << endl;
40:    cout << "theRect is " << theRect.GetWidth()
41:       << " feet wide." << endl;
42:    theRect.SetLength(20);
43:    theRect.SetWidth(10);
44:    cout << "theRect is " << theRect.GetLength()
45:       << " feet long." << endl;
46:    cout << "theRect is " << theRect.GetWidth()
47:       << " feet wide. " << endl;
48:    return 0;
49: }
```

输出：

```
theRect is 10 feet long.
theRect is 5 feet wide.
theRect is 20 feet long.
theRect is 10 feet wide.
```

分析：

第 11～12 行的存取器函数 SetLength( )和第 13～14 行的存取器函数 GetLength( )都显式地使用了 this 指针来访问 Rectangle 对象的成员变量；第 16～19 行的存取器函数 SetWidth ( )和 GetWidth( )没有这样做。虽然语法更容易理解，但它们在行为上没有任何差别。

如果 this 指针就这么点功能，则没有必要介绍它。然而，this 是一个指针，它存储了对象的地址，因此，它可以是一个功能强有力的工具。

第 10 章讨论运算符重载时，读者将看到 this 指针的实际应用。现在，读者只需知道有这么一个 this 指针以及它是什么：一个指向当前对象的指针。

你不必担心创建或删除 this 指针的问题，这些工作由编译器完成。

8.10 迷途指针

有关指针的争论备受瞩目。这是因为在程序中由于指针引发的错误可能是最难发现和最难解决的。在 C++，导致难以发现和解决的错误的罪魁祸首之一迷途（stray）指针。迷途指针也被称为失控（wild）指针或悬浮（dangling）指针，是将 delete 用于指针（从而释放它指向的内存），但没有将它设置为空时引发的。如果随后你在没有重新赋值的情况下使用该指针，后果将是不可预料的：程序崩溃算你走运。

这就如同 Acme Mail Order 公司变更了电话号码，但你仍去按原来绑定的按钮。这可能不会导致什么严重后果——也许这将拨打一个无人仓库中的电话。另一方面，这个号码也可能被重新分配给一个军工厂，你拨打电话可能引发爆炸，将整个城市摧毁。

总之，对指针使用 delete 后就不要再使用它。虽然这个指针仍指向原来的内存区域，但编译器可能已经将其他数据存储在这里。不重新给这个指针赋值就再次使用它可能导致程序崩溃；更糟糕的是，程序可能表面上运行正常，但过不了几分钟就崩溃了。这被称为定时炸弹，可不是好玩的。为安全起见，删除指针后，把其设置空（0）。这样便解除了它的武装。

注意：迷途指针通常也被称为失控指针或悬浮指针。

程序清单 8.9 演示了如何创建迷途指针。

警告：这个程序故意创建了一个迷途指针。不要运行它，如果这样做，程序崩溃算你走运。

程序清单 8.9　创建迷途指针

```
1:  // Listing 8.9 - Demonstrates a stray pointer
2:
3:  typedef unsigned short int USHORT;
4:  #include <iostream>
5:
6:  int main()
7:  {
8:    USHORT * pInt = new USHORT;
9:    *pInt = 10;
10:   std::cout << "*pInt: " << *pInt << std::endl;
11:   delete pInt;
12:
13:   long * pLong = new long;
14:   *pLong = 90000;
15:   std::cout << "*pLong: " << *pLong << std::endl;
16:
17:   *pInt = 20;     // uh oh, this was deleted!
18:
19:   std::cout << "*pInt: " << *pInt << std::endl;
20:   std::cout << "*pLong: " << *pLong << std::endl;
21:   delete pLong;
22:   return 0;
23: }
```

输出：

```
*pInt:  10
*pLong:  90000
*pInt:  20
*pLong:  65556
```

不要尝试去生成上面的输出；如果你走运，输出将与此不同；如果你不走运，计算机将崩溃。

分析：

读者不要运行该程序，因为它可能导致计算机死锁。

第 8 行将 pInt 声明为一个 UNSHORT 指针，并将其指向使用 new 分配的内存。第 9 行将 10 存储到 pInt 指向的内存中，然后在第 10 行打印这个值。打印这个值后，对指针使用 delete。第 11 行被执行后，pInt 将成为一个迷途指针。

第 13 行声明了一个新的指针 pLong，它指向 new 分配的内存。第 14 行将 90000 存储到 pLong 指向的内存中，第 15 行打印这个值。

带来麻烦的是第 17 行，它将 20 存储到 pInt 指向的内存，但 pInt 不再指向任何合法的内存。第 11 行使用 delete 释放了 pInt 指向的内存，将一个值存储到该内存中无疑将带来灾难。

第 19 行打印 pInt 指向的值，当然应该是 20。第 20 行打印 pLong 指向的值，它突然变成了 65556。这提出了两个问题：

（1）在没有动 pLong 的情况下，它指向的值怎么会变呢？

（2）第 17 行使用 pInt 时将 20 存储到哪里了？

读者可能猜到了，这两个问题是相关的。第 17 行将一个值存储到 pInt 指向的内存中时，编译器将 20 存储到 pInt 原来指向的内存区域中。然而，由于第 11 行已经释放了这个内存块，编译器可以再次使用它。第 13 行创建指针 pLong 时，它指向的是 pInt 原来指向的内存块（在有些计算机上，可能不会发生这种情况，这取决于这些值保存在内存的什么位置）。将 20 赋给 pInt 原先指向的内存时，将覆盖 pLong 指向的值。这被称为“重踏指针”，它通常是使用迷途指针产生的不幸后果。

这种错误很难处理，因为被修改值与迷途指针毫无关联。pLong 指向的值被修改只是误用指针 pInt 的副作用。在大型程序中，这种错误很难查获。

只是好玩

在程序清单中，pLong 指向的内存中的值变成 65556 的过程如下：

（1）指针 pInt 指向某个内存块，并将 10 存储到该内存块中。

（2）将 delete 用于指针 pInt，这相当于告诉编译器，可以将该内存块用于存储其他东西。然后，指针 pLong 指向该内存块。

（3）将 90 000 赋给*PLong。在这个例子中使用的计算机按字节交换顺序存储 4 字节值 90 000（00 01 5F 90），因此它被存储为 5F 90 00 01。

（4）将 20（十六进制表示为 00 14）赋给*pInt。由于 pInt 仍然指向原来的地址，因此 pLong 的前两个字节被覆盖，变成了 00 14 00 01。

（5）打印*pLong 的值，将字节反转为正确的顺序 00 01 00 14，然后被转换为 DOS 值 65 556。

FAQ

空指针和迷途指针的区别是什么？

答：将 delete 用于指针时，实际是让编译器释放内存，但指针本身依然存在。它现在是一个迷途指针。如果接下来使用 myPtr = 0;，该迷途指针将变为空指针。

通常，如果对指针使用 delete 后再次对其使用 delete，后果将是不确定的，也就是说，任何事情都可能发生；如果幸运的话，程序可能会崩溃。但如果对空指针使用 delete，什么事情也不会发生，这样做是安全的。

使用迷途指针或空指针（如 myPtr=5;）是非法的，可能导致程序崩溃。如果是空指针，程序将崩溃，这是空指针相对于迷途指针的另一个优点。可预测的崩溃更可取，因为更容易调试。

8.11 使用 const 指针

声明指针时，可以在类型前或后使用关键字 const，也可在这两个位置都使用。例如，以下都是合法的声明：

```
const int * pOne;
int * const pTwo;
const int * const pThree;
```

然而，这些声明的含义是不同的：

- pOne 是一个指向整型常量的指针。它指向的值是不能修改的。
- pTwo 是一个指向整型的常量指针。它指向的值可以修改，但 pTwo 不能指向其他变量。
- pThree 是一个指向整型常量的常量指针。它指向的值不能修改，且这个指针也不能指向其他变量。

理解这些声明的技巧在于，查看关键字 const 右边来确定什么被声明为常量。如果该关键字的右边是类型，则值是常量；如果该关键字的右边是指针变量，则指针本身是常量。下面的代码有助于说明这一点：

```
const int * p1;      // the int pointed to is constant
int * const p2;      // p2 is constant, it can't point to anything else
```

8.11.1 const 指针和 const 成员函数

第 6 章介绍过，可以将关键字 const 用于成员函数。当函数被声明为 const 时，如果它试图修改对象的数

据，编译器将视为错误。

如果声明了一个指向 const 对象的指针，则通过该指针只能调用 const 方法。程序清单 8.10 说明了这一点。

程序清单 8.10　使用指向 const 对象的指针

```
1:  // Listing 8.10 - Using pointers with const methods
2:
3:  #include <iostream>
4:  using namespace std;
5:
6:  class Rectangle
7:  {
8:    public:
9:      Rectangle();
10:     ~Rectangle();
11:     void SetLength(int length) { itsLength = length; }
12:     int GetLength() const { return itsLength; }
13:     void SetWidth(int width) { itsWidth = width; }
14:     int GetWidth() const { return itsWidth; }
15:
16:   private:
17:     int itsLength;
18:     int itsWidth;
19: };
20:
21: Rectangle::Rectangle()
22: {
23:   itsWidth = 5;
24:   itsLength = 10;
25: }
26:
27: Rectangle::~Rectangle()
28: {}
29:
30: int main()
31: {
32:   Rectangle* pRect = new Rectangle;
33:   const Rectangle * pConstRect = new Rectangle;
34:   Rectangle * const pConstPtr = new Rectangle;
35:
36:   cout << "pRect width: " << pRect->GetWidth()
37:     << " feet" << endl;
38:   cout << "pConstRect width: " << pConstRect->GetWidth()
39:     << " feet" << endl;
40:   cout << "pConstPtr width: " << pConstPtr->GetWidth()
41:     << " feet" << endl;
42:
43:   pRect->SetWidth(10);
44:   // pConstRect->SetWidth(10);
45:   pConstPtr->SetWidth(10);
46:
47:   cout << "pRect width: " << pRect->GetWidth()
48:     << " feet\n";
```

```
49:    cout << "pConstRect width: " << pConstRect->GetWidth()
50:       << " feet\n";
51:    cout << "pConstPtr width: " << pConstPtr->GetWidth()
52:       << " feet\n";
53:    return 0;
54: }
```

输出：

```
pRect width: 5 feet
pConstRect width: 5 feet
pConstPtr width: 5 feet
pRect width: 10 feet
pConstRect width: 5 feet
pConstPtr width: 10 feet
```

分析：

第 6～19 行声明了一个 Rectangle 类。第 14 行将成员方法 Getwidth( )声明为 const 的。

第 32 行声明了一个名为 pRect 的 Rectangle 指针；第 33 行声明了一个指向常量 Rectangle 的指针 pConstRect；第 34 行声明了一个指向 Rectangle 的常指针 pConstPtr。第 36～41 行打印这 3 个变量的值。

第 43 行使用 pRect 将矩形的宽度设置为 10。第 44 行的本意是使用 pConstRect 来设置宽度，但它被声明指向常量 Rectangle，不能通过它来调用非 const 成员函数。这不是一条合法的语句，因此这一行被注释掉了。

第 45 行通过 pConstPtr 调用函数 Setwidth( )。pConstPtr 被声明为一个指向矩形的常量指针。换句话说，该指针是常量，不能指向其他对象，但被指向的矩形不是常量，因此可以通过该指针调用诸如 GetWidth( )和 SetWidth( )等方法。

8.11.2 使用 const this 指针

将对象声明为 const 时，相当于将该对象的 this 指针声明为一个指向 const 对象的指针。const this 指针只能用来调用 const 成员函数。

应该：

如果对象不应被修改，则按引用传递它时应使用 const 进行保护。

务必将指针设置为空，而不要让它未被初始化（悬浮）。

不应该：

不要使用已被删除的指针。

不要将指针删除多次。

讨论指向常量对象的引用时，将再次讨论常量对象和常量指针。

8.12 小　　结

指针提供了一种间接访问数据的强有力的手段。每个变量都有地址，可以通过地址运算符&来获得。地址可以存储在指针中。

要声明指针，可指出它指向的对象的类型，然后加上间接运算符（*）和指针名。应将指针初始化为指向一个对象或空（0）。

要访问指针存储的地址处的值，可使用解除引用运算符（*）。

可以声明 const 指针和指向 const 对象的指针；对于前者，不能给它重新赋值使之指向其他对象；而后者不能用于修改它指向的对象。

要在自由存储区中创建对象，可使用关键字 new，并将它返回的地址赋给一个指针。要释放指针指向的内存，将关键字 delete 用于该指针，但这样做不会销毁指针。因此，释放指针指向的内存后，必须给它重新赋值。

8.13 问 与 答

问：为什么指针如此重要？

答：指针重要的原因很多，其中包括指针可用于存储对象的地址和按引用传递参数。第 14 章将介绍指针在类多态中的应用。另外，很多操作系统和类库为你创建对象并返回指向它们的指针。

问：为什么要在自由存储区中声明对象？

答：自由存储区中的对象在函数返回后仍然存在。另外，在自由存储区中创建对象的功能让你能够在运行时决定需要多少个对象，而不是事先进行声明。这将在下一章深入讨论。

问：既然 const 对象限制了你可对它执行的操作，为什么要声明这样的对象呢？

答：作为程序员，你想让编译器帮助你查找错误。一种很难发现的严重错误是，函数以对调用函数来说不明显的方式修改对象。将对象声明为 const 可防止这种修改。

8.14 作　　业

作业包括测验和练习，前者帮助加深读者对所学知识的理解，后者提供了使用新学知识的机会。请尽量先完成测验和练习题，然后再对照附录 D 中的答案，继续学习下一章之前，请务必弄懂这些答案。

8.14.1 测验

1. 哪个运算符用于获得变量的地址？
2. 哪个运算符用于获得存储指针指向的地址处的值？
3. 什么是指针？
4. 指针指向的地址和存储在该地址中的值有何不同？
5. 间接运算符和地址运算符之间有何不同？
6. const int * ptrOne 和 int * const ptrTwo 之间有何不同？

8.14.2 练习

1. 下述声明的作用是什么？
   a. int * pOne;
   b. int vTwo;
   c. int * pThree = &vTwo;
2. 如果有一个名为 yourAge 的 unsigned short 变量，如何声明一个指针来操纵 yourAge？
3. 使用练习 2 中声明的指针将 50 赋给 yourAge。
4. 编写一个小程序，在其中声明一个 int 变量和一个 int 指针，并将 int 变量的地址赋给指针，然后使用该指针来设置 int 变量的值。
5. 查错：下面的代码有何错误？

```
#include <iostream>
using namespace std;
int main()
{
```

```
    int *pInt;
    *pInt = 9;
    cout << "The value at pInt: " << *pInt;
    return 0;
}
```

6. 查错：下面的代码有何错误？

```
#include <iostream>
using namespace std;
int main()
{
      int SomeVariable = 5;
      cout << "SomeVariable: " << SomeVariable << endl;
      int *pVar = & SomeVariable;
      pVar = 9;
      cout << "SomeVariable: " << *pVar << endl;
      return 0;
}
```