# 4.7 指针和自由存储空间

计算机程序在存储数据时必须跟踪的3种基本属性。

* 信息存储在何处；
* 存储的值为多少；
* 存储的信息是什么类型。

前面我们使用了第一种策略来达到上述目的：定义一个简单变量。声明语句指出了值的类型和符号名，还让程序为值分配丙存，并在内部跟踪该内存单元。

下面来看一看另一种策略，它在开发C++类时非常重要。这种策略以指针为基础，指针是一个变量，其存储的是值的地址，而不是值本身。要找到常规变量的地址。必需对变量应用地址运算符（&），就可以获得它的位置。例如，如果home是一个变量，则&home是它的地址。程序清单4.14演示了这个运算符的用法。

**程序清单4.14 address.cpp**

// address.cpp -- using the & operator to find addresses

#include <iostream>

int main()

{

 using namespace std;

 int donuts = 6;

 double cups = 4.5;

 cout << "donuts value = " << donuts;

 cout << " and donuts address = " << &donuts << endl;

// NOTE: you may need to use unsigned (&donuts)

// and unsigned (&cups)

 cout << "cups value = " << cups;

 cout << " and cups address = " << &cups << endl;

 cin.get();

 return 0;

}

下面是该程序在某个系统上的输出：

donuts value = 6 and donuts address = 0x0065fd40

cups value = 4.5 and cups address = 0x0065fd44

显示地址时，该实现的cout使用十六进制表示法，因为这是常用于描述内存的表示法。在该实现中，donuts的存储位置比cups要低，两个地址的差为0x0065fd44 - 0x0065fd40（即4）。因为donuts的类型为int，而这种类型使用4个字节。不同系统给定的地址值可能不同。有些系统可能先存储cups，再存储donuts，这样两个地址值的差将为8个字节，因为cups的类型为double。另外，在有些系统中，可能不会将这两个变量存储在相邻的内存单元中。

使用常规变量时，值是指定的量，而地址为派生量。下面来看看指针策略，它是C++内存管理编程理念的核心。

**指针与C++基本原理**

面向对象编程与传统的过程性蝙程的区别在于，OOP强调的是在运行阶段（而不是编译阶段）进行决策。运行阶段指的是程序正在运行时，编译阶段指的是编译器将程序组合起来时。运行阶段决策就好比度假时，选择参观哪些景点取决于天气和当时的心情；而编译阶段决策更像不管在什么条件下，都坚持预先设定的日程安排。

运行阶段决策提供了灵活性，可以根据当时的情况进行调整。例如，考虑为数组分配内存的情况。传统的方法是声明一个数组。要在C++中声明数组，必须指定数组的长度。因此，数组长度在程序编译时就设定好了；这就是编译阶段决策。您可能认为，在80%的情况下，一个包含20个元素的数组足够了，但程序有时需要处理200个元素。为了安全起见，使用了一个包含200个元素的数组。这样，程序在大多数情况下都浪费了内存。OOP通过将这样的决策推迟到运行阶段进行，使程序更灵活。在程序运行后，可以这次告诉它只需要20个元素，而还可以下次告诉它需要205个元素。

总之，使用OOP时，您可能在运行阶段确定数组的长度。为使用这种方法，语言必须允许在程序运行时创建数组。稍后你看会到，C++采用的方法是，

使用关键字new请求正确数量的内存以及使用指针来跟踪新分配的内存的位置。

在运行阶段做决策并非OOP独有的，但使用C++编写这样的代码比使用C语言简单。处理存储数据的新策略刚好相反，将地址视为指定的量，而将值视为派生量。一种特殊类型的变量——指针用于存储值的地址。因此，指针名表示的是地址。\*运算符被称为**间接值（indirect value）**或**解除引用（dereferencing）运算符**，将其应用于指针，可以得到该地址处存储的值（这和乘法使用的符号相同；C++根据上下文来确定所指的是乘法还是解除引用）。例如，假设manly是一个指针，则manly表示的是一个地址，而\*manly表示存储在该地址处的值。\*manly与常规int变量等效。程序清单4.15说明了这几点，它还演示了如何声明指针。

**程序清单4.15 pointer.cpp**

// pointer.cpp -- our first pointer variable

#include <iostream>

int main()

{

 using namespace std;

 int updates = 6; // declare a variable

 int \* p\_updates; // declare pointer to an int

 p\_updates = &updates; // assign address of int to pointer

// express values two ways

 cout << "Values: updates = " << updates;

 cout << ", \*p\_updates = " << \*p\_updates << endl;

// express address two ways

 cout << "Addresses: &updates = " << &updates;

 cout << ", p\_updates = " << p\_updates << endl;

// use pointer to change value

 \*p\_updates = \*p\_updates + 1;

 cout << "Now updates = " << updates << endl;

 cin.get();

 return 0;

}

下面是该程序的输出：

Values: updates = 6, \*p\_updates = 6

Addresses: &updates = 0x0065fd4a, p\_updates= 0x0065fd4a

Now updates = 7

从中可知，int变量updates和指针变量p\_updates只不过是同一枚硬币的两面。变量updates表示值，并使用&运算符来获得地址；而变量p\_updates表示地址，并使用\*运算符来获得值。由于p\_updates指向updates，因此\*p\_updates和updates完全等价。可以像使用int变量那样使用\*p\_updates。如程序清单4.15所示，可以将值赋给\*p\_updates。这样做将修改指向的值，即updates。

## 4.7.1 声明和初始化指针

计算机需要跟踪指针指向的值得类型。例如，char的地址与double的地址看上去没什么两样，但它们使用的字节数是不同的，它们存储值时使用的内部格式也不同。因此，指针声明必须制定指针指向的数据的类型。

例如，前面的示例包含这样的声明：

int \* p\_updates;

这表明，\* p\_updates的类型是int。由于\*运算符被用于指针，因此p\_updates变量本身必须是指针。我们说p\_updates指向int类型，还可以说p\_updates的类型是指向int的指针。可以这样说，p\_updates是指针（地址），而\* p\_updates是int，不是指针。

\*运算符两边的空格是可选的。传统上，C程序员使用这种格式：

int \*ptr;

这强调\*ptr是一个int类型的值。而很多C++程序员使用这种格式：

int\* ptr;

这强调的是：int\*是一种类型一一指向int的指针。在哪里添加空格对于编译器来说没有任何区别，甚至可以这样做：

int\*ptr;

但要知道的是，下面的声明创建一个指针（p1）和一个int变量（p2）：

int\* p1,p2;

对每个指针变量名，都需要使用一个\*。在C++中，int\*是一种复合类型，是指向int的指针。

可以用同样的句法来声明指向其他类型的指针：

double\* tax\_ptr; // tax\_ptr points to type double

char\* str; // str points to type char

由于已将tax\_ptr声明为一个指向double的指针，因此编译器知道\*tax\_ptr是一个double类型的值。也就是说，它知道\*tax\_ptr是一个以浮点格式存储的值，这个值（在大多数系统上）占据8个字节。指针变量不仅仅是指针，而且是指向特定类型的指针。tax\_ptr的类型是指向double的指针（或double\*类型）。str是指向char的指针类型（或char\*）。尽管它们都是指针，却是不同类型的指针。和数组一样，指针都是基于其他类型的。

虽然tax\_ptr和str指向两种长度不同的数据类型，但这两个变量本身的长度通常是相同的。也就是说，char的地址与double的地址的长度相同，这就好比1016可能是超市的街道地址，而1024可以是小村庄的街道地址一样。地址的长度或值既不能指示关于变量的长度或类型的任何信息，也不能指示该地址上有什么建筑物。一般来说，地址需要2个还是4个字节，取决于计算机系统（有些系统可能需要更大的地址，系统可以针对不同的类型使用不同长度的地址）。

可以在声明语句中初始化指针。在这种情况下，被初始化的是指针，而不是它指向的值。也就是说，下面的语句将pt（而不是\*pt）的值设置为&higgens：

int higgens = 5;

int\* pt = &higgens;

程序清单4.16演示了如何将指针初始化为一个地址。

**程序清单4.16 init\_ptr.cpp**

// init\_ptr.cpp -- initialize a pointer

#include <iostream>

int main()

{

 using namespace std;

 int higgens = 5;

 int \* pt = &higgens;

 cout << "Value of higgens = " << higgens

 << "; Address of higgens = " << &higgens << endl;

 cout << "Value of \*pt = " << \*pt

 << "; Value of pt = " << pt << endl;

 cin.get();

 return 0;

}

下面是该程序的示例输出：

Value of higgens = 5; Address of higgens = 0012FED4

Value of \*pt = 5; Value of pt = 0012FED4

从中可知，程序将pt（而不是\*pt）初始化为higgens的地址。在您的系统上，显示的地址可能不同，显示格式也可能不同。

## 4.7.2 指针的危险

在C++中创建指针时，计算机将分配用来存储地址的内存，但不会分配用来存储指针所指向的数据的内存。为数据提供空间是一个独立的步骤，忽略这一步无疑是自找麻烦，如下所示：

long \* fellow; // create a pointer-to-long

\*fellow = 223323 // place a value in never-never land

fellow确实是一个指针，但它指向哪里呢？上述代码没有将地址赋给fellow。那么223323将被放在哪里呢？我们不知道。由于fellow没有被初始化，它可能有任何值。不管值是什么，程序都将它解释为存储223323的地址。如果fellow的值碰巧为1200，计算机将把数据放在地址1200上，即使这恰巧是程序代码的地址。fellow指向的地方很可能并不是所要存储223323的地方。这种错误可能会导致一些最隐匿、最难以跟踪的bug。

**警告：一定要在对指针应用解除引用运算符（\*）之前，将指针初始化为一个确定的、适当的地址。这是关于使用指针的金科玉律。**

## 4.7.3 指针和数字

指针不是整型，虽然计算机通常把地址当作整数来处理。从概念上看，指针与整数是截然不同的类型。整数是可以执行加、减、除等运算的数字，而指针描述的是位置，将两个地址相乘没有任何意义。从可以对整数和指针执行的操作上看，它们也是彼此不同的。因此，不能简单地将整数赋给指针：

int \* pt;

pt = 0xB8000000; //type mismatch

在这里，左边是指向int的指针，因此可以把它赋给地址，但右边是一个整数。您可能知道，0xB8000000是老式计算机系统中视频内存的组合段偏移地址，但这条语句并没有告诉程序，这个数字就是一个地址。在C99标准发布之前，C语言允许这样赋值。但C++在类型一致方面的要求更严格，编译器将显示一条错误消息，通告类型不匹配。要将数字值作为地址来使用，应通过强制类型转换将数字转换为适当的地址类型：

int \* pt;

pt = (int \*)0xB8000000; // types now match

这样，赋值语句的两边都是整数的地址，因此这样赋值有效。注意，pt是int值的地址并不意味着pt本身的类型是int。例如，在有些平台中，int类型是个2字节值，而地址是个4字节值。

## 4.7.4 使用new来分配内存

前面我们都将指针初始化为变量的地址；变量是在编译时分配的有名称的内存，而指针只是为可以通过名称直接访问的内存提供了一个别名。指针真正的用武之地在于，在运行阶段分配未命名的内存以存储值。在这种情况下，只能通过指针来访问内存。在C语言中，可以用库函数malloc()来分配内存，在C++中仍然可以这样做，但C++还有更好的方法——new运算符。

下面来试试这种新技术，在运行阶段为一个int值分配未命名的内存，并使用指针来访问这个值，这里的关键所在是C++的new运算符。程序员要告诉new，需要为哪种数据类型分配内存；new将找到一个长度正确的内存块，并返回该内存块的地址。程序员的责任是将该地址赋给一个指针。下面是一个这样的示例：

int \* pn = new int;

new int告诉程序，需要适合存储int的内存。new运算符根据类型来确定需要多少字节的内存。然后，它找到这样的内存，并返回其地址。接下来，将地址赋给pn，pn是被声明为指向int的指针。现在，pn是地址，而\*pn是存储在那里的值。将这种方法与将变量的地址赋给指针进行比较：

int higgens;

int \* pt = &higgens;

在这两种情况（pn和pt）下，都是将一个int变量的地址赋给了指针。在第一种情况下，可以通过名称higgens来访问该int，在第一种情况下，则只能通过该指针进行访问。这引出了一个问题：pn指向的内存没有名称，如何称呼它呢？我们说pn指向一个数据对象，这里的“对象”不是“面向对象编程”中的对象，而是一样“东西”。术语“数据对象”比“变量”更通用，它指的是为数据项分配的内存块。因此，变量也是数据对象，但pn指向的内存不是变量。乍一看，处理数据对象的指针方法可能不太好用，但它使程序在管理内存方面有更大的控制权。

为一个数据对象（可以是结构，也可以是基本类型）获得并指定分配内存的通用格式如下：

typeName \* pointer\_name = new typeName;

需要在两个地方指定数据类型：用来指定需要什么样的内存和用来声明合适的指针。当然，如果已经声明了相应类型的指针，则可以使用该指针，而不用再声明一个新的指针。程序清单4.17演示了如何将new用于两种不同的类型。

**程序清单4.17 use\_new.cpp**

// use\_new.cpp -- using the new operator

#include <iostream>

int main()

{

 using namespace std;

 int nights = 1001;

 int \* pt = new int; // allocate space for an int

 \*pt = 1001; // store a value there

 cout << "nights value = ";

 cout << nights << ": location " << &nights << endl;

 cout << "int ";

 cout << "value = " << \*pt << ": location = " << pt << endl;

 double \* pd = new double; // allocate space for a double

 \*pd = 10000001.0; // store a double there

 cout << "double ";

 cout << "value = " << \*pd << ": location = " << pd << endl;

 cout << "location of pointer pd: " << &pd << endl;

 cout << "size of pt = " << sizeof(pt);

 cout << ": size of \*pt = " << sizeof(\*pt) << endl;

 cout << "size of pd = " << sizeof pd;

 cout << ": size of \*pd = " << sizeof(\*pd) << endl;

 cin.get();

 return 0;

}

下面是该程序的输出：

nights value=1001: location 0028F7F8

int value=1001: location = 00033A98

double value=le+007: location = 000339B8

location of pointer pd: 002BF7FC

size of pt = 4:size of \*pt = 4

size of pd = 4:size of \*pd = 8

### 程序说明

该程序使用new分别为int类型和double类型的数据对象分配内存。这是在程序运行时进行的。指针pt和pd指向这两个数据对象，如果没有它们，将无法访问这些内存单元。有了这两个指针，就可以像使用变量那样使用\*pt和\*pd了。将值赋给\*pt和\*pd，从而将这些值赋给新的数据对象。同样，可以通过打印\*pt和\*pd来显示这些值。

该程序还指出了必须声明指针所指向的类型的原因之一。地址本身只指出了对象存储地址的开始，而没有指出其类型（使用的字节数）。从这两个值的地址可以知道，它们都只是数字，并没有提供类型或长度信息。另外，指向int的指针的长度与指向double的指针相同。它们都是地址，但由于use\_new.cpp声明了指针的类型，因此程序知道\*pd是8个字节的double值，\*pt是4个字节的int值。use\_new.cpp打印\*pd的值时，cout知道要读取多少字节以及如何解释它们。

对于指针，需要指出的另一点是，new分配的内存块通常与常规变量声明分配的内存块不同。变景nights和pd的值都存储在被称为栈（stack）的内存区域中，而new从被称为堆（heap）或自由存储区（free store）的内存区域分配内存。

**内存被耗尽？**

计算机可能会由于没有足够的内存而无法满足new的请求。在这种情况下，new通常会引发异常；而在较老的实现中，new将返回0。在C++中，值为0的指针被称为空指针（null pointer）。C++确保空指针不会指向有效的数据，因此它常被用来表示运算符或函数失败（如果成功，它们将返回一个有用的指针）。因此，C++提供了检测并处理内存分配失败的工具。

## 4.7.5 使用delete释放内存

当需要内存时，可以使用new来请求，这只是C++内存管理数据包中有魅力的一个方面。另一个方面是delete运算符，它使得在使用完内存后，能够将其归还给内存池，这是通向最有效地使用内存的关键一步。归还或释放（free）的内存可供程序的其他部分使用。使用delete时，后面要加上指向内存块的指针（这些内存块最初是用new分配的）：

int \* ps = new int; // allocate memory with new

… // use the memory

delete ps; // free memory with delete when done

这将释放ps指向的内存，但不会删除指针ps本身。例如，可以将ps重新指向另一个新分配的内存块。一定要配对地使用new和delete；否则将发生内存泄漏（memory leak），也就是说，被分配的内存再也无法使用了。如果内存泄漏严重，则程序将由于不断寻找更多内存而终止。

不要尝试释放已经释放的内存块，C++标准指出，这样做的结果将是不确定的，这意味着什么情况都可能发生。另外，不能使用delete来释放声明变量所获得的内存：

int \* ps = new int; // ok

delete ps; // ok

delete ps; // not ok now

int jugs = 5; // ok

int \* pi = &jugs; // ok

delete pi; //not allowed,memory not allocated by new

**警告：只能用delete来释放使用new分配的内存。然而，对空指针使用delete是安全的。**

注意，使用delete的关键在于，将它用于new分配的内存。这并不意味着要使用用于new的指针，而是用于new的地址：

int \* ps = new int; // allocate memory

int \* pq = ps; // set second pointer to same block

delete pq; // delete with second pointer

一般来说，不要创建两个指向同一个内存块的指针，因为这将增加错误地删除同一个内存块两次的可能性。但稍后你会看到，对于返回指针的函数，使用另一个指针确实有道理。

## 4.7.6 使用new来创建动态数组

如果程序只需要一个值，则可能会声明一个简单变量，因为对于管理一个小型数据对象来说，这样做比使用new和指针更简单。通常，对于大型数据（如数组、字符串和结构），应使用new，这正是new的用武之地。例如，假设要编写一个程序，它是否需要数组取决于运行时用户提供的信息。如果通过声明来创建数组，则在程序被编译时将为它分配内存空间。不管程序最终是否使用数组，数组都在那里，它占用了内存。在编译时给数组分配内存被称为静态联编（static binding），意味着数组是在编译时加入到程序中的。但使用new时，如果在运行阶段需要数组，则创建它；如果不需要，则不创建。还可以在程序运行时选择数组的长度。这被称为动态联编（dynamic binding），意味着数组是在程序运行时创建的。这种数组叫作动态数组（dynamic array）。使用静态联编时，必须在编写程序时指定数组的长度；使用动态联编时，程序将在运行时确定数组的长度。

下面来看一下关于动态数组的两个基本问题：如何使用C++的new运算符创建数组以及如何使用指针访问数组元素。

### 1．使用new创建动态数组

在C++中，创建动态数组很容易；只要将数组的元素类型和元素数目告诉new即可。必须在类型名后加上方括号，其中包含元素数目。例如，要创建一个包含10个int元素的数组，可以这样做：

int \* psome = new int [10]; // get a block of 10 ints

new运算符返回第一个元素的地址。在这个例子中，该地址被赋给指针psome。

当程序使用完new分配的内存块时，应使用detete释放它们。然而，对于使用new创建的数组，应使用另一种格式的delete来释放：

delete [] psome; // free a dynamic array

方括号告诉程序，应释放整个数组，而不仅仅是指针指向的元素。请注意delete和指针之间的方括号。如果使用new时，不带方括号，则使用delete时，也不应带方括号。如果使用new时带方括号，则使用delete时也应带方括号。C++的早期版本无法识别方括号表示法。然而，对于ANSI/ISO标准来说，new与delete的格式不匹配导致的后果是不确定的。

int \* pt = new int;

short \* ps = new short[500];

delete [] pt; // effects is undefined,don’t do it

delete ps; // effect is undefined,don't do it

总之，使用new和delete时，应遵守以下规则：

* 不要使用delete来释放不是new分配的内存。
* 不要使用delete释放同一个内存块两次。
* 如果使用new []为数组分配内存，则应使用delete[]来释放。
* 如果使用new为一个实体分配内存，则应使用delete（没有方括号）来释放。
* 对空指针应用delete是安全的。

psome是指向一个int（数组第一个元素）的指针。你的责任是跟踪内存块中的元素个数。也就是说，由于编译器不能对psome是指向10个整数中的第1个这种情况进行跟踪，因此编写程序时，必须让程序跟踪元素的数目。

实际上，程序确实跟踪了分配的内存量，以便以后使用detete []运算符时能够正确地释放这些内存。但这种信息不是公用的，例如，不能使用sizeof运算符来确定动态分配的数组包含的字节数。

为数组分配内存的通用格式如下：

type\_name \* pointer\_name = new type\_name [num\_elements];

使用new运算符可以确保内存块足以存储num\_elements个类型为type\_name的元素，而pointer\_name将指向第1个元素。

### 2．使用动态数组

下面的语句创建指针psome，它指向包含10个int值的内存块中的第1个元素：

int \* psome = new int [10] // get a block of 10 ints

从概念上看：可以将它看作是一根指向该元素的手指。假设int占4个字节，则将手指沿正确的方向移动4个字节，手指将指向第2个元素。总共有10个元素，这就是手指的移动范围。因此，new语句提供了识别内存块中每个元素所需的全部信息。

如何访问其中的元素呢？第一个元素不成问题。由于psome指向数组的第1个元素，因此\*psome是第1个元素的值。这样，还有9个元素。如果没有使用过C语言，下面这种最简单的方法可能会令您大吃一惊：只要把指针当作数组名使用即可。也就是说，对于第1个元素，可以使用psome[0]，而不是\*psome；对于第2个元素，可以使用psome[1]，依此类推。这样，使用指针来访问动态数组就非常简单了，虽然还不知道为何这种方法管用。可以这样做的原因是，C和C++内部都使用指针来处理数组。数组和指针基本等价是C和C++的优点之一。

程序清单4.18演示了如何使用new来创建动态数组以及使用数组表示法来访问元素：它还指出了指针和真正的数组名之间的根本差别。

**程序清单4.18 arraynew.cpp**

// arraynew.cpp -- using the new operator for arrays

#include <iostream>

int main()

{

 using namespace std;

 double \* p3 = new double [3]; // space for 3 doubles

 p3[0] = 0.2; // treat p3 like an array name

 p3[1] = 0.5;

 p3[2] = 0.8;

 cout << "p3[1] is " << p3[1] << ".\n";

 p3 = p3 + 1; // increment the pointer

 cout << "Now p3[0] is " << p3[0] << " and ";

 cout << "p3[1] is " << p3[1] << ".\n";

 p3 = p3 - 1; // point back to beginning

 delete [] p3; // free the memory

 cin.get();

 return 0;

}

下面是该程序的输出：

p3[1] is 0.5.

Now p3[0] is 0.5 and p3[1] is 0.8.

从中可知，arraynew.cpp将指针p3当作数组名来使用，p3[0]为第1个元素，依次类推。下面的代码行指出了数组名和指针之间的根本差别：

p3 = p3+1; // okay for pointers, wrong for array names

不能修改数组名的值。但指针是变量，因此可以修改它的值。请注意将p3加1的效果。表达式p3[0]现在指的是数组的第2个值。因此，将p3加1导致它指向第2个元素而不是第1个。将它减1后，指针将指向原来的值，这样程序便可以给delete []提供正确的地址。