单位来进行读写。

下面我们来看一个具体的示例。如代码清单 4-1 所示，这是一个往 *a*、*b*、*c* 这 3 个变量中写入数据 123 的 C 语言程序。这 3 个变量表示的是内存的特定区域。通过使用变量，即便不指定物理地址，也可以在程序中对内存进行读写。这是因为，在程序运行时，Windows 等操作系统会自动决定变量的物理地址。

代码清单 4-1　各种类型的变量

```
// 定义变量
char a;
short b;
long c;

// 给变量赋值
a = 123;
b = 123;
c = 123;
```

这 3 个变量的数据类型分别是，表示 1 字节长度的 char，表示 2 字节长度的 short，以及表示 4 字节长度的 long[①]。因此，虽然同样是数据 123，存储时其所占用的内存大小是不一样的。这里，我们假定采用的是将数据低位存储在内存低位地址的低字节序（little endian）[②]方式（图 4-4）。

① 在 C 语言中，int 这一数据类型经常会用到。int 也是 CPU 最容易处理的数据类型的长度。在 32 位的 CPU 中，int 是 32 位的。在以前的 16 位的 CPU 中，int 是 16 位的。

② 将多字节数据的低位字节存储在内存低位地址的方式称为低字节序，与此相反，把数据的高位字节存储在内存低位的方式称为高字节序。本章的示例图中使用的是奔腾等英特尔处理器所采用的低字节序方式。

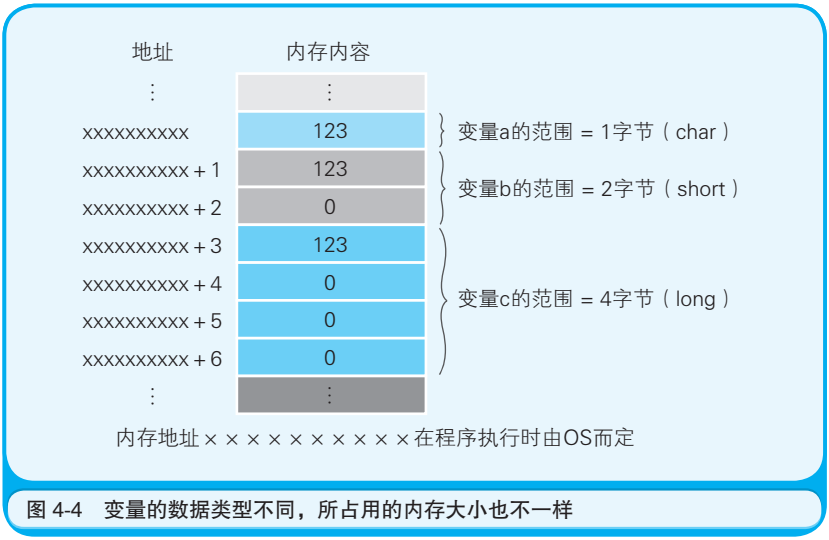

图 4-4　变量的数据类型不同，所占用的内存大小也不一样

仔细思考一下就会发现，根据程序中所指定的变量的数据类型的不同，读写的物理内存大小也会随之发生变化，这其实是非常方便的。大家不妨想一想，假如程序中只能逐个字节地对内存进行读写，那该多么不便啊。在处理超过 1 个字节的数据时，还必须要编写分割处理程序。此外，在不同的编程语言中，变量可以指定的数据类型的最大长度也不相同。C 语言中，8 字节（= 64 位）的 double 类型是最大的。

4.3　简单的指针

接下来，让我们一起来看一下指针。指针是 C 语言的重要特征，但很多人都说它难以理解，甚至还有人因无法理解指针而对 C 语言的学习产生了很强的挫败感。不过，对已经阅读到现在的各位读者来说，指针应该很容易理解。理解指针的关键点就是要弄清楚数据类型这个概念。

指针也是一种变量，它所表示的不是数据的值，而是存储着数据的内存的地址。通过使用指针，就可以对任意指定地址的数据进行读写。虽然前面所提到的假想内存 IC 中仅有 10 位地址信号，但大家在 Windows 计算机上使用的程序通常都是 32 位（4 字节）的内存地址。这种情况下，指针变量的长度也是 32 位。

请大家看一下代码清单 4-2。这是定义[①]了 *d*、*e*、*f* 这 3 个指针变量的 C 语言程序。和通常的变量定义有所不同，在定义指针时，我们通常会在变量名前加一个星号（*）。我们知道，*d*、*e*、*f* 都是用来存储 32 位（4 字节）的地址的变量。然而，为什么这里又用来指定 char（1 字节）、short（2 字节）、long（4 字节）这些数据类型呢？大家是不是也感到很奇怪？实际上，这些数据类型表示的是从指针存储的地址中一次能够读写的数据字节数。

代码清单 4-2　各种数据类型指针的定义

```
char *d;                //char 类型的指针 d 的定义
short *e;               //short 类型的指针 e 的定义
long *f;                //long 类型的指针 f 的定义
```

假设 *d*、*e*、*f* 的值都是 100。在这种情况下，使用 *d* 时就能够从编号 100 的地址中读写 1 个字节的数据，使用 *e* 时就是 2 个字节（100 地址和 101 地址）的数据，使用 *f* 时就是 4 个字节（100 地址～103 地址）的数据。怎么样？指针是不是很简单呢（图 4-5）。

① 在程序中，通过明确标记数据类型来记述变量的过程称为定义变量。例如，若将其记述为 short a;，则表示定义了 2 个字节的 short 类型的变量 *a*。变量定义后就可以进行读写了。

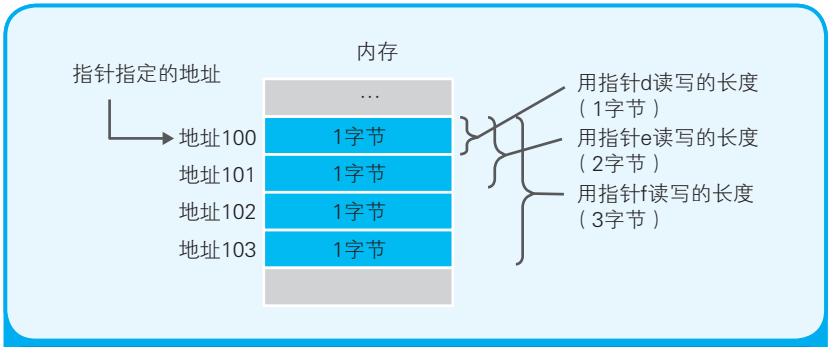

图 4-5　指针的数据类型表示一次可以读写的长度

4.4　数组是高效使用内存的基础

下面让我们回到主题，解释一下本章标题中出现的“熟练使用有棱有角的内存”。在熟练使用前，我们先来看一下内存最直接的使用方法。在这里，我们要用到数组。

数组是指多个同样数据类型的数据在内存中连续排列的形式。作为数组元素的各个数据会通过连续的编号被区分开来，这个编号称为**索引**（index）。指定索引后，就可以对该索引所对应地址的内存进行读写操作[①]。而索引和内存地址的变换工作则是由编译器自动实现的。

代码清单 4-3 表示的是在 C 语言中定义 char 类型、short 类型和 long 类型这三个数组。用括号围起来的 [100]，表示数组的元素有 100 个。由于在 C 语言中，数组的索引是从 0 开始的，因此，charg[100]; 表示的就是可以使用 g[0]～g[99] 这 100 个元素。

① CPU 是通过利用基址寄存器和变址寄存器来指定内存地址的，这一点第 1 章中已经进行了说明。

代码清单 4-3　各种类型的数组定义

```
char g[100];            //char 类型数组 g 的定义
short h[100];           //short 类型数组 h 的定义
long i[100];            //long 类型数组 i 的定义
```

数组的定义中所指定的数据类型，也表示一次能够读写的内存大小。char 类型的数组以 1 个字节为单位对内存进行读写，而 short 类型和 long 类型的数组则分别以 2 个字节、4 个字节为单位对内存进行读写。数组是使用内存的基本。本章后半部分会讲述各种各样的内存使用技能，其中每一种都需要以数组为基础。

之所以说数组是内存的使用方法的基础，是因为数组和内存的物理构造是一样的。特别是 1 字节类型的数组，它和内存的物理构造完全一致。不过，如果只能逐个字节地来读写，程序就会变得比较麻烦，因而可以指定任意数据类型来定义数组。这和将 1 层 = 1 单元的楼房改造成多个楼层 = 1 单元的楼房是同一个道理（图 4-6）。

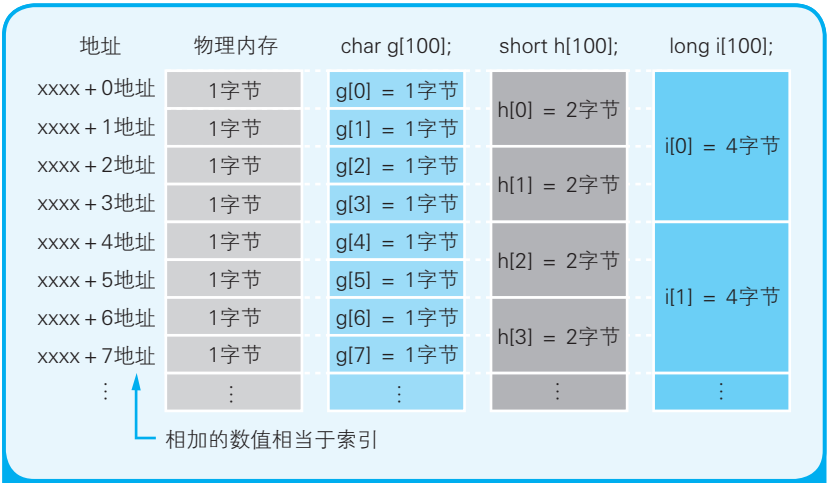

图 4-6　不同数据类型的数组

使用数组能够使编程工作变得更加高效。如果在反复运行的循环处理[①]中使用数组，很短的代码就能达到按顺序读出或写入数组元素的目的。不过，虽然是通过指定索引来使用数组，但这和内存的物理读写并没有特别大的区别。因此很多程序都会在数组的使用上花费大量工夫。接下来，我们就向大家介绍一下栈、队列、代码清单和二叉查找树这些数组的变形方法。对于一名优秀的程序员来说，不仅要了解，还要会灵活使用这些方法。

4.5 栈、队列以及环形缓冲区

栈[②]和队列，都可以不通过指定地址和索引来对数组的元素进行读写。需要临时保存计算过程中的数据、连接在计算机上的设备或者输入输出的数据时，都可以通过这些方法来使用内存。如果每次保存临时数据都需指定地址和索引，程序就会变得比较麻烦，因此要加以改进。

栈和队列的区别在于数据出入的顺序是不同的。在对内存数据进行读写时，栈用的是 LIFO（Last Input First Out，后入先出）方式，而队列用的则是 FIFO（First Input First Out，先入先出）方式。如果我们在内存中预留出栈和队列所需要的空间，并确定好写入和读出的顺序，就不用再指定地址和索引了。

如果要在程序中实现栈和队列，就需要以适当的元素数来定义一个用来存储数据的数组，以及对该数组进行读写的函数对。当然，在这些函数的内部，对数组的读写会涉及索引的管理，但从使用函数的角度来说，就没有必要考虑数组及索引了。

① 循环处理（loop）是指反复多次进行同样的处理。

② 这里所说的栈并不是第 1 章及第 10 章提到的函数调用时使用的栈，而是指程序员自身做成的 LIFO 形式的数据存储方式（该栈的实体是数组）。

这里，我们暂且把往栈中写入数据的函数命名为 Push[①]，把从栈中读出数据的函数命名为 Pop，把往队列中写入数据的函数命名为 EnQueue，把从队列中读出数据的函数命名为 DeQueue[②]。Push 和 Pop 以及 EnQueue 和 DeQueue 分别组成一对函数。Push 和 EnQueue 用于为函数的参数传递要写入的数据。Pop 和 DeQueue 用于将读出的数据作为函数返回值返回。通过使用这些函数，可以将数据临时保存（写入），然后再在需要时候把这些数据读出来（代码清单 4-4、代码清单 4-5）。

代码清单 4-4　使用栈的程序示例

```
// 往栈中写入数据
Push(123);       // 写入 123
Push(456);       // 写入 456
Push(789);       // 写入 789

// 从栈中读出数据
j = Pop();       // 读出 789
k = Pop();       // 读出 456
l = Pop();       // 读出 123
```

代码清单 4-5　使用队列的程序示例

```
// 往队列中写入数据
EnQueue(123);    // 写入 123
EnQueue(456);    // 写入 456
EnQueue(789);    // 写入 789

// 从队列中读出数据
m = DeQueue();   // 读出 123
n = DeQueue();   // 读出 456
o = DeQueue();   // 读出 789
```

虽然示例程序中没有展示实际的数组以及 Push、Pop、EnQueue、

① 汇编语言中有 push 和 pop 两个指令，但这里指的是程序员为了以 LIFO 形式对数组进行读写而做成的 Push 函数和 Pop 函数。

② 通常情况下，往栈写入数据称为 Push（入栈），从栈中读出数据称为 Pop（出栈）。往队列中写入数据称为 EnQueue（入列），从队列中读出数据称为 DeQueue（出列）。这里直接把它们各自的英文名称作为函数名字使用了。

DeQueue 的处理内容，不过还是希望大家能够据此对栈及队列是如何使用内存的有一个大体印象。

顾名思义，在栈中，LIFO 方式表示栈的数组中所保存的最后面的数据（Last In）会被最先读取出来（First Out）。代码清单 4-4 的程序运行后，按照 123、456、789 的顺序写入的数据，结果却按照 789、456、123 的顺序被读取出来（图 4-7）。

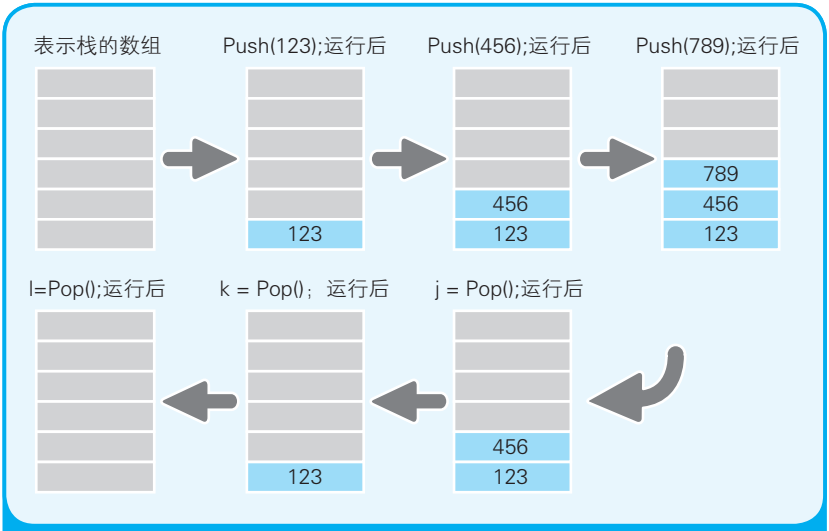

图 4-7　代码清单 4-4 运行时栈的变化

栈的原意是“干草堆积如山”。干草堆积成山后，最后堆的干草会被最先抽取出来。干草堆也是用来临时保存家禽饲料的方式。程序中也是如此，为了实现临时保存数据的目的，使用这种类似于干草堆的机制是非常方便的。而这种机制体现在内存上，就是栈。当我们需要暂时舍弃当前的数据，随后再原貌还原时，会使用栈。

与栈相对的是队列，顾名思义，FIFO 方式表示队列的数组中所保

存的最初数据（First Input）会最先被读取出来（First Out）。代码清单4-5中的程序运行后，按照123、456、789的顺序写入的数据，结果会按照123、456、789的顺序被读取出来（图4-8）。

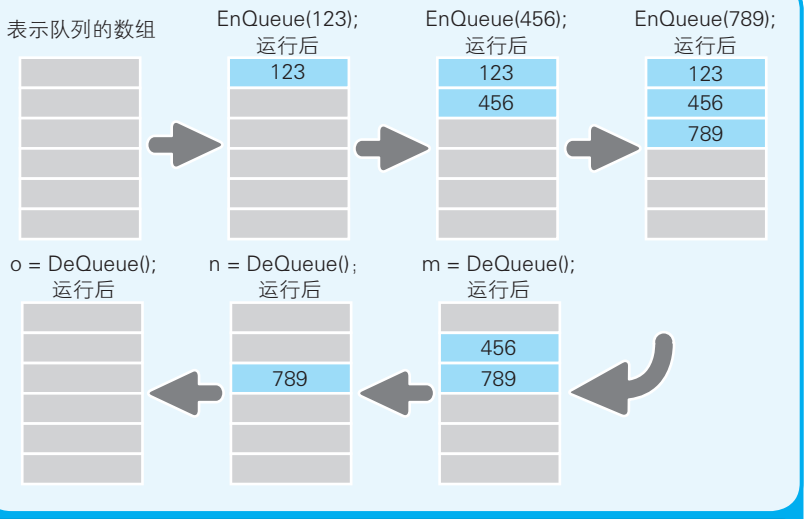

图4-8　代码清单4-5运行时队列的变化

队列这一方式也称为**排队**。排队指的是买车票时在自动售票机前等候的队列等。排队时，站在最前面的乘客先买票，购买后率先从队列中走出来。当随机前来的购票乘客数量和自动售票机的处理速度不相符时，排队能起到很好的缓冲作用。程序中也是如此，为了协调好数据输入和处理时机间的关系，采用类似于排队的机制是很方便的。在内存上，实现这种机制的方式就是队列。当我们需要处理通讯中发送的数据时，或由同时运行的多个程序所发送过来的数据时，会用到这种对队列中存储的不规则数据进行处理的方法。

队列一般是以环状缓冲区（ring buffer）的方式来实现的，也就是

本章标题中所说的“熟练使用有棱有角的内存”。例如，假设我们要用有 6 个元素的数组来实现一个队列。这时可以从数组的起始位置开始有序地存储数据，然后再按照存储时的顺序把数据读出。在数组的末尾写入数据后，后一个数据就会被写入数组的起始位置（此时数据已经被读出所以该位置是空的）。这样，数组的末尾就和开头连接了起来，数据的写入和读出也就循环起来了（图 4-9）。

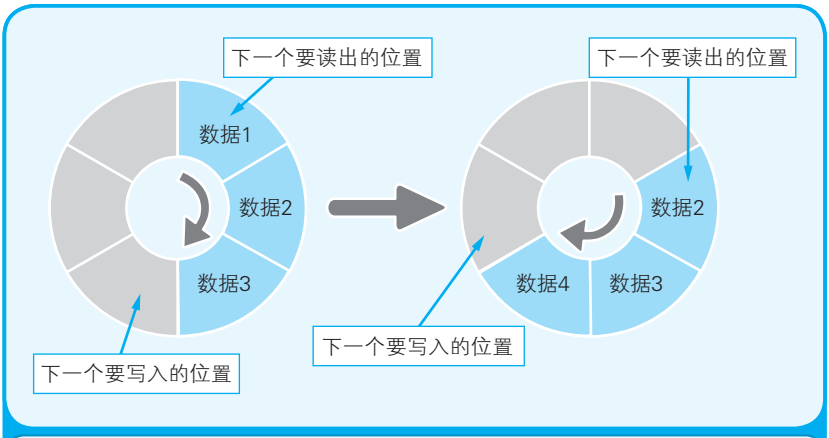

图 4-9 环状缓冲区的模型

4.6 链表使元素的追加和删除更容易

接下来介绍的链表和二叉查找树，都是不用考虑索引的顺序就可以对数组元素进行读写的方式。通过使用链表，可以更加高效地对数组数据（元素）进行追加和删除处理。而通过使用二叉查找树，则可以更加高效地对数组数据进行检索。

在数组的各个元素中，除了数据的值之外，通过为其附带上下一个元素的索引，即可实现**链表**。数据的值和下一个元素的索引组合在一起，就构成了数组的一个元素。这样，数组元素相连就构成了念珠