10.7 使用指针创建和使用动态数组

当预先并不知道数据的长度时，可以使用动态数组。绝大多数具有数组的编程语言都能够在运行时设置数组的长度。它们允许程序员计算需要处理的元素的数目，然后创建一个刚好能容纳这些元素的数组。历史比较悠久的语言如 Algol-60、PL/I 和 Algol-68 等也具备这个功能，比较新的语言如 Ada、Fortran90 和 GNU C（由 GNU C 编译器实现的语言版本）等也允许声明长度可在运行时设置的数组。

然而，在 ANSI C 中，数组是静态的——数组的长度在编译时便已确定不变。在这个领域，C 语言的支持很弱，你甚至不能使用像下面这样的常量形式：

```
const int limit = 100;
char plum[limit];
          ^^^
        error:integral constant expression expected（错误，期待整型常量表达式）
```

我们不想问“为什么一个 const int 不能被当作一个整型常量表达式”这样令人尴尬的问题。在 C++中，这样的语句是合法。

在 ANSI C 中引入动态数组应该是比较容易的，因为这个特性所需要的“前向艺术(prior art)”功能已经存在。所需要做的就是就是把标准 5.5.4 小节中下面这一行

```
direct-declarator [ constant-expression opt ]
```

改为

```
direct-declarator [ expression opt ]
```

如果去除这个人为限制，数组的定义事实上会更简单一些。如果真能这样做的话，C 语言的功能将会得到增强，而且仍然能与 K&R C 保持兼容。由于委员会强烈希望与 C 语言最初的简单设计保持一致，所以这个方案仍然没有被采纳。幸运的是，除此之外仍然有办法实现动态数组的功能（代价嘛就是我们必须亲自做一些指针操作喽）。

现在我们讨论 C 语言中如何实现动态数组。请系紧安全带，这次的学习之旅可是非常的颠簸噢！它的基本思路就是使用 malloc()库函数（内存分配）来得到一个指向一大块内存的指针。然后，像引用数组一样引用这块内存，其机理就是一个数组下标访问可以改写为一个指针加上偏移量。

```
#include <stdlib.h>
#include <stdio.h>
   ...
int size;
char *dynamic;
char input[10];
printf("Please enter size of array: ");
size = atoi(fgets(input, 7, stdin));
dynamic = (char *)malloc(size);
...
dynamic[0] = 'a';
dynamic[size-1] = 'z';
```

动态数组对于避免预定义的限制也是非常有用的。这方面的经典例子是在编译器中。我们不想把编译器符号表的记录数量限制在一个固定的数目上，但也不想一开始就建立一个非常巨大的固定长度的表，这样会导致其他操作的内存空间不够。到目前为止，这些内容还是

比较容易理解的。

我们真正需要实现的是使表具有根据需要自动增长的能力，这样它的惟一限制就是内存的总容量。如果你不是直接声明一个数组，而是在运行时在堆上分配数组的内存，这可以实现这个目标。有一个库函数 realloc()，它能够对一个现在的内存块大小进行重新分配（通常是使之扩大），同时不会丢失原先内存块的内容。当需要在动态表中增长一个项目时，可以进行如下操作：

1. 对表进行检查，看看它是否真的已满。
2. 如果确实已满，使用 realloc()函数扩展表的长度。并进行检查，确保 realloc()操作成功进行。
3. 在表中增加所需要的项目。

用 C 代码表示，大致如下：

```
int   current_element = 0;
int   total_element = 128;
char *dynamic = malloc(total_element);

void add_element(char c){
    if(current_element == total_element - 1){
        total_element *= 2;
        dynamic = (char *)realloc(dynamic, total_element);
        if(dynamic == NULL) error("Coundn't expand the table");
    }
    current_element++;
    dynamic[current_element] = c;
}
```

在实践中，不要把 realloc()函数的返回值直接赋给字符指针。如果 realloc()函数失败，它会使该指针的值变成 NULL，这样就无法对现有的表进行访问。

这种模拟动态数组的技巧在 SunOS 5.0 版本中得到了很广泛的使用。所有重要的固定长度的表（人们在实际使用中受到限制）都进行了修改，使之能够自动增长。这个技巧在其他许多系统软件中也得到了使用，如编译器和调试器。但这个技巧并不是在所有地方都应该使用，理由如下：

- 当一个大型表格突然需要增长时，系统的运行速度可能会慢下来，而且这在什么时候发生是无法预测的。内存分配成倍增长是最关键的原因。
- 重分配操作很可能把原先的整个内存块移到一个不同的位置，这样表格中元素的地址便不再有效。为避免麻烦，应该使用下标而不是元素的地址。
- 所有的“增加”和“删除”操作都必须通过函数来进行，这样才能维持表的完整性。只是这样一来，修改表所涉及到的东西就比仅仅使用下标要多得多。
- 如果表的项目数量减少，可能应该缩小表并释放多余的内存。这样内存收缩的操作对程序的运行速度有很大的影响。每次搜索表格时，编译器最好能够知道任一时刻表的大小。
- 当某个线程对表进行内存重新分配时，你可能想锁住表，保护表的访问，防止其他线程读取表。对于多线程代码，这种锁总是必要的。

数据结构动态增长的另一种方法是使用链表，但链表不能进行随机访问。你只能线性地访问链表（除非你把频繁访问的链表元素的地址保存在缓冲区内），而数组则允许随机访问，这可能在性能上造成很大的差别。

C 能做的不止这些。可以在程序运行时分配更多的内存。主要的工具是malloc()函数，该函数接受一个参数：所需的内存字节数。malloc()函数会找到合适的空闲内存块，这样的内存是匿名的。也就是说，malloc()分配内存，但是不会为其赋名。然而，它确实返回动态分配内存块的首字节地址。因此，可以把该地址赋给一个指针变量，并使用指针访问这块内存。因为char表示1字节，malloc()的返回类型通常被定义为指向char的指针。然而，从ANSI C标准开始，C使用一个新的类型：指向void的指针。该类型相当于一个“通用指针”。malloc()函数可用于返回指向数组的指针、指向结构的指针等，所以通常该函数的返回值会被强制转换为匹配的类型。在ANSI C中，应该坚持使用强制类型转换，提高代码的可读性。然而，把指向 void 的指针赋给任意类型的指针完全不用考虑类型匹配的问题。如果 malloc()分配内存失败，将返回空指针。

我们试着用 malloc()创建一个数组。除了用 malloc()在程序运行时请求一块内存，还需要一个指针记录这块内存的位置。例如，考虑下面的代码：

```
double * ptd;

ptd = (double *) malloc(30 * sizeof(double));
```

以上代码为30个double类型的值请求内存空间，并设置ptd指向该位置。注意，指针ptd被声明为指向一个double类型，而不是指向内含30个double类型值的块。回忆一下，数组名是该数组首元素的地址。因此，如果让ptd指向这个块的首元素，便可像使用数组名一样使用它。也就是说，可以使用表达式ptd[0]访问该块的首元素，ptd[1]访问第2个元素，以此类推。根据前面所学的知识，可以使用数组名来表示指针，也可以用指针来表示数组。

现在，我们有3种创建数组的方法。

声明数组时，用常量表达式表示数组的维度，用数组名访问数组的元素。可以用静态内存或自动内存创建这种数组。

声明变长数组（C99新增的特性）时，用变量表达式表示数组的维度，用数组名访问数组的元素。具有这种特性的数组只能在自动内存中创建。

声明一个指针，调用malloc()，将其返回值赋给指针，使用指针访问数组的元素。该指针可以是静态的或自动的。

使用第2种和第3种方法可以创建动态数组（dynamic array）。这种数组和普通数组不同，可以在程序运行时选择数组的大小和分配内存。例如，假设n是一个整型变量。在C99之前，不能这样做：

```
double item[n]; /* C99之前：n不允许是变量 */
```

但是，可以这样做：

```
ptd = (double *) malloc(n * sizeof(double)); /* 可以 */
```

如你所见，这比变长数组更灵活。

通常，malloc()要与free()配套使用。free()函数的参数是之前malloc()返回的地址，该函数释放之前malloc()分配的内存。因此，动态分配内存的存储期从调用malloc()分配内存到调用free()释放内存为止。设想malloc()和free()管理着一个内存池。每次调用malloc()分配内存给程序使用，每次调用free()把内存归还内存池中，这样便可重复使用这些内存。free()的参数应该是一个指针，指向由 malloc()分配的一块内存。不能用 free()释放通过其他方式（如，声明一个数组）分配的内存。malloc()和free()的原型都在stdlib.h头文件中。

使用malloc()，程序可以在运行时才确定数组大小。如程序清单12.14所示，它把内存块的地址赋给指针 ptd，然后便可以使用数组名的方式使用ptd。另外，如果内存分配失败，可以调用 exit()函数结束程序，其原型在stdlib.h中。EXIT_FAILURE的值也被定义在stdlib.h中。标准提供了两个返回值以保证在所有操作系统中都能正常工作：EXIT_SUCCESS（或者，相当于0）表示普通的程序结束， EXIT_FAILURE 表示程序异常中止。一些操作系统（包括 UNIX、Linux 和 Windows）还接受一些表示其他运行错误的整数值。

程序清单**12.14 dyn_arr.c**程序

```
/* dyn_arr.c -- 动态分配数组 */
#include  <stdio.h>
#include <stdlib.h> /* 为 malloc()、free()提供原型 */
int  main(void)
{
double * ptd;
int  max;
int  number;
int  i  =  0;
puts("What  is  the  maximum  number  of  type  double  entries?");
if  (scanf("%d",  &max)  !=  1)
{
puts("Number  not  correctly  entered  --  bye.");
exit(EXIT_FAILURE);
}
```

```
ptd = (double *) malloc(max * sizeof(double));
if (ptd == NULL)
{
puts("Memory allocation failed. Goodbye.");
exit(EXIT_FAILURE);
}
/* ptd 现在指向有max个元素的数组 */
puts("Enter the values (q to quit):");
while (i < max && scanf("%lf", &ptd[i]) == 1)
++i;
printf("Here are your %d entries:\n", number = i);
for (i = 0; i < number; i++)
{
printf("%7.2f ", ptd[i]);
if (i % 7 == 6)
putchar('\n');
}
```

```
if (i % 7 != 0)
putchar('\n');
puts("Done.");
free(ptd);
return 0;
}
```

下面是该程序的运行示例。程序通过交互的方式让用户先确定数组的大小，我们设置数组大小为 5。虽然我们后来输入了6个数，但程序也只处理前5个数。

```
What is the maximum number of entries?
5
Enter the values (q to quit):
20 30 35 25 40 80
Here are your 5 entries:
20.00 30.00 35.00 25.00 40.00
Done.
```

该程序通过以下代码获取数组的大小：

```
if (scanf("%d", &max) != 1)
{
puts("Number not correctly entered -- bye.");
exit(EXIT_FAILURE);
}
```

接下来，分配足够的内存空间以储存用户要存入的所有数，然后把动态分配的内存地址赋给指针ptd：

```
ptd = (double *) malloc(max * sizeof (double));
```

在C中，不一定要使用强制类型转换(double *)，但是在C++中必须使用。所以，使用强制类型转换更容易把C程序转换为C++程序。

malloc()可能分配不到所需的内存。在这种情况下，该函数返回空指针，程序结束：

```
if (ptd == NULL)
{
puts("Memory allocation failed. Goodbye.");
exit(EXIT_FAILURE);
}
```

如果程序成功分配内存，便可把ptd视为一个有max个元素的数组名。

注意，free()函数位于程序的末尾，它释放了malloc()函数分配的内存。free()函数只释放其参数指向的内存块。一些操作系统在程序结束时会自动释放动态分配的内存，但是有些系统不会。为保险起见，请使用free()，不要依赖操作系统来清理。

使用动态数组有什么好处？从本例来看，使用动态数组给程序带来了更多灵活性。假设你已经知道，在大多数情况下程序所用的数组都不会超过100个元素，但是有时程序确实需要10000个元素。要是按照平时的做法，你不得不为这种情况声明一个内含 10000 个元素的数组。基本上这样做是在浪费内存。如果需要10001个元素，该程序就会出错。这种情况下，可以使用一个动态数组调整程序以适应不同的情况。

静态内存的数量在编译时是固定的，在程序运行期间也不会改变。自动变量使用的内存数量在程序执行期间自动增加或减少。但是动态分配的内存数量只会增加，除非用 free()进行释放。例如，假设有一个创建数组临时副本的函数，其代码框架如下：

```
...
int  main()
{
double  glad[2000];
int  i;
  ...
  for  (i  =  0;  i  <  1000;  i++)
  gobble(glad,  2000);
  ...
  }
  void  gobble(double  ar[],  int  n)
  {
  double * temp = (double *) malloc( n * sizeof(double));
  .../* free(temp); // 假设忘记使用free() */
  }
```

第1次调用gobble()时，它创建了指针temp，并调用malloc()分配了16000字节的内存（假设double为8 字节）。假设如代码注释所示，遗漏了free()。当函数结束时，作为自动变量的指针temp也会消失。但是它所指向的16000字节的内存却仍然存在。由于temp指针已被销毁，所以无法访问这块内存，它也不能被重复使用，因为代码中没有调用free()释放这块内存。

第2次调用gobble()时，它又创建了指针temp，并调用malloc()分配了16000字节的内存。第1次分配的16000字节内存已不可用，所以malloc()分配了另外一块16000字节的内存。当函数结束时，该内存块也无法被再访问和再使用。

循环要执行1000次，所以在循环结束时，内存池中有1600万字节被占用。实际上，也许在循环结束之前就已耗尽所有的内存。这类问题被称为内存泄漏（memory leak）。在函数末尾处调用free()函数可避免这类问题发生。