7.6 动态存储管理

7.6.1 为什么需要动态存储管理

程序中需要用变量（各种简单类型变量、数组变量等）保存被处理的数据和各种状态信息，变量在使用之前必须安排好存储：放在哪里、占据多少存储单元，等等，这个工作被称作*存储分配*。用机器语言写程序时，所有存储分配问题都需要人处理，这个工作琐碎繁杂、很容易出错。在用高级语言写程序时，人通常不需要考虑存储分配的细节，主要工作由编译程序在加工程序时自动完成。这也是用高级语言编程序效率较高的一个重要原因。

C 程序里的变量分为几种。外部变量、局部静态变量的存储问题在编译时确定，其存储空间的实际分配在程序开始执行前完成。程序执行中访问这些变量，就是直接访问与之对应的固定位置。对于局部自动变量，在执行进入变量定义所在的复合语句时为它们分配存储。应该看到，这种变量的大小也是静态确定的。例如，局部自动数组的元素个数必须用静态可求值的表达式描述。这样，一个函数在调用时所需的存储量（用于安放该函数里定义的所有自动变量）在编译时就完全确定了。函数定义里描述了所需要的自动变量和参数，定义了数组的规模，这些就决定了该函数在执行时实际需要的存储空间大小。

以静态方式安排存储的好处主要是实现比较方便，效率高，程序执行中需要做的事情比较简单。但这种做法也形成了对写程序方式的一种限制，使某些问题在这个框架里不好解决。举个简单的例子：假设现在要写一个处理一组学生成绩数据的程序，被处理数据需要存储，因此应该定义一个数组。由于每次使用程序时要处理的成绩的项数可能不同，我们可能希望在程序启动后输入一个表示成绩项数的整数（或通过命令行参数提供一个整数，问题完全一样）。对于这个程序，应该怎样建立其内部的数据表示呢？

问题在于写程序时怎样描述数组元素的个数。一种理想方式是采用下面的程序框架：

```
int n;
...
scanf("%d", &n);
double scores[n];
... /* 读入成绩数据，然后进行处理 */
```

但是这一做法行不通。这里存在两个问题：首先是变量定义不能出现在语句之后。这个问题好解决，可以引进一个复合语句，把 scores 的定义放在复合语句里。第二个问题更本质，在上面程序段里，描述数组 scores 大小的表达式是一个变量，它无法静态求出值。也就是说，这个数组大小不能静态确定，C 语言不允许以这种方式定义数组。这个问题用至今讨论过的机制都无法很好解决。目前可能的解决方案有（一些可能性）：

1. 分析实际问题，定义适当大小的数组，无论每次实际需要处理多少数据都用这个数组。前面的许多程序采用了这种做法。如果前期分析正确，这样做一般是可行的。但如果某一次实际需要处理的数据很多，程序里定义数组不够大，这个程序就不能用了（当然，除非使用程序的人有源程序，而且知道如果修改程序，如何编译等等。在现实生活中，这种情况是例外）。
2. 定义一个很大的数组，例如在所用的系统里能定义的最大数组。这样做的缺点是可能浪费大量空间（存储器是计算机系统里最重要的一种资源）。如果在一个复杂系统里，有这种情况的数组不止一个，那就没办法了。如果都定义得很大，系统可能根本无法容纳它们。而在实际计算中，并不是每个数组都真需要那么大的空间。

上面只是一个说明情况的例子。一般情况是：许多运行中的存储需求在写程序时无法确定。通过定义变量的方式不能很好地解决这类问题。为此就需要一种机制，使我们能利用它写出

一类程序，其中可以根据运行时的实际存储需求分配适当大小的存储区，以便存放到在运行中才能确定大小的数据组。C 语言为此提供了*动态存储管理系统*。说是“动态”，因为其分配工作完全是在动态运行中确定的，与程序变量的性质完全不同。程序里可以根据需要，向动态存储管理系统申请任意大小的存储块。

现在有了动态存储分配，可以要求系统分配一块存储，但是怎么能在程序里掌握和使用这种存储块呢？对于普通的变量，程序里通过变量名去使用它们。动态分配的存储块无法命名（命名是编程序时的手段，不是程序运行中可以使用的机制），因此需要另辟蹊径。一般的语言里都通过指针实现这种访问，用指针指向动态分配得到的存储块（把存储块的地址存入指针），而后通过对指针的间接操作，就可以去使用存储块了。引用动态分配的存储块是指针的最主要用途之一。

与动态分配对应的是动态释放。如果以前动态分配得到的存储块不再需要了，就应该考虑把它们交回去。动态分配和释放的工作都由动态存储管理系统完成，这是支持程序运行的基础系统（称为*程序运行系统*）的一部分。这个系统管理一片存储区，如果需要存储块，就可以调用动态分配操作申请一块存储；如果以前申请的某块存储不需要了，可以调用释放操作将它交还管理系统。动态存储管理系统管理的这片存储区通常称为堆（heap）。

7.6.2 C 语言的动态存储管理机制

C 语言的动态存储管理由一组标准库函数实现，其原型在标准文件<stdlib.h>里描述，需要用这些功能时应包含这个文件。与动态存储分配有关的函数共有四个：

1）存储分配函数 malloc()。函数原型是：

```
void *malloc(size_t n);
```

这里的 size_t 是标准库里定义的一个类型，它是一个无符号整型。这个整型能够满足所有对存储块大小描述的需要，具体相当于哪个整型由具体的 C 系统确定。malloc 的返回值为(void *)类型（这是通用指针的一个重要用途），它分配一片能存放大小为 n 的数据的存储块，返回对应的指针值；如果不能满足申请（找不到能满足要求的存储块）就返回 NULL。在使用时，应该把 malloc 的返回值转换到特定指针类型，赋给一个指针。

例：利用动态存储管理机制，前面提出的问题可以采用如下方式解决：

```
int n;
double *scores;
...
scanf("%d", &n);
scores = (double *)malloc(n * sizeof(double));
if (scores == NULL) {
    ....  /* 出问题时的处理，根据实际情况考虑 */
}
..scores[i] ... *(scores+j) ... /* 读入数据进行处理 */
```

调用 malloc 时，应该利用 sizeof 计算存储块的大小，不要直接写整数，以避免不必要的错误。此外，每次动态分配都必须检查成功与否，并考虑两种情况的处理。

注意，虽然这里的存储块是通过动态分配得到的，但是它的大小也是确定的，同样不允许越界使用。例如上面程序段分配的块里能存 n 个双精度数据，随后的使用就必须在这个范围内进行。越界使用动态分配的存储块，尤其是越界赋值，可能引起非常严重的后果，通常会破坏程序的运行系统，可能造成本程序或者整个计算机系统垮台。

2）带计数和清 0 的动态存储分配函数 calloc。函数原型是：

```
void *calloc(size_t n, size_t size);
```

参数 size 意指数据元素的大小，n 指要存放的元素个数。calloc 将分配一块存储，其大小足以存放 n 个大小各为 size 的元素，分配之后还把存储块里全部清 0（初始化为 0 值）。如果不能满足要求就返回 NULL。

例：前面程序片段里的存储分配也可以用下面语句实现：

```
scores = (double *)calloc(n, sizeof(double));
```

注意，malloc 对于所分配区域不做任何事情，calloc 对整个区域进行初始化，这是两个函数的主要不同点。另外就是两个函数的参数不同，calloc 主要是为了分配“数组”。我们可以根据情况选用。

3）动态存储释放函数 free。原型是：

```
void free(void *p);
```

函数 free 释放指针 p 所指的存储块。指针 p 的值（存储块地址）必须是以前通过动态存储分配函数分配得到的。如果当时 p 的值是空指针，free 就什么也不做。

注意，调用 free(p) 不会改变 p 的值（在函数里不可能改变值参数 p），但被 p 指向的存储块的内容却可能变了（可能由于存储管理的需要）。释放后不允许再通过 p 去访问已释放的块，否则也可能引起灾难性后果。

为了保证动态存储区的有效使用，在知道某个动态分配的存储块不再用时，就应及时将它释放，这应该成为习惯。释放动态存储块只能通过调用 free 完成。下面是一个示例：

```
int fun (...) {
    int *p;
    ... ,..
    p = (int *)malloc(...);
    ...
    free(p);
    return ...;
}
```

这里的 free(p)在 fun 退出前释放了在函数里分配的存储块。如果没有最后的这个 free(p)，函数里分配的这个存储块就可能丢掉。因为 fun 的退出也是 p 的存在期结束，此后 p 保存的信息（动态存储块地址）就找不到，这个块就可能丢掉了[1]。丢失动态分配块的情况称为动态存储的“流失”。对于需要长时间执行的程序，存储流失就可能成为严重问题，可能造成程序执行一段后被迫停止。因此，实际系统不能容忍这种情况的发生。

4）分配调整函数 realloc。函数原型是：

```
void *realloc(void *p, size_t n);
```

这个函数用于更改以前的存储分配。在调用 realloc 时，指针变量 p 的值必须是以前通过动态存储分配得到的指针，参数 n 表示现在需要的存储块大小。realloc 在无法满足新要求时返回 NULL，同时也保持 p 所指的存储块的内容不变。如果能够满足要求，realloc 就返回一片能存放大小为 n 的数据的存储块，并保证该块的内容与原块一致：如果新块较小，其中将存放着原块里大小为 n 的范围内的那些数据；如果新块更大，原有数据存在新块的前面一部分里，新增的部分不自动初始化。如果分配成功，原存储块的内容就可能改变了，因此不允许再通过 p 去使用它。

假如要把一个现有的双精度块改为能存放 m 个双精度数，可以用下面程序段处理：

```
q = (double *)realloc(p, m * sizeof(double));
if (q == NULL) {
    ... ... /* 分配不成功，p 仍指向原块，处理这种情况 */
}
else {
    p = q;
    ... ... /* 分配成功，通过 p 可以去用新的存储块 */
}
```

上面的 q 是另一个双精度指针。这里没有把 realloc 的返回值赋给直接 p，是为了避免分

---

[1] 这种说法也有例外。我们可以在一个函数里申请存储块，而后在函数里把存储块的地址全局的指针变量，或者返回指向这个块的指针值，把这个块交给调用函数的地方用。这些做法也意味着把存储块的“拥有权”交给程序的其他部分，此时就不应该释放它了。

配失败时丢掉原存储块。如果直接赋值，指针 p 原来的值就会丢掉。如果当时的分配没有成功，p 将被赋空指针值，原来那个块可能就再也找不到了（除非在这次调整前已经让另一个指针指向了它）。

请注意：通过动态分配得到的块是一个整体，只能作为一个整体去管理（无论是释放还是改变大小）。在调用 free(p) 或者 realloc(p, ...) 时，p 当时的值必须是以前通过调用存储分配函数得到的，绝不能对指在动态分配块里其他位置的指针调用这两个函数（更不能对并不指向动态分配块的指针使用它们），那样做的后果不堪设想。

7.6.3 两个程序实例

例：修改筛法程序，令它由命令行参数得到所需的整数范围。如果没有命令行参数，就要求用户输入一个确定范围的整数值。

先考虑 main 的设计。为了使程序更加清晰，我们可以考虑把筛法计算写成一个函数。这里还有一个小问题：如果用户通过命令行参数给出工作范围，程序就需要从命令行参数字符串计算出对应的整数。为此我们定义如下函数：

```
int s2int(char s[]);
```

再利用原来的 getnumber 函数，这个程序的 main 可以定义为：

```
enum { LARGEST = 32767 };

int main(int argc, char **argv)
{
    int i, j, n, *ns;

    if (argc == 2) n = s2int(argv[1]);
    else getnumber("Largest number to test: ", 2, LARGEST, 5, &n);

    if (n < 2 || n > LARGEST) {
        printf("Largest number must in range [2, %d]", LARGEST);
        return 1;
    }

    if ((ns = (int*)malloc(sizeof(int)*(n+1))) == NULL) {
        printf("No enough memory!\n");
        return 2;
    }

    sieve(n, ns);

    for(j = 1, i = 2; i <= n; ++i)
        if (ns[i] == 1) {
            printf("%7d%c", i, (j%8 == 7 ? '\n' : ' '));
            ++j;
        }
    putchar('\n');

    free(ns);
    return 0;
}
```

主函数被清晰地分为三部分：准备工作，主要处理部分，输出与结束。如果程序得到的范围不合要求，它就打印错误信息并立即结束。正常情况下完成筛法计算并产生输出。

getnumber 可以直接利用已有的定义（这里又可以看到函数的价值），剩下的工作就是定义程序里需要的两个函数。从数字字符串转换产生整数的函数很简单，它顺序算出各数字字符的整数值并将其加入累加值，每处理一个数位都需要将原值乘 10：

```
int s2int(char s[]) {
    int n;
    for (n = 0; isdigit(*s); ++s)
        n = 10 * n + (*s - '0');
    return n;
}
```

在这个函数里没有检查计算的溢出问题。如果需要，很容易加进这种检查。这里也可以直接用标准库函数 atoi，该函数完成的就是从数字字符串到整数的转换。有关 atoi 的情况请查阅本书第 11 章的介绍。

把筛法计算包装为函数的工作很容易完成，下面是函数的定义：

```
void sieve(int lim, int an[]) {
    int i, j, upb = sqrt(lim+1);

    an[0] = an[1] = 0; // 建立起初始向量
    for (i = 2; i <= lim; ++i) an[i] = 1;

    for (i = 2; i <= upb; ++i)
        if (an[i] == 1) // i 是素数
            for (j = i*2; j <= lim; j += i)
                an[j] = 0; // 这些数都是 i 的倍数，因此不是素数
}
```

使用动态存储管理的要点

1）必须检查分配的成功与否。人们常用的写法是：

```
if ((p = (... *)malloc(...)) == NULL) {
    .. ... /* 对分配未成功情况的处理 */
}
```

2）系统对动态分配块的使用不做任何检查。编程序的人需要保证使用的正确性，绝不可以超出实际存储块的范围进行访问。这种越界访问可能造成大灾难。

3）一个动态分配块的存在期并不依赖于分配这个块的地方。在一个函数里分配的存储块的存在期与该函数的执行期无关。函数结束时不会自动回收这种存储块，要回收这种块，唯一的方法就是通过 free 释放（完全由写程序的人控制）。

4）如果在函数里分配了一个存储块，并用局部变量指向它，在这个函数退出前就必须考虑如何处理这个块。如果这个块已经没用了，那么就应该把它释放掉；如果这个块还有用（其中保存着有用的数据），那么就应该把它的地址赋给存在期更长的变量（例如全局变量），或者把这个地址作为函数返回值，让调用函数的地方去管理它。

5）其他情况也可能造成存储块丢失。例如给一个指向动态存储块的指针赋其他值，如果此前没有其他指针指向这个块，此后就再也无法找到它了。如果一个存储块丢失了，在这个程序随后的运行中，将永远不能再用这个存储块所占的存储。

6）计算机系统里的存储管理分很多层次。一个程序运行时，操作系统分给它一部分存储，供它保存代码和数据。其数据区里包括一块动态存储区，由这个程序的动态存储管理系统管理。该程序运行中的所有动态存储申请都在这块空间里分配，释放就是把不用的存储块交还程序的动态存储管理系统。一旦这个程序结束，操作系统就会收回它占用的所有存储空间。所以，这里说“存储流失”是我们程序内部的问题，并不是整个系统的问题。当然，操作系统也是程序，它也有存储管理问题，那是另一个层次的问题。

把这些函数定义（包括 getnumber 的定义）放到一起，适当安排函数位置，必要时加入原型。在源文件前部加入适当 #include 命令行，整个程序就完成了。

在这个程序里需要存储一批数据，但是数据的数目在写程序时无法确定，因此只能采用动态存储分配的方式。程序里申请了一个大存储块，其中可以存放所需的 int 值。用指针指向这样得到的存储块，用起来就像是在使用一个 int 数组。

例：改造第 6 章的学生成绩统计和直方图生成程序，使之能处理任意多的学生成绩。

本例的重点是讨论一种常见问题的处理技术：通过动态分配的数组，保存事先完全无法确定数量的输入数据。前一个例子是先确定了数据量，而后做一次动态分配。假如直到开始读入数据的时候还不知道有多少数据项，那又该怎么办？下面我们解决这个问题。

在前面的成绩直方图程序用了一个数组，因此也限制了能处理的成绩数。现在我们想修改 readscores，由它全权处理输入工作，在输入过程中根据需要申请适当大小的存储块，把输入数据存入其中。这样，readscores 结束时就需要返回两项信息：保存数据的动态存储块地址，以及存于其中的数据项数。一个函数只能有一个返回值，另一“返回值”需要通过参数送出来。下面是修改后 readscores 的原型和 main 的定义：

```
double* readscores(int* np); /*读入数据，返回动态块地址，通过 np 送回项数*/

int main()
{
    int n;
    double *scores;
    if ((scores = readscores(&n)) == NULL)
        return 1;
    statistics(n, scores);
    histogram(n, scores, HISTOHIGH);
    return 0;
}
```

由于原程序的组织比较合理，在进行当前这个功能扩充时，我们只需要修改其中的输入部分，并对 main 做很局部的修改，其他部分根本无须任何变动。

现在考虑如何写 readscores。一种可行考虑是先做某种初始分配，在发现数据项数太多，当前的分配无法满足需要时进行存储调整。例如把动态数据块的初始大小定为 20（或其他合理的大小），随后如何调整是一个值得研究的问题。下面采用的策略是每次调整时把容量加倍，有关不同调整方式的分析在后面的方框中。这样定义出的函数如下：

```
enum { INITNUM = 20 };

double* readscores(int* np) {
    usigned curnum, n;
    double *p, *q, x;

    if ((p = (double*)malloc(INITNUM*sizeof(double))) == NULL) {
        printf("No memory. Stop\n");
        *np = 0;
        return NULL;
    }

    for(curnum = INITNUM, n = 0; scanf("%lf", &x) == 1; ++n) {
        if (n == curnum) {
            q = (double*)realloc(p, 2*curnum*sizeof(double));
            if (q == NULL) {
                printf("No enough memory. Process %d scores.\n", n);
                break;
            }
            p = q; curnum *= 2;
        }
```

```
        p[n] = x;
    }
    *np = n;

    return p;
}
```

函数里用变量 curnum 记录当前分配块的大小，用 n 记录当前存入的数据项数。一旦遇到数据块满而且还有新项时，我们就扩大存储，把容量加倍。

这个函数定义主要显示了在处理类似问题时常用的一种基本技术，其中并没有刻意追求函数的进一步完善。例如，如果读入数据的过程中遇到一个错误数据，这个函数就会立即结束，返回的是至此读入的数据。有关数据检查和处理等都是前面讨论过的问题，进一步修改这个输入函数，使之能合理处理输入数据中的错误，给出有用的出错信息，或者进一步增加其他有用的功能等等的工作并不困难，都留给读者作为进一步的练习。

7.6.4 函数、指针和动态存储

如果需要在函数里处理一组数据，并把处理结果反应到调用函数的地方，最合适的办法就是在函数调用时提供数组的起始位置和元素数目（或者结束位置）。这种传递成组数据的方式在本章和前一章里反复使用。这时函数完全不必知道用的是程序里定义的数组变量，还是动态分配的存储块。例如，我们完全可以用如下方式调用筛法函数：

```
int ns[1000];

int main()
{
    int i, j;
```

动态调整策略

要实现一个能在使用中根据需要增长的“动态”数组（一个动态分配的，能存储许多元素的存储块可以看成一个“数组”），需要考虑所采用的增长策略。

一种简单的想法是设定一个增量，例如 10，一旦存储区满时就把存储区扩大 10 个单元。仔细考虑和计算会发现这样做有很大的缺陷。实际中对存储量的需要常常是逐步增加的。一般说，在遇到存储区满时，实际上需要另外分配一块更大的存储区，并需要把原块里已有的元素复制到新块里。realloc 完成这种操作的代价（虽然没有显露出来）通常与已有的元素个数成正比。

假设输入过程中执行了一系列扩大存储的动作，如果每加入 10 个元素做一次复制，把数组从 20 增加到包含 1000 个元素，总的复制数将是 20+…+980+990 = 49990。这样，加入每个元素平均大约做 n/20 次复制，n 是最后的元素个数。当数组增大到 1000000 个元素时，每加入一个元素平均要做 50000 次复制，这个代价比较高。

一种合理的增长方式是每次让存储块加倍。假设存储块从 1 开始增长，增长到 1024 时所复制元素为 1+2+4+…+512 = 1023。进一步增长到 1024×1024≈1000000 时，元素复制的总次数大约也为 1000000 次，加入一个元素，平均需要复制一次。可见，增长策略的作用确实很大。当然，如果数组很小，两种策略的差异就不那么明显了。

采用后一增长策略也有代价（世界上没有免费的午餐），那就是存储空间。每次加倍后数组中就出现了一大块空区。例如，当数组有 513 个元素时，空位有 511 个之多。随着数组的加倍，最大的空位数也差不多增加一倍。也就是说，按照这种方案，最坏情况下浪费了一半空间。而按照第一种增长策略，空闲元素最多只有 9 个。

总结一下，这里也是在时间和空间之间做交易。在计算机科学技术领域里，这种时间与空间交换的事情到处都可以看到。问题是要考虑需求，综合权衡。

```
    sieve(1000, ns);
    for(j = 1, i = 2; i <= n; ++i)
        if (ns[i] == 1) {
            printf("%7d%c", i, (j%8 ? ' ' : '\n'));
            ++j;
        }

    putchar('\n');
    return 0;
}
```

在前一节的筛法程序实例里，我们在主函数里通过动态分配取得存储，而后调用函数 sieve，最后还是由 main 函数释放这块存储。这样，分配和释放的责任位于同一层次，由同一个函数（函数 main）完成。这样做最清晰，易于把握，是最好的处理方案。

但也存在一些情况，其中不能采用上述做法，例如上面的直方图程序。程序里定义了一个读入函数，它需要根据输入情况确定如何申请动态存储。这时动态存储的申请在被调用函数 readscores 的内部，该函数完成向存储块里填充数据的工作，最后把做好的存储块（就像是一个数组）的地址通过返回值送出来。调用函数（main）用类型合适的指针接收这个地址值，而后通过这个指针使用这一存储块里的数据。

首先，这一做法完全正确，因为动态分配的存储块将一直存在到明确调用 free 释放它为止。虽然上述存储块是在函数 readscores 里面分配的，但它的生命周期（生存期）并不随该函数的退出而结束。语句：

```
    scores = readscores(&n);
```

使 scores 得到函数 readscores 的运行中申请来并填充好数据的存储块，在 main 里继续用这个块是完全没问题的。当然，采用这种方式，readscores 就不应该在退出前释放该块。注意：上面的调用除了传递有关的数据外，实际上还有存储管理责任的转移问题。在 readscores 把一块存储的指针通过返回值送出来时，也把释放这块存储的责任转交给 main。这样，我们也可以看出前面的程序里忽略了一件事情，在那里没有释放这一存储块。应做的修改就是在 main 的最后加一个释放语句（当然，由于 main 的结束也就是整个程序的结束，未释放的这块存储也不会再有用了。如前所述，在这个程序结束后，操作系统将会收回这个程序占用的全部存储）。

现在考虑 readscores 的设计里的一个问题。在前面的程序里，readscores 通过 int 指针参数（实参应该是一个 int 变量的地址）传递实际读入数据的个数。另一种可能做法是让函数返回这一整数值，例如将其原型改成：

```
int readscores(???);
```

这样，我们在 main 里就可以写如下形式的调用：

```
if (readscores(... ...) <= 0) { ... } /* 产生错误信息并结束程序 */
```

（这一写法使人想起标准库的输入函数 scanf）。如果这样设计函数，调用 readscores 的地方就需要通过实参取得函数里动态分配的存储块地址。也就是说，要从参数获得一个指针值。问题是，这个函数的参数应该如何定义呢？

答案与其他情况完全一样。如果我们想通过实参取得函数里送出来的一个 int 值，就要把一个 int 变量的地址送进函数，要求函数间接地给这个变量赋值。同理，现在需要得到一个指针值，就应该通过实参把这种指针变量的地址送进去，让函数通过该地址给调用时指定的指针变量赋值。这样，修改后的函数 readscores 的原型应该是：

```
int readscores(double **dpp);
```

因为 double 指针的类型是(double*)，其地址的类型就是指向(double*)的指针，也就是(double**)。调用 readscores 时应该把这种指针的地址传给它：

```
if (readscores(&scores) <= 0) {  /* 产生错误并结束程序 */ }
```

由于 scores 的类型是(double*)，表达式 &scores 的类型就是(double**)。函数

readscores 也需要做相应的修改：

```
int readscores(double **dpp) {
    size_t curnum, n;
    double *p, *q, x;

    if ((p = (double*)malloc(INITNUM*sizeof(double))) == NULL) {
        printf("No memory. Stop\n");
        *dpp = NULL;
        return 0;
    }

    for(curnum = INITNUM, n = 0; scanf("%lf", &x) == 1; ++n) {
        if (n == curnum) {
            q = (double*)realloc(p, 2*curnum*sizeof(double));
            if (q == NULL) {
                printf("No enough memory. Process %d scores.\n", n);
                break;
            }
            p = q; curnum *= 2;
        }
        p[n] = x;
    }
    *dpp = p;

    return n;
}
```

这里展示的也是 C 程序里常用的一种技术。在这一处理方案中，我们同样是把函数里分配的存储块送到函数之外，同时也把管理这一存储块的责任转交给调用函数的程序段。不同的是，这次是通过参数传递存储块的地址。

在这一节里，我们介绍了指针、函数与动态分配之间的一些关系，并讨论了几种不同的处理技术。只要有可能，在程序里最好使用第一种设计，因为它最清晰，也最不容易出现忘记释放的情况。如果不得已而采用了其他方式，那么就一定要记得存储管理责任的交接问题，并在适当的地方释放动态分配的存储区。