第七章　指针

本章讨论指针及其在程序中的重要作用，包括变量地址与指针的概念，C 程序里指针的基本定义和使用，指针和数组的关系，函数的指针参数，动态存储分配等问题。

7.1 地址与指针

程序执行中所用的数据都存于内存。任何数据对象在能用期间都有存储位置，占据一定数量的存储单元。内存单元按顺序排，每个单元有一个称为*内存地址*的编号，内存数据都是通过地址访问的。高级语言把存储单元、地址等低级概念用变量等高级概念掩盖起来，使写程序时可以不必过多关心这方面细节。但内存与地址等仍是最基本的重要概念。

前面讲的变量存在期概念实际上与存储有密切关系。建立变量就是为它分配所需的（一批）存储单元。给变量赋值是将值存入对应单元；使用变量值时从相应单元中取用。外部变量和静态局部变量的存在期贯穿整个程序执行期，其存储位置在程序开始前确定，并保持到程序结束。局部自动变量则不同，设 x 是函数 fun 里定义的自动变量，只有执行进入 x 的定义所在的复合语句时才为 x 确定存储。x 占据该块存储直至执行离开这个复合语句。如果执行再次进入该复合语句（包括 fun 再次执行），就会再次为 x 分配存储，但是其位置与前一次无关。这些情况决定了自动变量的各种特性。

变量存在期就是它占据被分配存储位置的期间。虽然不同变量在这方面的性质不同，但它们在存在期里都有一个固定地址*。既然变量都有地址，地址也用二进制编码，那么就有可能将地址作为处理的数据。问题是这样做有什么价值。

许多高级语言把程序对象（如变量）的地址作为一种可处理数据，称为*地址值*或*指针值*，以地址为值的变量称为*指针变量*，简称*指针*（pointer）。我们知道，机器语言层对各种对象的操作都要通过地址。指针变量里保存程序对象的地址，通过它们就可以访问和处理有关对象。高级语言里的指针是访问程序对象的手段，以便能更灵活方便地实施操作。

对指针变量的操作包括：（1）将程序对象的地址（如变量地址，还有其他情况。为简单起见，下面以变量为例）存入指针变量，这称为*指针赋值*。当一个指针变量保存了某个变量的地址时，也说该指针*指向*了那个变量。（2）通过指针访问被指对象（变量），称为*间接访问*。指针变量 p 指向变量 x 的情况用图 7.1 示意。

由于指针值是数据，指针变量可以赋值，所以一个指针的指向在程序执行中可以改变。指针 p 在执行中某时刻指向变量 x（如图 7.1 所示），在另一时刻也可以指向变量 y（不应对此感到奇怪，就像一个整型变量在某时可能保存着 0，另一时刻可能保存着 2）。这样，同一个通过 p 使用被它指向的对象的语句，在前一时刻访问的就是 x，后一时刻访问的就是 y。这样就带来了新的灵活性，下面我们将看到这种功能的价值。

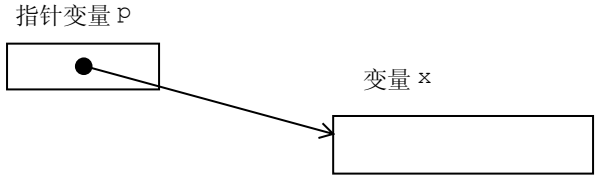

图 7.1　指针与被指的变量

C 语言的指针比其他语言的指针更灵活，功能更强，理解和

* 这种情况的例外是 C 语言的寄存器变量，它们可能被安排在 CPU 的寄存器中。这种变量没有可用的存储地址。本章所讨论的问题都把寄存器变量排除在外。

掌握都有一定难度，也比较容易用错。因此请读者在学习中特别注意思考，以逐步理解其本质，掌握正确的使用方法。下面也特别解释了指针使用中的一些常见错误，请读者留意。

指针是 C 语言的一种重要机制。用好指针机制常常可以使写出的程序更简洁有效。也有些问题必须借助指针才能处理。指针在较大的复杂软件的中使用广泛。可以说，指针使用能力是评价一个人 C 程序设计水平高低的一个重要方面。此外，指针也是大部分高级语言都提供的重要机制。掌握了 C 语言的指针机制后也可以触类旁通。

7.2 指针变量的定义和使用

C 语言的指针有类型，每个指针只能指向一种特定类型的变量，保存这种类型变量的地址。例如，如果 p 是指向 int 变量的指针，那么 p 就只能指向 int 型变量，而不能指向其他类型的变量。因此，程序里也认为 p 所指向的总是 int，从 p 间接访问的东西总作为整型变量看待。指向整型变量的指针也简称为整型指针。人们常说“int 指针 p1”、“double 指针 p2”等等。

定义指针变量时需要用类型名说明指向类型，在被定义的指针变量名前面加星号，说明定义的是指针变量。多个同类型指针可以一起定义，例如，下面定义了两个指向整型变量的指针变量（int 指针）p 和 q：

```
int *p, *q;
```

指针变量也可以与其他变量一起定义。在下面定义里，不仅定义了三个整型指针，还定义了一个整型数组和另外两个整型变量：

```
int *p, n, a[10], *q, *p1, m;
```

整型指针的类型用(int *)表示，其他指针的类型表示形式类似。

对每个类型都可以定义相关的指针类型，C 语言把指针类型也看作基本类型。所有指针占用的存储都一样大，通常是一个机器字的大小。

7.2.1 指针操作

取变量地址的操作用一元运算符&；间接访问操作用一元运算符*，也称间接操作。这两个运算符与其他一元运算符的优先级相同，自右向左结合。

取地址运算

将取地址运算符&放在变量描述（最简单情况就是变量名）前，就求出该变量的地址，这是一个相应类型的指针值，可以赋给类型合适的指针（变量）。有了前面定义，可以写：

```
p = &n;
q = p;
p1 = &a[1];
```

第一个语句将 n 的地址值赋给指针 p。这个赋值合法，因为 n 的类型与 p 所需的类型匹配，int 指针可以指向任何 int 变量。赋值后 p 指向变量 n，通过 p 就可以间接访问 n 了。第二个语句把 p 的值赋给指针 q，这将使 q 也指向变量 n。可见，两个同类型指针可以指向同一个变量。p 和 q 都指向 n 的情况如图 7.2 所示。

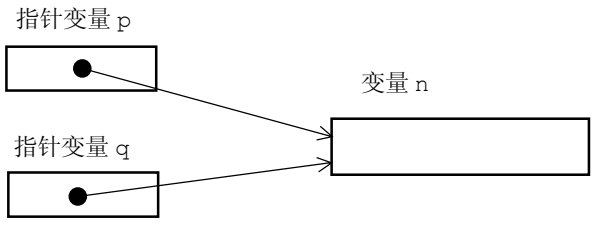

图 7.2 两个指针指向同一个变量的情况

指针变量可以做相等判断。相等就是值相等，对指针变量而言，值相等意味着两个指针指向同一位置。在图 7.2 的情况下 p 和 q 相等。

上面第三个语句的右边表达

式取出数组 a 中下标为 1 的元素的地址。由于 a 是整型数组，其元素相当于整型变量，这一地址也是指向整型变量的指针值，可以赋给指针 p1。由于数组成员访问运算[]优先级更高，赋值号右边的表达式里不必写括号。

间接运算

间接运算由指针得到被指变量。这种表达式可以像普通变量一样用：放在表达式里表示取值参加运算；或放在赋值运算符左边给被指变量赋值。下面是一个间接赋值：

```
*p = 17;
```

因为当时 p 指向变量 n，写 *p 就相当于直接写变量 n，因此这个操作完成的是给变量 n 赋值 17。下面是另一个赋值语句：

```
m = *p + *q * n;
```

在这个语句里实际访问了变量 n 三次（两次间接访问、一次直接访问）。由于变量 n 当时的值是 17，变量 m 被赋的值是由表达式计算出的 306。

下面是另一些指针使用的例子及其解释（假定接着上面语句继续做）：

```
++ *p;          /* 使变量 n 的值加 1，变成 18 */
(*p)++;         /* 使变量 n 的值再加 1，变成 19。
                   由于结合性的规定，*p++ 的意义与此不同 */
*p += *q + n;   /* 变量 n 被赋以新值 57 */
q = &a[0];      /* 指针 q 指向了数组 a 的元素 */
*q = *p / 16;   /* a[0]被赋值 3 */
```

7.2.2 指针作为函数的参数

仅从上面讨论还看不到指针的意义。现在讨论一个在 C 程序里必须借助指针解决的问题：函数的指针参数。利用这种参数能写出可以改变函数调用时环境的函数。所谓函数调用时环境，指的是在函数调用处能访问的所有变量。下面从一个例子谈起。

假设程序里常要交换两个整型变量的值，我们想为此写函数 swap，希望调用 swap 能交换两个变量的值。由于操作中需要改变两个变量，显然不能靠返回值（返回值只有一个）。不仔细考虑也可能认为这个问题很简单，有人可能写出下面函数定义：

```
void swap0 (int x, int y) {
    int t = x;
    x = y;
    y = t;
}
```

写一段程序定义变量并实际调用这个函数，例如写出如下程序段：

```
int m = 1, n = 2;
swap0(m, n);
```

执行后会发现变量 m 和 n 的值没有变。上述定义失败的原因在于 C 语言的参数机制：调用 swap0 时 m 和 n 的值送给形参 x 和 y，虽然函数里面交换了 x 和 y 的值，但不会影响调用的实参 m 和 n。调用结束时局部变量 x 和 y 被撤消，m 和 n 的值没有变。

前面所有函数有一个共同点：它们可以通过参数使用调用环境中变量的值，但不能改变那里的变量值*。在函数 f 里以局部变量 m 调用 g(m)，绝不会改变 m。

要想让 g 能改变调用处可用的 m，必须在 g 内部把握住 m。利用指针机制可以解决这个问题：在调用时把 m 的地址（这也是值，地址值）通过指针参数传进函数 g，在 g 里对参数指针间接就能完成对 m 的各种操作，包括对 m 赋值。总结一下，利用指针解决问题的方案包括三方面：函数定义时用指针参数；函数里通过指针参数间接访问被指变量；函数调用

* 以数组作为函数的实际参数是例外。关于数组参数的实际意义，本章也将给出一个明确解释。这种情况下能改变实际参数数组元素的原因也与指针有关。

时把变量地址传给函数。图 7.3 是调用的现场情况。

这样，函数 swap 应定义为：

```
void swap(int *p, int *q) {
    int t = *p;
    *p = *q;
    *q = t;
}
```

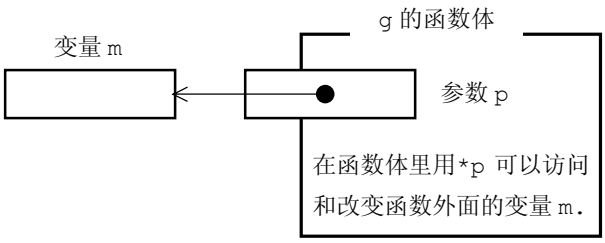

图 7.3　在函数里通过指针可以访问外面的变量

现在 swap 有两个整型指针参数。假设需要交换值的变量是 m 和 n，调用形式应该是：

```
swap(&m, &n);
```

调用中 m 和 n 的地址传递给了函数的指针参数 p 和 q。函数体里通过对 p 和 q 的间接访问，就能交换 m 和 n 值了。函数调用时形参与实参的关系如图 7.4 所示。

请注意，swap 定义的参数类型是 (int *)，调用时的实参必须是合法的整型变量地址。假设有下面变量定义：

```
int a[10], k;
```

下面两个调用都是合法的：

```
swap(&a[0], &a[5]);
swap(&a[1], &k);
```

如果有关整型变量和数组都已有值，这些调用将完成值交换工作。

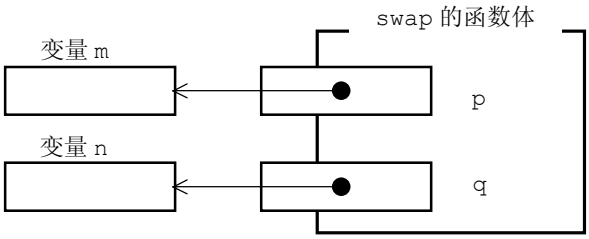

图 7.4　函数调用 swap(&m, &n)形成的现场

读者应想到标准库函数 scanf。前面介绍 scanf 时我们反复强调，接受输入值的变量前必须写符号&。那就是为了取得变量地址并把这个地址传入函数。scanf 采用的就是上面定义 swap 所用的技术，通过间接访问方式为指定变量赋值。根据什么介绍，不难想清楚 scanf 如何把它得到的输入值赋给我们指定的变量。

例：改造上一章最后的输入整数值并检查数值范围的函数，通过引进一个指针参数，使之能更好地处理整数输入中出现错误的问题。

前面讨论了函数的实现方法，但那里要用一个特殊整数值指明输入出错情况，遗留下了一个需要改进的缺陷。指针参数提供了一种解决问题的新方法。

我们可以给函数增加一个指针参数，通过它送回实际读入的值。这样就可以使函数返回值空出来，专门用于传递函数的执行状态信息。我们用 1 表示函数成功读入了一个整数，用 0 表示输入遇到麻烦而没有正常完成输入。修改后的函数定义是：

```
int getnumber(char prompt[], int imin, int imax, int repeat, int *np) {
    int i;
    *np = 0; // 为了安全，保证函数里一定给 *np 赋了值。
    for (i = 0; repeat <= 0 || i < repeat; ++i) {
        printf("%s", prompt);
        if (scanf("%d", np) != 1 || *np < imin || *np > imax) {
            printf("Wrong input. Correct range [%d, %d].\n",
                   imin, imax);
            while (getchar() != '\n') ;
        }
        else return 1;
    }
    return 0;
}
```

前面的调用现在可以重写为：

```
getnumber("Choose a range [0, n]. Input n: ", 2, 32767, 5, &m);
getnumber("Your guess: ", 0, m-1, 5, &guess);
```

调用这一函数的更合适形式应该是，例如：

```
if (getnumber("Your guess: ", 0, m-1, 5, &guess) == 0) {
    /* 输入出错处理程序片段 */
}
```

注意，虽然这类函数的参数被定义为指针，作为实际参数必须是某个变量的地址。将这一函数代换到原程序中是非常简单的，这一工作请读者自己完成。

从这个函数实例中可以清楚地看到标准库函数的影子。scanf 也是通过指针参数得到存放输入值的变量地址，也用返回值通知执行中遇到的情况，通告输入成功完成或是遇到特殊情况。调用它地程序段检查函数的返回值后，就可以针对不同情况采取适当对策了。不难看出，标准库的许多函数（尤其是输入输出函数）的设计都采用了这种模式。在实际的 C 程序设计中，采用函数返回值表示操作状态信息的技术得到普遍应用。

7.2.3 与指针有关的一些问题

在讨论更多与指针有关的编程技术与应用之前，先介绍一些与指针有关的基本情况。

空指针

空指针是个特殊指针值，也是唯一对任何指针类型都合法的指针值。一个指针变量具有空指针值，表示它当时没指向有意义的东西，处于闲置状态。空指针值用 0 表示，这个值绝不会是任何程序对象的地址。给一个指针赋值 0 就表示要它不指向任何有意义的东西。为了提高程序的可读性，标准库定义了一个与 0 等价的符号常量 NULL，程序里可以写：

```
p = NULL;
```

或者：

```
p = 0;
```

这两种写法都将指针 p 置为空指针值。前一种写法使读程序的人容易意识到这里是一个指针赋值，采用这种写法时必须通过 #include 包含了某个标准头文件。

指针初始化

在定义指针变量时，可以用合法的指针值对它进行初始化。例如下面定义：

```
int n, *p = &n, *q = NULL;
```

如果定义指针变量时没做初始化，外部变量和局部静态变量将自动初始化为空指针（0 值），局部自动变量和寄存器变量不自动初始化，建立后的值不确定。

指针使用中的常见错误

使用指针的最常见错误就是非法的间接访问，也就是说，在一个指针并没有指向合法变量的情况下对它做间接访问。例如下面的程序片段：

```
int f (...) {
    int *p, n = 3;
    *p = 2;
    ...
}
```

语句“*p = 2;”是错误的。因为在执行这个语句时 p 没有指向任何整型变量，定义 p 时没对它做初始化，后面也没做过指针赋值。这时通过 p 间接赋值会把值赋到何处？这个语句是个大错误。请不要到计算机上试验这种语句，它可能引起的后果无法预料。

当一个指针没有保存当时合法的变量地址时，人们称它是悬空指针或者野指针。通过悬空指针进行间接访问是极危险的，我们完全不能保证这种操作访问的目标位于合法数据区内，由此引起的问题可能极其严重。通过悬空的指针做间接赋值就更加危险，因为这种操作得以执行将改变某些内存单元的值，后果根本无法预料。这方面常见的错误写法包括（假设下面语句执行时，p 是一个悬空的整型指针，n 是一个整型变量）：

```
int p, n = 3;
swap(p, &n);
```

```
scanf("...", p);
scanf("...", n);
```

第一个调用的第二个参数不错，但不知要将 n 值与哪里的值交换（因为 p 当时是悬空的）。第二个调用不知把 scanf 读入的值送到那里；第三个调用里没写&，scanf 会把 n 的值当作地址（指针值）去做间接赋值。编译程序通常不能查出这些错误，例如 scanf 的两个调用一般都查不出错。但这里确实有严重语义错误，这些语句的执行后果无法预料。

另外，指针具有空指针值时也没指向合法变量，显然，通过这种指针的间接访问也是没道理的、非法的。

通用指针

前面一直说 C 语言的指针都有类型，实际上也存在一种例外。C 语言里有一种通用指针，它们可以指向任何类型的变量。通用指针的类型用 (void *) 表示，因此也称为 void 指针。下面的第三行定义了两个通用指针：

```
int n, *p;
double *q;
void *gp1, *gp2;
```

可以直接把任何变量的地址赋给通用指针。例如，有了上面定义，下面赋值是合法的：

```
gp1 = &n;
```

可以把通用指针的值赋给普通的指针，规定：如果通用指针 gpt 当时指着变量 g，而 g 的类型是另一普通指针 pt 的类型，那么把 gpt 的值赋给 pt 后，通过 pt 就可以正确访问变量 g。如果被赋值指针与通用指针所指变量的类型不符，这种赋值操作没有任何保证。对符合规定的赋值也需要写强制转换。例如，有了前面定义和赋值，现在可以写：

```
p = (int *)gp1;
```

由于 gp1 当时指向整型变量 n，所以，把 gp1 的值赋给 p 是合理的，赋值后通过 p 可以正确访问变量 n。在同样情况下，下面赋值就不合法：

```
q = (double *)gp1;
```

因为当时通用指针 gp1 实际指向整型变量，把这个指针值赋给双精度类型的指针显然没有意义，是严重错误。编译程序不能识别这种语义错误，需要写程序的人注意。

通用指针的另一特殊之处在于这种指针不能做间接运算，取得被指对象。普通指针可以做间接运算，是因为这些指针的指向类型是明确的，间接后就可以作为某类型的变量用，得到的值类型、如何参加运算、赋值该怎样做等等都完全是清楚的。通用指针可以指向任何变量，上面这些问题就都无法确定了。

指针转换*

初学者往往对指针转换感到很迷惑。实际上，因为指针就是地址，指针类型转换并不需要改变指针值。例如，把 int 变量 n 的地址（整型指针值）赋给通用指针 gpt，gpt 保存的就是 n 的地址。把这个地址转换回整型指针赋给整型指针 p，显然不会出问题。因为这就是使 p 指向了变量 n。

从更深入的观点看，一个指针类型代表着一种观点。通过整型指针的间接总看成是一个整型变量；被双精度指针指向的东西也总看成是双精度类型的变量。这也就是指针需要类型的根本原因，因为这样才能有效地通过指针去使用被指变量。

通用指针对被指的东西不提供任何类型信息，所以我们就不能通过通用指针去直接使用被指的东西。从这个角度看，通用指针也是最没有用的指针，这种指针唯一的用途是保存和提供指针值。

这样，指针转换就可以看作一种观点转换。把一个整型指针转换为一个通用指针，地址并没有改变，但却把类型信息丢掉了。C 语言保证，如果恢复指针原来的类型（例如在这个情况下，就是从通用指针转回整型指针），仍然可以通过它正常使用被指的变量。

7.3 指针与数组

C语言的指针和数组间有密切关系，在这里可以以指针作为媒介，方便地完成对数组成员的各种操作。人们在写 C 程序时常常采用这种方式，本节讨论这方面的问题。当然，通过指针访问数组元素时，同样需要注意不出现数组越界访问的错误。

C语言中指针和数组的关系是它所特有的，除了由C语言派生出来的一些语言（如C++等）之外，一般程序语言里并没有这种关系。值得提出的是，这种关系现在已经被借用到其他地方，成为一种很有用访问数据集合的一般性技术。

7.3.1 指向数组元素的指针

数组元素可以看作是相应类型的变量。因此，只要类型匹配，完全可以让指针指向数组元素。前面给出过这方面的例子。本节的讨论假设已有了下面变量定义：

```
int *p1, *p2, *p3, *p4;
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

此后可以写：

```
p1 = &a[0];
p2 = p1;
p3 = &a[5];
p4 = &a[10];
```

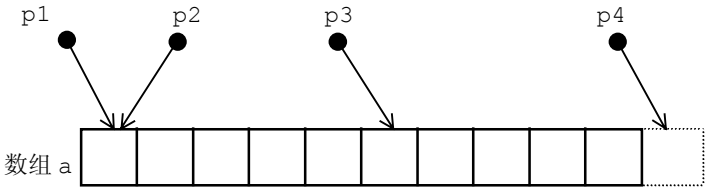

图 7.5　指向数组 a 的元素的 4 个指针

在完成了这些赋值后，指针 p1 和 p2 都指向 a 的首元素，p3 指向数组元素 a[5]，p4 当时指向的并不是 a 的元素，而是数组 a 最后一个元素之后的下一个位置。C语言保证这种“数组之后一个位置”的地址一定存在，因此可以用这种位置作为指针值。当然，此时 p4 并不指向合法的数组元素，因此通过 p4 的间接访问是错误的，其效果无定义。上面写了&a[0]一类的表达形式时，由于数组元素访问运算符[]的优先级更高，这里不必写括号。

图 7.5 显示出上述语句执行后各个指针与数组 a 的关系，以及它们间的位置关系。在目前情况下，通过对指针 p1 或 p2 的间接操作都可以间接访问 a 的首元素，通过对 p3 的间接操作可以访问元素 a[5]，但是现在不能对 p4 做间接，因为它没有指向合法元素。下面马上就能看到像 p4 这样指向数组末元之后一个位置的指针的用处。

语言规定，如果在表达式里直接写出的数组名，求值得到的将是该数组的首元素的地址。因此 p1 = &a[0] 可以简写为 p1 = a，效果完全一样。

指针运算

当一个指针指向数组中的某个元素时，程序里不但可以通过该指针访问被指元素，还可以通过它访问数组里的其他元素。例如，在图 7.5 的情况下，p1 指向 a 的首元素。此时不仅 p1 是合法指针（指向元素 a[0]），通过它可以访问 a[0]，表达式 p1+1 也代表一个合法指针，它的值就是元素 a[1]的地址。也就是说，通过 p1+1 可以间接访问 a[1]。与此类似，表达式 p1+2、p1+3、…、p+9 也是合法指针值，通过它们可以去间接访问数组 a 的其他元素。例如可以写：

```
*(p1 + 2) = 3;
p2 = p1 + 5;
```

这里的第一个语句给数组元素 a[2]赋值 3；第二个语句给指针 p2 赋了由表达式 p1 + 5 计算出的指针值，也就是数组元素 a[5]的地址。

即使一个指针所指向的不是数组首元素，只要它指向数组中的某个元素，就可以通过它去访问数组的其他元素。假如在上面的一系列赋值语句之后又有语句：

```
*(p2 + 2) = 5;
```

由于p2当时指向a[5]，p2 + 2表示由此向后两个元素位置，也就是a[7]的地址。所以这个语句给a[7]赋了值5。当指针指向数组中间的元素时，还可以通过减去一个整数的方式访问位于当时指针所指位置之前的元素，例如：

```
*(p2 - 2) = 4;
```

这个语句实现对a[3]的赋值（因为p2当时指向a[5]）。这类由指针值出发进行的运算称为**指针运算**。显然，可以用指针运算得到的值进行指针更新，例如可以写：

```
p2 = p2 - 2;
```

这使指针p2在数组里向下标小的方向移了两个位置，使它指向数组元素a[3]。在指针指在数组里的情况下，人们经常用增量、减量操作做指针值更新。例如写：

```
p3 = p2;
++p3;
--p2;
p3 += 2;
```

这些操作可以使指针改指向数组中的下一元素或前一元素，或者向某个方向移动几个元素的位置。当然，在用这种方式移动指针时，也要保证指针值位于合法范围内。

请特别注意，通过指向数组的指针访问数组元素，同样必须保证所有间接访问都在数组合法范围之内，否则也是越界访问。此外，还要求通过指针运算取得的指针值（即使不做间接）不超出数组首元素，不超出末元素之外一个元素位置，否则得到的值也没有保证。这种规定既有实际使用价值，也牵涉到计算机系统本身的限制。

当两个指针指在同一数组里面时，可以求它们的差，得到的结果是这两个指针之间的数组元素个数（是一个带符号整数）。例如，对于图7.5的情况，语句：

```
n = p3 - p1;
```

将使变量n取得值5。与此类似，p1 - p3将得到-5。显然，当两个指针并不指向同一数组时，求它们的差就完全没有意义了，因为不知道会得到什么值。

如果两个指针指在同一个数组里，那么还可以比较它们的大小。例如可以写：

```
if (q1 > p1) ....
```

如果q1所指元素位于p1所指元素之后，则条件成立（条件表达式求出值1），否则条件就不成立（值0）。如果两个指针不指在同一个数组里，对它们做大小比较也没有意义。

相等和不等比较的使用面更广。同类型指针可以比较相等或不等，任何指针都可以与通用指针比较相等或不等，任何指针都可以与空指针值（0或NULL）比较相等或不等。当两个指针指向同一数据元素或同时具有空指针值时认为它们相等，否则就是不相等。

通用指针可以与任何指针比较相等和不等，也可以与任何指针比较大小（如果它们指在同一数组里，但不能做其他指针运算（如加减整数等）。

数组写法与指针写法

当一个指针指在某数组里时，要通过指针去访问数组元素，也可以采用普通的数组元素访问形式，将这个指针写在数组名的地方。假设指针p1指向数组a的首元素，指针p3指

> 指针运算原理
>
> 为什么当一个指针指向数组时，可能计算出它所指数组中下一元素的位置？原因是普通指针具有确定类型，总是指向确定类型的数据对象，而这种数据对象的大小总可以静态确定。当指针p指向数组a时，由于p的指向类型与a的元素类型一致，而这种类型的一个数据项占据的存储大小已知，显然指针值p+1可以根据指针p当时的值和数组元素的大小算出来。知道数组里一个元素的位置，就可以计算出下一个元素的位置，或几个元素之后的元素位置，这是指针运算的基础。
>
> 对于通用指针，即使它指在数组里的某个地方，因为没有确定的指向类型，对它也不可能做一般的指针计算。对通用指针有效的指针运算只有比较。

向数组 a 下标为 5 的元素，我们就可以写下面形式的语句：

```
p1[3] = 5;
p3[2] = 8;
```

第一个语句里的 p1[3] 相当于*(p1+3)，这是给元素 a[3] 赋值。p3[2] 相当于*(p3+2)，因此第二个语句给元素 a[7] 赋值 8。

p[3] 这样一类的写法被称为*数组写法*，而*(p+3) 一类的写法称为*指针写法*，这两类写法具有同等效力，写程序时可以根据自己的喜好自由选择。

另一方面，在从数组名出发去访问数组元素时也可以采用指针写法。实际上，这不过是前面讲过的一个情况的自然推论。读者应记得：对程序里写出的数组名求值，将得到指向数组首元素的指针值。例如，对定义好的数组 a，对 a 的求值结果是指向 a[0] 的指针值。这个值自然可以参加指针运算。所以，访问数组成员 a[3] 也可以写为*(a+3)，因为 a+3 的计算得到的是指向 a[3] 的指针值，对它间接访问就是访问 a[3]。

数组名还可以参加另一些指针运算，例如与其他指针比较大小，比较相等或不等，等等。但是必须注意，数组名并不是指针变量，不能把它当作指针变量使用，特别是不能对它赋值，也不能做其他更新操作。例如，下面操作都是错误的：

```
++a;
a += 3;
a = p;
```

因为这些操作试图更新 a 的值。有些运算虽然不是对数组名赋值，但也可能没有意义。例如：表达式 a - 3 显然不能得到合法指针值，因为它超出了数组的范围。

7.3.2 基于指针运算的数组程序设计

指针运算也是程序设计的需要，它提供了另一种处理数组元素的方式，这种方式有时非常方便。假定有了上面定义的数组 a 和各指针，我们可以通过下面各种方式遍历数组 a，打印出 a 中各个元素：

```
for (p1 = a, p2 = a+10; p1 < p2; ++p1)
    printf("%d\n", *p1);

for (p1 = a; p1 < a+10; ++p1)
    printf("%d\n", *p1);

for (p1 = p2 = a; p1 - p2 < 10; ++p1)
    printf("%d\n", *p1);

for (p1 = a; p1 - a < 10; ++p1)
    printf("%d\n", *p1);
```

还能写出另一些功能相同的循环。

7.3.3 数组参数与指针

现在可以给函数的数组参数的一个清楚解释了。C 语言规定，函数的数组参数实际上就是相应类型的指针参数。例如，下面两个函数头部：

```
int f(int d[]) {... ...}
int f(int *d) {... ...}
```

完全等价。数组参数就是这样实现。

根据前面的讨论，在调用 f 处理数组时，实参应是被处理数组的名字，对这个名字求值得到一个指针值，正好符合函数对实参的类型要求，指针值被传给形参。至于函数里的参数使用，前面例子里一直采用数组元素访问的形式。上面已解释过，对指针确实可以用这种写法。这样，通过这种指针形参，函数里实际访问的就是调用时通过指针传来的数组。C 语言正是通过这一套方式实现对数组的直接操作，使我们能在函数里修改实参数组。这一解释也说明，对于数组参数，函数里完全可以用指针描述形式写数组元素访问。

这一说明也解释了前面讨论过的另一个问题。如果在函数里用 sizeof 计算“数组参数”的大小，得到的就是一个指针变量的大小，因为这种参数就是指针参数。C语言里所有指针的大小都一样，因为它们保存的都是地址值，而地址值都用同样方式表示。

前面采用下面形式写处理数组的函数：

```
double sum(int n, double a[]) {
    int i;
    double s = 0.0;
    for (i = 0; i < n; ++i)
        s += a[i];
    return s;
}
```

这里的 n 是为了传递被处理数组的长度而设的附加参数。假设程序里有双精度数组：

```
double b[40];
```

并且数组 b 的元素已经有了值。用函数 sum 可以求出该数组所有元素的和：

```
x = sum(40, b);
```

用这个函数也可以求数组 b 前面一段元素的和，例如求数组中前 20 个元素之和：

```
y = sum(20, b);
```

实际上， sum 根本就不知道 b 的大小，它只知道由第一个参数得到数组首元素地址，从这里开始求出连续 20 个元素的和，这正是我们所需要的。这种函数也可用于求数组中一段的和。例如，如果希望求出从下标为 17 的元素开始的 11 个元素之和，可以写：

```
y = sum(11, b+17);
```

7.3.4 指针与数组操作的程序实例

本节里用几个完成字符串操作的函数实例说明通过指针操作数组的方法。

例 1，用指针方式实现计算字符串长度的函数。第一种实现方式是：

```
int strLength (const char *s) {
    int n = 0;
    while (*s != '\0') {
        ++s;
        ++n;
    }
    return n;
}
```

函数里不断更新局部指针变量，逐个扫描参数串中的字符，直到字符串结束，得到的计数值就是字符串长度。这里参数用关键字 const，说明这一函数不会改变对应实参（被处理的字符数组）。当然，把参数定义为常量，就要求在函数体里确实不改变实际参数。

函数的另一种实现方法是：

```
int strLength (const char *s) {
    char *p = s;
    while (*p != '\0') ++p;
    return p - s;
}
```

这个方法很有趣，而且循环中只需更新一个变量。当指针 p 指到串结尾的空字符时，两个指针之差就是它们之间的字符个数，也就是字符串的长度。应特别指出，虽然上面两个函数的参数定义为指针类型(char*)，调用时却应该用字符串常量或字符数组（其中应存着字符串）作为实际参数。当然，也可以用一个实际指向字符串的指针作为参数。

例 2，用指针方式实现字符串复制。下面函数把由第二个参数确定的字符串的内容复制给由第一个参数确定的字符数组：

```
void strCopy (char *s, const char *t) {
    while ((*s = *t) != '\0') {
        s++;
        t++;
    }
}
```

在复制空字符的同时循环条件失败，字符串的复制正好完成。显然只有第二个参数可以用 const 说明。由于赋值表达式也有值，而空字符'\0'的值就是 0，上述程序可以简化为：

```
void strCopy (char *s, const char *t) {
    while (*s = *t) {
        s++;
        t++;
    }
}
```

熟悉 C 语言的人们常把指针更新操作也写在循环的测试条件里，修改后的程序是：

```
void strCopy (char *s, const char *t) {
    while (*s++ = *t++)
        ;
}
```

这里的循环体是空语句。要理解这个程序的意义，必须注意运算符的优先级与结合顺序，增量运算的作用与计算出的值，赋值表达式的值等等。理解这个程序对于第一次见到它的人都是一个小测验，请读者自己设法把程序的意义弄明白。

这两个函数的意义与使用方法与标准库函数中相应的字符串函数相同。

例 3，一些借助于指针的数组函数。

下面是一个利用指针输出整型数组元素里任一子序列的函数：

```
void prt_sequence(int *begin, int *end) {
    for (; begin != end; ++begin)
        printf("%d\n", *begin);
}
```

在调用这一函数时，只要保证对应 begin 和 end 的实参都指向同一数组，保证 begin 的实参不在 end 的实参之后（也就是说，保证它们确实描述了某数组中的一个子序列，可以是空序列），这个函数就能顺序输出该子序列中的各个元素。下面是一些使用实例：

```
prt_sequence(a, a+10);
prt_sequence(a+5, a+10);
prt_sequence(a, a+3);
prt_sequence(a+2, a+6);
prt_sequence(a+4, a+4);
prt_sequence(a+10, a+10);
```

最后两个调用处理的是空序列，循环将在第一次检测时失败，函数立即结束。注意，这里由 end 所指的元素（可能并不存在）不在打印之列，这是一种通行做法。人们常说这样的两个指针描述了一个“半闭半开”的序列，因为它不包含最后指针所指的那个元素（也可能不是真正的元素）。

采用同样思想，我们可以写出许多通过指针操作数组中元素序列的函数。例如，下面定义了一个“设置”函数，它将一个整型元素序列中的元素都设置为某个确定的值：

```
void set_sequence(int *begin, int *end, int v) {
   for (; begin != end; ++begin) *begin = v;
}
```

下面函数将序列里的每个元素都用其平方根取代：

```
void sqrt_sequence(double *begin, double *end) {
   for (; begin != end; ++begin) *begin = sqrt(*begin);
}
```

也可以采用这种方式定义求数组的子序列中元素平均值的函数：

```
double avrg(double *begin, double *end) {
    double *p, x = 0.0;
    if (begin == end) return 0.0;
    for (p = begin; p != end; ++p) x += *p;
    return x / (end - begin);
}
```

例 4，考虑用指针参数的方式写出上一章的数组元素划分函数。

当然可以重复前面写法，只是将数组参数换成指针参数。现在考虑一种更符合指针精神的定义方式：为函数定义一对界定数组范围（或数组中一段）的指针参数和一个划分值；让函数返回划分后的分界位置，这里用指向数组元素的指针值，令它指向大于等于划分值的第一个元素。如果不存在这种元素，返回指向（数组）序列后面一个位置的指针值。

这里也采用“半闭半开”的序列，第一个指针参数指向序列的第一个元素，第二个指针参数指向序列最后元素之后一个位置。按这些考虑和有关分析，可以给出下面的定义：

```
double* partition(double *begin, double *end, double cut) {
    while (begin < end) {
        while (begin < end && *begin < cut) ++begin;
        while (begin < end && *(--end) >= cut) ;
        if (begin < end) {
            double x = *begin;
            *begin = *end;
            *end = x;
            ++begin;
        }
    }
    return begin;
}
```

这里的大循环内部还是用了两个 while 循环，其中的一个更新向右移的指针，另一个更新向左移的指针。在条件语句内部定义了一个用于交换元素的临时变量。第二个内部的 while 语句值得注意。该循环以空语句为体，循环条件中还包括对变量 end 的更新。当然也可以用更普通的形式写这个循环。但目前写法很紧凑，熟悉 C 语言的人们经常采用这类简洁写法，因此，在学习 C 语言和程序设计中也应逐渐习惯这类写法。

图 7.6 展示了上述函数对一个数组的处理过程，假定函数调用时以 5 作为数据的分界值。(1) 函数开始时，两个参数指针分别指向序列两端（半闭半开）；(2) 两个内部循环第一次执行之后，两个指针分别指向一对需要交换的元素；(3) 交换元素并更新指针之后的现场；(4) 两个内部循环执行之后，begin 指针没有动，end 指针向左移了一个位置；(5) 交换并移动指针之后，循环条件失败，循环终止。返回的指针值指向划分之后大于等于划分值的那一段

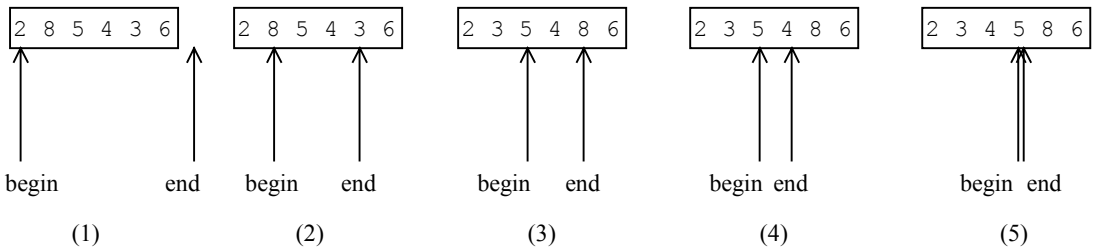

图 7.6 用 5 作为分界值划分数组

的第一个元素。

7.3.5 字符指针与字符数组

按照类型，字符指针（char*）应是指向字符变量。但在实际应用中人们常用字符指针指向字符数组的元素，以便通过这种指针使用字符数组的内容。最常见情况是令字符指针指向字符串：或是指向一个常量字符串，或是指向存储着字符串的字符数组。通常情况是字符指针指向字符串的开始*。

可以在字符指针定义时用字符串常量进行初始化，例如：

```
char *p = "Programming";
```

这个定义有多重含义，实际上完成了三件事：（1）定义了字符指针 p；（2）建立了字符串常量"Programming"，它以字符数组形式存储，最后有一个空字符；（3）给 p 指定初值，使它指向刚建立的那个字符串常量。结果如图 7.7 (a) 所示，其中的\0 表示空字符。

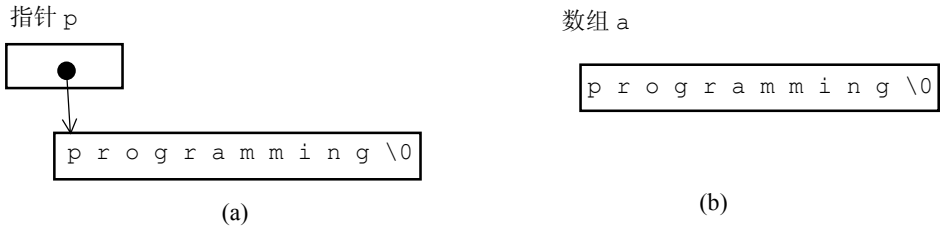

图 7.7 指针指向的字符串和存储在数组里的字符串

显然，上述定义的效果与定义一个字符数组并用同一个字符串进行初始化不同。如果我们定义了字符数组 a 并给以初始化：

```
char a[] = "Programming";
```

这个定义产生的就是一个字符数组，数组元素给了初始值，产生的现场形式如图 7.7 (b)。这里并没有建立字符串常量，只是在初始化描述中借用了字符串常量的写法；这里也没有指针，a 是数组名。可以看到上面两个定义的意义有相近之处，也有许多差异。相近之处请读者考虑，两者之间差异至少有如下一些方面：

1. 所定义的 p 是指针，虽然定义时令它指向创建的字符串常量，但这并不妨碍随后的程序里重新给它赋值，使之指向其他地方，或给它赋空指针值。而数组名 a 总表示被分配的那一块存储区域，也不能给它赋值。有了上面定义后，我们可以写语句：

   ```
   p = "Programming Language C";
   ```

   这个语句建立另一字符串常量，并把这个字符串常量的起始地址赋给指针 p。把这里的指针 p 换成 a 就是一个非法操作。
2. p 和 a 的类型不同，大小也不同。p 只占一个指针所需的存储空间，而 a 占据 12 个字符的空间，其中存放字符序列 Programming 的各字符和一个空字符。
3. 数组 a 的成分可以重新赋值。例如可以做下面的操作：

   ```
   a[8] = 'e';
   a[9] = 'r';
   a[10] = '\0';
   ```

   在这几个赋值后，存在数组 a 里的字符串变成了"Programmer"，在有效字符序列之后还有两个空字符。在指针指向字符串常量时，C 语言不允许通过它去做间接赋值（不允许修改字符串常量的值）。做这种修改是语义错误，其后果无法预料。

仔细分析，还可能找出一些其他的不同点。

我们自然可以定义字符指针变量，并让它指向已有的字符数组。人们写 C 程序时常采

* 实际上，当一个指针指到某个字符串的中间时，我们仍然可以把被它指向的东西当作字符串使用。当然，这时实际使用的只是整个字符串的后面一部分而已。

用这种方式去使用和操作字符数组内容。例如，下面程序段统计字符串里字符 e 的个数：

```
char uname[] = "What is done can't be undone.";
char *p;
int count;
for (count = 0, p = uname; *p != '\0'; ++p)
    if (*p == 'e') ++count;
```

7.4 指针数组

指针也是数据，自然就可以定义指针的数组。指针数组在复杂的程序里使用广泛，本节讨论与之相关的一些问题。这里的讨论主要以字符指针数组为例，对于其他类型指针的数组，这里的讨论也可以作为参考。后面章节里将有这方面的例子。

字符指针数组是 C 程序的一种常用结构。假如程序里需要一组字符串，一种常见做法就是用一个字符指针数组表示它们。一个典型实例是软件系统的错误信息。软件在运行中出现错误时可能需要显示某些错误信息，将情况通报给使用软件的人。这种错误信息通常用一些字符串表示。但是，很可能程序里的许多地方需要显示同样的错误信息。虽然可以用字符串常量形式把这些信息分散写在各处，但这使信息管理变得非常困难，需要统一修改时，就会很不方便。此外，重复的信息字符串也可能占据大量额外存储空间。一种常用方法是定义一个全局的指针数组，让其中的指针分别指向表示输出信息的字符串常量。程序里任何地方需要有关信息串，都通过这个指针数组去使用。这样统一管理所有输出信息，可以给复杂程序的开发和维护带来很大方便。

下面是一个字符指针数组的定义：

```
char *pa[10];
```

运算符优先级也适用于定义和说明。在上面例子里，由于[]运算符的优先级更高，所以它定义的是数组 pa，其元素类型是(char *)。也就是说，pa 是字符指针的数组。

字符指针数组也可以在定义时做初始化。人们常用字符指针指向字符串，也常用字符串常量为数组中指针提供初始值。下面是这种用法的一个例子：

```
char *days[] = {
    "Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"
};
```

这就定义了一个含有 7 个字符指针的数组，同时还建立了 7 个字符串常量，并分别用各字符串对数组中的指针进行初始化。有了这个定义，通过数组元素指针 days[0]、days[1]等等就可以访问各个字符串，用在需要它们的地方了。例如，可以写下面语句：

```
printf("Work days: ");
for (i = 1; i < 6; ++i)
    printf("%s ", days[i]);
printf("\nWeekend: ");
printf("%s %s\n", days[6], days[0]);
```

这个程序片段将打印出：

```
Work days: Monday Tuesday Wednesday Thursday Friday
Weekend: Saturday Sunday
```

作为字符指针数组的实例，我们改写前面讨论过的 C 语言关键字统计程序，把原程序里的两维字符数组改为字符指针数组，用关键字字符串初始化。定义如下：

```
char *keywords[] = {
    "auto",    "break",    ....
    ....       "volatile", "while"
};
```

整个程序的其他部分完全不需要修改，就能够正常工作了。

7.4.1 指针数组与两维数组

读者可能会问：前面程序里使用两维的字符数组，现在使用字符指针的数组，它们之间有哪些不同呢？让我们用一个小例子说明这个问题。假设有下面两个定义：

```
char color1[][6] = {"RED", "GREEN", "BLUE"};
char *color[] = {"RED", "GREEN", "BLUE"};
```

第一个定义建立了一个两维（3×6）字符数组，它占着一片连续存储区。数组里的字符元素用三个字符串分别给定初值，所有未指定值的元素都给了空字符值。这个定义建立的现场情况如图 7.8 (a) 所示。三个成员数组里各存着一个字符串，包括表示结束的“\0”，随后的“\0”（如果存在）是自动填入的。

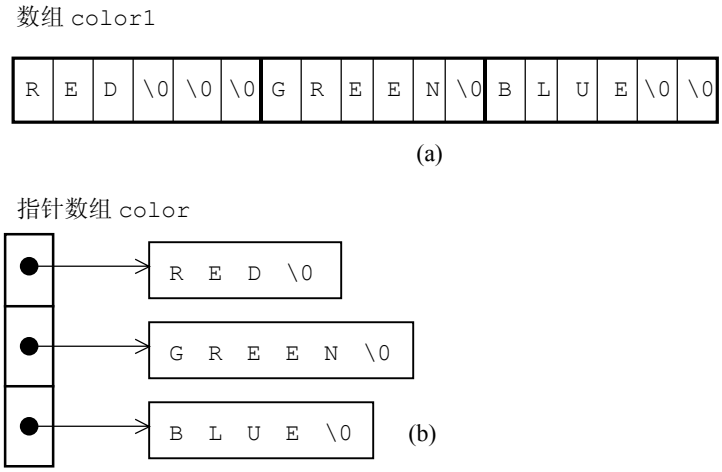

图 7.8　两维数组和指针数组

第二行定义的是一个包含三个指针的指针数组，另外还建立了三个字符串常量。这个定义也使数组里各指针分别指向相应的字符串常量，变量定义后的现场情况如图 7.8 (b) 所示。

指针可以看成数据间的一种联系机制。通过指针连接，上面第二个定义建立起的结构具有分散性：指针数组存于一个地方，三个字符串常量可以与它在一起，也可以不在一起（具体情况依赖于系统，一般系统把字符串常量存在另外的存储区域）。同样，这几个字符串可以连续存放，也可以分散存放。采用这种定义方式，系统更容易安排数据对象的存储问题。第一个定义建立的是一个“大”数组，为能在程序中正常使用，数组元素必须连续存放，这样就需要一块较大的连续存储区。如果数组非常大（例如包含数以十万计或百万计的元素），所产生的影响就更大了。本章后面还会再次提起这些问题。

有关这两个定义还有一些具体问题，前面已经讨论过。第二个定义建立起一组字符串常量，按照规定，这些常量是不能修改的。对于第一个定义显然没有这一限制，作为普通数组元素总可以根据需要随意重新赋值。另一方面，字符指针数组的成员是指针，这些指针都可以重新赋值，程序执行中可以令它们指向其他类型合适的数据对象。

虽然这里用字符指针数组和两维字符数组作为例子，实际上所讨论的情况具有一般性。无论数组的基本元素是什么类型的，这里的讨论都有意义。

7.4.2 命令行参数及其处理

要启动一个程序，基本方式是在操作系统命令状态下由键盘输入一个命令。操作系统根据命令名去查找相应的程序代码文件，把它装入内存并令其开始执行。“命令行”就是为启动程序而在操作系统状态下输入的表示命令的字符行。当然，目前许多操作系统采用图形用户界面，在要求执行程序时，常常不是通过命令行形式发出命令，而是通过点击图标或菜单项等。但实际的命令行仍然存在，它们存在于图标或菜单的定义中。

在要求执行一个命令时，所提供的命令行里往往不仅是命令名，可能还需要提供另外的信息。例如在 DOS 系统里，要用系统的编辑器编辑一个文件，我们可能输入：

```
edit file1.txt
```

文件名 `file1.txt` 就是命令的附加信息。在使用 DOS 的列目录命令时可能写出：

```
dir \windows\system /p
```

命令行中的这些额外信息以字符序列形式出现，这就是本节要讨论的命令行参数。

读者已经写过许多程序。例如，在常见微机系统中，如果源程序文件名是 abcd.c，经过编译通常产生出名为 abcd.exe 的可执行程序文件，在命令状态下输入命令：

```
abcd
```

这个程序就会装入执行。但至今我们还没有考虑过命令行参数的处理问题。要考虑这种程序，就需要了解 C 语言的命令行参数机制。

处理程序的命令行参数很像处理函数的参数。因为，写这种程序时需要考虑的是如何处理将来别人输入命令行、执行这个程序时所提供的信息。就像在定义函数时，要考虑的是处理函数被调用时提供的信息。两者确实很像。

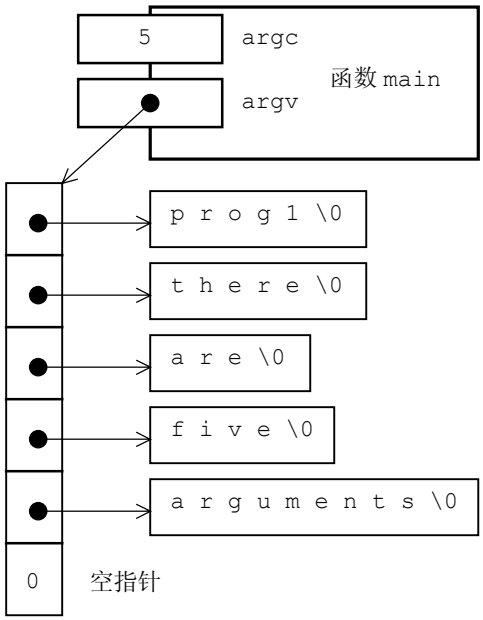

图 7.9　执行形成的命令行参数现场情况

要写能处理命令行参数的程序，实现需要了解 C 程序如何看待命令行。这里总把命令行中的字符看成由空格分隔的若干字段，每段是一个命令行参数。命令名本身是编号为 0 的参数，后面的参数依次编号。在程序启动后正式开始执行前，每个命令行参数被自动做成一个字符串，程序里可以按规定方式使用这些字符串，以接受和处理各个命令行参数。

假设有一个程序，它的名字是 prog1。假设调用这个程序时写的命令行是：

```
prog1 there are five arguments
```

对于这个命令行，字符序列“prog1”就被看着是编号为 0 的命令行参数，“there”是编号为 1 的命令行参数，其余类推。在这个命令行里一共有 5 个参数。对于另一个命令行：

```
prog1 I don't know what is the number
```

这次程序 prog1 的执行时将得到 8 个命令行参数。

C 程序通过 main 的参数获取命令行参数信息。前面程序中的 main 函数都没有参数，那就表示它们不处理命令行参数。实际上 main 可以有两个参数，这时的原型是：

```
int main (int argc, char *argv[]);
```

人们常用 argc、argv 作为 main 两个参数的名字。当然，根据对函数性质的了解，我们应该知道，这两个参数完全可以用任何其他名字，但它们的类型是确定的。只要我们在定义 main 函数写出上面这样类型正确的函数原型，就能保证在程序启动执行时正确得到有关命令行参数的信息。

当一个用 C 编写的程序被装入内存准备执行时，main 的两个参数被自动给定初值：argc 的值是启动命令行中的命令行参数的个数；指针 argv（前面讲过，数组参数实际是指针参数）指向一个字符指针数组，这个数组里共有 argc+1 个字符指针，其中的前 argc 个指针分别指向表示各命令行参数的字符串，最后是一个空指针，表示数组结束。

对于前面讨论过的程序调用

```
prog1 there are five arguments
```

当程序执行进入主函数 main 时，与命令行参数有关的现场情况如图 7.9 所示。其中 main 的整型参数 argc 保存着 5，指针参数 argv 指向一个包含 6 个成员的字符指针数组，其中前 5 个指针分别指向相应字符串，最后是一个空指针。这些都是在 main 开始执行前自动建立的。这样，在函数 main 里就可以通过 argc 和 argv 访问命令行的各个参数了：由 argc 可得到命令行参数的个数，由 argv 可以找到各个命令行参数字符串。通过编号为 0 的参数

还可以访问启动程序的命令名本身* 。

下面是一个使用命令行参数的简单例子，从中可以看到命令行参数的基本使用方法。例：写出程序 echo，它依次打印程序调用时提供的各个命令行参数。

虽然我们写程序时并不知道调用时可能提供的命令行参数是什么，但是，利用上面介绍的命令行机制，却可以让写出的程序把它们打印出来。这个程序非常简单：

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; ++i)
        printf("Args[%d]: %s\n", i, argv[i]);
    return 0;
}
```

假定这个程序的源文件是 echo.c，编译后的可执行文件是 echo.exe。执行下面命令：

```
echo programming is understanding
```

将会产生下面几行输出：

```
echo
programming
is
understanding
```

由于 argv 本身就是一个指针，利用它所指的指针数组最后有一个空指针的事实，可以给出上述程序的另一种写法（这也是命令行参数采用现在这种表示方式的原因）：

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    while(*agrv != NULL)
        printf("%s\n", *argv++);
return 0;
}
```

注意这里 *argv++ 的意思。

现在，人们经常在集成开发环境（IDE）里开发程序，程序的编辑、调试、执行等工作都可以在同一个环境里完成。这时如何为执行程序提供命令行参数呢？实际上，集成开发环境都有专门的机制为启动命令行提供参数，请读者自己找一找，查查系统手册。提供命令行参数另一种方法是转到开发环境之外，在操作系统的命令行状态下启动程序。例如启动一个命令式交互的窗口，在命令窗口里执行程序。

在图形用户界面的系统里，这里关于命令行参数机制的讨论并没有过时。前面已经说过，命令行仍然存在于启动程序的按钮或者菜单中之中。另外还有一些情况。例如，在许多系统里可以把某个文件拖到一个程序文件那里作为程序的启动参数。实际上这就是要求系统产生一个实现这种命令的命令行。此外，要建立程序项、命令菜单项等，也需要写出实际命令行，其中包括提供所需要的各个命令行参数。

7.5 多维数组作为参数的通用函数

前一章里提出了实现处理多维数组的通用函数问题。对于一维数组，通过引进长度参数。就可以定义出具有通用性的函数。它们可以正确处理特定类型的各种一维数组。多维数组的情况不那么简单。按语言规定，当函数参数是两维或更多维数组时，参数说明必须给出除第一维外其他各维的大小。这就使定义的函数失去了大部分通用性。例如写：

```
int fun1(int n, int mat[][10]) {
```

* 在一些系统里，表示命令本身的 0 号参数还包括完整的启动目录路径。

```
    ... ...
    ... mat[i][j] ...
}
```

这一函数将只能用于第二个维长为10的各种数组，并不很通用。如果把函数头部改成：

```
int fun2(int n, int m, int mat[][]) {
    ... ...
    ... mat[i][j] ...
}
```

编译时将无法通过。为什么呢？

请注意，mat是一个指针参数（数组参数就是指针参数），其指向类型是整型数组（两维数组的元素是一维数组）。但后一定义里没给出mat所指整型数组的大小，造成了指针的类型描述不完全。这样，虽然函数里知道mat[0]（是整型数组）的开始位置，却无法计算mat[1]和其他子数组的位置（因为没有mat[0]的完整类型），因此无法计算数组的基本元素mat[i][j]的位置。因此编译就无法完成。

然而实际中确实需要定义处理多维数组的通用函数。例如，科学计算中最常用的一种数据形式是矩阵，矩阵的最直接表示就是用两维数组。对于矩阵有许多标准操作，如求矩阵的和或乘积，求矩阵行列式的值，求矩阵的特征值等。我们当然不希望为每种不同大小的矩阵定义一套几乎完全一样的函数，最好是能够定义一套通用的矩阵操作函数。

C语言没提供写这类函数的标准方法，但人们为解决这个问题提出了许多实用技术。这里介绍一种可行方法，采用它可以写出处理多维数组的通用函数。下面的讨论以两维数组为例，多维数组的情况可以同样处理。

我们先看看，一般性地处理两维数组需要哪些信息。显然需要知道数组基本元素的类型，也需要两个维的长度，还需要数组的开始位置。对数组处理函数而言，由数组参数可以知道元素的类型和数组首元素的位置（开始位置）；由于数组元素按行排列，根据数组元素类型和行长，就可以确定各行首元素的位置，进而也就可以算出其他元素的位置了。

举一个例子：假设被考查的数组是：

```
int a[10][8];
```

a的首元素位置是&a[0][0]，这是个整型指针值。由于数组a每行有8个元素，所以下标为1的行的首元素位置应该是：

```
&a[0][0] + 8
```

通过表达式：

```
*(&a[0][0] + 8)
```

可以访问数组a里下标为1的行的首元素。一般说，表达式：

```
*(&a[0][0] + i * 8)
```

访问数组a里下标为i的那行的首元素。所以，表达式：

```
*(&a[0][0] + i * 8 + j)
```

访问的就是数组a里下标为i的行中下标为j的元素，即a[i][j]。也就是说，有了上面提出的各种信息，我们完全能够找到数组 a 的各个元素，因此就可以写出处理数组的通用函数了。

下面是采用这种方法写通用函数的一个例子：

例：写一个打印输出两维整型数组的函数，它把数组每行的元素打印在一个字符行里。

按上面讨论可以写出下面函数定义，函数由两个整型参数得到数组两个维的大小：

```
void prtMatrix (int m, int n, int *mp) {
    int i, j;
    for (i = 0; i < m; ++i) {
        for (j = 0; j < n; ++j)
            printf("%d ", *(mp + i * n + j));
        putchar('\n');
    }
}
```

如果要打印上面定义的数组 a，正确的函数调用形式是：

```
prtMatrix(10, 8, &a[0][0]);
```

这里&a[0][0]在类型上是指向整型的指针，这正是函数参数 mp 所要求的类型。如果需要打印另一个 20×36 的整型数组 mat，函数的正确调用形式应当是：

```
prtMatrix(20, 36, &mat[0][0]);
```

请读者根据这些函数调用实例，分析各个实际参数的写法以及函数里的计算过程。利用同样的技术也可以写出通用的多维数组处理函数。

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）保存被处理数据和各种状态信息，变量在使用前必须安排好存储：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、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。这也是用高级语言工作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，就是直接访问它们的固定存储位置。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（用于安放其中的所有自动变量）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。但这些也对写程序的方式形成了一种限制，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。由于每次使用该程序时要处理的数据项数可能不同，我们希望能在程序启动后输入一个表示成绩项数的整数（或者通过命令行参数提供一个整数，随后的问题完全一样）。对于这个程序，应该怎样建立程序内部的数据表示呢？

问题在于写程序时怎样描述数组元素的个数。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 */
```

这一做法行不通。这里有两个问题：第一是变量定义不能出现在语句之后。这个问题还不严重，可以通过引进一个复合语句，把 scores 的定义放在复合语句里的方式解决。第二个问题更根本，上面程序段里说明数组 scores 大小的表达式是一个变量，不能静态求出值。也就是说，数组大小不能静态确定，C 语言不允许这种写法。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（例如）：

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。前面许多程序采用了这一做法。如果前期分析正确，这样做一般是可行的。但如果实际需要处理的数据很多，所定义数组不够大，程序就不能用了。
2. 定义一个大数组，例如所用系统中可能定义的最大数组。这样做的缺点是可能浪费大量

空间（存储是最重要的一种资源）。如果一个复杂系统里类似数组不止一个，那就没办法了。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解决这类问题。因此需要一种机制，使我们能利用它写出一类程序，其中能根据运行时的实际需求进行存储分配，取得适当大小的存储块作为变量用。这就是本节要讨论的*动态存储分配*机制。说是“动态分配”，因为其分配方式完全是动态确定的，与程序变量的性质完全不同。

假设有了动态存储分配，可以要求系统分配一个存储块，但怎么才能在程序里掌握和使用这种存储块呢？对普通变量，程序里通过变量名实现对其存储位置和内容的访问。动态分配的存储块无法命名（命名是写程序时的手段），因此需要另辟途径。一般语言都是通过指针实现这种访问。我们用一个指针指向动态分配得到的存储块（将存储块的地址存入指针），此后通过对指针的间接操作，就可以使用这个存储块了。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。如果动态分配的存储块不用了，就应该考虑把它们交回去。动态分配和释放的工作由*动态存储管理系统*完成，这是支持程序运行的基础系统（通常称为*程序运行系统*）的一部分。这个系统管理着一片存储区，在需要存储块时，可以调用动态分配操作，申请一块存储；如果申请到的某块存储不再需要了，就调用释放操作将它交还管理系统。由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 是标准库定义的一个类型，它是一个无符号整型。这个整型能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 C 系统确定。malloc 的返回值为 (void *)类型（通用指针的重要用途），它分配一块能存放大小为 n 的数据的存储块，返回相应指针。申请不能满足时（例如申请块太大，动态存区没有足够大的块）malloc 返回 NULL。使用时应将 malloc 的返回值转换到特定指针类型，赋给有关的指针。

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 */
```

在调用 malloc 时，应该通过 sizeof 计算存储块大小，不要直接写整数，以免出现不必要的错误。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，一次动态分配得到的存储块也有确定的大小，不允许越界使用。例如上面程序段里分配的存储块能存 n 个双精度数据，随后使用时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，其中能存放 n 个大小各为 size 的元素，分配时还把存储块里全部清 0（初始化为 0 值）。如果要求不能满足，函数返回 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 对所分配区域不做任何事情，而 calloc 对整个区域进行自动初始化，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。

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。这个存储块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 将什么也不做。

注意，调用 free(p) 并不改变 p 本身的值（因为 p 是值参数，函数里不可能改变它），但 p 所指存储块的内容却可能改变了（由于动态存储管理的需要）。此后，不允许再通过 p 去访问已释放的块，否则可能引起灾难性后果。还有一点也要注意：绝不能对并非指向动态分配存储块的指针使用 free，那样做的后果不堪设想。

为保证有效使用动态存储区，在知道某块动态分配的存储不再用时，就应及时将它释放，这应该成为一种习惯。释放动态存储块只能通过调用 free 完成。下面是使用示例：

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

这里在函数 fun 退出前释放了函数内分配的存储块。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 退出也使 p 的存在期结束，此后 p 保存的信息（动态存储块地址）就找不到了。如果程序里没有产生访问该块的其他途径（例如，函数里可以把存储块地址赋给外部指针变量，这样就建立起了另一条访问路径），程序后面将永远不能再用这个块。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。许多实际系统不能容忍这种情况的发生。假如函数有多个 return 语句，在从各处退出前，都应该释放在函数里分配而且已经不再使用的动态存储块。

4）分配调整函数 realloc。函数原型是：

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

本函数用于更改以前做过的存储分配。在调用函数 realloc 时，指针 p 应指向一个以前分配的块，参数 n 表示现在需要的块大小。新要求无法满足时 realloc 返回 NULL，与此同时 p 所指存储块的内容保持不变。如果要求能满足，realloc 返回的指针指向能存放大小为 n 的数据的块，并且保证该块的内容与原存储块一致：如果新块较小，其中将存着原块在 n 范围内的数据；如果新块更大，原有数据存于新块的前面部分，新增部分不自动初始化。如果分配成功，原存储块内容就可能改变，因此不允许再通过 p 去使用它。

假如需要把现有的一个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) {
    ... /* 分配未成功，p仍然指向原存储块，处理这种情况 */
}
```

```
else {
    p = q;
    ... ... /* 分配成功，通过p可以使用具有新大小的存储块 */
}
```

这里的q是另一个双精度指针。程序段里没有直接将realloc的返回值赋给指针p，是为了避免分配失败时存储块丢失。如果直接赋值，指针 p 原来的值就会丢掉。如果分配没成功，原来的存储块就再也无法找到了。

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。为了整洁清晰，我们考虑将筛法计算写成函数。这里还有一个小问题：如果用户通过命令行参数指明要求的工作范围，程序里就需要从由参数得到的字符串求出对应的整数。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函数可以定义为：

```
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);
```

使用动态存储管理

1）注意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）系统对所分配存储块的使用完全不进行检查。写程序的人需要保证这种使用的正确性，切不可超出实际存储块的范围进行访问。

3）动态分配存储块的存在期不依赖于分配该块的位置。如果在某函数里分配了一个块，这个块的存在期与该函数的执行期无关。只有通过free释放这个块，才使其存在期结束。注意，变量存在期的结束时刻就是它所占存储被收回的时刻。

4）如果在某函数里分配了动态存储，并通过局部指针访问这种存储块，那么在函数退出前就必须考虑如何处理这些存储块：或是将它们释放；或是将它们的地址赋给存在期更长的指针变量（如全局变量）。否则，在函数退出时局部指针变量撤销，它们所指的尚未释放的存储块就再也找不到了（流失了）。

5）其他情况也可能造成存储块丢失，例如一个指向动态块的指针赋了其他值，如果原被指存储块没有其他访问路径，那么就再也无法找到它了。如果存储块丢失，在本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）请注意计算机系统里存储管理的关系。一个程序运行时将从操作系统取得一部分存储空间，用于保存其代码和数据。用于数据存储的空间里包括一部分动态存储区，由程序里的动态存储管理系统管理。在这个程序的运行期间，所有动态存储申请都由这块空间里分配。程序代码中释放存储，就是将不用的存储块交还程序的动态存储管理系统。一旦该程序结束，操作系统将收回这个程序所获得的所有存储区域。所以，我们所说“存储流失”是程序内部的问题，而不是整个系统的问题。

```
    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;
}
```

主函数清晰地分为三部分：准备工作，处理，输出与结束。如果程序得到的检查范围不合要求，它就打印错误信息并立即结束。正常情况下进入程序完成筛法并产生输出。

函数 getnumber 可以直接利用已有的定义（从这里又看到函数的价值），剩下的工作就是定义程序里所需要的两个函数。从数字字符串转换产生整数的函数很简单：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;
}
```

这个函数里没有检查计算溢出的问题。如果需要，也可以设法加入这种检查。此外，在这里也可以直接使用标准库函数 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 章的有关介绍。

将筛法计算包装为函数的工作很容易完成，下面是函数定义：

```
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
}
```

把这些函数定义（包括 getnumber 的定义）写入一个源文件，适当安排函数位置，必要时加入原型说明。在源文件前部加入适当 #include 命令行，程序就完成了。

在这个程序里需要存储一批数据。由于数据的数目在写程序时无法确定，因此我们采用动态存储分配的方式。这里申请了一个大存储块，其中可以存放一系列 int 值。用指针指向这种存储块，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，使之能处理任意个学生的成绩。

这里的重点是讨论一种常见问题的处理技术：用动态分配的数组保存事先无法确定数量的输入数据。前面程序用了一个全局数组，因此限制了能处理的成绩数目。现在想修改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，将输入数据存入其中。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，进行目前的功能扩充时只需要修改其输入部分，并对 main 做非常局部的修改，其他部分则根本无须任何变动。

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。举例来说，可以将动态数据块的初始大小确定为 20，但随后如何调整却是一个值得研究的问题。下面采用的策略时每次调整时将容量加倍，有关不同调整方式的分析在后面的方框中。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;
}
```

这个函数里用变量 curnum 记录当前分配块的大小，用 n 记录当前存入的数据项数。一旦数据块存满而且还有新项时就扩大存储量。

这个函数定义主要是显示了在处理类似问题时常用的一种基本技术，在这里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。有关数据检查和处理等都是前面已经讨论过的问题，进一步修改这个输入函数，使之能够合理地处理输入数据错误，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，这些都留给读者作为进一步的工作。

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，并将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都反复使用了。这时函数里甚至不知道被处理的是程序里定义的数组变量，还是动态分配的存储块。例如，完全可以用如下方式调用筛法函数：

```
int ns[1000];

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

在前面的筛法程序实例中，我们先在主函数里通过动态分配动态取得存储，而后调用函数 sieve，最后还是由 main 函数释放动态存储。这样，分配和释放的责任位于同一层次，由同一个函数完成。这样做事情比较清晰，也易于把握，是最好的处理方案。

动态调整策略

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”），首先需要考虑所采用的增长策略。

一个简单的考虑是设定一个增量，例如 10，每次存储区满时就将存储区扩大 10 个元素位置。经过仔细考虑和计算会发现，这样做存在着很大缺陷。实际中对存储量的需要常常是逐步增加的。一般说，是遇到存储区满时另行分配一块更大的存储，将向量中已有的元素复制到新块中。realloc 完成这种操作的代价（虽然不显露出来）通常与数组已有的元素个数成正比。

假设构造中执行了一系列的增加动作，如果每加入 10 个元素做一次复制，将这个数组从 20 增加到包含 1000 个元素，所做的总复制数将是 20+…+980+990 = 49990。这样，加入每个元素平均大约 n/20 次复制，n 是元素个数。加入到 1000000 个元素时，平均每加入一个元素要做 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，平均加入一个元素做一次复制。增长策略的作用确实很大。当然，如果实际数组很小，两种策略的差异就不明显了。

采用后一种增长策略也有代价（没有免费的午餐），代价就是存储空间。由于每次加倍，数组中就可能出现一大块空区。例如，在数组有 513 个元素时，空位数目是 511。所以这里的考虑也是拿时间和空间做交换。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。

但也有一些情况，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，最后将做好的存储块（就像是一个数组）的地址通过返回值送出来。调用函数（main）用一个类型合适的指针接收这个地址值，而后通过该指针使用这块数据。

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 的内部分配的，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 运行中建立起的数据块，在 main 里继续使用这个块完全没问题。当然，采用这种方式，readscores 就不应该在函数退出前释放该块。在上面调用中，除了传递了有关的数据之外，实际上还出现了一种对存储管理责任的转移问题：readscores 也将释放这块存储的责任转交给 main。由此也可以看出前面程序里忽略了一件事情：那里没有释放这一存储块。修改就是在 main 的最后加上释放语句。

现在考虑 readscores 的设计问题。在前面程序里，readscores 通过 int 指针参数（实参应是 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，例如将其原型改为：

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

这样，程序的 main 里可以写如下形式的调用：

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

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，应该如何定义这样的函数呢？

答案与其他情况完全一样。如果我们想通过函数的实参取得函数里送出来的一个 int 值，就需要将一个 int 变量的地址送进函数，要求函数里间接地给这个变量赋值。同理，现在需要得到一个指针值，就应该通过实参将一个指针变量的地址送进去，让函数里通过这个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**)。实际调用 readscores 时应该把一个指针的地址传给它：

```
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，我们同样是将函数里分配的存储块送到函数之外，同时也将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，现在通过参数传递这种存储块的地址。

在这一节里，我们介绍了指针、函数与动态分配之间的关系，并讨论了几种不同的处理技术。如果可能的话，最好是在程序里使用第一种设计，因为它最清晰，也最不容易出现忘记释放的情况。

7.7 定义类型

C 语言为各种基本类型提供了类型名，利用它们可方便地定义变量，定义函数参数与返回值，或用在其他需要类型名的地方。现在有了数组、指针等机制，说明一个东西“类型”的形式变得越来越复杂，总写复杂的东西就容易出错。这种情况还带来了一些新问题，例如，有时需要在多个地方写同样类型描述，保证它们的一致性又是新的负担，尤其是在需要修改描述时。如果我们能根据需要，把复杂的“类型形式描述”看作类型，为它定义名字，就会给写程序带来极大方便，特别是在实现复杂的程序或软件系统时。

写程序者定义的类型称为*用户定义类型*。类型定义机制是一些新语言的核心机制，这样的类型可以与基本类型一样使用，在那里起着极重要的作用。C 语言的类型定义机制比较弱，其作用基本上是简化描述，方便人们编写程序。下面介绍这方面的情况。

C 程序里的类型定义用关键字 typedef 引导，出现在关键字 typedef 后面的描述形式上与变量定义类似，出现在原来变量位置上的标识符成为所定义的类型名，类型名可以根据需要选取。下面是一个简单例子：

```
typedef unsigned long int ULI;
```

这一描述定义了类型名 ULI，它表示的类型就是 unsigned long int。这个新类型名的使用与基本类型名一样，可以用于定义变量、说明函数原型等。下面是几个例子：

```
ULI x, y, *p;
ULI fun1(double x, ULI n);
p = (ULI *)malloc(n * sizeof(ULI));
```

这种类型定义可以简化程序书写，为此目的定义类型有一定实际价值。

有时定义新类型名是为了提高程序的可读性，使程序更清晰。例如下面的定义：

```
typedef double LENGTH;
typedef double AREA;
```

程序中的长度和面积可能都用双精度数表示，但是把表示长度的变量与表示面积的变量相加却不合理。定义这种类型名字可能帮人在阅读程序时看到这种内在的不一致性。

注意，C 语言并不认为这样定义的类型是新类型，而认为它们只是原有类型的新名字（别名）。所以，即使我们定义了 LENGTH 和 AREA，并分别定义两个类型的变量，C 语言编译程序还是认为通过它们定义的变量都是双精度类型的变量。这也就是前面所说“为提高程序的可读性”的意思。

7.7.1 定义数组类型

前面几个例子似乎可以不用 typedef，用预处理命令也能产生类似效果。例如：

```
#define LENGTH double
#define AREA   double
```

在程序里两者的使用形式相同，最终效果也相同。当然，这两种写法的处理过程是不同的，

预处理命令由预处理程序处理，而类型定义则由编译程序处理。

有些类型定义不能通过宏定义描述，例如数组类型。数组类型的定义形式也符合前面的解释，例如，要定义一种具有4个元素的双精度数组类型，可以写下面的定义：

```
typedef double VECT4[4];
```

正如前面所说，出现原来写变量处的标识符是现在定义的类型名。这里的类型名是VECT4，可以将它用在各种定义说明之中。如在程序里写：

```
VECT4 v1, v2;
```

这就定义了两个4元素的双精度数组。我们还可以定义例如5×5的双精度数组类型：

```
typedef double MAT[5][5];
```

在此之后就可以写：

```
MAT a1, a2, a3; /* 定义三个5×5的数组变量 */
double det(MAT m); /* 用于说明函数参数 */
```

程序里还可以写：

```
MAT *p;
p = (MAT *)malloc(sizeof(MAT));
```

同样可以定义指针类型。例如写：

```
typedef int * IP;
typedef MAT * MATP;
```

这就定义了两个指针类型。后面章节里还会讨论其他类型的定义问题。

7.7.2 复杂类型描述与解读

类型描述可能变得很复杂，这也是C语言的一个缺点。我们不赞成写过于复杂的类型描述，建议在情况比较复杂时利用typedef分解有关类型描述，这样能使程序容易理解，也不容易出错。因为定义之后可以多次使用，此后写程序也更方便，可能节省时间，也降低了由多次写复杂类型描述而带来细节错误的可能性，减少修改维护的代价。

但在实际中，并不是所有人都能这样做。在阅读别人的程序时也可能遇到复杂的类型描述，因为C语言允许人们那样写。由于这一客观情况，我们也应了解C语言类型描述的一般规则，知道如何解读有关的描述。了解了有关情况，自己写类型描述时也能更清楚些，知道应该怎样写，在什么地方需要加括号等等。

类型描述中除能使用已有类型名外，可用的构造符号就是我们已经很熟悉的三个（组）运算符：

*	[]	()
指针	数组	函数

它们（总是）分别表示指针、数组和函数。圆括号还可以用于改变结合关系，这也可能使类型描述更难阅读。

首先应记住，类型描述中各种符号的意义，它们优先级和结合方式都与用在表达式里时相同。也就是说，[]和()的结合性较强，星号*结合力较弱；[]和()从左向右结合，而星号*从右向左结合。阅读类型描述时，应该首先抓住其中被说明（定义）的名字，从这个标识符出发进行解读，剩下的问题就迎刃而解了。被说明（定义）的标识符往往出现在描述的中间，被许多其他符号包围着。

下面分析一些例子，结合介绍类型描述中的问题：

1. `int *f(int);`

   这里被说明的名字是f。因为f的上下文中有*和()，而()的优先级高，所以f是一个函数，位于它前面的描述说明函数的返回值类型（int指针）；右边括号里是函数的参数类型。所以，f是具有一个int参数的函数，返回int指针。

2. `int (*dp)[16];`

   被说明（定义）的dp是一个指针（由于括号的缘故），随后的方括号说明这一指针指

向数组。左边是数组元素类型。因此，dp 的指向类型是 16 个元素的整型数组。

3. `int *dp1[16];`
   被说明（定义）的 dp1 是包含 16 个元素的数组（因为[]优先级更高），这一数组的元素是 int 指针。
4. `char **argv;`
   这里说明（定义）的是 argv，它是一个指针。被 argv 指向的又是指针类型：字符指针类型。所以 argv 是一个指向字符指针类型的指针。
5. `int (*fp)(int);`
   这里被说明（定义）名字的是 fp。附加的括号使*首先起作用，所以 fp 是指针。再向外看可以看到 fp 的指向类型。由于其右边有一对括号，所以这个指针的指向对象是函数。fp 是指向函数的指针（简称函数指针），它指向具有一个整型参数、返回整型值的函数。有关函数指针的情况是下一节讨论的题目。
6. `int (*(*x(int))[4])(double);`
   首先应该这里被说明的 x。这是一个函数（因为 x 的右边是圆括号），它有一个整型参数，返回的是指针值。这种指针指向包含 4 个元素的数组，而这一数组的元素又是指向函数的指针，被指函数有一个 double 参数并返回 int 值。

从上面例子中可以看到类型描述的一些情况。虽然实际中很复杂的类型极其少见，但 C 语言确实允许这种类型描述一层层地构造上去。所有类型描述都是使用上述三种构造符，再加上表示结合顺序的括号。也有些构造方式是不合法的。例如，函数不能返回数组；函数不能以函数作为参数，也不能返回函数；函数不能作为数组的元素等等。

前面说过，利用 typedef 可以更清晰地描述复杂类型。我们考虑上面最后的例子，看看可能如何分解它的描述。实际中对复杂类型的分解应根据具体情况和需要，根据其中哪个（哪些）类型具有逻辑意义，在程序里有用。这里给出的分解则完全是主观的。

首先可以定义下面类型：

```
typedef int (*funp)(double);
typedef funp fparray[4];
```

这定义了一个函数指针类型和一个函数指针数组类型。有了这些定义之后，x 的类型描述就很简单了：

```
fparray *x(int);
```

它是一个返回指向 fparray 类型的指针的函数。当然，所定义的类型都可作他用。

7.8 指向函数的指针

假设程序里需要一个求数学函数根的函数。虽然前面给过实现这种功能的函数，但在实际程序里它却未必合用。请回忆当时的定义方式，在求根函数的定义里规定了被求根的数学函数，因为给定了被求根函数的名字与函数原型。确定函数原型是很合理的，这并没有过分限制求根函数的通用性，需要处理的数学函数都具有这种原型。而给出被处理函数的名字，效果就不同了，这将使写出的函数只能对具有这个名字的函数求根。

作为程序试验，可以在程序中给出一个数学函数定义，令它具有所需名字，编译后求出结果。要想处理另一函数就必须修改程序，用程序中规定的名字定义这个新函数。

这一方式不实用。例如，一个程序里可能有多个需要求根的函数，这时除非为每个需要求根的函数定义一个求根函数。这些求根函数的基本代码完全相同，只是其中调用的函数名不同。问题的症结仍然是：前面的求根函数不具通用性，只能对具有某个特殊名字的函数求根。提高函数通用性只有一个方法，那就是为函数引进新参数。由于现在希望所定义的函数能处理不同的数学函数，所以应当为求根函数引进与函数有关的参数。

C 语言里不能把函数作为函数的参数，而需要使用指向函数的指针（函数指针）。在函数调用时，可以通过这种指针把所需函数传进去，从而达到在不同调用中使用不同函数的目的。利用指针传递函数参数是C语言里指针的另一个重要用途*。函数指针在实际程序中应用广泛，在本课程后的进一步学习中，读者可能遇到许多函数指针的应用。

要定义以函数指针为参数的函数，最好是先定义出有关的函数指针类型。下面是指向数学函数的指针类型：

```
typedef double (* MFP)(double);
```

这里假定数学函数都是有一个 double 参数返回 double 值的函数，定义的类型名是 MFP，用括号括起说明它是指针类型。(* MFP)的括号是必需的，前面已经解释过这个问题。

假设要定义另一种函数指针类型，这类指针能指向具有两个整型参数，返回双精度指针值的函数，这个类型的定义应该是：

```
typedef double * (*FUNP)(int, int);
```

7.8.1 函数指针的定义和使用

有了函数指针类型，使用就很方便了。下面定义了两个指向数学函数的指针变量：

```
MFP p1, p2;
```

如果没有先定义指针类型，定义同样两个变量需要写：

```
double (*p1)(double), (*p2)(double);
```

这样写比较烦琐，被定义的东西裹在其他描述中，不容易看清。此外，反复写这种东西也很麻烦，不易保持一致性，也容易出错。读者在许多书籍中可以看到这种写法，我们不提倡这种方式，先定义指针类型的方式要好得多。

下一个问题是如何使函数指针指向函数，以便通过指针去使用函数。语言规定，对单独出现的函数名求值，得到的就是指向该函数的指针值，可以把这种值赋给函数指针。注意，函数指针值也有类型，被赋值的指针应当具有正确类型。下面语句使函数指针 p1 指向标准库函数 sin（使用这个语句的程序应已包含了标准头文件<math.h>）：

```
p1 = sin;
```

在函数指针有了合法值后，就可以把它当函数名使用了（未初始化的函数指针绝不能用）。将这种指针写在通常写函数的位置，就能调用被指函数。例如，现在可以写：

```
x = p1(3.24);
```

这相当于写：

```
x = sin(3.24);
```

这里不需要写间接。写间接访问也是正确的（也可写为“(*p1)(3.24)”）。

我们原本可以通过函数名直接去用函数，现在为什么要引进一个麻烦，先定义函数指针并让它指向函数，然后再通过它去调用那个函数？这样不是多此一举吗？当然不是！看看一般的指针，我们也可以通过变量名访问变量，又能通过指针访问被指变量。两种情况很类似，也可以有类似的解释：如果通过函数名使用函数，那么每次执行时调用的总是同一个函数；而如果通过函数指针使用函数，执行中使用的是哪个函数，就要看指针当时的值了。显然，采用函数指针的方式带来了新的灵活性。

函数指针还有另外一些重要的使用方式，有许多高级的程序设计技术，它们已经超出了本书的范围。函数指针的一个重要用途就是作为函数参数，这一用途不是直接调用函数所能完成的。下面将讨论这方面的问题。

7.8.2 函数指针作为函数的参数

现在看如何利用函数指针解决本节开始提的问题。实际上，所用方法已经很清楚了。我

* 其他语言的处理方式与C语言不同。C采用函数指针是为了简化语言定义，又能提供最大的灵活性。这种方式初学时较难理解，请读者注意。

们用一个函数指针作为求根函数的参数，调用时根据需要为求根函数提供合乎类型要求的函数指针值，这样就能得到一种新的通用性。下面是解决问题的程序例子：

例：利用函数指针重新定义采用弦线法求函数根的函数。函数定义可重写为：

```
double cross(MFP fp, double x1, double x2) {
    double y1 = fp(x1), y2 = fp(x2);
    return (x1 * y2 - x2 * y1) / (y2 - y1);
}

double root(MFP fp, double x1, double x2) {
    double x, y, y1 = fp(x1);

    do {
        x = cross(fp, x1, x2);
        y = fp(x);
        if (y * y1 > 0.0) {
            y1 = y;
            x1 = x;
        }
        else
            x2 = x;
    } while (y >= 1E-6 || y <= -1E-6);

    return x;
}
```

看起来函数的改动并不大：各个函数都增加了函数指针参数，将函数里的直接调用改成通过参数的间接调用。然而，这样改动之后的函数已经完全是“通用的”了。下面是在程序里使用这个函数的两个例子，其中假设 fun 是另一个已定义好的数学函数：

```
x = root(sin, 0.4, 4.5);
y = root(fun, 1.26, 7.03);
```

上面的定义假定已经定义好类型 MFP，因此写起来清晰简单。当然，在定义这种函数时，也可以采用直接描述函数指针参数类型的方式，例如，可以写：

```
double root(double (*fp)(double), ...) ...
```

这种写法显然不如前一写法清晰易读，如果每个函数都这样写就更麻烦了。

过时函数指针形式

读者可能会在一些书籍里看到用下面形式定义的函数指针：

```
double (*p1)(), (*p2)();
```

或者下面形式定义函数指针参数：

```
int f(double (*fp)(), ...) ...
```

这是一种极其危险的过时形式，是不良书写方式。问题还是没给出函数指针所要求的函数参数类型，使编译系统无法检查和处理。如果采用这种形式，通过指针使用函数时就必须特别小心，否则可能出现隐藏很深的语义错误，这种错误极难发现。

下面例子说明这种书写方式的潜在危险性。假设程序里有下面的定义及使用：

```
int f(double (*fp)() ... ) {
    ... ...
    x = fp(3);
    ... ...
}

... ...
    x = f(sin, ...);
    ... ...
    y = f(pow, ...);
... ...
```

首先可以看到，函数 f 体中通过 fp 的函数调用将直接使用所提供的整型参数（不会对它做转换）。对 f 第一个调用将 sin 作为函数指针的对应实参，后果是把调用中的整数参数直接

当双精度数据使用，显然不能得到正确结果。第二个调用也不会发现错误，虽然这里的实际参数函数 pow 要求两个双精度参数。这种程序显然不可能得到有意义的结果。

上面这种貌似简单的写法可能留下极难发现的隐患，采用这种方式是得不偿失。因此应请读者特别注意，在定义函数指针类型或函数指针参数时，一定要给出被指函数的完整参数类型和返回值类型，以便编译系统能正确地检查和处理。如果觉得这种定义写起来不方便，那么就应该先定义一个指针类型，而不是在后面的书写中偷工减料。

7.8.3 数值积分函数

现在用求数值积分的函数作为函数指针参数使用的另一个例子，同时也讨论一下如何实现这种在数值计算中常用的计算过程。一个通用的数值积分函数应该有 3 个参数：一个函数指针参数，实参是相应被积数学函数的地址。还需两个参数表示积分下限和上限，这里用双精度类型。积分函数返回表示积分值的双精度结果，原型应是：

```
double numInt(MFP fp, double a, double b);
```

由微积分的知识可知，如果一个函数的积分收敛，就可以用区域分割法逼近实际积分值，可以采用矩形法或梯形法等。在各分割区域长度趋于 0 时，这些计算方法有共同的极限。为简单起见，下面对积分区间采用等长划分和矩形方法，对每个区间用左端点计算（当然也可以用右端点或其他点），这样写出的积分函数定义如下：

```
enum { DIVN = 30 };

double numInt(MFP fp, double a, double b) {
    double res = 0.0, step = (b - a) / DIVN;

    int i;
    for (i = 0; i < DIVN; ++i)
        res += fp(a + i * step) * step;

    return res;
}
```

这个函数采用固定划分数 DIVN（30 只是一个例子），函数定义很简单。

上面函数定义有许多不完善的地方。首先，各种数学函数的性质千差万别，同一函数在不同区间里的性质也可能差别很大，统一划分方式（例如平均分为 30 份）不可能满足各种实际情况的需要。这样定义的函数可能对一些情况算得很好，但对另一些情况的计算结果就可能与实际积分值偏差很大。进一步说，虽然可以通过增加划分数提高结果精度，但由人来确定区间的划分数也很困难。为此，我们希望能有一种方法，使定义出的函数能自动确定合适的区间划分，去适应被积分的函数和区间的各种情况。

人们提出的一种方法是通过多次计算积分值去解决问题。在每次计算后增加区间数并再做计算。按照数学知识，如果被积函数可积，这种过程得到的结果将逐步逼近实际积分值。当然，为了得到合理近似结果的重复计算不能无限进行下去，它必须在某个时候终止。一种合理方法是保留前次计算的结果，在算出的新值与保留值很接近时结束工作。下面函数定义实现了这种想法，其中用两次积分值之差小于 10^{-6} 作为结束条件：

```
double numInt(MFP fp, double a, double b) {
    long i, divn = 10;
    double step, dif, res0,
           res = (fp(b) + fp(a)) * (b - a) / 2;

    for (dif = 1.0; dif > 1E-6 || dif < -1E-6; divn *= 2) {
        res0 = res;
        step = (b - a) / divn;
        for (res = 0.0, i = 0; i < divn; ++i)
            res += fp(a + i * step) * step;
```

```
            dif = res - res0;
        }

        return res;
    }
```

这一函数所用的初始划分数为 10，每次后续计算时将划分数加倍。函数最开始时用被积函数在两端点值形成的梯形作为积分值的近似，这是一个简单的处理。每次大循环完成积分的一次近似计算。变量 dif 记录连续的两次计算之差，初始时把它设置为 1 就是为了保证它有一个值，而且这个值能使循环继续下去。在这里可以用满足条件的任何值，用 1 是随意的选择。后面的程序代码都很容易理解。

需要进一步给读者提出的问题是：后一个函数能应付各种数学函数的实际积分问题吗？如果不能，在什么情况下可能出问题？会出什么问题？这些问题有什么解决办法？其中有没有根本无法解决的问题？如果深入思考这些问题，也可能学到一些东西。

7.8.4 若干以函数指针为参数的数组操作实用函数

在本章开始介绍指针与数组的关系时，给出过一些利用指针操作数组的实用函数。有了函数指针之后，我们可以给这类函数增加一个操作函数，写出另外一些通用的对数组元素进行操作的实用函数。本节里将举出几个这方面的例子。

程序中经常需要对一个数组或者数组中的一段元素做某种“变换”，也就是说，将这些元素修改为从它们的原值出发计算出的一个新值。利用函数指针，我们可以写出下面的“通用”变换函数，它将某个对 int 类型的操作提升为对数组中元素序列的操作：

```
void trans_int(int *begin, int *end, int (*fp)(int)) {
    for ( ; begin != end; ++begin)
        *begin = fp(*begin);
}
```

当然，也可以对 double 类型写出类似函数：

```
void trans_double(double *begin, double *end, MFP fp) {
    for ( ; begin != end; ++begin)
        *begin = fp(*begin);
}
```

假定我们有一个 double 数组 a，现在希望将这一数组的前 20 个元素修改为各元素的 sin 值，这一工作就可以用下面语句完成：

```
trans_double(a, a+20, sin);
```

考虑另一个例子。假定我们有两个 int 的数组（元素序列），现在希望逐个比较数组中处于同样位置的元素，找出第一对“不匹配”的元素。在这样做时，我们希望将元素是否匹配作为一种参数化的东西，可以根据具体需要选择。为此可以定义出如下通用的“不匹配”查找函数，它返回在第一个数组里不匹配元素的位置（一个指针值），如果没有找到不匹配情况，就返回 end 的值：

```
int * mismatch_int(int *begin, int *end, int *second,
                   int (*fp)(int, int)) {
    for ( ; begin != end; ++begin, ++second)
        if (fp(*begin, *second) == 0) break;
    return begin;
}
```

在这个函数里，实际假定了由 second 表示的第二个序列（数组）至少也和第一个数组一样大，函数中不会出现越界访问（这要求使用函数的人保证）。下面看两个使用实例。

先考虑一个简单情况，假定需要确定第一对不等的元素。这时只要定义一个相等函数：

```
int equal_int(int m, int n) { return m == n; }
```

假定 a 和 b 是被查找的数组，现在需要考虑其中的前 20 个元素。可以采用下面写法：

```
int *p;
p = mismatch_int(a, a+20, b, equal_int);
```

这时如果 p 不等于 a+20，那么就是找到了第一个不匹配元素的位置，否则就是没有这种元素。假设想找出两个数组里第一对奇偶性不同的元素，我们只需要定义下面函数：

```
int parity_check(int m, int n) { return m%2 == n%2; }
```

而后写下面的函数调用：

```
p = mismatch_int(a, a+20, b, parity_check);
```

我们还可以使用同样技术，在第一个数组里找出第一个小于（大于等等）第二个数组中对应元素的元素，还可以利用这种技术解决许多问题。

本章讨论的重要概念

地址值（指针值），指针（指针变量），指针的类型，取地址运算，间接运算，函数的执行环境，指针参数，空指针，NULL，悬空的指针，通用指针，指针运算，数组写法和指针写法，字符指针和字符数组，指针数组，命令行，命令行参数，多维数组作为参数，数组元素位置的计算，动态存储分配，动态存储管理系统，堆，存储释放，类型定义，数组类型，指向函数的指针，数值积分，复杂类型描述和解读。

练习

1. 把前面用数组方式定义的一些函数改写为用指针运算的方式定义。
2. 把前面练习中定义的各种求根函数改写为使用函数指针的函数。
3. 写一个程序，其命令行包括一个字符串参数 s，运行中由标准输入读入一系列正文行。该程序把所有输入行依次输出，并在那些包含字符串 s 的行前面标一个星号。
4. 1）写一个函数，它检查两个字符串是否由同样一些字符组成。2）写一个函数，它判断一个字符串是不是可以由另一个字符串通过重排字符而得到，例如 dare 和 read、dear 都有这种关系。
5. 1）修改第五章的猜数程序，通过命令行参数为它提供数的范围。2）将前面的某些程序改造成为使用命令行参数的，通过命令行为程序提供运行的数据。
6. 矩阵是数值计算中最常用的结构之一，在程序里可以很自然地采用两维数组表示。请按照本章提出的处理两维数组的通用函数的技术，写出几个通用函数实现矩阵的常用操作，包括数乘、矩阵加法、矩阵乘法、矩阵转置、行列式求值等。提示：实现矩阵行列式的求值请采用消去法（高斯消去法），将矩阵变换为三角矩阵后完成计算。
7. 利用函数指针功能，写出利用梯形法求数值积分的函数。
8. 辛普生方法是用二次曲线逼近被积函数的数值积分方法，有辛普生公式：

   $$\int_a^b f(x)dx = h/3\cdot\{[f(a)+f(b)]+4[f(a+h)+f(a+3h)+\cdots+f(a+(2m-1)h)]$$
   $$+2[f(a+2h)+f(a+4h)+\cdots f(a+(2m-2)h)]\}$$

   积分区间 $[a,b]$ 划分为 $2m$ 个等分小区间，$h=(b-a)/2m$。用函数指针实现按这一公式计算数值积分的函数，请考虑采用能根据情况自动调整区间分割数的方法。

9. 魔阵是由 1 到 n^2 这些整数排成$n\times n$方阵，其中每一行、每一列和两个对角线上的数

之和相同。写程序构造出 3 阶和 4 阶的魔阵。

下面是一个构造奇数阶魔阵的通用算法：首先把 1 放在第一行中间。当数 k 放好后，考虑数 k+1 的安放，总把它放在向上一行、向右一个位置。下面是各种特殊情况的处理：

1）要从最上一行向上，那么就转移到最下一行；

2）要从最右一列向右，那么就转移到最左一列；

3）如果企图放数的位置已经有了数，那么就把这个数放在它前面一个数的下面。

写程序实现这个算法。另外写一个函数，它能检查一个 n 阶方阵是否为一个魔阵。

10. 写一个小游戏程序，它内部存储着一些英语单词（在写程序时给定单词集合）。程序运行中每次由这些单词中随机地选出一个，要求游戏者猜。做游戏者反复询问某些字母是否出现在单词里面，程序给出回答。直至人猜出这个单词（或者放弃）。
11. 修改前一题提出的猜单词程序，设计实现某种较好的选词策略和提示策略。
12. 写一个函数 `select(int n, double a[], double b[], double x)`，它将数组 b 中大于等于 x 的数顺序复制到数组 a 中。假定 n 是两个数组的大小。请分别用数组写法和指针写法完成这一工作。
13. 写一个程序，它可以输入并保存至多 100 个长度不超过 80 个字符的字符行。在读完所有的输入之后，它先输出其中长度不超过 40 个字符的行，而后输出其他行。考虑下面两种实现方式：1）用两维字符数组保存字符行；2）用一个字符指针数组，将字符行保存在通过动态分配的存储块里。
14. 完善本章中处理任意数量学生成绩的程序，完成文中提出的各种改造，并加入你所设想的合理的进一步改造。
15. 修改 7.6 节的筛法程序，将分配数组的工作移到实现筛法的函数 `sieve` 里，使它返回做好的数组，并对主函数做适当改造。从程序的清晰性和功能分配的合理性等方面比较这两种实现方式。
16. 请定义：1）一个数组类型，该类型的数组包含 10 个元素，其元素是函数指针，被指函数都是有一个 `double` 参数并返回 `double` 值（请直接定义，并将写出的定义与借助于 `MFP` 的定义做一个比较；2）一个函数的原型说明，它的参数是一个指向“数学函数”的指针和一个 `double`，返回值是指向数组的指针。这种数组中包含着 12 个 `double` 值（请分别采用直接的方式和预先定义其他类型的方式定义它）。
17. 多项式可以数组表示，例如用一个 n+2 个元素的整数数组 `a` 表示一个 n 次的整系数多项式，其中的在 `a[0]` 中保存多项式最高次的次数（由此也可以决定多项式的项数），其余元素顺序表示一个多项式的 0 次项、1 次项、2 次项等等的系数，多项式的变量隐含假定为 x。这种数组可以通过动态存储分配创建。请基于这种表示方式实现一元多项式的输入（自己设计一种输入方式、输出可尽可能采用直观的类似数学中的表示形式（例如输出为 `3+2x+4x^2`，系数为 0 的项不输出）。请进一步实现各种多项式运算，完成一个简单的一元多项式计算系统。
18. 假定所用 C 语言支持的整数类型 `long`（32 位）无法满足目前应用的需要，我们需要一种 64 位的整数。请设法利用现有的功能实现 64 位整数的表示、输入、输出和计算。