第13章

高级指针话题

本章收集了各种各样的涉及指针的技巧。有些技巧非常实用，另外一些技巧则学术味更浓一些，还有一些则纯属找乐。但是，这些技巧都很好地说明了这门语言的各种原则。

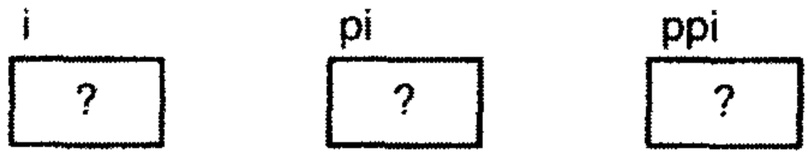
13.1 进一步探讨指向指针的指针

在上一章，我们使用了指向指针的指针，用于简化向单链表插入新值的函数。另外还存在许多领域，指向指针的指针能够发挥重要的作用。

这里有一个通用的例子。

```
int     i;
int     *pi;
int     **ppi;
```

这些声明在内存中创建了下列变量。如果它们是自动变量，我们无法猜测它们的初始值。



有了上面这些信息之后，请问下面各条语句的效果是什么呢？

```
①  printf( "%d\n", ppi );
②  printf( "%d\n", &ppi );
③  *ppi = 5;
```

① 如果 ppi 是个自动变量，它就未被初始化，这条语句将打印一个随机值。如果它是个静态变量，这条语句将打印 0。

② 这条语句将把存储 ppi 的地址作为十进制整数打印出来。这个值并不是很有用。

③ 这条语句的结果是不可预测的。对 ppi 不应该执行间接访问操作，因为它尚未被初始化。

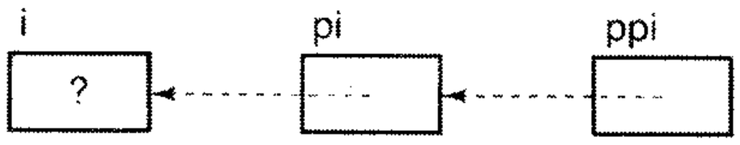
接下来的两条语句用处比较大。

```
ppi = &pi;
```

这条语句把 ppi 初始化为指向变量 pi。以后我们就可以安全地对 ppi 执行间接访问操作了。

```
*ppi = &i;
```

这条语句把 pi（通过 ppi 间接访问）初始化为指向变量 i。经过上面最后两条语句之后，这些变量变成了下面这个样子：



现在，下面各条语句具有相同的效果：

```
i='a';
*pi='a';
**ppi='a';
```

在一条简单的对 i 赋值的语句就可以完成任务的情况下，为什么还要使用更为复杂的涉及间接访问的方法呢？这是因为简单赋值并不总是可行，例如链表的插入。在那些函数中，我们无法使用简单赋值，因为变量名在函数的作用域内部是未知的。函数所拥有的只是一个指向需要修改的内存位置的指针，所以要对该指针进行间接访问操作以访问需要修改的变量。

在前一个例子中，变量 i 是一个整数，pi 是一个指向整型的指针。但 ppi 是一个指向 pi 的指针，所以它是一个指向整型的指针的指针。假定我们需要另一个变量，它需要指向 ppi。那么，它的类型当然是“指向整型的指针的指针的指针”，而且它应该像下面这样声明：

```
int  ***pppi;
```

间接访问的层次越多，你需要用到它的次数就越少。但是，一旦你真正理解了间接访问，无论出现多少层间接访问，你应该都能十分轻松地应付。

提示：

只有当确实需要时，你才应该使用多层间接访问。不然的话，你的程序将会变得更庞大、更缓慢并且更难于维护。

13.2 高级声明

在使用更高级的指针类型之前，我们必须观察它们是如何声明的。前面的章节介绍了表达式声明的思路以及 C 语言的变量如何通过推论进行声明。我们在第 8 章声明指向数组的指针时已经看到过一些推论声明的例子。让我们通过观察一系列越来越复杂的声明进一步探索这个话题。

首先让我们来看几个简单的例子。

```
int        f;   /* 一个整型变量 */
int        *f;  /* 一个指向整型的指针 */
```

不过，请回忆一下第 2 个声明是如何工作的：它把**表达式*f** 声明为一个整数。根据这个事实，你肯定能推断出 f 是个指向整型的指针。C 声明的这种解释方法可以通过下面的声明得到验证。

```
int* f, g;
```

它并没有声明两个指针。尽管它们之间存在空白，但星号是作用于 f 的，只有 f 才是一个指针。g 只是一个普通的整型变量。

下面是另外一个例子，你以前曾见过：

```
int        f();
```

它把 f 声明为一个函数，它的返回值是一个整数。旧式风格的声明对函数的参数并未提供任何信息。它只声明 f 的返回值类型。现在我将使用这种旧式风格，这样例子看上去简单一些，后面我再回到完整的原型形式。

下面是一个新例子：

```
int        *f();
```

要想推断出它的含义，你必须确定表达式*f( )是如何进行求值的。首先执行的是函数调用操作符（），因为它的优先级高于间接访问操作符。因此，f 是一个函数，它的返回值类型是一个指向整型的指针。

如果“推论声明”看上去令你觉得有点讨厌，你只要这样考虑就可以了：用于声明变量的表达式和普通的表达式在求值时所使用的规则相同。你不需要为这类声明学习一套单独的语法。如果你能够对一个复杂表达式求值，你同样可以推断出一个复杂声明的含义，因为它们的原理是相同的。

接下来的一个声明更为有趣：

```
int   (*f)();
```

确定括号的含义是分析这个声明的一个重要步骤。这个声明有两对括号，每对的含义各不相同。第 2 对括号是函数调用操作符，但第 1 对括号只起到聚组的作用。它迫使间接访问在函数调用之前进行，使 f 成为一个函数指针，它所指向的函数返回一个整型值。

函数指针？是的，程序中的每个函数都位于内存中的某个位置，所以存在指向那个位置的指针是完全可能的。函数指针的初始化和使用将在本章后面详述。

现在，下面这个声明应该是比较容易弄懂了：

```
int        *(*f)();
```

它和前一个声明基本相同，f 也是一个函数指针，只是所指向的函数的返回值是一个整型指针，必须对其进行间接访问操作才能得到一个整型值。

现在，让我们把数组也考虑进去。

```
int        f[];
```

这个声明表示 f 是个整型数组。数组的长度暂时省略，因为我们现在关心的是它的类型，而不是它的长度[1]。

下面这个声明又如何呢？

```
int        *f[];
```

这个声明又出现了两个操作符。下标的优先级更高，所以 f 是一个数组，它的元素类型是指向整型的指针。

下面这个例子隐藏着一个圈套。不管怎样，让我们先推断出它的含义。

```
int        f()[];
```

f 是一个函数，它的返回值是一个整型数组。这里的圈套在于这个声明是非法的——函数只能返回标量值，不能返回数组。

这里还有一个例子，颇费思量。

```
int        f[]();
```

现在，f 似乎是一个数组，它的元素类型是返回值为整型的函数。这个声明也是非法的，因为

---

[1] 如果它们的链接属性是 external 或者是作用函数的参数，即使它们在声明时未注明长度，也仍然是合法的。

数组元素必须具有相同的长度，但不同的函数显然可能具有不同的长度。

但是，下面这个声明是合法的：

```
int        (*f[])();
```

首先，你必须找到所有的操作符，然后按照正确的次序执行它们。同样，这里有两对括号，它们分别具有不同的含义。括号内的表达式*f[]首先进行求值，所以 f 是一个元素为某种类型的指针的数组。表达式末尾的( )是函数调用操作符，所以 f 肯定是一个数组，数组元素的类型是函数指针，它所指向的函数的返回值是一个整型值。

如果你搞清楚了上面最后一个声明，下面这个应该是比较容易的了：

```
int        *(*f[])();
```

它和上面那个声明的唯一区别就是多了一个间接访问操作符，所以这个声明创建了一个指针数组，指针所指向的类型是返回值为整型指针的函数。

到现在为止，我使用的是旧式风格的声明，目的是为了让例子简单一些。但 ANSI C 要求我们使用完整的函数原型，使声明更为明确。例如：

```
int        (*f)( int, float );
int        *(*g[])( int, float );
```

前者把 f 声明为一个函数指针，它所指的函数接受两个参数，分别是一个整型值和浮点型值，并返回一个整型值。后者把 g 声明为一个数组，数组的元素类型是一个函数指针，它所指向的函数接受两个参数，分别是一个整型值和浮点型值，并返回一个整型指针。尽管原型增加了声明的复杂度，但我们还是应该大力提倡这种风格，因为它向编译器提供了一些额外的信息。

提示：

如果你使用的是 UNIX 系统，并能访问 Internet，你可以获得一个名叫 cdecl 的程序，它可以在 C 语言的声明和英语之间进行转换。它可以解释一个现存的 C 语言声明：

```
cdecl> explain int (*(*f)())[10];
declare f as pointer to function returning pointer to
    array 10 of int
```

或者给你一个声明的语法：

```
cdecl> declare x as pointer to array 10 of pointer to
    function returning int
int (*(*x)[10])()
```

cdecl 的源代码可以从 comp.sources.unix.newsgroup 存档文件第 14 卷中获得。

13.3 函数指针

你不会每天都使用函数指针。但是，它们确有用武之地，最常见的两个用途是转换表(jump table)和作为参数传递给另一个函数。本节，我们将探索这两方面的一些技巧。但是，首先容我指出一个常见的错误，这是非常重要的。

警告：

简单声明一个函数指针并不意味着它马上就可以使用。和其他指针一样，对函数指针执行间接访问之前必须把它初始化为指向某个函数。下面的代码段说明了一种初始化函数指针的方法。

```
int        f( int );
```

```
int       (*pf)( int ) = &f;
```

第 2 个声明创建了函数指针 pf，并把它初始化为指向函数 f。函数指针的初始化也可以通过一条赋值语句来完成。在函数指针的初始化之前具有 f 的原型是很重要的，否则编译器就无法检查 f 的类型是否与 pf 所指向的类型一致。

初始化表达式中的&操作符是可选的，因为**函数名**被使用时总是由编译器把它转换为**函数指针**。&操作符只是显式地说明了编译器将隐式执行的任务。

在函数指针被声明并且初始化之后，我们就可以使用三种方式调用函数：

```
int     ans;

ans = f( 25 );
ans = (*pf)( 25 );
ans = pf( 25 );
```

第 1 条语句简单地使用名字调用函数 f，但它的执行过程可能和你想象的不太一样。函数名 f 首先被转换为一个函数指针，该指针指定函数在内存中的位置。然后，函数调用操作符调用该函数，执行开始于这个地址的代码。

第 2 条语句对 pf 执行间接访问操作，它把函数指针转换为一个函数名。这个转换并不是真正需要的，因为编译器在执行函数调用操作符之前又会把它转换回去。不过，这条语句的效果和第 1 条语句是完全一样的。

第 3 条语句和前两条语句的效果是一样的。间接访问操作并非必需，因为编译器需要的是一个函数指针。这个例子显示了函数指针通常是如何使用的。

什么时候我们应该使用函数指针呢？前面提到过，两个最常见的用途是把函数指针作为参数传递给函数以及用于转换表。让我们各看一个例子。

13.3.1　回调函数

这里有一个简单的函数，它用于在一个单链表中查找一个值。它的参数是一个指向链表第 1 个节点的指针以及那个需要查找的值。

```
Node *
search_list( Node *node, int const value )
{
        while( node != NULL ){
                if( node->value == value )
                        break;
                node = node->link;
        }
        return node;
}
```

这个函数看上去相当简单，但它只适用于值为整数的链表。如果你需要在一个字符串链表中查找，你不得不另外编写一个函数。这个函数和上面那个函数的绝大部分代码相同，只是第 2 个参数的类型以及节点值的比较方法不同。

一种更为通用的方法是使查找函数与类型无关，这样它就能用于任何类型的值的链表。我们必须对函数的两个方面进行修改，使它与类型无关。首先，我们必须改变比较的执行方式，这样函数就可以对任何类型的值进行比较。这个目标听上去好像不可能，如果你编写语句用于比较整型值，它怎么还可能用于其他类型如字符串的比较呢？解决方案就是使用函数指针。调用者编写一个函数，用于比较两个值，然后把一个指向这个函数的指针作为参数传递给查找函数。然后查找函数调用这

个函数来执行值的比较。使用这种方法，任何类型的值都可以进行比较。

我们必须修改的第 2 个方面是向函数传递一个指向值的**指针**而不是值本身。函数有一个 void * 形参，用于接收这个参数。然后指向这个值的指针便传递给比较函数。这个修改使字符串和数组对象也可以被使用。字符串和数组无法作为参数传递给函数，但指向它们的指针却可以。

使用这种技巧的函数被称为**回调函数**(callback function)，因为用户把一个函数指针作为参数传递给其他函数，后者将“回调”用户的函数。任何时候，如果你所编写的函数必须能够在不同的时刻执行不同类型的工作或者执行只能由函数调用者定义的工作，你都可以使用这个技巧。许多窗口系统使用回调函数连接多个动作，如拖拽鼠标和点击按钮来指定用户程序中的某个特定函数。

我们无法在这个上下文环境中为回调函数编写一个准确的原型，因为我们并不知道进行比较的值的类型。事实上，我们需要查找函数能作用于任何类型的值。解决这个难题的方法是把参数类型声明为 void *，表示“一个指向未知类型的指针”。

提示：

在使用比较函数中的指针之前，它们必须被强制转换为正确的类型。因为强制类型转换能够躲过一般的类型检查，所以你在使用时必须格外小心，确保函数的参数类型是正确的。

在这个例子里，回调函数比较两个值。查找函数向比较函数传递两个指向需要进行比较的值的指针，并检查比较函数的返回值。例如，零表示相等的值，非零值表示不相等的值。现在，查找函数就与类型无关，因为它本身并不执行实际的比较。确实，调用者必须编写必需的比较函数，但这样做是很容易的，因为调用者知道链表中所包含的值的类型。如果使用几个分别包含不同类型值的链表，为每种类型编写一个比较函数就允许单个查找函数作用于所有类型的链表。

程序 13.1 是类型无关查找函数的一种实现方法。注意函数的第 3 个参数是一个函数指针。这个参数用一个完整的原型进行声明。同时注意虽然函数绝不会修改参数 node 所指向的任何节点，但 node 并未被声明为 const。如果 node 被声明为 const，函数将不得不返回一个 const 结果，这将限制调用程序，它便无法修改查找函数所找到的节点。

```
/*
** 在一个单链表中查找一个指定值的函数。它的参数是一个指向链表第 1 个节点的
** 指针，一个指向我们需要查找的值的指针和一个函数指针，它所指向的函数用于比
** 较存储于链表中的类型的值。
*/
#include <stdio.h>
#include "node.h"

Node *
search_list( Node *node, void const *value,
    int (*compare)( void const *, void const * ) )
{
        while( node != NULL ){
             if( compare( &node->value, value ) == 0 )
                 break;
        node = node->link;
        }
        return node;
}
```

程序 13.1　类型无关的链表查找　　search.c

指向值参数的指针和&node->value 被传递给比较函数。后者是我们当前所检查的节点的值。在选择比较函数的返回值时，我选择了与直觉相反的约定，就是相等返回零值，不相等返回非零值。它的目的是为了与标准库的一些函数所使用的比较函数规范兼容。在这个规范中，不相等操作数的报告方式更为明确——负值表示第 1 个参数小于第 2 个参数，正值表示第 1 个参数大于第 2 个参数。

在一个特定的链表中进行查找时，用户需要编写一个适当的比较函数，并把指向该函数的指针和指向需要查找的值的指针传递给查找函数。例如，下面是一个比较函数，它用于在一个整数链表中进行查找。

```
int
compare_ints( void const *a, void const *b )
{
        if( *(int *)a == *(int *)b )
                return 0;
        else
                return 1;
}
```

这个函数将像下面这样使用：

```
desired_node = search_list( root, &desired_value,
    compare_ints );
```

注意强制类型转换：比较函数的参数必须声明为 void *以匹配查找函数的原型，然后它们再强制转换为 int *类型，用于比较整型值。

如果你希望在一个字符串链表中进行查找，下面的代码可以完成这项任务：

```
#include <string.h>
...
desired_node = search_list( root, "desired_value",
    strcmp );
```

碰巧，库函数 strcmp 所执行的比较和我们需要的完全一样，不过有些编译器会发出警告信息，因为它的参数被声明为 char *而不是 void *。

13.3.2　转移表

转移表最好用个例子来解释。下面的代码段取自一个程序，它用于实现一个袖珍式计算器。程序的其他部分已经读入两个数（op1 和 op2）和一个操作符（oper）。下面的代码对操作符进行测试，然后决定调用哪个函数。

```
switch( oper ){
case ADD:
        result = add( op1, op2 );
        break;

case SUB:
        result = sub( op1, op2 );
        break;

case MUL:
        result = mul( op1, op2 );
        break;

case DIV:
        result = div( op1, op2 );
        break;
...
```

对于一个新奇的具有上百个操作符的计算器，这条 switch 语句将会非常之长。

为什么要调用函数来执行这些操作呢？把具体操作和选择操作的代码分开是一种良好的设计方案。更为复杂的操作将肯定以独立的函数来实现，因为它们的长度可能很长。但即使是简单的操作也可能具有副作用，例如保存一个常量值用于以后的操作。

为了使用 switch 语句，表示操作符的代码必须是整数。如果它们是从零开始连续的整数，我们可以使用转换表来实现相同的任务。转换表就是一个函数指针数组。

创建一个转换表需要两个步骤。首先，声明并初始化一个函数指针数组。唯一需要留心之处就是确保这些函数的原型出现在这个数组的声明之前。

```
double  add( double, double );
double  sub( double, double );
double  mul( double, double );
double  div( double, double );
...
double  (*oper_func[])( double, double ) = {
        add, sub, mul, div, ...
};
```

初始化列表中各个函数名的正确顺序取决于程序中用于表示每个操作符的整型代码。这个例子假定 ADD 是 0，SUB 是 1，MUL 是 2，接下去以此类推。

第 2 个步骤是用下面这条语句替换前面整条 switch 语句！

```
result = oper_func[ oper ]( op1, op2 );
```

oper 从数组中选择正确的函数指针，而函数调用操作符将执行这个函数。

警告：

在转换表中，越界下标引用就像在其他任何数组中一样是不合法的。但一旦出现这种情况，把它诊断出来要困难得多。当这种错误发生时，程序有可能在三个地方终止。首先，如果下标值远远越过了数组的边界，它所标识的位置可能在分配给该程序的内存之外。有些操作系统能检测到这个错误并终止程序，但有些操作系统并不这样做。如果程序被终止，这个错误将在靠近转换表语句的地方被报告，问题相对而言较易诊断。

如果程序并未终止，非法下标所标识的值被提取，处理器跳到该位置。这个不可预测的值可能代表程序中一个有效的地址，但也可能不是这样。如果它不代表一个有效地址，程序此时也会终止，但错误所报告的地址从本质上说是一个随机数。此时，问题的调试就极为困难。

如果程序此时还未失败，机器将开始执行根据非法下标所获得的虚假地址的指令，此时要调试出问题根源就更为困难了。如果这个随机地址位于一块存储数据的内存中，程序通常会很快终止，这通常是由于非法指令或非法的操作数地址所致（尽管数据值有时也能代表有效的指令，但并不总是这样）。要想知道机器为什么会到达那个地方，唯一的线索是转移表调用函数时存储于堆栈中的返回地址。如果任何随机指令在执行时修改了堆栈或堆栈指针，那么连这个线索也消失了。

更糟的是，如果这个随机地址恰好位于一个函数的内部，那么该函数就会快乐地执行，修改谁也不知道的数据，直到它运行结束。但是，函数的返回地址并不是该函数所期望的保存于堆栈上的地址，而是另一个随机值。这个值就成为下一个指令的执行地址，计算机将在各个随机地址间跳转，执行位于那里的指令。

问题在于指令破坏了机器如何到达错误最后发生地点的线索。没有了这方面的信息，要查明问题的根源简直难如登天。如果你怀疑转移表有问题，可以在那个函数调用之前和之后各打印一条信息。如果被调用函数不再返回，用这种方法就可以看得很清楚。但困难在于人们很难认识到程序某个部分的失败可以是位于程序中相隔甚远的且不相关部分的一个转移表错误所引起的。

提示：

一开始，保证转移表所使用的下标位于合法的范围是很容易做到的。在这个计算器例子里，用于读取操作符并把它转换为对应整数的函数应该核实该操作符是有效的。

13.4　命令行参数

处理命令行参数是指向指针的指针的另一个用武之地。有些操作系统，包括 UNIX 和 MS-DOS，让用户在命令行中编写参数来启动一个程序的执行。这些参数被传递给程序，程序按照它认为合适的任何方式对它们进行处理。

13.4.1　传递命令行参数

这些参数如何传递给程序呢？C 程序的 main 函数具有两个形参[1]。第 1 个通常称为 argc，它表示命令行参数的数目。第 2 个通常称为 argv，它指向一组参数值。由于参数的数目并没有内在的限制，所以 argv 指向这组参数值（从本质上说是一个数组）的第 1 个元素。这些元素的每个都是指向一个参数文本的指针。如果程序需要访问命令行参数，main 函数在声明时就要加上这些参数：

```
int
main( int argc, char **argv )
```

注意这两个参数通常取名为 argc 和 argv，但它们并无神奇之处。如果你喜欢，也可以把它们称为“fred”和“ginger”，只不过程序的可读性会差一点。

图 13.1 显示了下面这条命令行是如何进行传递的：

```
$ cc -c -o main.c insert.c -o test
```

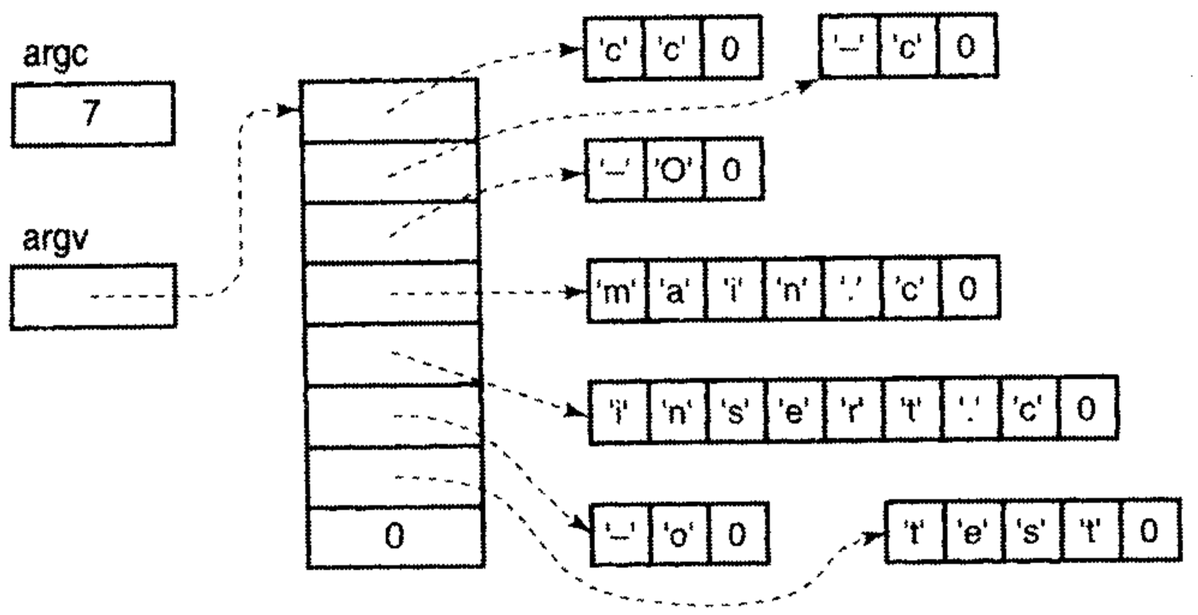


图 13.1　命令行参数

注意指针数组：这个数组的每个元素都是一个字符指针，数组的末尾是一个 NULL 指针。argc 的值和这个 NULL 值都用于确定实际传递了多少个参数。argv 指向数组的第 1 个元素，这就是它为什么被声明为一个指向字符的指针的指针的原因。

---

[1] 实际上，有些操作系统向 main 函数传递第 3 个参数，它是一个指向环境变量列表以及它们的值的指针。请参考你的编译器或操作系统文档，了解更多细节。

最后一个需要注意的地方是第 1 个参数就是程序的名称。把程序名作为参数传递有什么用意呢？程序显然知道自己的名字，通常这个参数是被忽略的。不过，如果程序通常采用几组不同的选项进行启动，此时这个参数就有用武之地了。UNIX 中用于列出一个目录的所有文件的 ls 命令就是一个这样的程序。在许多 UNIX 系统中，这个命令具有几个不同的名字。当它以名字 ls 启动时，它将产生一个文件的简单列表；当它以名字 l 启动，它就产生一个多列的简单列表；如果它以名字 ll 启动，它就产生一个文件的详细列表。程序对第 1 个参数进行检查，确定它是由哪个名字启动的，从而根据这个名字选择启动选项。

在有些系统中，参数字符串是挨个存储的。这样当你把指向第 1 个参数的指针向后移动，越过第 1 个参数的尾部时，就到达了第 2 个参数的起始位置。但是，这种排列方式是由编译器定义的，所以你不能依赖它。为了寻找一个参数的起始位置，你应该使用数组中合适的指针。

程序是如何访问这些参数的呢？程序 13.2 是一个非常简单的例子——它简单地打印出它的所有参数（除了程序名），非常像 UNIX 的 echo 命令。

```
/*
** 一个打印其命令行参数的程序
*/
#include <stdio.h>
#include <stdlib.h>

int
main( int argc, char **argv )
{
        /*
        ** 打印参数，直到遇到 NULL 指针（未使用 argc）。程序名被跳过。
        */
        while( *++argv != NULL )
                printf( "%s\n", *argv );
        return EXIT_SUCCESS;
}
```

程序 13.2　打印命令行参数　　echo.c

while 循环增加 argc 的值，然后检查*argv，看看是否到达了参数列表的尾部，方法是把每个参数都与表示列表末尾的 NULL 指针进行比较。如果还存在另外的参数，循环体就执行，打印出这个参数。在循环一开始就增加 argc 的值，程序名就被自动跳过了。

printf 函数的格式字符串中的%s 格式码要求参数是一个指向字符的指针。printf 假定该字符是一个以 NUL 字节结尾的字符串的第 1 个字符。对 argv 参数使用间接访问操作产生它所指向的值，也就是一个指向字符的指针——这正是格式所要求的。

13.4.2　处理命令行参数

让我们编写一个程序，用一种更加现实的方式处理命令行参数。这个程序将处理一种非常常见的形式——文件名参数前面的选项参数。在程序名的后面，可能有零个或多个选项，后面跟随零个或多个文件名，像下面这样：

```
prog -a -b -c name1 name2 name3
```

每个选项都以一条横杠开头，后面是一个字母，用于在几个可能的选项中标明程序所需的一个。

每个文件名以某种方式进行处理。如果命令行中没有文件名，就对标准输入进行处理。

为了让这些例子更为通用，我们的程序设置了一些变量，记录程序所找到的选项。一个现实程序的其他部分可能会测试这些变量，用于确定命令所请求的处理方式。在一个现实的程序中，如果程序发现它的命令行参数有一个选项，其对应的处理过程就可能也会执行。

下面的程序 13.3 和程序 13.2 颇为相似，因为它包含了一个循环，检查所有的参数。它们的主要区别在于我们现在必须区分选项参数和文件名参数。当循环到达并非以横杠开关的参数时就结束。第 2 个循环用于处理文件名。

```
/*
** 处理命令行参数
*/
#include <stdio.h>
#define   TRUE 1

/*
** 执行实际任务的函数的原型。
*/
void process_standard_input( void );
void process_file( char *file_name );

/*
** 选项标志，缺省初始化为 FALSE。
*/
int   option_a, option_b  /* etc. */ ;

void
main( int argc, char **argv )
{
          /*
          ** 处理选项参数: 跳到下一个参数，并检查它是否以一个横杠开头。
          */
          while( *++argv != NULL && **argv == '-' ){
          /*
          ** 检查横杠后面的字母。
          */
                switch( *++*argv ){
                case 'a':
                option_a = TRUE;
                break;

          case 'b':
                option_b = TRUE;
                break;

          /* etc. */
          }
}

/*
** 处理文件名参数
*/
if( *argv == NULL )
    process_standard_input();
```

```
            else {
                do {
                    process_file( *argv );
                } while( *++argv != NULL );
            }
}
```

程序 13.3　处理命令行参数　　cmd_line.c

注意，在程序 13.3 的 while 循环中，增加了下面这个测试：

```
**argv == '-'
```

双重间接访问操作访问参数的第 1 个字符，如图 13.2 所示。如果这个字符不是一个横杠，那就表示不再有其他的选项，循环终止。注意在测试**argv 之前先测试*argv 是非常重要的。如果*argv 为 NULL，那么**argv 中的第 2 个间接访问就是非法的。

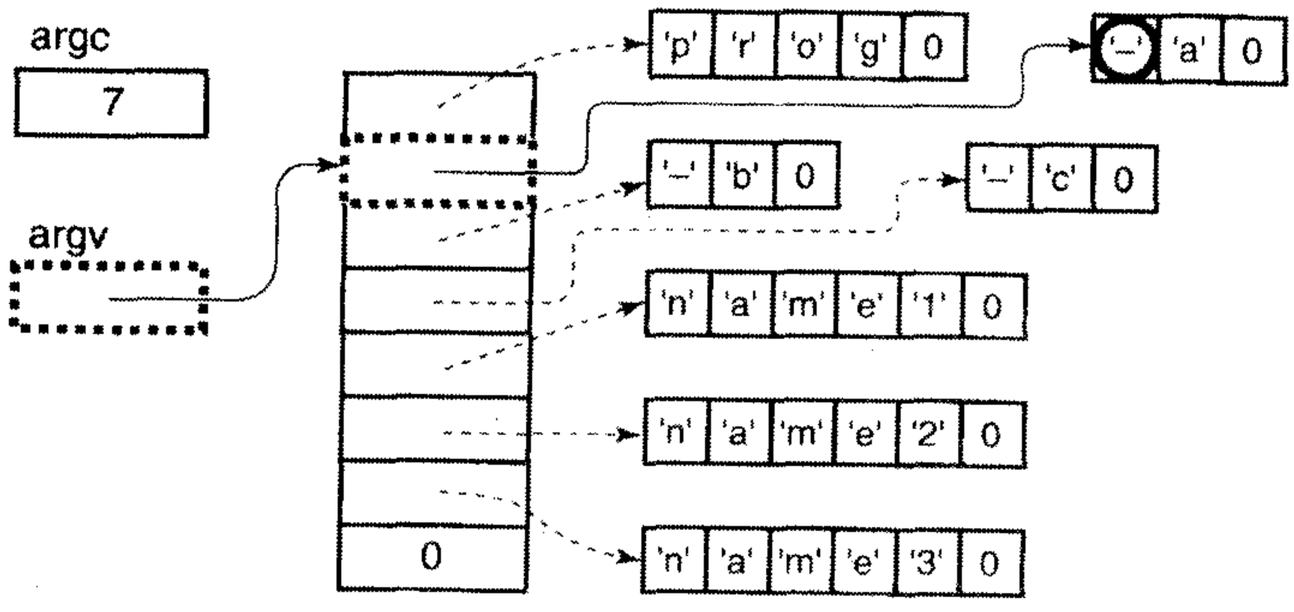


图 13.2　访问参数

switch 语句中的*++*argv 表达式你以前曾见到过。第 1 个间接访问操作访问 argv 所指的位置，然后这个位置执行自增操作。最后一个间接访问操作根据自增后的指针进行访问，如图 13.3 所示。switch 语句根据找到的选项字母设置一个变量，while 循环中的++操作符使 argv 指向下一个参数，用于循环的下一次迭代。

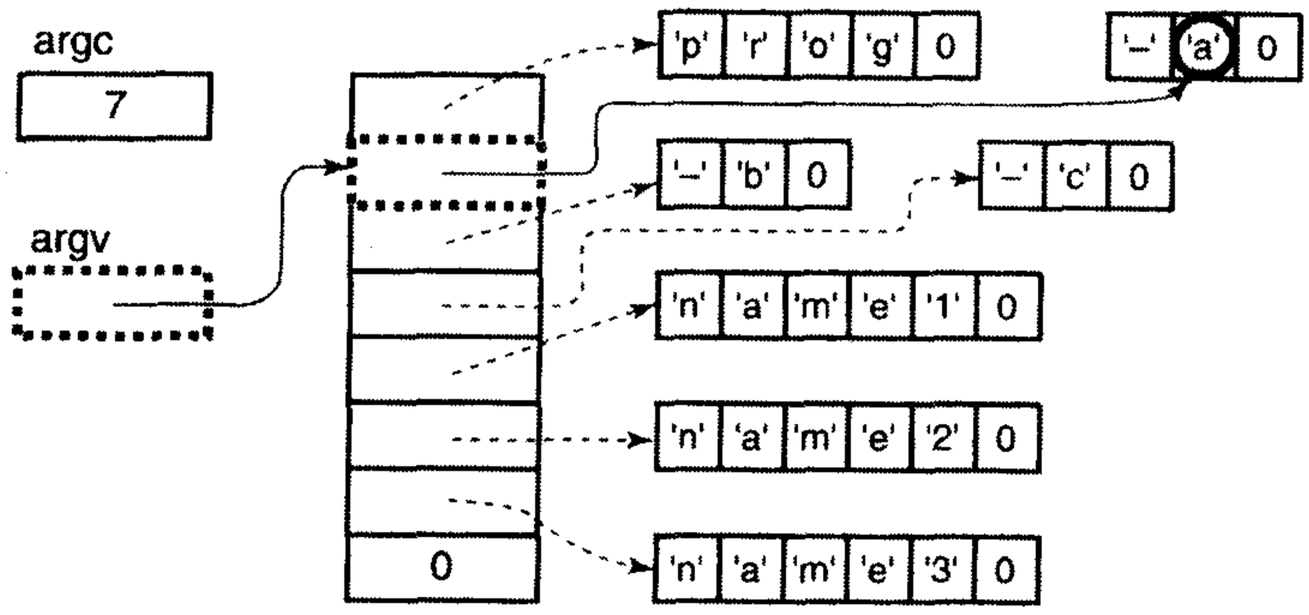


图 13.3　访问参数中的下一个字符

当不再存在其他选项时，程序就处理文件名。如果 argv 指向 NULL 指针，命令行参数里就没有别的东西了，程序就处理标准输入。否则，程序就逐个处理文件名。这个程序的函数调用较为通用，

它们并未显示一个现实程序可能执行的任何实际工作。然而，这个设计方式是非常好的。Main 程序处理参数，这样执行处理过程的函数就无需担心怎样对选项进行解析或者怎样挨个访问文件名。

有些程序允许用户在一个参数中放入多个选项字母，像下面这样：

```
prog -abc name1 name2 name3
```

一开始你可能会觉得这个改动会使我们的程序变得复杂，但实际上它很容易进行处理。每个参数都可能包含多个选项，所以我们使用另一个循环来处理它们。这个循环在遇到参数末尾的 NUL 字节时应该结束。

程序 13.3 中的 switch 语句由下面的代码段代替。

```
while( ( opt = *++*argv ) != '\0' ){
        switch( opt ){
        case 'a':
                option_a = TRUE;
                break;
        /* etc. */
        }
}
```

循环中的测试使参数指针移动到横杠后的那个位置，并复制一份位于那里的字符。如果这个字符并非 NUL 字节，那么就像前面一样使用 switch 语句来设置合适的变量。注意选项字符被保存到局部变量 opt 中，这可以避免在 switch 语句中对**argv 进行求值。

提示：

注意，使用这种方式，命令行参数可能只能处理一次，因为指向参数的指针在内层的循环中被破坏。如果必须多次处理参数，当你挨个访问列表时，对每个需要增值的指针都作一份拷贝。

在处理选项时还存在其他的可能性。例如，选项可能是一个单词而不是单个字母，或者可能有一些值与某些选项联系在一起，如下面的例子所示：

```
cc -o prog prog.c
```

本章的其中一个问题就是对这个思路的扩展。

13.5　字符串常量

现在是时候对以前曾提过的一个话题进行更深入的讨论了，这个话题就是字符串常量。当一个字符串常量出现于表达式中时，它的值是个指针常量。编译器把这些指定字符的一份拷贝存储在内存的某个位置，并存储一个指向第 1 个字符的指针。但是，当数组名用于表达式中时，它们的值也是指针常量。我们可以对它们进行下标引用、间接访问以及指针运算。这些操作对于字符串常量是不是也有意义呢？让我们来看一些例子。

下面这个表达式是什么意思呢？

```
"xyz" + 1
```

对于绝大多数程序员而言，它看上去像堆垃圾。它好像是试图在一个字符串上面执行某种类型的加法运算。但是，当你记得字符串常量实际上是个指针时，它的意义就变得清楚了。这个表达式计算“指针值加上 1”的值。它的结果是个指针，指向字符串中的第 2 个字符：y。

那么这个表达式又是什么呢？

```
*"xyz"
```

对一个指针执行间接访问操作时，其结果就是指针所指向的内容。字符串常量的类型是“指向字符的指针”，所以这个间接访问的结果就是它所指向的字符：x。注意表达式的结果并不是整个字符串，而只是它的第 1 个字符。

下一个例子看上去也是有点奇怪，不过现在你应该能够推断出这个表达式的值就是字符 z。

```
"xyz" [2]
```

最后这个例子包含了一个错误。偏移量 4 超出了这个字符串的范围，所以这个表达式的结果是一个不可预测的字符。

```
*( "xyz" + 4 )
```

什么时候人们可能想使用类似上面这些形式的表达式呢？程序 13.4 的函数是一个有用的例子。你能够推断出这个神秘的函数执行了什么任务吗？提示：用几个不同的输入值追踪函数的执行过程，并观察它的打印结果。答案将在本章结束时给出。

同时，让我们来看一个另外的例子。程序 13.5 包含了一个函数，它把二进制值转换为字符并把它们打印出来。你第 1 次看到这个函数是在程序 7.6 中。我们将修改这个例子，以十六进制的形式打印结果值。第 1 个修改很容易：只要把结果除以 16 而不是 10 就可以了。但是，现在余数可能是 0～15 的任何值，而 10～15 的值应该以字母 A～F 来表示。下面的代码是解决这个问题的一种典型方法。

```
remainder = value % 16;
if( remainder < 10 )
        putchar( remainder + '0' );
else
        putchar( remainder - 10 + 'A' );
```

我使用了一个局部变量来保存余数，而不是三次分别计算它。对于 0～9 的余数，就和以前一样打印一个十进制数字。但对于其他余数，就把它们以字母的形式打印出来。代码中的测试是必要的，因为在任何常见的字符集中，字母 A～F 并不是立即位于数字的后面。

```
/*
**  神秘函数
**
**    参数是一个 0～100 的值
*/
#include <stdio.h>

void
mystery( int n )
{
     n += 5;
     n /= 10;
     printf( "%s\n", "**********" + 10 - n );
}
```

程序 13.4　神秘函数　　mystery.c

```
/*
** 接受一个整型值（无符号），把它转换为字符，并打印出来。前导零被去除。
*/
#include <stdio.h>
```

```
void
binary_to_ascii( unsigned int value )
{
     unsigned int   quotient;

     quotient = value / 10;
     if( quotient != 0 )
          binary_to_ascii( quotient );
     putchar( value % 10 + '0' );
}
```

程序 13.5　把二进制值转换为字符　　btoa.c

下面的代码用一种不同的方法解决这个问题。

```
putchar( "0123456789ABCDEF" [value % 16 ] );
```

同样，余数将是一个 0～15 的值。但这次它使用下标从字符串常量中选择一个字符进行打印。前面的代码是比较复杂的，因为字母和数字在字符集中并不是相邻的。这个方法定义了一个字符串，使字母和数字相邻，从而避免了这种复杂性。余数将从字符串中选择一个正确的数字。

第 2 种方法比传统的方法要快，因为它所需要的操作更小。但是，它的代码并不一定比原来的方法更小。虽然指令减少了，但它付出的代价是多了一个 17 个字节的字符串常量。

提示：

但是，如果程序的可读性大幅度下降，对于因此获得的执行速度的略微提高是得不偿失的。当你使用一种不寻常的技巧或语句时，确保增加一条注释，描述它的工作原理。一旦解释清楚了这个例子，它实际上比传统的代码更容易理解，因为它更短一些。

现在让我们回到神秘函数。你是不是已经猜出它的意思？它根据参数值的一定比例打印相应数量的星号。如果参数为 0，它就打印 0 个星号；如果参数为 100，它就打印 10 个星号；位于 0～100 的参数值就打印出 0～10 个的星号。换句话说，这个函数打印一幅柱状图的一横，它比传统的循环方案要容易得多，效率也高得多。

13.6　总结

如果声明得当，一个指针变量可以指向另一个指针变量。和其他的指针变量一样，一个指向指针的指针在它使用之前必须进行初始化。为了取得目标对象，必须对指针的指针执行双重的间接访问操作。更多层的间接访问也是允许的（比如一个指向整型的指针的指针的指针），但它们与简单的指针相比用的较少。你也可以创建指向函数和数组的指针，还可以创建包含这类指针的数组。

在 C 语言中，声明是以推论的形式进行分析的。下面这个声明

```
int      *a;
```

把表达式*a 声明为一个整型。你必须随之推断出 a 是个指向整型的指针。通过推论声明，阅读声明的规则就和阅读表达式的规则一样了。

你可以使用函数指针来实现回调函数。一个指向回调函数的指针作为参数传递给另一个函数，后者使用这个指针调用回调函数。使用这种技巧，你可以创建通用型函数，用于执行普通的操作如在一个链表中查找。任何特定问题的某个实例的工作，如在链表中进行值的比较，由客户提供的回

调函数执行。

转移表也使用函数指针。转移表像 switch 语句一样执行选择。转移表由一个函数指针数组组成（这些函数必须具有相同的原型）。函数通过下标选择某个指针，再通过指针调用对应的函数。你必须始终保证下标值处于适当的范围之内，因为在转移表中调试错误是非常困难的。

如果某个执行环境实现了命令行参数，这些参数是通过两个形参传递给 main 函数的。这两个形参通常称为 argc 和 argv。argc 是一个整数，用于表示参数的数量。argv 是一个指针，它指向一个序列的字符型指针。该序列中的每个指针指向一个命令行参数。该序列以一个 NULL 指针作为结束标志。其中第 1 个参数就是程序的名字。程序可以通过对 argv 使用间接访问操作来访问命令行参数。

出现在表达式中的字符串常量的值是一个常量指针，它指向字符串的第 1 个字符。和数组名一样，你既可以用指针表达式也可以用下标来使用字符串常量。

13.7 警告的总结

1. 对一个未初始化的指针执行间接访问操作。
2. 在转移表中使用越界下标。

13.8 编程提示的总结

1. 如果并非必要，避免使用多层间接访问。
2. cdecl 程序可以帮助你分析复杂的声明。
3. 把 void *强制转换为其他类型的指针时必须小心。
4. 使用转移表时，应始终验证下标的有效性。
5. 破坏性的命令行参数处理方式使你以后无法再次进行处理。
6. 不寻常的代码始终应该加上一条注释，描述它的目的和原理。

13.9 问题

1. 下面显示了一列声明。

```
a.  int     abc();
b.  int     abc[3];
c.  int     **abc();
d.  int     (*abc)();
e.  int     (*abc)[6];
f.  int      *abc();
g.  int      **(*abc[6])();
h.  int      **abc[6];
i.  int      *(*abc)[6];
j.  int      *(*abc())();
k.  int      (**(*abc))());
l.  int      (*(*abc))[6];
m.  int      *(*(*(*abc))[6])();
```

从下面的列表中挑出与上面各个声明匹配的最佳描述。

Ⅰ. int 型指针（指向 int 的指针）。
Ⅱ. int 型指针的指针。
Ⅲ. int 型数组。
Ⅳ. 指向“int 型数组”的指针。
Ⅴ. int 型指针数组。
Ⅵ. 指向“int 型指针数组”的指针。
Ⅶ. int 型指针的指针数组。
Ⅷ. 返回值为 int 的函数。
Ⅸ. 返回值为“int 型指针”的函数。
Ⅹ. 返回值为“int 型指针的指针”的函数。
Ⅺ. 返回值为 int 的函数指针。
Ⅻ. 返回值为 int 型指针的函数指针。
XIII. 返回值为 int 型指针的指针的函数指针。
XIV. 返回值为 int 的函数指针的数组。
XV. 指向“返回值为 int 型指针的函数”的指针的数组。
XVI. 指向“返回值为 int 型指针的指针的函数”的指针的数组。
XVII. 返回值为“返回值为 int 的函数指针”的函数。
XVIII. 返回值为“返回值为 int 的函数的指针的指针”的函数。
XIX. 返回值为“返回值为 int 型指针的函数指针”的函数。
XX. 返回值为“返回值为 int 的函数指针”的函数指针。
XXI. 返回值为“返回值为 int 的函数指针的指针”的函数指针。
XXII. 返回值为“返回值为 int 型指针的函数指针”的函数指针。
XXIII. 返回值为“指向 int 型数组的指针”的函数指针。
XXIV. 返回值为“指向 int 型指针数组的指针”的函数指针。
XXV. 返回值为“指向‘返回值为 int 型指针的函数指针’的数组的指针”的函数指针。
XXVI. 非法。

2. 给定下列声明：

```
char      *array[10];
char      **ptr = array;
```

如果变量 ptr 加上 1，它的效果是什么样的？

3. 假定你将要编写一个函数，它的起始部分如下所示：

```
void func( int ***arg ){
```

参数的类型是什么？画一张图，显示这个变量的正确用法。如果想取得这个参数所指代的整数，你应该使用怎样的表达式？

4. 下面的代码可以如何进行改进？

```
Transaction *trans;
trans->product->orders += 1;
trans->product->quantity_on_hand -= trans->quantity;
trans->product->supplier->reorder_quantity
    += trans->quantity;
if( trans->product->export_restricted ){
        ...
}
```

5. 给定下列声明：

```
typedef         struct  {
        int     x;
        int     y;
} Point;

Point   p;
Point   *a = &p;
Point   **b = &a;
```

判断下面各个表达式的值。

a. a

b. *a

c. a->x

d. b

e. b->a

f. b->x

g. *b

h. *b->a

i. *b->x

j. b->a->x

k. (*b)->a

l. (*b)->x

m. **b

6. 给定下列声明：

```
typedef         struct  {
        int     x;
        int     y;
} Point;

Point   x, y;
Point   *a = &x, *b = &y;
```

解释下列各语句的含义。

a. x = y;

b. a = y;

c. a = b;

d. a = *b;

e. *a = *b;

7. 许多 ANSI C 的实现都包含了一个函数，称为 getopt。这个函数用于帮助处理命令行参数。但是，getopt 在标准中并未提及。拥有这样一个函数，有什么优点？又有什么缺点？

8. 下面的代码段有什么错误（如果有的话）？你如何修正它？

```
char    * pathname = "/usr/temp/xxxxxxxxxxxxxxx"
…
/*
**Insert the filename in to the pathname.
*/
strcpy ( pathname+10 , "abcde");
```

9. 下面的代码段有什么错误（如果有的话）？你如何修正它？

```
char     pathname[] = "/usr/temp/";
...
/*
** Append the filename to the pathname.
*/
strcat( pathname, "abcde" );
```

10. 下面的代码段有什么错误（如果有的话）？你如何修正它？

```
char     *pathname [20] = "/usr/temp/ ";
…
/*
**  Append the filename to the pathname.
*/
stroat (pathrame,filename);
```

11. 标准表示如果对一个字符串常量进行修改，其效果是未定义的。如果你修改了字符串常量，有可能会出现什么问题呢？

13.10　编程练习

★★ 1. 编写一个程序，从标准输入读取一些字符，并根据下面的分类计算各类字符所占的百分比：

控制字符
空白字符
数字
小写字母
大写字母
标号符号
不可打印字符

这些字符的分类是根据 ctype.h 中的函数定义的。不能使用一系列的 If 语句。

★ 2. 编写一个通用目的的函数，遍历一个单链表。它应该接受两个参数：一个指向链表第 1 个节点的指针和一个指向一个回调函数的指针。回调函数应该接受单个参数，也就是指向一个链表节点的指针。对于链表中的每个节点，都应该调用一次这个回调函数。这个函数需要知道链表节点的什么信息？

★★ 3. 转换下面的代码段，使它改用转移表而不是 switch 语句。

```
Node *list;
Node *current;
Transaction  *transaction;
typedef enum  { NEW, DELETE, FORWARD, BACKWARD,
     SEARCH, EDIT } Trans_type;
...
switch( transaction->type ) {
case    NEW
```

```
            add_new_trans(list,transaction);
            break;
    case DELETE:
            current = delete_trans( list, current );
            break;

    case FORWARD:
            current = current->next;
            break;

    case BACKWARD:
            current = current->prev;
            break;

    case SEARCH:
            current = search( list, transaction );
            break;

    case EDIT:
            edit( current, transaction );
            break;

    default:
            printf( "Illegal transaction type!\n" );
            break;
    }
```

★★★★4. 编写一个名叫 sort 的函数，它用于对一个任何类型的数组进行排序。为了使函数更为通用，它的其中一个参数必须是一个指向比较回调函数的指针，该回调函数由调用程序提供。比较函数接受两个参数，也就是两个指向需要进行比较的值的指针。如果两个值相等，函数返回零；如果第 1 个值小于第 2 个，函数返回一个小于零的整数；如果第 1 个值大于第 2 个，函数返回一个大于零的整数。

sort 函数的参数将是：

1. 一个指向需要排序的数组的第 1 个值的指针。
2. 数组中值的个数。
3. 每个数组元素的长度。
4. 一个指向比较回调函数的指针。

sort 函数没有返回值。

你将不能根据实际类型声明数组参数，因为函数应该可以对不同类型的数组进行排序。如果你把数据当作一个字符数组使用，你可以用第 3 个参数寻找实际数组中每个元素的起始位置，也可以用它交换两个数组元素（每次一个字节）。

对于简单的交换排序，你可以使用下面的算法，当然也可以使用你认为更好的算法。

```
for i = 1 to 元素数-1 do
    for j = i + 1 to 元素数 do
        if 元素 i > 元素 j then
            交换元素 i 和元素 j
```

★★★★★5. 编写代码处理命令行参数是十分乏味的，所以最好有一个标准函数来完成这项工作。但是，不同的程序以不同的方式处理它们的参数。所以，这个函数必须非常灵活，以便使它能用于更多的程序。在本题中，你将编写这样一个函数。你的函数通过寻找和提取参数来提供灵活性。用户所提供的回调函数将执行实际的处理工作。

下面是函数的原型。注意它的第 4 个和第 5 个参数是回调函数的原型。

```
char **
do_args( int argc, char **argv, char *control,
    void (*do_arg)( int ch, char *value ),
    void (*illegal_arg)( int ch ) );
```

头两个参数就是 main 函数的参数，main 函数对它们不作修改，直接传递给 do_args 第 3 个参数是个字符串，用于标识程序期望接受的命令行参数。最后两个参数都是函数指针，它们是由用户提供的。

do_args 函数按照下面这样的方式处理命令行参数：

```
跳过程序名参数
while 下一次参数以一个横杠开头
      对于参数横杠后面的每个字符
         处理字符
返回一个指针，指向下一个参数指针。
```

为了“处理字符”，你首先必须观察该字符是否位于 control 字符串内。如果它并不位于那里，调用 illegal_arg 所指向函数，把这个字符作为参数传递过去。如果它位于 control 字符串内，但它的后面并不是跟一个+号，那么就调用 do_arg 所指向的函数，把这个字符和一个 NULL 指针作为参数传递过去。

如果该字符位于 control 字符串内并且后面跟一个+号，那么就应该有一个值与这个字符相联系。如果当前参数还有其他字符，它们就是我们需要的值。否则，下一个参数才是这个值。在任何一种情况下，你应该调用 do_arg 所指向的函数，把这个字符和指向这个值的指针传递过去。如果不存在这个值（当前参数没有其他字符，且后面不再有参数），那么你应该改而调用 illegal_arg 函数。**注意**：你必须保证这个值中的字符以后不会被处理。

当所有以一个横杠开头的参数被处理完毕后，你应该返回一个指向下一个命令行参数的指针的指针（也就是一个诸如&argv[4]或 argv+4 的值）。如果所有的命令行参数都以一个横杠开头，你就返回一个指向“命令行参数列表中结尾的 NULL 指针”的指针。

这个函数必须既不能修改命令行参数指针，也不能修改参数本身。为了说明这一点，假定程序 prog 调用这个函数：下面的例子显示了几个不同集合的参数的执行结果。

命令行：	$ prog –x –y z
control：	“x”
do_args 调用：	(*do_arg)(‘x’, 0)
	(*illegal_arg)(‘y’)
并且返回：	&argv[3]
命令行：	$ prog –x –y –z
control：	“x+y+z+”
do_args 调用：	(*do_arg)(‘x’, “-y”)
	(*illegal_arg)(‘z’)
并且返回：	&argv[4]

续表

命令行：	$ prog –abcd –ef ghi jkl
control：	“ab+cdef+g”
do_args 调用：	(*do_arg)(‘a’, 0)
	(*do_arg)(‘b’, “cd”)
	(*do_arg)(‘e’, 0)
	(*do_arg)(‘f’, “ghi”)
并且返回：	&argv[4]
命令行：	$ prog –a b –c –d –e –f
control：	“abcdef”
do_args 调用：	(*do_arg)(‘a’, 0)
并且返回：	&argv[2]