8.4.2 指向字符的指针、字符数组和字符串

字符数组常常用作字符串来使用。这里提醒读者，作为字符串的字符数组的末尾必须有一个‘\0’作为结束标志。不过，字符数组的使用总有些不太方便，因此经常用字符指针来表示字符串。请看如下声明：

```
char str1[] = { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’ };
char str2[] = “Hello”;
char *str3 = “Hello”;
```

这里，用了 3 种不同的方式来声明字符串。虽然表面上看它们的效果一样，但实际的内部实现是完全不同的。

- str1 是数组而非指针，它的长度是 6。str1 数组被一系列字符常量初始化。
- str2 也是数组而非指针，它的长度是 6。str2 数组被长度为 5 的字符串常量初始化。在初始化时，字符串常量的每一个字符被复制到了数组 str2 的对应元素中，包括结尾‘\0’。
- str3 是指针而非数组，它被初始化指向了一个字符串常量。换句话说就是，str3 获得了字符串的首地址，也就是字符‘H’的地址。
- str1 和 str2 数组的每一个元素都是变量，因此可以被改变；而 str3 本身是个变量（它可以指向任意其他的字符串），但现在它指向了一串常量，而常量串中的每一个字符都不能被修改。

【例 8-11】指针、字符数组和字符串。

```
//8-11.c
#include <stdio.h>

int main()
{
    char str1[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
    char str2[] = "Hello";
    char *str3 = "Hello";

    printf("%s\n%s\n%s\n", str1, str2, str3);

    str1[0] = 'h';
    str2[0] = 'h';
    //str3[0] = 'h'; //这条语句会引起一个运行时错误
    printf("%s\n%s\n%s\n", str1, str2, str3);

    return 0;
}
```

程序的运行结果是：

```
Hello
Hello
Hello
hello
hello
Hello
```

在将字符指针作为字符串对待时，应该注意以下问题：

（1）含义问题。如例中所示，表达式 str3 代表了字符串，而*str3 则是一个字符；

（2）超界问题。指针和数组（包括字符串）连用总是存在这个问题，并且不好把握。幸好字符串有一个结尾标志，可以通过指针是否指向了这个标志来判断；

（3）初始化问题。未初始化的指针是不能使用的，这点读者都已经知道。特别是在输入字符串的时候，往往需要用一个足够大的字符数组来缓存，然后通过一个指针来访问该数组。

【例 8-12】从键盘输入一个字符串，求出它的长度。

```
//8-12.c
#include <stdio.h>
```

```
int main()
{
    char str[80];
    char *p;

    printf("Please input a string:\n");
    gets(str);

    p = str;
    while (*p != '\0') p++;
    printf("The length of the string is %d.\n", p - str);

    return 0;
}
```

运行结果如下：

```
Please input a string:
to be or not to be, it's a question.↙
The length of the string is 36.
```

代码中用到了一个技巧：首先将指针从字符串的头移到尾，然后求头尾之间地址的差（以单元计）。由于一个字符单元的大小是 1 字节，因此这个差恰好就是字符串的长度。

在程序中移动字符指针到串尾的语句是：

```
while (*p != '\0') p++;
```

而这条语句常常被程序员精简为：

```
while (*p++);
--p;
```

8.4.3 指针数组

数组元素的基类型可以是任意类型，也包括了指针类型。由指针构成的数组就是指针数组。例如：

```
char *weekday[7];
```

现在解析这个声明如下。

（1）先解析出变量名 weekday。

（2）weekday 前后都有修饰符，但“[]”的优先级高于“*”，因此 weekday 先与“[]”结合。这样，weekday 就不是指针，而是一个数组，并且是一维的，其长度为 7。

（3）weekday 前面的所有符号都是它的基类型。此例中是 char *。因此，weekday 数组的每一个元素都是指针，这些指针指向了字符单元。根据前面的讲解，这些字符指针往往作为字符串使用。

将声明扩展一下，为数组的每个元素赋初值：

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

该声明的内存布局情况如图 8-10 所示。

【例 8-13】指针数组的使用。

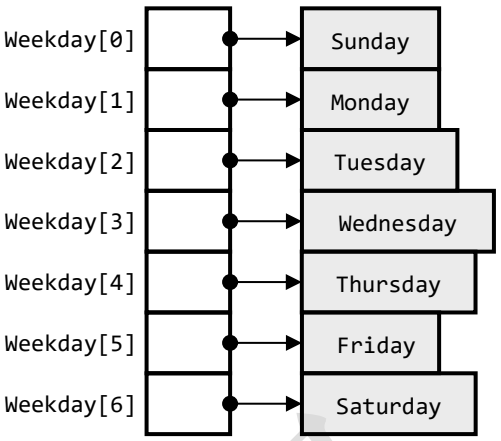


图 8-10 指针数组

```
//8-13.c
#include <stdio.h>

int main()
{
     const int WEEK = 7;
     char *weekday[] = { "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday" };
     int i;

     printf("Weekdays\n----------\n");
     for (i = 0; i < WEEK; i++)
     printf("%d.  %s\n", i + 1, weekday[i]);

     printf("Please choose a number:\n");
     scanf("%d", &i);
     if (i >= 1 && i <= WEEK)
          printf("You chose %s\n", weekday[i-1]);
     else
          printf("You chose an illegal one.\n");

     return 0;
}
```

程序的运行结果是：

```
Weekdays
----------
1. Sunday
2. Monday
3. Tuesday
4. Wednesday
5. Thursday
6. Friday
7. Saturday
Please choose a number:
4↙
You chose Wednesday
```

8.4.4 指向数组的指针

如果有声明：

```
int a[10];
int *p = &a[0]; //或者 int *p = a;
```

指针 p 获得了数组 a 的首元素的地址，因此可以说它们是指向了数组元素的指针。在这种情况下，通过指针 p，看到的仅仅是一个整型单元。不过由于数组的连续存储性，可以通过移动这个指针来访问数组的所有元素。

如果要把数组作为一个整体来对待，那么与它相联系的指针就是一个指向数组的指针。这种指针的声明如下：

```
int (*q)[10];
```

对这个声明的解析如下。

（1）变量名 q 由于()的作用，首先与“*”结合。这样，q 一定是个指针，而不是一个数组。

（2）剩下的部分都是 q 的基类型。在此例中，是 int[10]。很明显，这是一个数组。

综上所述，变量 q 是一个指向数组的指针（简称数组指针），它指向了一个长度为 10 的一维数组，而该数组的所有元素都是整型的。对比以下声明：

```
int *t[10];
```

前面已经分析过，变量 t 是一个一维指针数组，该数组长度为 10，数组的每个元素都是一个指针，并且这些指针都应该指向一个整型单元。基于此，t 的类型是 int*[10]，而 q 的类型是 int(*)[10]，二者完全不同。

数组指针在使用前应该初始化。如果期望 q 指向 a,那么有理由相信读者的第一想法是这样的：

```
q = a; //或者 q = &a[0];
```

但实际上，这种赋值是错误的。为什么呢?

这里来分析一下上述语句。q 是一个指针没错，但它的基类型是 int [10]；虽然 a 或者 &a[0]的值也是一个指针，但它们的基类型是 int。因此，两种不同类型的指针之间赋值当然是错误的。虽然这种错误在编译时仅仅是一个警告，但一定要清楚地认识到它在概念上的混淆以及潜在的危害。那么怎样才能正确地初始化 q 呢?

对比 q 和 a 的声明：

```
int a[10];
int (*q)[10];
```

如果 q 指向了 a，那么很明显，a 和*q 是等价的。也就是说，*q 等价于 a，而 q 就应该等价于&a。有了这样的理解，那么初始化 q 就变得简单了：

```
q = &a;
```

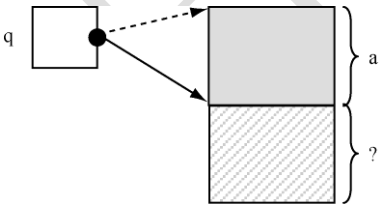


图 8-11 ++q 使 q 跳过整个数组 a

在使用 q 时，下面的语句是错误的：

```
int b = q[i];
```

正确的语句应该是：

```
int b = (*q)[i];
```

现在再来对比一下本小节中声明的两个指针 p 和 q 的算术运算情况。读者已经知道，如果 p 指向数组 a 的首元素，那么++p 会使其指向 a 的第二个元素。而指针 q 指向了整个数组 a，因此++q 将使 q 跳过整个数组 a 而指向其尾元素后面的那个单元，并将其后的连续 10 个单元解释为一个整型数组。而这种解释是否成立要视情况而定。++q 将使 q 跳过整个整组 a 的这种情况如图 8-11 所示。

【例 8-14】使用指向数组的指针来访问数组

```
//8-14.c
#include <stdio.h>

int main()
{
```

```
    int a[5] = {1, 2, 3, 4, 5};
    int (*p)[5];
    int i;

    p = &a;
    for (i = 0; i < 5; ++i) printf("%4d", (*p)[i]);

    return 0;
}
```

程序的运行结果是：

```
   1   2   3   4   5
```

8.5 指向指针的指针

在【例 8-13】中，weekday 是一个指针数组，那么 weekday[1]就是一个指针，它指向字符串“Monday”。那么表达式&weekday[1]又是个什么类型的值呢?

一个单元的地址是一个指针。如果那个单元本身也是一个指针，那么它的地址就是指针的指针。即一个指针的指针指向了一个指针，而后者又指向了一个特定的对象，想要访问该对象，就必须经过两次指针间接访问。

定义一个指向指针的指针变量其实很简单，其语法如下。

```
int **pp;
```

指针运算符“*”是单目运算符，其结合方式是从右向左结合，因此变量名 pp 首先和最靠近它的“*”结合，非常明显地说明了 pp 是个指针，那么声明中所有在*pp 左边的部分都是 pp 指向的基类型，在例中是 int *，而这又是一个指针类型，所以，变量 pp 是一个指向整型指针的指针变量。可以这样来解读声明

```
(int *) *pp;
```

又给出声明

```
int a = 111;
int *p = &a;
int **pp = &p;
```

那么图 8-12 就示意了 pp 变量的指向情况：

表达式*pp、*p 分别间接引用了变量 p 和 a。

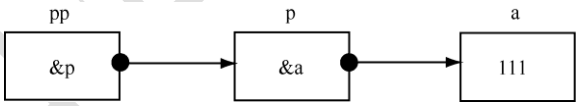


图 8-12　指向指针的指针

【例 8-15】指向指针的指针。

```
//8-15.c
#include <stdio.h>

int main()
{
    int a = 111;
    int *p = &a;
    int **pp = &p;

    printf("a=%d, *p=%d, **p=%d\n", a, *p, **pp);
    printf("&a=%p, p=%p\n&p=%p, pp=%p", &a, p, &p, pp);

    return 0;
}
```

程序的运行结果如下：

```
a=111, *p=111, **p=111
&a=0022FF18, p=0022FF18
&p=0022FF14, pp=0022FF14
```

从程序的运行结果可以清楚地看到：p 是 a 的地址，而 pp 是 p 的地址。

8.6 指针和函数

指针常出现在函数的参数列表和函数的返回值中，这为满足一些特殊的要求带来了方便。

8.6.1 指针作为函数的参数

1．指针作为函数的参数

为了说明指针作为函数参数的特性，首先来看一个例子。

【例 8-16】编写一个 swap()函数来交换两个变量的值。

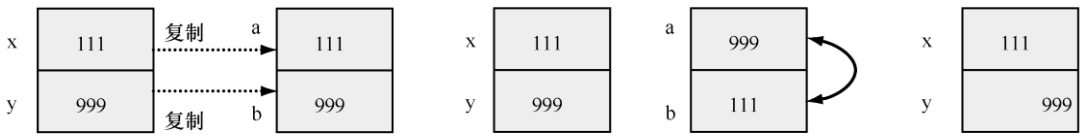
```
//8-16.c
#include <stdio.h>

void swap(int a, int b);

int main()
{
int x = 111, y = 999;

printf("Beginning: x = %d, y = %d\n", x, y);
swap(x, y);
printf("Done: x = %d, y = %d\n", x, y);

return 0;
}

void swap(int a, int b)
{
int t = a;
a = b;
b = t;

printf("Swapping: a = %d, b = %d\n", a, b);
}
```

程序的运行结果是：

```
Beginning: x = 111, y = 999
Swapping: a = 999, b = 111
Done: x = 111, y = 999
```

可以看到，函数 swap()并没有完成所要求的工作。这是为什么呢?

这里用图 8-13 来解释在调用函数 swap()时内存的情况。



(a) 调用swap( )时　　(b) swap( )中交换a和b　　(c) swap( )返回后

图 8-13　swap()调用前后实参和形参的情况

从图中可以清楚地看到，当 main()调用 swap()时，实参 x、y 被复制到了作为临时单元的形参 a、b 中。虽然 swap()确实交换了 a、b 的值，但对实参没有任何影响。swap()返回后，形参 a 和 b 失效，实参 x 和 y 保持不变。这就是 C 语言传值调用的结果。

要真正使形参的改变影响到实参，可以使用指针。请看【例 8-17】。

【例 8-17】使用指针交换两个变量的值。

```
//8-17.c
#include <stdio.h>

void swap(int *pa, int *pb);

int main()
{
int x = 111, y = 999;

printf("Beginning: x = %d, y = %d\n", x, y);
swap(&x, &y);
printf("Done: x = %d, y = %d\n", x, y);

return 0;
}

void swap(int *pa, int *pb)
{
int t = *pa;
*pa = *pb;
*pb = t;

printf("Swapping: *pa = %d, *pb = %d\n", *pa, *pb);
}
```

运行结果如下：

```
Beginning: x = 111, y = 999
Swapping: *pa = 999, *pb = 111
Done: x = 999, y = 111
```

图 8-14 解释了函数调用时形参和实参发生的变化。

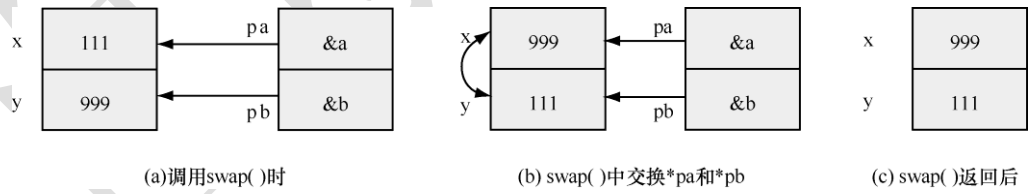


图 8-14　swap()调用前后实参和形参的情况

当 main()调用 swap()时，实参 x、y 的地址被分别复制到了形参 pa、pb 中，pa 和 pb 就分别指向了实参 x 和 y；swap()函数没有交换 pa 和 pb，而交换的是*pa 和*pb，也就是交换了它们指向的单元中的值，实参 x 和 y 的值就这样被交换了。

通过形参指针可以间接地修改实参。换句话说，就是指针参数具有携带结果返回 caller 中的能力。因此，在函数会产生多个结果的情况下，可以通过指针参数来实现结果的返回。一般地，具有这种能力的参数称为“**输出(out)参数**”，而不能携带结果的参数称为“**输入(in)参数**”。例如，如果一个函数 f()要向其 caller 返回两个计算结果，那么可以将其原型声明为：

```
int f(int *x, int *y);
```

在 f()内部可以通过改变*x 和*y 的值而将结果传回 caller。

2．再谈 scanf()的参数

读者已经知道，scanf()的参数列表是一系列地址。实际上，这些参数就是一些指针，而指针参数拥有携带结果返回到 caller 的能力。而这正是 scanf()希望的。

如果用 scanf()读取普通变量的值，那么必须在变量前加上“&”运算符；而如果参数本身已经是个指针（包括字符数组名），那么就不需要加“&”符号了。例如：

```
int a, *p = &a;
char s[128], *str=s;
scanf("%d", &a);
scanf("%d", p);
scanf("%s", s);
scanf("%s", str);
```

> 除了作为字符串使用的字符数组外，其他类型的数组不能一次性输入。

3．指针参数和数组参数是等效的

读者已经知道，C 编译器将数组名视为一个常量指针。因此，使用数组作为参数和使用指向数组元素的指针作为参数是完全等效的。例如以下 3 个函数声明完全等效：

```
void f(int a[]);
void f(int a[5]);
void f(int *p);
```

因为以上声明隐含的函数类型都是：

```
void (int *)
```

4．用 const 约束指针参数的行为

在有些情况下，用指针作为参数，但又不希望实参被修改，那么可以将形参指针用 const 关键字来修饰。

【例 8-18】两个字符串的比较。

【解题思路】字符串的比较只有两个结果：严格的相等和不相等。字符串相等的定义是两个字符串不仅长度一样，并且对应位置上的每个字符都一样。

但是，根据 C 语言的规定，类似于“abc”==“abc”这样的字符串直接比较虽然在语法上是正确的，但其含义却不是字面暗示的那样，而是在比较两个字符串的**首地址**是否相等。

> 表达式“abc”==“abc”的确会得到真结果，但这是因为编译器认为这两个字符串实际上是放在同一位置的同一个常量。如果改成 char s1[]=“abc”，s2[]=“abc”；那么 s1==s2 就会得到假结果。

因此，根据上面的定义，可以设计出这样的算法：从第一个字符起两两进行比较，如果不相等就直接返回假，否则进行下一个位置的比较，直到遇到任何一个串的结尾为止。如果两个串同时结束，则返回真，否则返回假。

```
//8-18.c
#include <stdio.h>

int Strcmp(const char *s1, const char *s2);

int main()
{
char *s1 = "Hello", *s2 = "Hello!";

printf(Strcmp(s1, s2) ? "Identical" : "Different");

return 0;
```

```
}

int Strcmp(const char *s1, const char *s2)
{
for (; *s1 != '\0' && *s2 != '\0'; ++s1, ++s2)
     if (*s1 != *s2) return 0;

if (*s1 + *s2 == 0) return 1; //两串同时结束
return 0;
}
```

程序的运行结果如下：

```
Different
```

5．main()函数的参数

在前面所有的例子中，main()函数都没有带任何参数。实际上，main()函数可以带有参数，其形式如下：

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

读者不禁要问：main()的参数值由谁来提供呢？这些参数又有什么含义呢？

书中曾经提到过，main()函数是 C 程序的入口，而 C 程序的运行是在操作系统之下的。从这个角度来看：操作系统就是 main()函数的调用者，因此 main()的参数应该由它来提供。

操作系统在启动一个应用程序的时候，例如启动 gcc 编译器，常采用这样的方式：

```
gcc 8-1.c
```

这就是常说的“命令行(*command line*)”。其中，gcc 是命令，8-1.c 是命令行参数，该参数由命令来接收。

如果命令行中的命令是程序员编写的 C 程序，那么命令行参数就会由 main()函数来接收。

回到 main()函数的声明上。参数中，argc 参数规定了命令行参数的个数，包括了命令本身；argv 参数是个字符指针数组，其中存储了命令行中各参数的文本字符串。例如上面的命令行中，argc=2，argv[0]=“gcc”，argv[1]=“8-1.c”。

使用命令行参数可以为程序的运行带来更多的选择。

【例 8-19】编写一个程序，计算 1+2+…+n。其中 n 为一个大于 0 的正整数，它不是从键盘输入，而是由 main()函数的命令行参数指定。

【解题思路】n 的值可以从 main()的命令行参数获得。不过一个问题是：在 argv[]中保存的是字符串，因此需要编写一个函数将字符串转换成整数。

字符串转换成整数的算法可以这样描述：用一个指针指向字符串的首字符 c1，令 d=c1-‘0’，这样可以获得字符 c1 对应的十进制数字。然后后移指针使其指向后续字符 c2，再令 d=d*10+(c2-‘0’)。以此类推，直到指针指向字符串的末尾‘\0’。最后的 d 就是转换后的值。

程序的编码如下：

```
//8-19.c
#include <stdio.h>

int stoi(const char *s)
{
int d = 0;
char *p = (char *)s;

for (; *p != '\0'; ++p) d = d * 10 + (*p - '0');

return d;
}

int sigma(int n)
{
```

```
int i, sum;

for (i = 1, sum = 0; i <= n; ++i) sum += i;

return sum;
}

int main(int argc, char *argv[])
{
if (argc < 2)
{
     printf("Usage:8-19 n\ne.g. 8-19 100");
     return -1;
}

printf("1+2+...+%s=%d", argv[1], sigma(stoi(argv[1])));

return 0;
}
```

程序的 3 次运行情况如下（Windows 下）：

```
E:\>8-19↙
Usage:8-19 n
e.g. 8-19 100
E:\>8-19 100↙
1+2+...+100=5050
E:\>8-19 10000↙
1+2+...+10000=50005000
```

8.6.2 函数返回指针

一般情况下函数会返回一个简单类型的值。可以简单地认为这个返回值存储在一个临时常量对象中。而有的时候，函数的运算会产生一个左值对象，并且希望将这个对象传递给调用者。在这种情况下，可以让函数返回一个指向这个数据对象的指针，而不是将整个对象返回给调用者。

【例 8-20】在字符串中查询是否存在给定的字符。

```
//8-20.c
#include <stdio.h>

char * Strchr(char ch, const char *s);

int main()
{
char *s = "The quick fox jumps over a lazy dog";

printf(Strchr('x', s) != NULL ? "Found" : "Not found");

return 0;
}

char * Strchr(char ch, const char *s)
{
for (; *s != '\0'; ++s)
     if (*s == ch) return (char *)s; //类型转换是为了避免编译警告

return NULL;
}
```

程序的运行结果如下：

Found

读者已经知道，函数返回后函数中所有局部变量都会失效，包括形参。所以，如果函数要返回指针，那么该指针指向的数据对象在函数结束后还必须存在。【例 8-21】是个反例。

【例 8-21】函数返回一个已经失效的对象的指针。

```
//8-21.c
#include <stdio.h>

double * foo();

int main()
{
double *p = foo();

return 0;
}

double * foo()
{
double d = 12.345;

return &d;
}
```

编译时将会得到如下警告（VC9）：

```
warning C4172: returning address of local variable or temporary
```

这个警告是不能忽略的。修改的方案可以非常简单，其中一个就是将

```
double d = 12.345;
```

改为

```
static double d = 12.345;
```