第7章

函数

C 的函数和其他语言的函数（或过程、方法）相似之处甚多。所以到现在为止，尽管我们对函数只是进行了一点非正式的讨论，但你已经能够使用它们了。但是，函数的有些方面并不像直觉上应该的那样，所以本章将正式描述 C 的函数。

7.1 函数定义

函数的定义就是函数体的实现。函数体就是一个代码块，它在函数被调用时执行。与函数定义相反，函数声明出现在函数被调用的地方。函数声明向编译器提供该函数的相关信息，用于确保函数被正确地调用。首先让我们来看一下函数的定义。

函数定义的语法如下：

类型

函数名(***形式参数***)

代码块

回忆一下，代码块就是一对花括号，里面包含了一些声明和语句（两者都是可选的）。因此，最简单的函数大致如下所示：

```
function_name()
{
}
```

当这个函数被调用时，它简单地返回。然而，它可以实现一种有用的**存根**（stub）目的，为那些此时尚未实现的代码保留一个位置。编写这类存根，或者说为尚未编写的代码“占好位置”，可以保持程序在结构上的完整性，以便于你编译和测试程序的其他部分。

形式参数列表包括变量名和它们的类型声明。代码块包含了局部变量的声明和函数调用时需要执行的语句。程序 7.1 是一个简单函数的例子。

把函数的类型与函数名分写两行纯属风格问题。这种写法可以使我们在使用视觉或某些工具程序追踪源代码时更容易查找函数名。

K&R C

在 K&R C 中，形式参数的类型以单独的列表进行声明，并出现在参数列表和函数体的左花括号之间，如下所示：

```
int *
find_int(key, array, array_len)
int key;
int array[];
int array_len;
{
```

这种声明形式现在仍为标准所允许，主要是为了让较老的程序无需修改便可通过编译。但我们应该提倡新声明风格，理由有二：首先，它消除了旧式风格的冗余。其次，也是更重要的一点，它允许函数原型的使用，提高了编译器在函数调用时检查错误的能力。关于函数原型，我们将在本章后面的内容里讨论。

```
/*
** 在数组中寻找某个特定整型值的存储位置，并返回一个指向该位置的指针。
*/
#include <stdio.h>

int *
find_int( int key, int array[], int array_len )
{
        int i;

        /*
        ** 对于数组中的每个位置 ...
        */
        for( i = 0; i < array_len; i += 1 )
            /*
            ** 检查这个位置的值是否为需要查找的值。
            */
            if( array[ i ] == key )
                return &array[ i ];

        return NULL;
}
```

程序 7.1 在数组中寻找一个整型值 find_int.c

return 语句

当执行流到达函数定义的末尾时，函数就将返回(return)，也就是说，执行流返回到函数被调用的地方。return 语句允许你从函数体的任何位置返回，并不一定要在函数体的末尾。它的语法如下所示：

```
return  expression;
```

表达式 expression 是可选的。如果函数无需向调用程序返回一个值，它就被省略。这类函数在绝大多数其他语言中被称为过程（procedure）。这些函数执行到函数体末尾时隐式地返回，它们没有返回值。这种没有返回值的函数在声明时应该把函数的类型声明为 void。

真函数是从表达式内部调用的，它必须返回一个值，用于表达式的求值。这类函数的 return 语

句必须包含一个表达式。通常，表达式的类型就是函数声明的返回类型。只有当编译器可以通过寻常算术转换把表达式的类型转换为正确的类型时，才允许返回类型与函数声明的返回类型不同的表达式。

有些程序员更喜欢把 return 语句写成下面这种样子：

```
return ( x );
```

语法并没有要求你加上括号。但如果你喜欢，尽管加上，因为在表达式两端加上括号总是合法的。

在 C 中，子程序不论是否存在返回值，均被称为函数。调用一个真函数（即返回一个值的函数）但不在任何表达式中使用这个返回值是完全可能的。在这种情况下，返回值就被丢弃。但是，从表达式内部调用一个过程类型的函数（无返回值）是一个严重的错误，因为这样一来在表达式的求值过程中会使用一个不可预测的值（垃圾）。幸运的是，现代的编译器通常可以捕捉这类错误，因为它们较之老式编译器在函数的返回类型上更为严格。

7.2　函数声明

当编译器遇到一个函数调用时，它产生代码传递参数并调用这个函数，而且接收该函数返回的值（如果有的话）。但编译器是如何知道该函数期望接受的是什么类型和多少数量的参数呢？如何知道该函数的返回值（如果有的话）类型呢？

如果没有关于调用函数的特定信息，编译器便假定在这个函数的调用时参数的类型和数量是正确的。它同时会假定函数将返回一个整型值。对于那些返回值并非整型的函数而言，这种隐式认定常常导致错误。

7.2.1　原型

向编译器提供一些关于函数的特定信息显然更为安全，我们可以通过两种方法来实现。首先，如果同一源文件的前面已经出现了该函数的定义，编译器就会记住它的参数数量和类型，以及函数的返回值类型。接着，编译器便可以检查该函数的所有后续调用（在同一个源文件中），确保它们是正确的。

K&R C：

如果函数是以旧式风格定义的，也就是用一个单独的列表给出参数的类型，那么编译器就只记住函数的返回值类型，但不保存函数的参数数量和类型方面的信息。由于这个缘故，只要有可能，你都应该使用新式风格的函数定义，这点非常重要。

第 2 种向编译器提供函数信息的方法是使用**函数原型**(function prototype)，你在第 1 章已经见过它。原型总结了函数定义的起始部分的声明，向编译器提供有关该函数应该如何调用的完整信息。使用原型最方便（且最安全）的方法是把原型置于一个单独的文件，当其他源文件需要这个函数的原型时，就使用#include 指令包含该文件。这个技巧避免了错误键入函数原型的可能性，它同时简化了程序的维护任务，因为这样只需要该原型的一份物理拷贝。如果原型需要修改，你只需要修改它的一处拷贝。

举个例子，这里有一个 find_int 函数的原型，取自前面的例子：

```
int *find_int( int key, int array[], int len );
```

注意最后面的那个分号：它区分了函数原型和函数定义的起始部分。原型告诉编译器函数的参数数量和每个参数的类型以及返回值的类型。编译器见过原型之后，就可以检查该函数的调用，确保参数正确、返回值无误。当出现不匹配的情况时（例如，参数的类型错误），编译器会把不匹配的实参或返回值转换为正确的类型，当然前提是这样的转换必须是可行的。

提示：

注意我在上面的原型中加上了参数的名字。虽然它并非必需，但在函数原型中加入描述性的参数名是明智的，因为它可以给希望调用该函数的客户提供有用的信息。例如，你觉得下面这两个函数原型哪个更有用？

```
char *strcpy( char *, char * );
char *strcpy( char *destination, char *source );
```

警告：

下面的代码段例子说明了一种使用函数原型的危险方法。

```
Void
a()
{
        int     *func( int *value, int len);
        ...
}

void
b()
{
        int     func( int len, int *value );
        ...
}
```

仔细观察一下这两个原型，你会发现它们是不一样的。参数的顺序倒了，返回类型也不同。问题在于这两个函数原型都写于函数体的内部，它们都具有代码块作用域，所以编译器在每个函数结束前会把它记住的原型信息丢弃，这样它就无法发现它们之间存在的不匹配情况。

标准表示，在同一个代码块中，函数原型必须与同一个函数的任何先前原型匹配，否则编译器应该产生一条错误信息。但是，在这个例子里，第 1 个代码块的作用域并不与第 2 个代码块重叠。因此，原型的不匹配就无法被检测到。这两个原型至少有一个是错误的（也可能两个都错），但编译器看不到这种情况，所以不会发出任何错误信息。

下面的代码段说明了一种使用函数原型的更好方法。

```
#include "func.h"
void
a()
{
        ...
}

void
b()
{
        ...
}
```

文件 **func.h** 包含了下面的函数原型

```
int  *func( int *value, int len );
```

从几个方面看，这个技巧比前一种方法更好。

1．现在函数原型具有文件作用域，所以原型的一份拷贝可以作用于整个源文件，较之在该函数每次调用前单独书写一份函数原型要容易得多。

2．现在函数原型只书写一次，这样就不会出现多份原型的拷贝之间的不匹配现象。

3．如果函数的定义进行了修改，我们只需要修改原型，并重新编译所有包含了该原型的源文件即可。

4．如果函数的原型同时也被#include 指令包含到定义函数的文件中，编译器就可以确认函数原型与函数定义的匹配。

通过只书写函数原型一次，我们消除了多份原型的拷贝间不一致的可能性。然而，函数原型必须与函数定义匹配。把函数原型包含在定义函数的文件中可以使编译器确认它们之间的匹配。

考虑下面这个声明，它看上去有些含糊：

```
int *func();
```

它既可以看作是一个旧式风格的声明（只给出 func 函数的返回类型），也可以看作是一个没有参数的函数的新风格原型。它究竟是哪一个呢？这个声明必须被解释为旧式风格的声明，目的是保持与 ANSI 标准之前的程序的兼容性。一个没有参数的函数的原型应该写成下面这个样子：

```
int *func( void );
```

关键字 void 提示没有任何参数，而不是表示它有一个类型为 void 的参数。

7.2.2　函数的缺省认定

当程序调用一个无法见到原型的函数时，编译器便认为该函数返回一个整型值。对于那些并不返回整型值的函数，这种认定可能会引起错误。

警告：

所有的函数都应该具有原型，尤其是那些返回值不是整型的函数。记住，值的类型并不是值的内在本质，而是取决于它被使用的方式。如果编译器认定函数返回一个整型值，它将产生整数指令操纵这个值。如果这个值实际上是个非整型值，比如说是个浮点值，其结果通常将是不正确的。

让我们看一个这种错误的例子。假设有一个函数 xyz，它返回 float 值 3.14。在 Sun Sparc 工作站中，用于表示这个浮点数的二进制位模式如下：

```
01000000010010001111010111000011
```

现在假定函数是这样被调用的：

```
float  f;
...
f = xyz();
```

如果在函数调用之前编译器无法看到它的原型，它便认定这个函数返回一个整型值，并产生指令将这个值转换为 float，然后再赋值给变量 f。

函数返回的位如上所示。转换指令把它们解释为整型值 1 078 523 331，并把这个值转换为 float 类型，结果存储于变量 f 中。

为什么函数的返回值实际上已经是浮点值的形式时，还要执行类型转换呢？编译器并没有办法知道这个情况，因为没有原型或声明告诉它这些信息。这个例子说明了为什么返回值不是整型的函

数具有原型是极为重要的。

7.3 函数的参数

C 函数的所有参数均以“传值调用”方式进行传递，这意味着函数将获得参数值的一份拷贝。这样，函数可以放心修改这个拷贝值，而不必担心会修改调用程序实际传递给它的参数。这个行为与 Modula 和 Pascal 中的值参数（不是 var 参数）相同。

C 的规则很简单：所有参数都是传值调用。但是，如果被传递的参数是一个数组名，并且在函数中使用下标引用该数组的参数，那么在函数中对数组元素进行修改实际上修改的是调用程序中的数组元素。函数将访问调用程序的数组元素，数组并不会被复制。这个行为被称为“传址调用”，也就是许多其他语言所实现的 var 参数。

数组参数的这种行为似乎与传值调用规则相悖。但是，此处其实并无矛盾之处——数组名的值实际上是一个指针，传递给函数的就是这个指针的一份拷贝。下标引用实际上是间接访问的另一种形式，它可以对指针执行间接访问操作，访问指针指向的内存位置。参数（指针）实际上是一份拷贝，但在这份拷贝上执行间接访问操作所访问的是原先的数组。我们将在下一章再讨论这一点，此处只要记住两个规则：

1. 传递给函数的标量参数是传值调用的。
2. 传递给函数的数组参数在行为上就像它们是通过传址调用的那样。

```
/*
** 对值进行偶校验。
*/

int
even_parity( int value, int n_bits )
{
        int parity = 0;

        /*
        ** 计数值中值为 1 的位的个数。
        */
        while( n_bits > 0 ){
                parity += value & 1;
                value >>= 1;
                n_bits -= 1;
        }

        /*
        ** 如果计数器的最低位是 0，返回 TRUE(表示 1 的位数为偶数个)。
        */
        return ( parity % 2 ) == 0;
}
```

程序 7.2 奇偶校验 parity.c

程序 7.2 说明了标量函数参数的传值调用行为。函数检查第 1 个参数是否满足偶校验，也就是它的二进制位模式中 1 的个数是否为偶数。函数的第 2 个参数指定第 1 个参数中有效位的数目。函数一次一位地对第 1 个参数值进行移位，所以每个位迟早都会出现在最右边的那个位置。所有的位

逐个加在一起，所以在循环结束之后，我们就得到第 1 个参数值的位模式中 1 的个数。最后，对这个数进行测试，看看它的最低有效位是不是 1。如果不是，那么说明 1 的个数就是偶数个。

这个函数的有趣特性是在它的执行过程中，它会破坏这两个参数的值。但这并无妨，因为参数是通过传值调用的，函数所使用的值是实际参数的一份拷贝。破坏这份拷贝并不会影响原先的值。

程序 7.3a 则有所不同：它希望修改调用程序传递的参数。这个函数的目的是交换调用程序所传递的这两个参数的值。但这个程序是无效的，因为它实际交换的是参数的拷贝，原先的参数值并未进行交换。

```
/*
** 交换调用程序中的两个整数(没有效果!)
*/

void
swap( int x, int y )
{
        int temp;

        temp = x;
        x = y;
        y = temp;
}
```

程序 7.3a 整数交换：无效的版本　　swap1.c

为了访问调用程序的值，你必须向函数传递指向你希望修改的变量的指针。接着函数必须对指针使用间接访问操作，修改需要修改的变量。程序 7.3b 使用了这个技巧：

```
/*
** 交换调用程序中的两个整数。
*/

void
swap( int *x, int *y )
{
        int temp;

        temp = *x;
        *x = *y;
        *y = temp;
}
```

程序 7.3b 整数交换：有效版本　　swap2.c

因为函数期望接受的参数是指针，所以我们应该按照下面的方式调用它：

```
swap (&a, &b);
```

程序 7.4 把一个数组的所有元素都设置为 0。n_elements 是一个标量参数，所以它是传值调用的。在函数中修改它的值并不会影响调用程序中的对应参数。另一方面，函数确实把调用程序的数组的所有元素设置为 0。数组参数的值是一个指针，下标引用实际上是对这个指针执行间接访问操作。

这个例子同时说明了另外一个特性。在声明数组参数时不指定它的长度是合法的，因为函数并不为数组元素分配内存。间接访问操作将访问调用程序中的数组元素。这样，一个单独的函数可以

访问任意长度的数组。对于 Pascal 程序员而言，这应该是个福音。但是，函数并没有办法判断数组参数的长度，所以函数如果需要这个值，它必须作为参数显式地传递给函数

```
/*
** 把一个数组的所有元素都设置为零。
*/

void
clear_array( int array[], int n_elements )
{
        /*
        ** 从数组最后一个元素开始，逐个清除数组中的所有元素。注意前缀自增避免了越出数组边界的可能性。
        */
        while( n_elements > 0 )
             array[ --n_elements ] = 0;
}
```

程序 7.4 将一个数组设置为零 clrarray.c

K&R C：

回想一下，在 K&R C 中，函数的参数是像下面这样声明的：

```
int
func(a, b, c)
int a;
char b;
float c;
{
...
```

避免使用这种旧风格的另一个理由是 K&R 编译器处理参数的方式稍有不同：在参数传递之前，char 和 short 类型的参数被提升为 int 类型，float 类型的参数被提升为 double 类型。这种转换被称为缺省参数提升(default argument promotion)。由于这个规则的存在，在 ANSI 标准之前的程序中，你会经常看到函数参数被声明为 int 类型，但实际上传递的是 char 类型。

警告：

为了保持兼容性，ANSI 编译器也会为旧式风格声明的函数执行这类转换。但是，使用原型的函数并不执行这类转换，所以混用这两种风格可能导致错误。

7.4 ADT 和黑盒

C 可以用于设计和实现**抽象数据类型**(ADT，abstract data type)，因为它可以限制函数和数据定义的作用域。这个技巧也被称为**黑盒**(black box)设计。抽象数据类型的基本想法是很简单的——模块具有功能说明和接口说明，前者说明模块所执行的任务，后者定义模块的使用。但是，模块的用户并不需要知道模块实现的任何细节，而且除了那些定义好的接口之外，用户不能以任何方式访问模块。

限制对模块的访问是通过 static 关键字的合理使用实现的，它可以限制对那些并非接口的函数和数据的访问。例如，考虑一个用于维护一个地址/电话号码列表的模块。模块必须提供函数，根据一个指定的名字查找地址和电话号码。但是，列表存储的方式是依赖于具体实现的，所以这个信息为模块所私有，客户并不知情。

下一个例子程序说明了这个模块的一种可能的实现方法。程序 7.5a 定义了一个头文件，它定义了一些由客户使用的接口。程序 7.6b 展示了这个模块的实现[1]。

```
/*
** 地址列表模块的声明。
*/

/*
** 数据特征
**
**    各种数据的最大长度（包括结尾的 NUL 字节）和地址的最大数量。
*/
#define    NAME_LENGTH    30          /*允许出现的最长名字 */
#define    ADDR_LENGTH    100         /* 允许出现的最长地址 */
#define    PHONE_LENGTH   11          /* 允许出现的最长电话号码 */

#define    MAX_ADDRESSES  1000        /* 允许出现的最多地址个数 */

/*
** 接口函数
**
**    给出一个名字，查找对应的地址。
*/
char const *
lookup_address( char const *name );

/*
**    给出一个名字，查找对应的电话号码。
*/
char const *
lookup_phone( char const *name );
```

程序 7.5a 地址列表模块：头文件　　addrlist.h

```
/*
** 用于维护一个地址列表的抽象数据类型。
*/

#include "addrlist.h"
#include <stdio.h>

/*
**    每个地址的三个部分，分别保存于三个数组的对应元素中。
*/
static  char    name[MAX_ADDRESSES][NAME_LENGTH];
static  char    address[MAX_ADDRESSES][ADDR_LENGTH];
static  char    phone[MAX_ADDRESSES][PHONE_LENGTH];

/*
**    这个函数在数组中查找一个名字并返回查找到的位置的下标。
**    如果这个名字在数组中并不存在，函数返回-1。
*/
static int
```

[1] 如果每个名字、地址和电话号码存储在一个结构中更好一些，但我们要等到第 10 章才讲述结构。

```
find_entry( char const *name_to_find )
{
     int entry;

     for( entry = 0; entry < MAX_ADDRESSES; entry += 1 )
          if( strcmp( name_to_find, name[ entry ] ) == 0 )
               return entry;

     return -1;
}

/*
**    给定一个名字，查找并返回对应的地址。
**    如果名字没有找到，函数返回一个 NULL 指针。
*/
char const *
lookup_address( char const *name )
{
     int entry;

     entry = find_entry( name );
     if( entry == -1 )
          return NULL;
     else
          return address[ entry ];
}

/*
**    给定一个名字，查找并返回对应的电话号码。
**    如果名字没有找到，函数返回一个 NULL 指针。
*/
char const *
lookup_phone( char const *name )
{
     int entry;

     entry = find_entry( name );
     if( entry == -1 )
          return NULL;
     else
          return phone[ entry ];
}
```

程序 7.5b 地址列表模块：实现 addrlist.c

程序 7.5 是一个黑盒的好例子。黑盒的功能通过规定的接口访问，在这个例子里，接口是函数 lookup_address 和 lookup_phone。但是，用户不能直接访问和模块实现有关的数据，如数组或辅助函数 find_entry，因为这些内容被声明为 static。

提示：

这种类型的实现威力在于它使程序的各个部分相互之间更加独立。例如，随着地址列表的记录条数越来越多，简单的线性查找可能太慢，或者用于存储记录的表可能装满。此时你可以重新编写查找函数，使它更富效率，可能是通过使用某种形式的散列表查找来实现。或者，你甚至可以放弃使用数组，转而为这些记录动态分配内存空间。但是，如果用户程序可以直接访问存储记录的表，

表的组织形式如果进行了修改，就有可能导致用户程序失败。

黑盒的概念使实现细节与外界隔绝，这就消除了用户试图直接访问这些实现细节的诱惑。这样，访问模块唯一可能的方法就是通过模块所定义的接口。

7.5 递归

C 通过运行时堆栈支持递归函数的实现[1]。递归函数就是直接或间接调用自身的函数。许多教科书都把计算阶乘和菲波那契数列用来说明递归，这是非常不幸的。在第 1 个例子里，递归并没有提供任何优越之处。在第 2 个例子中，它的效率之低是非常恐怖的。

这里有一个简单的程序，可用于说明递归。程序的目的是把一个整数从二进制形式转换为可打印的字符形式。例如，给出一个值 4267，我们需要依次产生字符 '4'、'2'、'6' 和 '7'。如果在 printf 函数中使用了%d 格式码，它就会执行这类处理。

我们采用的策略是把这个值反复除以 10，并打印各个余数。例如，4267 除 10 的余数是 7，但是我们不能直接打印这个余数。我们需要打印的是机器字符集中表示数字 '7' 的值。在 ASCII 码中，字符 '7' 的值是 55，所以我们需要在余数上加上 48 来获得正确的字符。但是，使用字符常量而不是整型常量可以提高程序的可移植性。考虑下面的关系：

```
'0' + 0 = '0'
'0' + 1 = '1'
'0' + 2 = '2'
etc.
```

从这些关系中，我们很容易看出在余数上加上 '0' 就可以产生对应字符的代码[2]。接着就打印出余数。下一步是取得商，4267/10 等于 426。然后用这个值重复上述步骤。

这种处理方法存在的唯一问题是它产生的数字次序正好相反，它们是逆向打印的。程序 7.6 使用递归来修正这个问题。

程序 7.6 中的函数是递归性质的，因为它包含了一个对自身的调用。乍一看，函数似乎永远不会终止。当函数调用时，它将调用自身，第 2 次调用还将调用自身，以此类推，似乎会永远调用下去。但是，事实上并不会出现这种情况。

这个程序的递归实现了某种类型的螺旋状 while 循环。while 循环在循环体每次执行时必须取得某种进展，逐步迫近循环终止条件。递归函数也是如此，它在每次递归调用后必须越来越接近某种限制条件。当递归函数符合这个限制条件时，它便不再调用自身。

在程序 7.6 中，递归函数的限制条件就是变量 quotient 为零。在每次递归调用之前，我们都把 quotient 除以 10，所以每递归调用一次，它的值就越来越接近零。当它最终变成零时，递归便告终止。

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

void
```

[1] 有趣的是，标准并未说明递归需要堆栈。但是，堆栈非常适合于实现递归，所以许多编译器都使用堆栈来实现递归。

[2] 这些关系要求数字在字符集中必须连续。所有常用的字符集都符合这个要求。

```
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 7.6 将二进制整数转换为字符 btoa.c

递归是如何帮助我们以正确的顺序打印这些字符呢？下面是这个函数的工作流程。

1. 将参数值除以 10。
2. 如果 quotient 的值为非零，调用 binary_to_ascii 打印 **quotient** 当前值的各位数字。
3. 接着，打印步骤 1 中除法运算的余数。

注意在第 2 个步骤中，我们需要打印的是 quotient 当前值的各位数字。我们所面临的问题和最初的问题完全相同，只是变量 quotient 的值变小了。我们用刚刚编写的函数（把整数转换为各个数字字符并打印出来）来解决这个问题。由于 quotient 的值越来越小，所以递归最终会终止。

一旦你理解了递归，阅读递归函数最容易的方法不是纠缠于它的执行过程，而是相信递归函数会顺利完成它的任务。如果你的每个步骤正确无误，你的限制条件设置正确，并且每次调用之后更接近限制条件，递归函数总是能够正确地完成任务。

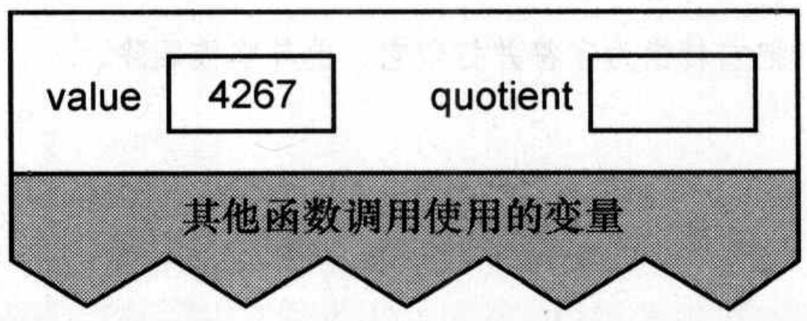
7.5.1 追踪递归函数

但是，为了能理解递归的工作原理，你需要追踪递归调用的执行过程，所以让我们来进行这项工作。追踪一个递归函数执行过程的关键是理解函数中所声明的变量是如何存储的。当函数被调用时，它的变量的空间是创建于运行时堆栈上的。以前调用的函数的变量仍保留在堆栈上，但它们被新函数的变量所掩盖，因此是不能被访问的。

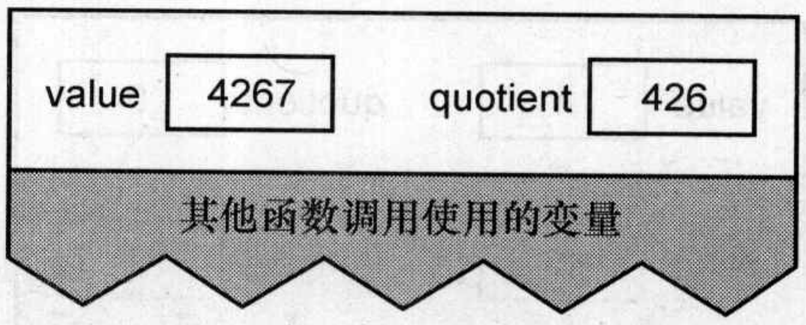
当递归函数调用自身时，情况也是如此。每进行一次新的调用，都将创建一批变量，它们将掩盖递归函数前一次调用所创建的变量。当我们追踪一个递归函数的执行过程时，必须把分属不同次调用的变量区分开来，以避免混淆。

程序 7.6 的函数有两个变量：参数 value 和局部变量 quotient。下面的一些图显示了堆栈的状态，当前可以访问的变量位于栈顶。所有其他调用的变量饰以灰色阴影，表示它们不能被当前正在执行的函数访问。

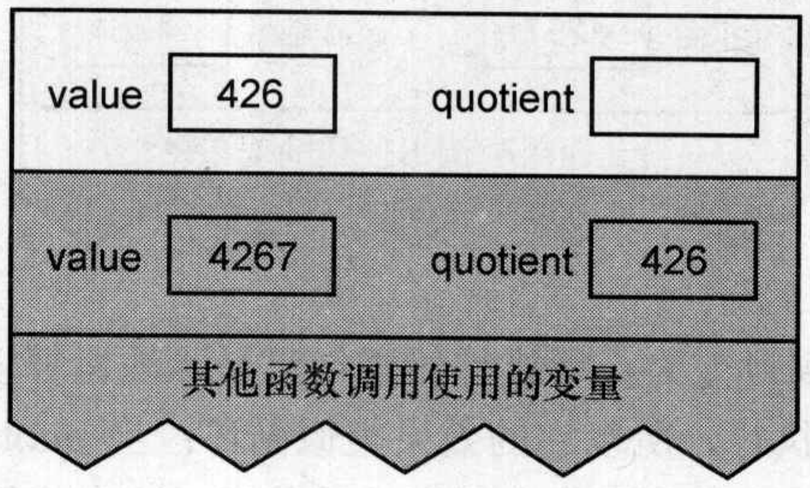
假定我们以 4267 这个值调用递归函数。当函数刚开始执行时，堆栈的内容如下图所示。



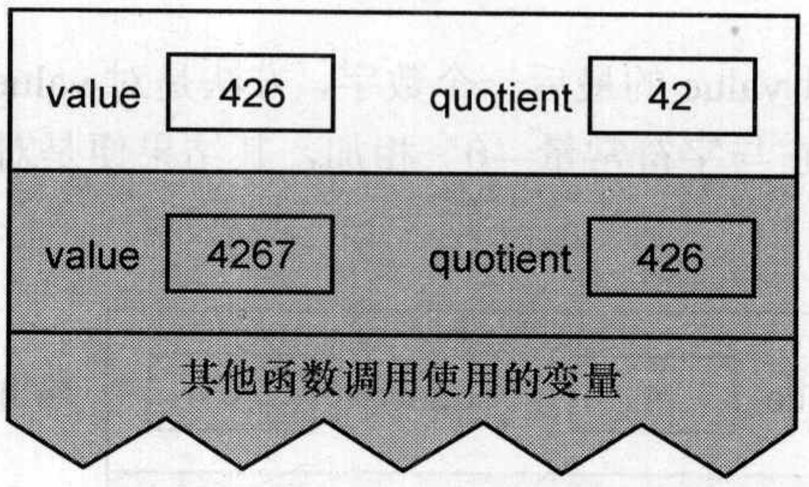
执行除法运算之后，堆栈的内容如下：



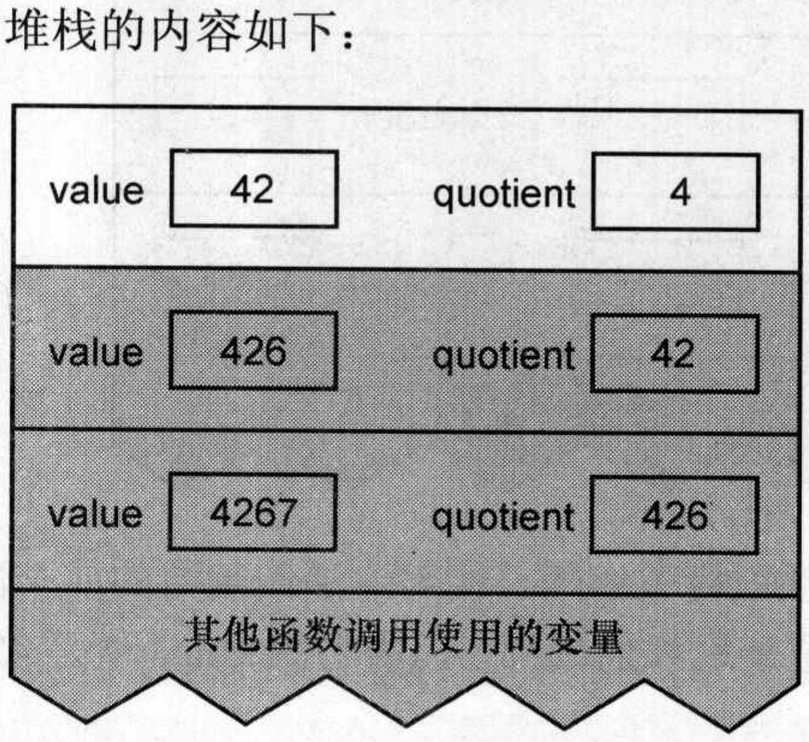
接着，if 语句判断出 quotient 的值非零，所以对该函数执行递归调用。当这个函数第二次被调用之初，堆栈的内容如下：



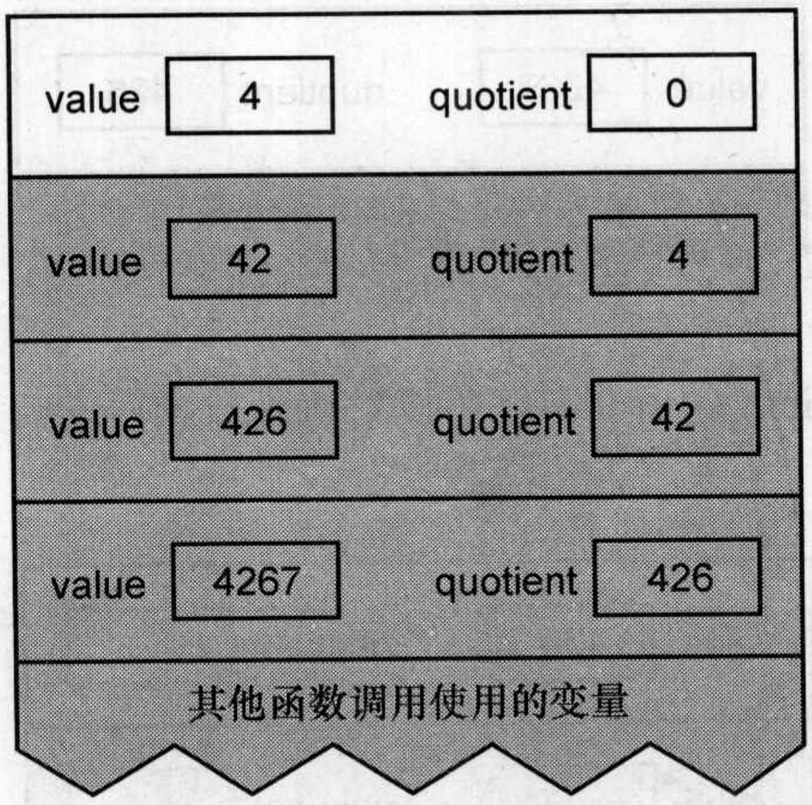
堆栈上创建了一批新的变量，隐藏了前面的那批变量，除非当前这次递归调用返回，否则它们是不能被访问的。再次执行除法运算之后，堆栈的内容如下：



quotient 的值现在为 42，仍然非零，所以需要继续执行递归调用，并再创建一批变量。在执行完这次调用的除法运算之后，堆栈的内容如下：



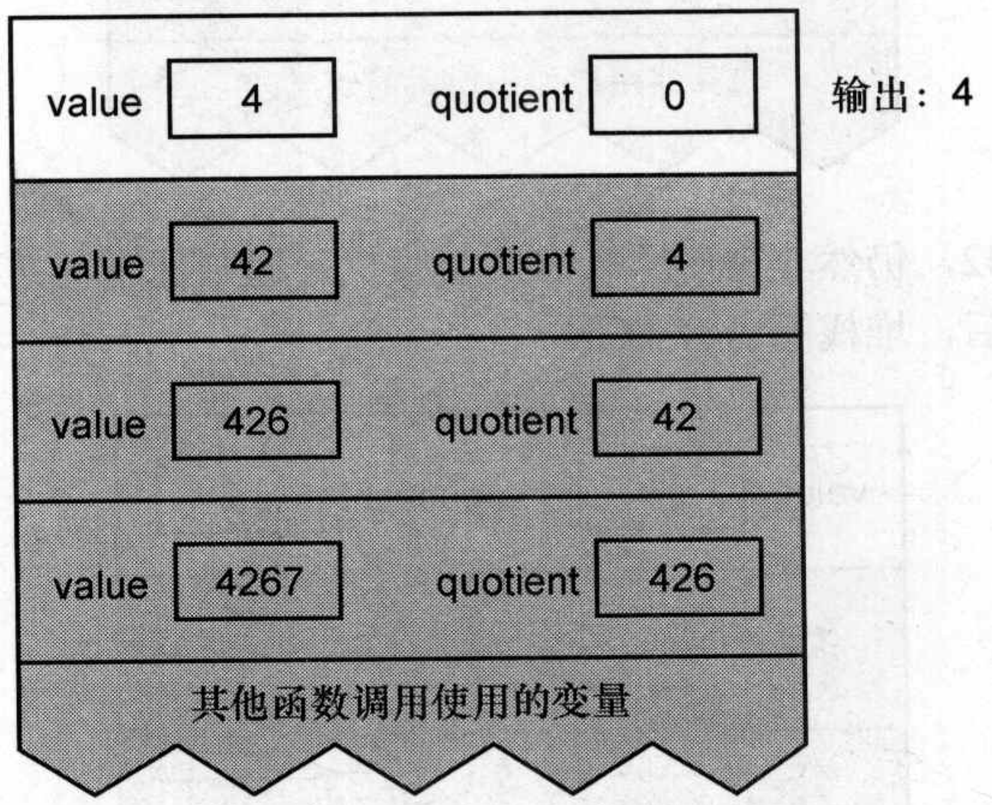
此时，quotient 的值还是非零，仍然需要执行递归调用。在执行除法运算之后，堆栈的内容如下：



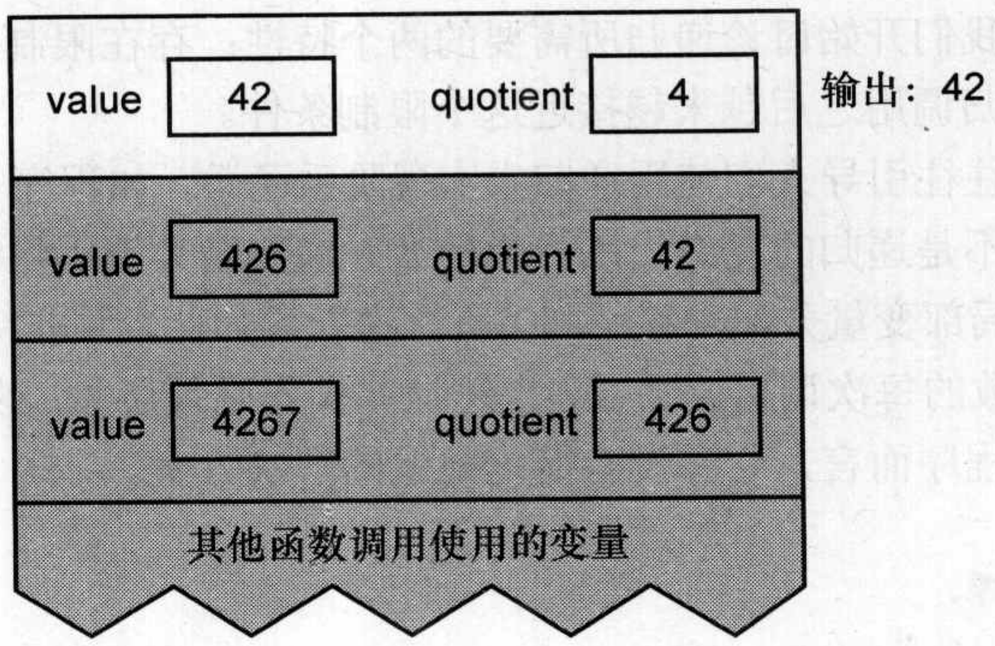
不算递归调用语句本身，到目前为止所执行的语句只是除法运算以及对 quotient 的值进行测试。由于递归调用使这些语句重复执行，所以它的效果类似循环：当 quotient 的值非零时，把它的值作为初始值重新开始循环。但是，递归调用将会保存一些信息（这点与循环不同），也就是保存在堆栈中的变量值。这些信息很快就会变得非常重要。

现在 quotient 的值变成了零，递归函数便不再调用自身，而是开始打印输出。然后函数返回，并开始销毁堆栈上的变量值。

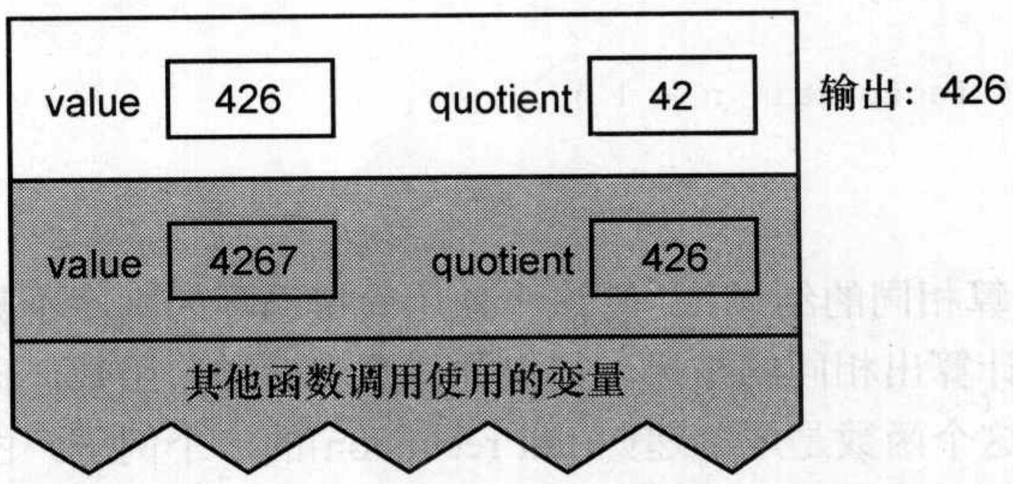
每次调用 putchar 得到变量 value 的最后一个数字，方法是对 value 进行模 10 取余运算，其结果是一个 0 到 9 之间的整数。把它与字符常量‘0’相加，其结果便是对应于这个数字的 ASCII 字符，然后把这个字符打印出来。



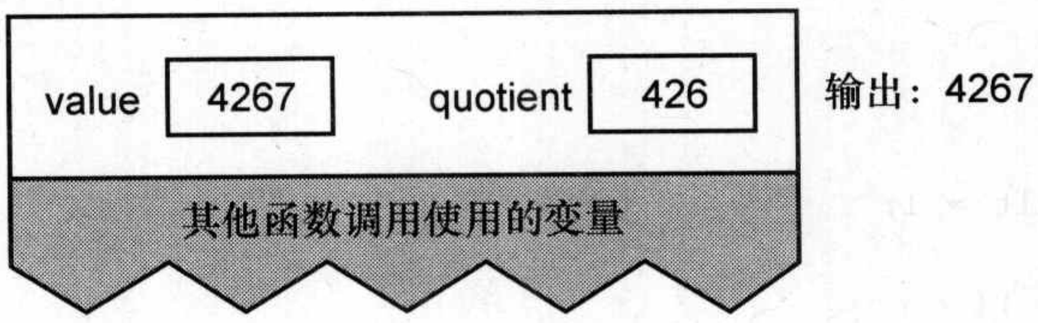
接着函数返回，它的变量从堆栈中销毁。接着，递归函数的前一次调用重新继续执行，它所使用的是自己的变量，它们现在位于堆栈的顶部。因为它的 value 值是 42，所以调用 putchar 后打印出来的数字是 2。



接着递归函数的这次调用也返回，它的变量也被销毁，此时位于堆栈顶部的是递归函数再前一次调用的变量。递归调用从这个位置继续执行，这次打印的数字是 6。在这次调用返回之前，堆栈的内容如下：



现在我们已经展开了整个递归过程，并回到该函数最初的调用。这次调用打印出数字 7，也就是它的 value 参数除 10 的余数。



然后，这个递归函数就彻底返回到其他函数调用它的地点。

如果你把打印出来的字符一个接一个排在一起，出现在打印机或屏幕上，你将看到正确的值：4267。

7.5.2　递归与迭代

递归是一种强有力的技巧，但和其他技巧一样，它也可能被误用。这里就有一个例子。阶乘的定义往往就是以递归的形式描述的，如下所示：

$$\text{factorial}(n) = \begin{cases} n \leqslant 0: 1 \\ n > 0: n \times \text{factorial}(n-1) \end{cases}$$

这个定义同时具备了我们开始讨论递归所需要的两个特性：存在限制条件，当符合这个条件时递归便不再继续；每次递归调用之后越来越接近这个限制条件。

用这种方式定义阶乘往往引导人们使用递归来实现阶乘函数，如程序 7.7a 所示。这个函数能够产生正确的结果，但它并不是递归的良好用法。为什么？递归函数调用将涉及一些运行时开销——参数必须压到堆栈中，为局部变量分配内存空间（所有递归均如此，并非特指这个例子），寄存器的值必须保存等。当递归函数的每次调用返回时，上述这些操作必须还原，恢复成原来的样子。所以，基于这些开销，对于这个程序而言，它并没有简化问题的解决方案。

```
/*
** 用递归方法计算 n 的阶乘。
*/

long
factorial( int n )
{
        if( n <= 0 )
            return 1;
        else
            return n * factorial( n - 1 );
}
```

程序 7.7a 递归计算阶乘 fact_rec.c

程序 7.7b 使用循环计算相同的结果。尽管这个使用简单循环的程序不甚符合前面阶乘的数学定义，但它却能更为有效地计算出相同的结果。如果你仔细观察递归函数，你会发现递归调用是函数所执行的最后一项任务。这个函数是尾部递归(tail recursion)的一个例子。由于函数在递归调用返回之后不再执行任何任务，所以尾部递归可以很方便地转换成一个简单循环，完成相同的任务。

```
*
** 用迭代方法计算 n 的阶乘。
*/

long
factorial( int n )
{
        int     result = 1;

        while( n > 1 ){
            result *= n;
            n -= 1;
        }

        return result;
}
```

程序 7.7b 迭代计算阶乘 fact_itr.c

提示：

许多问题是以递归的形式进行解释的，这只是因为它比非递归形式更为清晰。但是，这些问题的迭代实现往往比递归实现效率更高，虽然代码的可读性可能稍差一些。当一个问题相当复杂，难以用迭代形式实现时，此时递归实现的简洁性便可以补偿它所带来的运行时开销。

在程序 7.7a 中，递归在改善代码的可读性方面并无优势，因为程序 7.7b 的循环方案也同样简单。

这里有一个更为极端的例子，菲波那契数就是一个数列，数列中每个数的值就是它前面两个数的和。这种关系常常用递归的形式进行描述：

$$\text{Fibonacci(n)} = \begin{cases} n \leqslant 1: & 1 \\ n = 2: & 1 \\ n > 2: & \text{Fibonacci(n-1)} + \text{Fibonacci(n-2)} \end{cases}$$

同样，这种递归形式的定义容易诱导人们使用递归形式来解决问题，如程序 7.8a 所示。这里有一个陷阱：它使用递归步骤计算 Fibonacci(n-1)和 Fibonacci(n-2)。但是，在计算 Fibonacci(n-1)时也将计算 Fibonacci(n-2)。这个额外的计算代价有多大呢？

答案是：它的代价远远不止一个冗余计算：每个递归调用都触发另外两个递归调用，而这两个调用的任何一个还将触发两个递归调用，再接下去的调用也是如此。这样，冗余计算的数量增长得非常快。例如，在递归计算 Fibonacci(10)时，Fibonacci(3)的值被计算了 21 次。但是，在递归计算 Fibonacci(30)时，Fibonacci(3)的值被计算了 317 811 次。当然，这 317 811 次计算所产生的结果是完全一样的，除了其中之一外，其余的纯属浪费。这个额外的开销真是相当恐怖！

```
/*
** 用递归方法计算第 n 个菲波那契数的值。
*/

long
fibonacci( int n )
{
        if( n <= 2 )
            return 1;

        return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

程序 7.8a 用递归计算菲波那契数　　fib_rec.c

现在考虑程序 7.8b，它使用一个简单循环来代替递归。同样，这个循环形式不如递归形式符合前面菲波那契数的抽象定义，但它的效率提高了几十万倍！

当你使用递归方式实现一个函数之前，先问问你自己使用递归带来的好处是否抵得上它的代价。而且你必须小心：这个代价可能比初看上去要大得多。

```
/*
** 用迭代方法计算第 n 个菲波那契数的值。
*/

long
fibonacci( int n )
{
        long    result;
        long    previous_result;
        long    next_older_result;

        result = previous_result = 1;
```

```
        while( n > 2 ){
          n -= 1;
          next_older_result = previous_result;
          previous_result = result;
          result = previous_result + next_older_result;
        }
        return result;
}
```

程序 7.8b 用迭代计算菲波那契数 fib_iter.c

7.6 可变参数列表

在函数的原型中，列出了函数期望接受的参数，但原型只能显示固定数目的参数。让一个函数在不同的时候接受不同数目的参数是不是可以呢？答案是肯定的，但存在一些限制。考虑一个计算一系列值的平均值的函数。如果这些值存储于数组中，这个任务就太简单了，所以为了让问题变得更有趣一些，我们假定它们并不存储于数组中。程序 7.9a 试图完成这个任务。

这个函数存在几个问题。首先，它不对参数的数量进行测试，无法检测到参数过多这种情况。不过这个问题很好解决，简单加上测试就是了。其次，函数无法处理超过 5 个的值。要解决这个问题，你只有在已经很臃肿的代码中再增加一些类似的代码。

但是，当你试图用下面这种形式调用这个函数时，还存在一个更为严重的问题：

```
avg1 = average( 3, x, y, z );
```

这里只有 4 个参数，但函数具有 6 个形参。标准是这样定义这种情况的：这种行为的后果是未定义的。这样，第 1 个参数可能会与 n_values 对应，也可能与形参 v2 对应。你当然可以测试一下你的编译器是如何处理这种情况的，但这个程序显然是不可移植的。我们需要的是一种机制，它能够以一种良好定义的方法访问数量未定的参数列表。

```
/*
** 计算指定数目的值的平均值（差的方案）。
*/

float
average( int n_values, int v1, int v2, int v3, int v4, int v5 )
{
        float sum = v1;

        if( n_values >= 2 )
            sum += v2;
        if( n_values >= 3 )
            sum += v3;
        if( n_values >= 4 )
            sum += v4;
        if( n_values >= 5 )
            sum += v5;
        return sum / n_values;
}
```

程序 7.9a 计算标量参数的平均值：差的版本 average1.c

7.6.1　stdarg 宏

可变参数列表是通过宏来实现的，这些宏定义于 stdarg.h 头文件，它是标准库的一部分。这个头文件声明了一个类型 va_list 和三个宏——va_start、va_arg 和 va_end[1]。我们可以声明一个类型为 va_list 的变量，与这几个宏配合使用，访问参数的值。

程序 7.9b 使用这三个宏正确地完成了程序 7.9a 试图完成的任务。注意参数列表中的省略号：它提示此处可能传递数量和类型未确定的参数。在编写这个函数的原型时，也要使用相同的记法。

函数声明了一个名叫 var_arg 的变量，它用于访问参数列表的未确定部分。这个变量通过调用 va_start 来初始化。它的第 1 个参数是 va_list 变量的名字，第 2 个参数是省略号前最后一个有名字的参数。初始化过程把 var_arg 变量设置为指向可变参数部分的第 1 个参数。

为了访问参数，需要使用 va_arg，这个宏接受两个参数：va_list 变量和参数列表中下一个参数的类型。在这个例子中，所有的可变参数都是整型。在有些函数中，你可能要通过前面获得的数据来判断下一个参数的类型[2]。va_arg 返回这个参数的值，并使 var_arg 指向下一个可变参数。

最后，当访问完毕最后一个可变参数之后，我们需要调用 va_end。

7.6.2　可变参数的限制

注意，可变参数必须从头到尾按照顺序逐个访问。如果你在访问了几个可变参数后想半途中止，这是可以的。但是，如果你想一开始就访问参数列表中间的参数，那是不行的。另外，由于参数列表中的可变参数部分并没有原型，所以，所有作为可变参数传递给函数的值都将执行缺省参数类型提升。

```
/*
** 计算指定数量的值的平均值。
*/

#include <stdarg.h>

float
average( int n_values, ... )
{
    va_list  var_arg;
    int    count;
    float sum = 0;

    /*
    ** 准备访问可变参数。
    */
    va_start( var_arg, n_values );

    /*
    ** 添加取自可变参数列表的值。
    */
    for( count = 0; count < n_values; count += 1 ){
```

[1] 宏是由预处理器实现的，它将在第 14 章讨论。

[2] 例如，printf 检查格式字符串中的字符来判断它需要打印的参数的类型。

```
            sum += va_arg( var_arg, int );
        }

        /*
        ** 完成处理可变参数。
        */
        va_end( var_arg );

        return sum / n_values;
    }
```

程序 7.9b 计算标量参数的平均值：正确版本 average2.c

你可能同时注意到参数列表中至少要有一个命名参数。如果连一个命名参数也没有，你就无法使用 va_start。这个参数提供了一种方法，用于查找参数列表的可变部分。

对于这些宏，存在两个基本的限制。一个值的类型无法简单地通过检查它的位模式来判断，这两个限制就是这个事实的直接结果。

1．这些宏无法判断实际存在的参数的数量。

2．这些宏无法判断每个参数的类型。

要回答这两个问题，就必须使用命名参数。在程序 7.9b 中，命名参数指定了实际传递的参数数量，不过它们的类型被假定为整型。printf 函数中的命名参数是格式字符串，它不仅指定了参数的数量，而且指定了参数的类型。

警告：

如果你在 va_arg 中指定了错误的类型，那么其结果是不可预测的。这个错误是很容易发生的，因为 va_arg 无法正确识别作用于可变参数之上的缺省参数类型提升。char、short 和 float 类型的值实际上将作为 int 或 double 类型的值传递给函数。所以你在 va_arg 中使用后面这些类型时应该小心。

7.7 总结

函数定义同时描述了函数的参数列表和函数体（当函数被调用时所执行的语句），参数列表有两种可以接受的形式。K&R C 风格用一个单独的列表说明参数的类型，它出现在函数体的左花括号之前。新式风格（也是现在提倡的那种）则直接在参数列表中包含了参数的类型。如果函数体内没有任何语句，那么该函数就称为存根，它在测试不完整的程序时非常有用。

函数声明给出了和一个函数有关的有限信息，当函数被调用时就会用到这些信息。函数声明也有两种可以接受的形式。K&R 风格没有参数列表，它只是声明了函数返回值的类型。目前所提倡的新风格又称为函数原型，除了返回值类型之外，它还包含了参数类型的声明，这就允许编译器在调用函数时检查参数的数量和类型。你也可以把参数名放在函数的原型中，尽管不是必需，但这样做可以使原型对于其他读者更为有用，因为它传递了更多的信息。对于没有参数的函数，它的原型在参数列表中有一个关键字 void。常见的原型使用方法是把原型放在一个单独的文件中，当其他源文件需要这个原型时，就用#include 指令把这个文件包含进来。这个技巧可以使原型必需的拷贝份数降到最低，有助于提高程序的可维护性。

return 语句用于指定从一个函数返回的值。如果 return 语句没有包含返回值，或者函数不包含任何 return 语句，那么函数就没有返回值。在许多其他语言中，这类函数被称为过程。在 ANSI C

中，没有返回值的函数的返回类型应该声明为 void。

当一个函数被调用时，编译器如果无法看到它的任何声明，那么它就假定函数返回一个整型值。对于那些返回值不是整型的函数，在调用之前对它们进行声明是非常重要的，这可以避免由于不可预测的类型转换而导致的错误。对于那些没有原型的函数，传递给函数的实参将进行缺省参数提升：char 和 short 类型的实参被转换为 int 类型，float 类型的实参被转换为 double 类型。

函数的参数是通过传值方式进行传递的，它实际所传递的是实参的一份拷贝。因此，函数可以修改它的形参（也就是实参的拷贝），而不会修改调用程序实际传递的参数。数组名也是通过传值方式传递的，但它传给函数的是一个指向该数组的指针的拷贝。在函数中，如果在数组形参中使用了下标引用操作，就会引发间接访问操作，它实际所访问的是调用程序的数组元素。因此，在函数中修改参数数组的元素实际上修改的是调用程序的数组。这个行为被称为传址调用。如果你希望在传递标量参数时也具有传址调用的语义，你可以向函数传递指向参数的指针，并在函数中使用间接访问来访问或修改这些值。

抽象数据类型，或称黑盒，由接口和实现两部分组成。接口是公有的，它说明客户如何使用 ADT 所提供的功能。实现是私有的，是实际执行任务的部分。将实现部分声明为私有可以访止客户程序依赖于模块的实现细节。这样，当需要的时候，我们可以对实现进行修改，这样做并不会影响客户程序的代码。

递归函数直接或间接地调用自身。为了使递归能顺利进行，函数的每次调用必须获得一些进展，进一步靠近目标。当达到目标时，递归函数就不再调用自身。在阅读递归函数时，不必纠缠于递归调用的内部细节。你只要简单地认为递归函数将会执行它的预定任务即可。

有些函数是以递归形式进行描述的，如阶乘和菲波那契数列，但它们如果使用迭代方式来实现，效率会更高一些。如果一个递归函数内部所执行的最后一条语句就是调用自身时，那么它就被称为尾部递归。尾部递归可以很容易地改写为循环的形式，它的效率通常更高一些。

有些函数的参数列表包含可变的参数数量和类型，它们可以使用 stdarg.h 头文件所定义的宏来实现。参数列表的可变部分位于一个或多个普通参数（命名参数）的后面，它在函数原型中以一个省略号表示。命名参数必须以某种形式提示可变部分实际所传递的参数数量，而且如果预先知道的话，也可以提供参数的类型信息。当参数列表中可变部分的参数实际传递给函数时，它们将经历缺省参数提升。可变部分的参数只能从第 1 个到最后 1 个依次进行访问。

7.8　警告的总结

1. 错误地在其他函数的作用域内编写函数原型。
2. 没有为那些返回值不是整型的函数编写原型。
3. 把函数原型和旧式风格的函数定义混合使用。
4. 在 va_arg 中使用错误的参数类型，导致未定义的结果。

7.9　编程提示的总结

1. 在函数原型中使用参数名，可以给使用该函数的用户提供更多的信息。
2. 抽象数据类型可以减少程序对模块实现细节的依赖，从而提高程序的可靠性。
3. 当递归定义清晰的优点可以补偿它的效率开销时，就可以使用这个工具。

7.10 问题

1. 具有空函数体的函数可以作为存根使用。你如何对这类函数进行修改，使其更加有用？
2. 在 ANSI C 中，函数的原型并非必需。请问这个规定是优点还是缺点？
3. 如果在一个函数的声明中，它的返回值类型为 A，但它的函数体内有一条 return 语句，返回了一个类型为 B 的表达式。请问，这将导致什么后果？
4. 如果一个函数声明的返回类型为 void，但它的函数体内包含了一条 return 语句，返回了一个表达式。请问，这将导致什么后果？
5. 如果一个函数被调用之前，编译器无法看到它的原型，那么当这个函数返回一个不是整型的值时，会发生什么情况？
6. 如果一个函数被调用之前，编译器无法看到它的原型，如果当这个函数被调用时，实际传递给它的参数与它的形式参数不匹配，会发生什么情况？
7. 下面的函数有没有错误？如果有，错在哪里？

```
int
find_max( int array[10] )
{
        int     i;
        int     max = array[0];

        for( i = 1; i < 10; i += 1 )
                if( array[i] > max )
                        max = array[i];

        return max;
}
```

8. 递归和 while 循环之间是如何相似的？
9. 请解释把函数原型单独放在#include 文件中的优点。
10. 在你的系统中，进入递归形式的菲波那契函数，并在函数的起始处增加一条语句，它增加一个全局整型变量的值。现在编写一个 main 函数，把这个全局变量设置为 0 并计算 Fibonacci(1)。重复这个过程，计算 Fibonacci(2)至 Fibonacci(10)。在每个计算过程中分别调用了几次 Fibonacci 函数（用这个变量值表示）？这个全局变量值的增加和菲波那契数列本身有没有任何关联？基于上面这些信息，你能不能计算出 Fibonacchi(11)、Fibonacci(25)和 Fibonacci(50)分别调用了多少次 Fibonacci 函数？

7.11 编程练习

★★ 1. Hermite Polynomials（厄密多项式）是这样定义的：

$$H_n(x) = \begin{cases} n \leq 0: & 1 \\ n = 1: & 2x \\ n \geq 2: & 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x) \end{cases}$$

例如，$H_3(2)$的值是 40。请编写一个递归函数，计算 $H_n(x)$的值。你的函数应该与下面的原型匹配：

```
int hermite( int n, int x)
```

★★ 2. 两个整型值 M 和 N（M、N 均大于 0）的最大公约数可以按照下面的方法计算：

$$gcd(M, N) = \begin{cases} M \% N = 0: & N \\ M\%N = R,\ R>0: & gcd(N, R) \end{cases}$$

请编写一个名叫 gcd 的函数，它接受两个整型参数，并返回这两个数的最大公约数。如果这两个参数中的任何一个不大于零，函数应该返回零。

★★ 3. 为下面这个函数原型编写函数定义：

```
int ascii_to_integer( char *string );
```

这个字符串参数必须包含一个或多个数字，函数应该把这些数字字符转换为整数并返回这个整数。如果字符串参数包含了任何非数字字符，函数就返回零。请不必担心算术溢出。**提示**：这个技巧很简单——你每发现一个数字，把当前值乘以 10，并把这个值和新数字所代表的值相加。

★★★ 4. 编写一个名叫 max_list 的函数，它用于检查任意数目的整型参数并返回它们中的最大值。参数列表必须以一个负值结尾，提示列表的结束。

★★★★ 5. 实现一个简化的 printf 函数，它能够处理%d、%f、%s 和%c 格式码。根据 ANSI 标准的原则，其他格式码的行为是未定义的。你可以假定已经存在函数 print_integer 和 print_float，用于打印这些类型的值。对于另外两种类型的值，使用 putchar 来打印。

★★★★ 6. 编写函数

```
void written_amount( unsigned int amount, char *buffer );
```

它把 amount 表示的值转换为单词形式，并存储于 buffer 中。这个函数可以在一个打印支票的程序中使用。例如，如果 amount 的值是 16 312，那么 buffer 中存储的字符串应该是

```
SIXTEEN THOUSAND THREE HUNDRED TWELVE
```

调用程序应该保证 buffer 缓冲区的空间足够大。

有些值可以用两种不同的方法进行打印。例如，1 200 可以是 ONE THOUSAND TWO HUNDRED 或 TWELVE HUNDRED。你可以选择一种你喜欢的形式。